ГЛАВА 2. МЕТОДЫ МАШИННОГО ОБУЧЕНИЯ С УЧИТЕЛЕМ

Как мы уже говорили ранее, машинное обучение с учителем является одним из наиболее часто используемых и успешных видов машинного обучения. В этой главе мы более подробно расскажем о машинном обучении с учителем и объясним работу нескольких популярных алгоритмов. Мы уже разбирали применение машинного обучения с учителем в главе 1: классификацию ирисов по нескольким сортам с использованием измерений физических характеристик цветов.

Вспомним, что обучение с учителем используется всякий раз, когда мы хотим предсказать определенный результат (ответ) по данному объекту, и у нас есть пары объект-ответ. Мы строим модель машинного обучения на основе этих пар объект-ответ, которые составляют наш обучающий набор данных. Наша цель состоит в том, чтобы получить точные прогнозы для новых, никогда ранее не встречавшихся данных. Машинное обучение с учителем часто требует вмешательства человека, чтобы получить обучающий набор данных, но потом оно автоматизирует и часто ускоряет решение трудоемких или неосуществимых задач.

Классификация и регрессия

Есть два основные задачи машинного обучения с учителем: *классификация* (*classification*) и *регрессия* (*regression*).

Цель классификации состоит в том, чтобы спрогнозировать *метку класса* (*class label*), которая представляет собой выбор из заранее определенного списка возможных вариантов. В главе 1 мы использовали пример классификации ирисов, когда относили цветок к одному из трех возможных сортов. Классификация иногда разделяется на *бинарную классификацию* (*binary classification*), которая является частным случаем разделения на два класса, и мультиклассовую классификацию (*multiclass classification*), когда в классификации участвует более двух классов. Бинарную классификацию можно представить как попытку ответить на поставленный вопрос в формате «да/нет». Классификация электронных писем на спам и не-спам является примером бинарной классификации. В данной задаче бинарной классификации ответ «да/нет» дается на вопрос «является ли это электронное письмо спамом?»

В бинарной классификации мы часто говорим о том, что один класс является *положительным* (*positive*) классом, а другой класс является *отрицательным* (*negative*) классом. При этом «положительный» означает здесь не наличие выгоды (ценности), а объект исследования. Таким образом, при поиске спама, положительным классом может быть класс «спам». Вопрос о том, какой из этих двух классов будет положительным, часто субъективен и зависит от предметной области исследования.

С другой стороны, пример классификации ирисов является примером мультиклассовой классификации. Еще один пример – прогнозирование языка веб-сайта. Классами здесь будет заранее определенный список возможных языков.

Цель регрессии состоит в том, чтобы спрогнозировать непрерывное число или *число с плавающей точкой* (*floating-point number*), если использовать термины программирования, или *вещественное число* (*real number*), если говорить языком математических терминов. Прогнозирование годового дохода человека в зависимости от его образования, возраста и места жительства является примером регрессионной задачи. Прогнозируемое значение дохода представляет собой *сумму* (*amount*) и может быть любым числом в заданном диапазоне. Другой пример регрессионной задачи – прогнозирование объема урожая зерна на ферме в зависимости от таких атрибутов, как объем предыдущего урожая, погода, и количество сотрудников, работающих на ферме. И снова объем урожая может быть любым числом.

Самый простой способ отличить классификацию от регрессии – спросить, заложена ли в полученном ответе определенная непрерывность (преемственность). Если полученные результаты непрерывно связаны друг с другом, то решаемая задача является задачей регрессии. Возьмем прогнозирование годового дохода. Здесь ясно видна непрерывность ответа. Разница между годовым доходом в 40000$ или 40001$ не существенна, хотя речь идет о разных денежных суммах. Если наш алгоритм предсказывает 39999$ или 40001$, в то время как он должен предсказать 40000$ (реальное значение годового дохода), мы не будем настаивать на том, что разница существенна. Наоборот, в задаче распознавании языка веб-сайта (задаче классификации) ответы четко определены. Контент сайта может быть написан либо на одном конкретном языке, либо на другом. Между языками нет непрерывной связи, не существует языка, находящегося между английским и французским.[6]

[6] Мы просим прощения у лингвистов за упрощенное представление о языках как отдельных и фиксированных объектах.

Обобщающая способность, переобучение и недообучение

В машинном обучении с учителем нам нужно построить модель на обучающих данных, а затем получить точные прогнозы для новых, еще не встречавшихся нам данных, которые имеют те же самые характеристики, что и использованный нами обучающий набор. Если модель может выдавать точные прогнозы на ранее не встречавшихся данных, мы говорим, что модель обладает способностью *обобщать* (*generalize*) результат на тестовые данные. Нам необходимо построить модель, которая будет обладать максимальной обобщающей способностью.

Обычно мы строим модель таким образом, чтобы она давала точные прогнозы на обучающем наборе. Если обучающий и тестовый наборы имеют много общего между собой, можно ожидать, что модель будет точной и на тестовом наборе. Однако в некоторых случаях этого не происходит. Например, если мы строим очень сложные модели, необходимо помнить, что на обучающей выборке можно получить произвольную правильность.

Давайте взглянем на выдуманный пример, чтобы проиллюстрировать этот тезис. Скажем, начинающий специалист по анализу данных хочет спрогнозировать покупку клиентом лодки на основе записей о клиентах, которые ранее приобрели лодку, и клиентах, которые не заинтересованы в покупке лодки.[7] Цель состоит в том, чтобы отправить рекламные письма клиентам, которые, вероятно, хотят совершить покупку, и не беспокоить клиентов, не заинтересованных в покупке.

Предположим, у нас есть записи о клиентах, приведенные в таблице 2.1.

[7] В действительности это сложная проблема. Хотя мы знаем, что остальные клиенты еще не купили у нас лодку, они могли купить ее у кого-то еще, или они могут откладывать сбережения и планировать покупку лодки в будущем.

Возраст	Количество автомобилей в собственности	Есть собственный дом	Количество детей	Семейное положение	Есть собака	Купил лодку
66	1	да	2	вдовец	нет	да
52	2	да	3	женат	нет	да
22	0	нет	0	женат	да	нет
25	1	нет	1	холост	нет	нет
44	0	нет	2	разведен	да	нет
39	1	да	2	женат	да	нет
26	1	нет	2	холост	нет	нет
40	3	да	1	женат	да	нет
53	2	да	2	разведен	нет	да
64	2	да	3	разведен	нет	нет
58	2	да	2	женат	да	да
33	1	нет	1	холост	нет	нет

Таблица 2.1 Пример данных о клиентах

Поработав с данными некоторое время, наш начинающий специалист формулирует следующее правило «если клиент старше 45 лет, у него менее трех детей, либо у него трое и он женат, то он скорее всего купит лодку». Если спросить, насколько хорошо работает это правило, наш специалист воскликнет «оно дает 100%-ную правильность!» И в самом деле, для данных, приведенных в таблице, это правило демонстрирует идеальную правильность. Мы могли бы сформулировать массу правил, объясняющих покупку лодки. Значения возраста появляются в наборе данных лишь один раз, таким образом, мы могли бы сказать, что люди в возрасте 66, 52, 53 и 58 лет хотят купить лодку, тогда как все остальные не собираются ее покупать. Несмотря на то что можно сформулировать множество правил, которые хорошо работают для этих наблюдений, следует помнить о том, что нам не интересны прогнозы для этого набора данных, мы уже знаем ответы для этих клиентов. Мы хотим знать, могут ли новые клиенты купить лодку. Поэтому нам нужно правило, которое будет хорошо работать для новых клиентов, и достижение 100%-ной правильности на обучающей выборке не поможет нам в этом. Нельзя ожидать, что правило, сформулированное нашим специалистом, будет так же хорошо работать и для новых клиентов. Похоже, с этим у нас сложность, ведь у нас мало данных. Например, часть правила «либо трое и он женат» сформулировано по одному клиенту.

Единственный показатель качества работы алгоритма на новых данных – это использование тестового набора. Однако интуитивно[8] мы ожидаем, что простые модели должны лучше обобщать результат на новые данные. Если бы правило звучало «люди старше 50 лет хотят купить лодку» и оно объясняло бы поведение всех клиентов, мы

[8] И совершенно обоснованно с точки зрения математики.

доверяли бы ему больше, чем правилу, которое помимо возраста включало бы количество детей и семейное положение. Поэтому нам всегда нужно искать самую простую модель. Построение модели, которая слишком сложна для имеющегося у нас объема информации (что и сделал наш начинающий специалист по анализу данных), называется *переобучением* (*overfitting*). Переобучение происходит, когда ваша модель слишком точно подстраивается под особенности обучающего набора и вы получаете модель, которая хорошо работает на обучающем наборе, но не умеет обобщать результат на новые данные. С другой стороны, если ваша модель слишком проста, скажем, вы сформулировали правило «все, у кого есть собственный дом, покупает лодку», вы, возможно, не смогли охватить все многообразие и изменчивость данных, и ваша модель будет плохо работать даже на обучающем наборе. Выбор слишком простой модели называется *недообучением* (*underfitting*).

Чем сложнее модель, тем лучше она будет работать на обучающих данных. Однако, если наша модель становится слишком сложной, мы начинаем уделять слишком много внимания каждой отдельной точке данных в нашем обучающем наборе, и эта модель не будет хорошо обобщать результат на новые данные.

Существует оптимальная точка, которая позволяет получить наилучшую обобщающую способность. Собственно это и есть модель, которую нам нужно найти.

Компромисс между переобучением и недообучением показан на рис. 2.1.

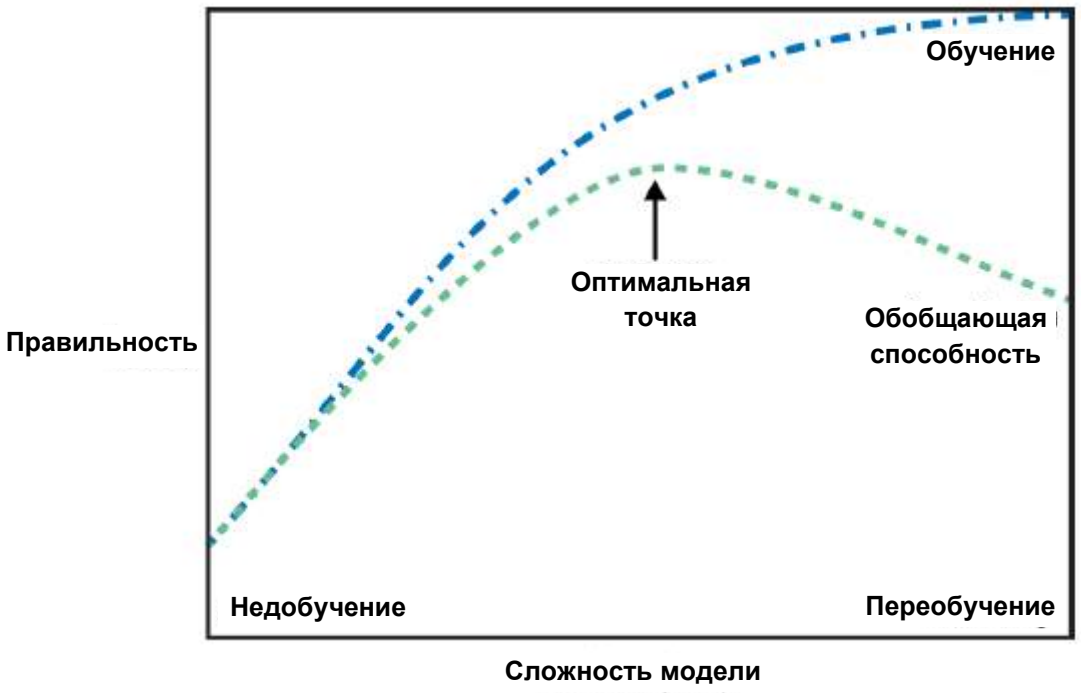

Рис. 2.1 Компромисс между сложностью модели и правильностью на обучающей и тестовой выборках

Взаимосвязь между сложностью модели и размером набора данных

Важно отметить, что сложность модели тесно связана с изменчивостью входных данных, содержащихся в вашем обучающем наборе: чем больше разнообразие точек данных в вашем наборе, тем более сложную модель можно использовать, не беспокоясь о переобучении. Обычно больший объем данных дает большее разнообразие, таким образом, большие наборы данных позволяют строить более сложные модели. Однако простое дублирование одних и тех же точек данных или сбор очень похожих данных здесь не поможет.

Возвращаясь к продажам лодок, можно сказать, что если бы у нас было более 10000 строк данных о клиентах и все они подчинялись бы правилу «если клиент старше 45 лет, у него менее трех детей, либо трое и он женат, то он скорее всего купит лодку», мы бы с гораздо большей вероятностью поверили в это правило, чем если бы оно было сформулировано лишь по 12 строкам таблицы 2.1.

Увеличение объема данных и построение более сложных моделей часто творят чудеса при решении задач машинного обучения с учителем. В этой книге мы сосредоточимся на работе с данными фиксированного размера. В действительности вы, как правило, сами можете определить объем собираемых данных, и это может оказаться более полезным, чем корректировка и настройка вашей модели. Никогда не стоит недооценивать преимущества увеличения объема данных.

Алгоритмы машинного обучения с учителем

Теперь мы рассмотрим наиболее популярные алгоритмы машинного обучения и объясним, как они обучаются на основе данных и как вычисляют прогнозы. Кроме того, мы расскажем о том, как принцип сложности реализуется для каждой из этих моделей, и покажем, как тот или иной алгоритм строит модель. Мы рассмотрим преимущества и недостатки каждого алгоритма, а также расскажем о том, применительно к каким данным лучше всего использовать тот или иной алгоритм. Мы также объясним значение наиболее важных параметров и опций. Многие алгоритмы имеют опции классификации и регрессии, поэтому мы опишем обе опции.

Необязательно детально вчитываться в описание каждого алгоритма, но понимание модели даст вам лучшее представление о различных способах работы алгоритмов машинного обучения. Кроме того, эту главу можно использовать в качестве справочного руководства, и вы можете вернуться к ней, если не знаете, как работает тот или иной алгоритм.

Некоторые наборы данных

Для иллюстрации различных алгоритмов мы будем использовать несколько наборов данных. Некоторые наборы данных будет небольшими и синтетическими (то есть выдуманными), призванными подчеркнуть отдельные аспекты алгоритмов. Другие наборы данных будут большими, реальными примерами.

Примером синтетического набора данных для двухклассовой классификации является набор данных `forge`, который содержит два признака. Программный код, приведенный ниже, создает диаграмму рассеяния (рис. 2.2), визуализируя все точки данных в этом наборе. На графике первый признак отложен на оси x, а второй – по оси y. Как это всегда бывает в диаграммах рассеяния, каждая точка данных представлена в виде одного маркера. Цвет и форма маркера указывает на класс, к которому принадлежит точка:

In[2]:
```
# генерируем набор данных
X, y = mglearn.datasets.make_forge()
# строим график для набора данных
%matplotlib inline
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(["Класс 0", "Класс 1"], loc=4)
plt.xlabel("Первый признак")
plt.ylabel("Второй признак")
print("форма массива X: {}".format(X.shape))
```

Out[2]:
```
форма массива X: (26, 2)
```

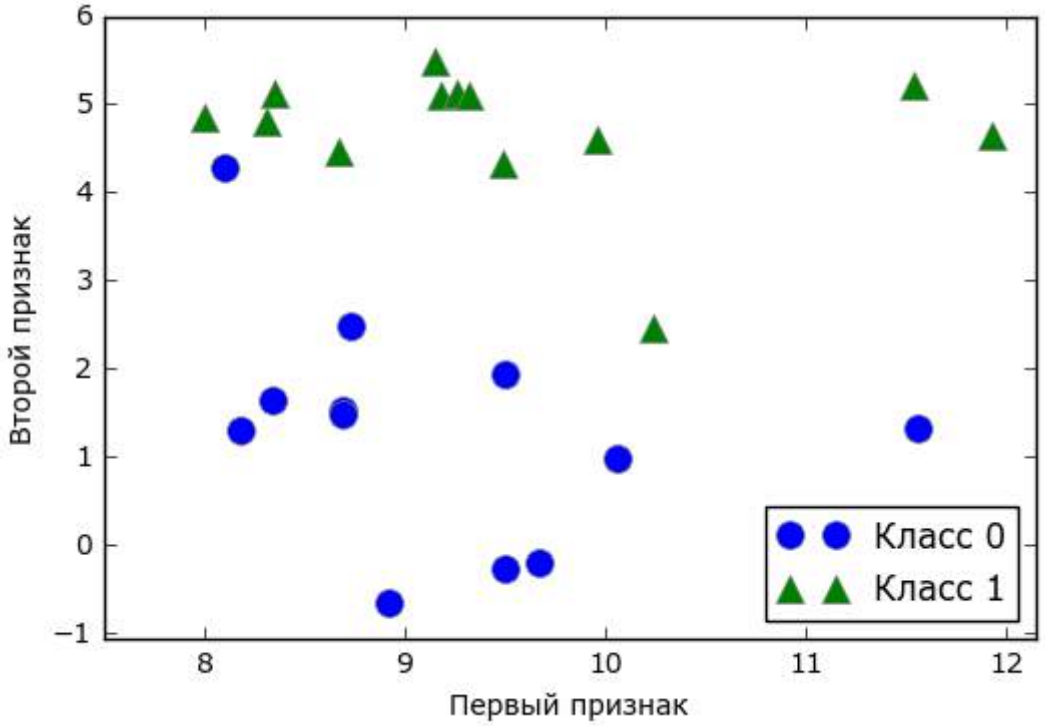

Рис. 2.2 Диаграмма рассеяния для набора данных forge

Как видно из сводки по массиву `X`, этот набор состоит из 26 точек данных и 2 признаков. Для иллюстрации алгоритмов регрессии, мы воспользуемся синтетическим набором `wave`. Набор данных имеет единственный входной признак и непрерывную целевую переменную или *отклик* (*response*), который мы хотим смоделировать. На рисунке, построенном здесь (рис. 2.3), по оси x располагается единственный признак, а по оси y – целевая переменная (ответ).

```
In[3]:
X, y = mglearn.datasets.make_wave(n_samples=40)
plt.plot(X, y, 'o')
plt.ylim(-3, 3)
plt.xlabel("Признак")
plt.ylabel("Целевая переменная")
```

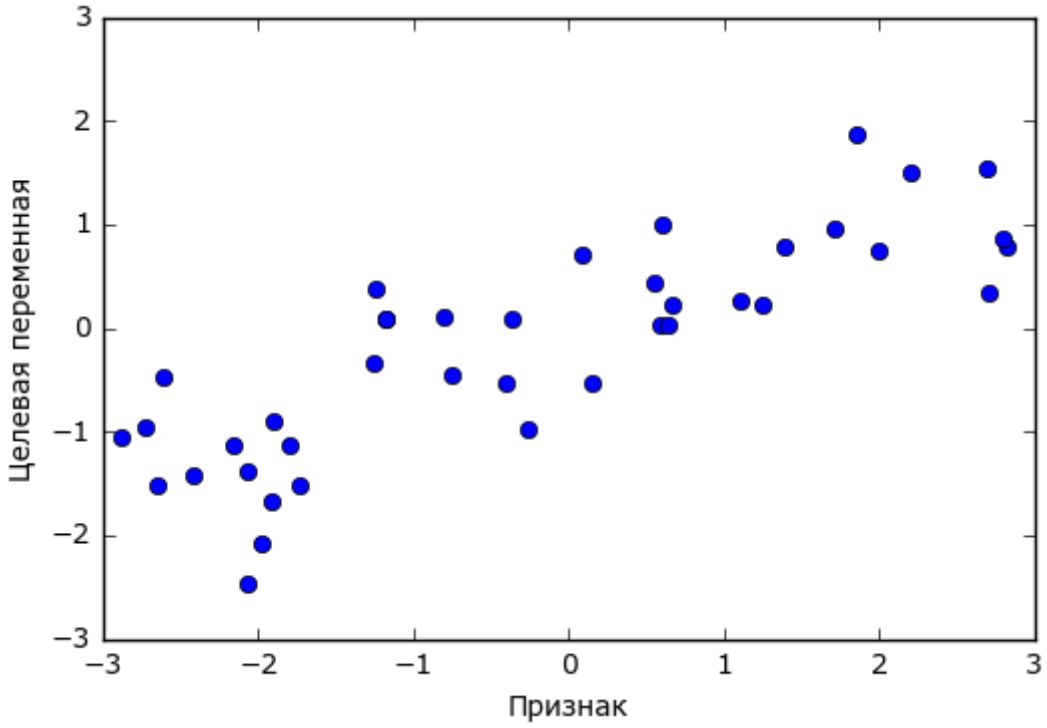

Рис. 2.3 График для набора данных wave, по оси x отложен признак, по оси y – целевая переменная

Мы используем эти очень простые, низкоразмерные наборы данных, потому что их легко визуализировать – печатная страница имеет два измерения, и данные, которые содержат более двух признаков, графически представить трудно. Вывод, полученный для набора с небольшим числом признаков или *низкоразмерном* (*low-dimensional*) наборе, возможно, не подтвердится для набора данных с большим количеством признаков или *высокоразмерного* (*high-dimensional*) набора. Если вы помните об этом, проверка алгоритма на низкоразмерном наборе данных может оказаться очень полезной.

Мы дополним эти небольшие синтетические наборы данных двумя реальными наборами, которые включены в `scikit-learn`. Один из них – набор данных по раку молочной железы Университета Висконсин (`cancer` для краткости), в котором записаны клинические измерения опухолей молочной железы. Каждая опухоль обозначается как «benign» («доброкачественная», для неагрессивных опухолей) или malignant («злокачественная», для раковых опухолей), и задача состоит в том, чтобы на основании измерений ткани дать прогноз, является ли опухоль злокачественной.

Данные можно загрузить из `scikit-learn` с помощью функции `load_breast_cancer`:

```
In[4]:
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("Ключи cancer(): \n{}".format(cancer.keys()))

Out[4]:
Ключи cancer():
dict_keys(['feature_names', 'data', 'DESCR', 'target', 'target_names'])
```

Наборы данных, которые включены в `scikit-learn`, обычно хранятся в виде объектов `Bunch`, которые содержат некоторую информацию о наборе данных, а также фактические данные. Все, что вам нужно знать об объектах `Bunch` – это то, что они похожи на словари, с тем преимуществом, что вы можете прочитать значения, используя точку (`bunch.key` вместо `bunch['key']`)

Набор данных включает 569 точек данных и 30 признаков.

```
In[5]:
print("Форма массива data для набора cancer: {}".format(cancer.data.shape))

Out[5]:
Форма массива data для набора cancer: (569, 30)
```

Из 569 точек данных 212 помечены как злокачественные, а 357 как доброкачественные.

```
In[6]:
print("Количество примеров для каждого класса:\n{}".format(
      {n: v for n, v in zip(cancer.target_names, np.bincount(cancer.target))}))

Out[6]:
Количество примеров для каждого класса:
{'benign': 357, 'malignant': 212}
```

Чтобы получить содержательное описание каждого признака, взглянем на атрибут `feature_names`:

```
In[7]:
print("Имена признаков:\n{}".format(cancer.feature_names))
```

```
Out[7]:
Имена признаков:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Если вам интересно, то более подробную информацию о данных можно получить, прочитав `cancer.DESCR`.

Кроме того, для задач регрессии мы будем использовать реальный набор данных – набор данных Boston Housing. Задача, связанная с этим набором данных, заключается в том, чтобы спрогнозировать медианную стоимость домов в нескольких районах Бостона в 1970-е годы на основе такой информации, как уровень преступности, близость к Charles River, удаленность от радиальных магистралей и т.д. Набор данных содержит 506 точек данных и 13 признаков:

```
In[8]:
from sklearn.datasets import load_boston
boston = load_boston()
print("форма массива data для набора boston: {}".format(boston.data.shape))

Out[8]:
форма массива data для набора boston: (506, 13)
```

Опять же, вы можете получить более подробную информацию о наборе данных, прочитав атрибут `boston.DESCR`. В данном случае мы более детально проанализируем набор данных, учтя не только 13 измерений в качестве входных признаков, но и приняв во внимание все *взаимодействия* (*interactions*) между признаками. Иными словами, мы будем учитывать в качестве признаков не только уровень преступности и удаленность от радиальных магистралей по отдельности, но и взаимодействие уровень преступности–удаленность от радиальных магистралей. Включение производных признаков называется *конструированием признаков* (*feature engineering*), которое мы рассмотрим более подробно в главе 4. Набор данных с производными признаками можно загрузить с помощью функции `load_extended_boston`:

```
In[9]:
X, y = mglearn.datasets.load_extended_boston()
print("форма массива X: {}".format(X.shape))

Out[9]:
форма массива X: (506, 104)
```

Полученные 104 признака – 13 исходных признаков плюс 91 производный признак.

Мы будем использовать эти наборы данных, чтобы объяснить и проиллюстрировать свойства различных алгоритмов машинного обучения. Однако сейчас давайте перейдем к самим алгоритмам. Во-первых, мы вернемся к алгоритму *k* ближайших соседей, который рассматривали в предыдущей главе.

Метод *k* ближайших соседей

Алгоритм *k* ближайших соседей, возможно, является самым простым алгоритмом машинного обучения. Построение модели заключается в запоминании обучающего набора данных. Для того, чтобы сделать прогноз для новой точки данных, алгоритм находит ближайшие к ней точки обучающего набора, то есть находит «ближайших соседей».

Классификация с помощью *k* соседей

В простейшем варианте алгоритм *k* ближайших соседей рассматривает лишь одного ближайшего соседа – точку обучающего набора, ближе всего расположенную к точке, для которой мы хотим получить прогноз. Прогнозом является ответ, уже известный для данной точки обучающего набора. На рис. 2.4 показано решение задачи классификации для набора данных `forge`:

```
In[10]:
mglearn.plots.plot_knn_classification(n_neighbors=1)
```

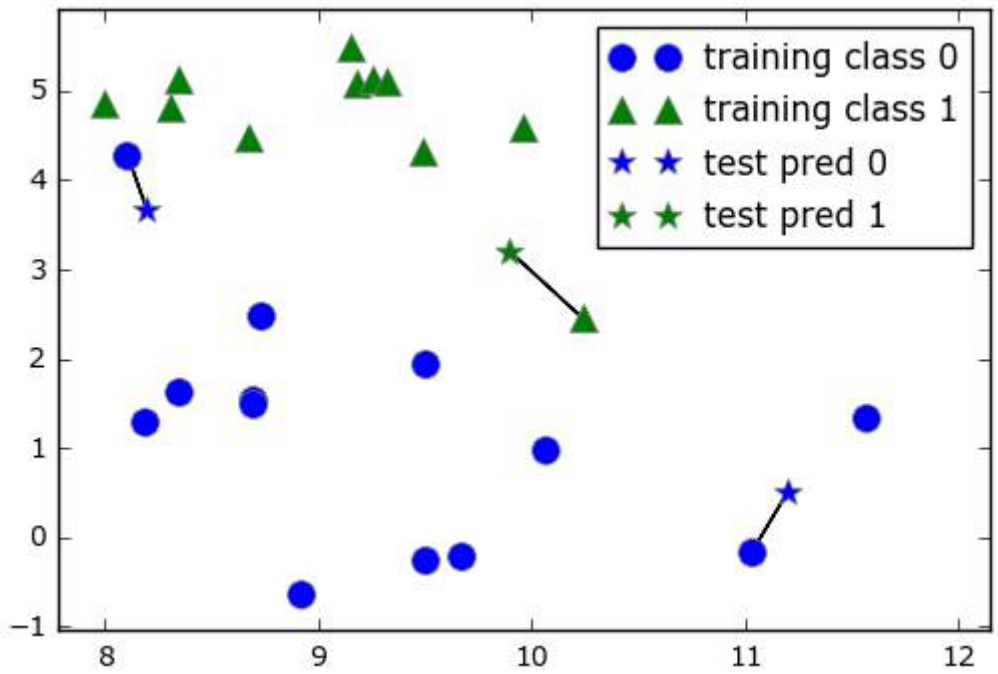

Рис. 2.4 Прогнозы, полученные для набора данных forge с помощью модели одного ближайшего соседа

Здесь мы добавили три новые точки данных, показанные в виде звездочек. Для каждой мы отметили ближайшую точку обучающего набора. Прогноз, который дает алгоритм одного ближайшего соседа – метка этой точки (показана цветом маркера).

Вместо того, чтобы учитывать лишь одного ближайшего соседа, мы можем рассмотреть произвольное количество (k) соседей. Отсюда и происходит название алгоритма k ближайших соседей. Когда мы рассматриваем более одного соседа, для присвоения метки используется *голосование* (*voting*). Это означает, что для каждой точки тестового набора мы подсчитываем количество соседей, относящихся к классу 0, и количество соседей, относящихся к классу 1. Затем мы присваиваем точке тестового набора наиболее часто встречающийся класс: другими словами, мы выбираем класс, набравший большинство среди k ближайших соседей. В примере, приведенном ниже (рис. 2.5), используются три ближайших соседа:

```
In[11]:
mglearn.plots.plot_knn_classification(n_neighbors=3)
```

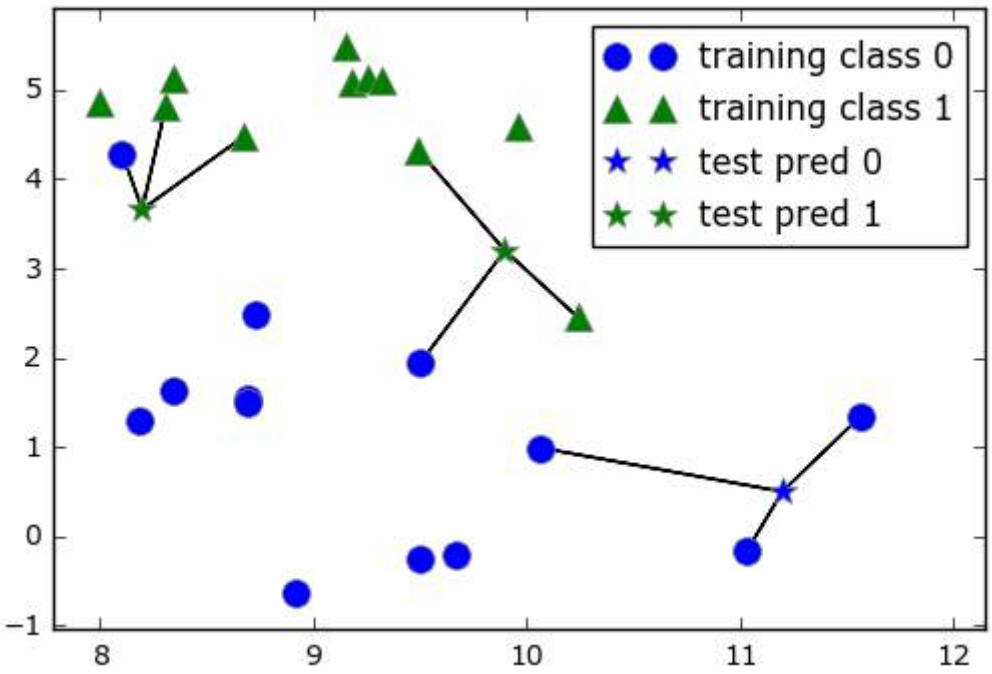

Рис. 2.5 Прогнозы, полученные для набора данных forge с помощью модели трех ближайших соседей

И снова прогнозы переданы цветом маркера. Видно, что прогноз для новой точки данных в верхнем левом углу отличается от прогноза, полученного при использовании одного ближайшего соседа.

Хотя данный рисунок иллюстрирует задачу бинарной классификации, этот метод можно применить к наборам данных с любым количеством классов. В случае мультиклассовой классификации мы подсчитываем

количество соседей, принадлежащих к каждому классу, и снова прогнозируем наиболее часто встречающийся класс.

Теперь давайте посмотрим, как можно применить алгоритм *k* ближайших соседей, используя `scikit-learn`. Во-первых, мы разделим наши данные на обучающий и тестовый наборы, чтобы оценить обобщающую способность модели, рассмотренную в главе 1:

```
In[12]:
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Далее выполняем импорт и создаем объект-экземпляр класса, задавая параметры, например, количество соседей, которое будем использовать для классификации. В данном случае мы устанавливаем его равным 3:

```
In[13]:
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

Затем подгоняем классификатор, используя обучающий набор. Для `KNeighborsClassifier` это означает запоминание набора данных, таким образом, мы можем вычислить соседей в ходе прогнозирования:

```
In[14]:
clf.fit(X_train, y_train)
```

Чтобы получить прогнозы для тестовых данных, мы вызываем метод `predict`. Для каждой точки тестового набора он вычисляет ее ближайших соседей в обучающем наборе и находит среди них наиболее часто встречающийся класс:

```
In[15]:
print("Прогнозы на тестовом наборе: {}".format(clf.predict(X_test)))

Out[15]:
Прогнозы на тестовом наборе: [1 0 1 0 1 0 0]
```

Для оценки обобщающей способности модели мы вызываем метод `score` с тестовыми данными и тестовыми метками:

```
In[16]:
print("Правильность на тестовом наборе: {:.2f}".format(clf.score(X_test, y_test)))

Out[16]:
Правильность на тестовом наборе: 0.86
```

Мы видим, что наша модель имеет правильность 86%, то есть модель правильно предсказала класс для 86% примеров тестового набора.

Кроме того, для двумерных массивов данных мы можем показать прогнозы для всех возможных точек тестового набора, разместив в плоскости xy. Мы зададим цвет плоскости в соответствии с тем классом, который будет присвоен точке в этой области. Это позволит нам сформировать *границу принятия решений* (*decision boundary*), которая разбивает плоскость на две области: область, где алгоритм присваивает класс 0, и область, где алгоритм присваивает класс 1.

Программный код, приведенный ниже, визуализирует границы принятия решений для одного, трех и девяти соседей (показаны на рис. 2.6):

```
In[17]:
fig, axes = plt.subplots(1, 3, figsize=(10, 3))

for n_neighbors, ax in zip([1, 3, 9], axes):
# создаем объект-классификатор и подгоняем в одной строке
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("количество соседей:{}".format(n_neighbors))
    ax.set_xlabel("признак 0")
    ax.set_ylabel("признак 1")
axes[0].legend(loc=3)
```

Рис. 2.6 Границы принятия решений, созданные моделью ближайших соседей для различных значений n_neighbors

На рисунке слева можно увидеть, что использование модели одного ближайшего соседа дает границу принятия решений, которая очень хорошо согласуется с обучающими данными. Увеличение числа соседей приводит к сглаживанию границы принятия решений. Более гладкая граница соответствует более простой модели. Другими словами, использование нескольких соседей соответствует высокой сложности модели (как показано в правой части рис. 2.1), а использование большого количества соседей соответствует низкой сложности модели (как показано в левой части рис. 2.1). Если взять крайний случай, когда количество соседей будет равно количеству точек данных обучающего

набора, каждая точка тестового набора будет иметь одних и тех же соседей (соседями будет все точки обучающего набора) и все прогнозы будут одинаковыми: будет выбран класс, который является наиболее часто встречающимся в обучающем наборе.

Давайте выясним, существует ли взаимосвязь между сложностью модели и обобщающей способностью, о которой мы говорили ранее. Мы сделаем это с помощью реального набора данных Breast Cancer. Начнем с того, что разобьем данные на обучающий и тестовый наборы. Затем мы оценим качество работы модели на обучающем и тестовом наборах с использованием разного количества соседей. Результаты показаны на рис. 2.7:

```
In[18]:
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, stratify=cancer.target, random_state=66)

training_accuracy = []
test_accuracy = []
# пробуем n_neighbors от 1 до 10
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    # строим модель
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # записываем правильность на обучающем наборе
    training_accuracy.append(clf.score(X_train, y_train))
    # записываем правильность на тестовом наборе
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(neighbors_settings, training_accuracy, label="правильность на обучающем наборе")
plt.plot(neighbors_settings, test_accuracy, label="правильность на тестовом наборе")
plt.ylabel("Правильность")
plt.xlabel("количество соседей")
plt.legend()
```

На этом графике по оси y отложена правильность на обучающем наборе и правильность на тестовом наборе, а по оси x – количество соседей. В реальности подобные графики редко бывают гладкими, мы по-прежнему можем увидеть некоторые признаки переобучения и недообучения (обратите внимание, что поскольку использование небольшого количества соседей соответствует более сложной модели, график представляет собой изображение рис. 2.1, зеркально отраженное по горизонтали). При использовании модели одного ближайшего соседа правильность на обучающем наборе идеальна. Однако при использовании большего количества соседей модель становится все проще и правильность на обучающем наборе падает. Правильность на тестовом наборе в случае использования одного соседа ниже, чем при использовании нескольких соседей. Это указывает на то, что

использование одного ближайшего соседа приводит к построению слишком сложной модели. С другой стороны, когда используются 10 соседей, модель становится слишком простой и она работает еще хуже. Оптимальное качество работы модели наблюдается где-то посередине, когда используются шесть соседей. Однако посмотрим на шкалу y. Худшая по качеству модель дает правильность на тестовом наборе около 88%, что по-прежнему может быть приемлемым результатом.

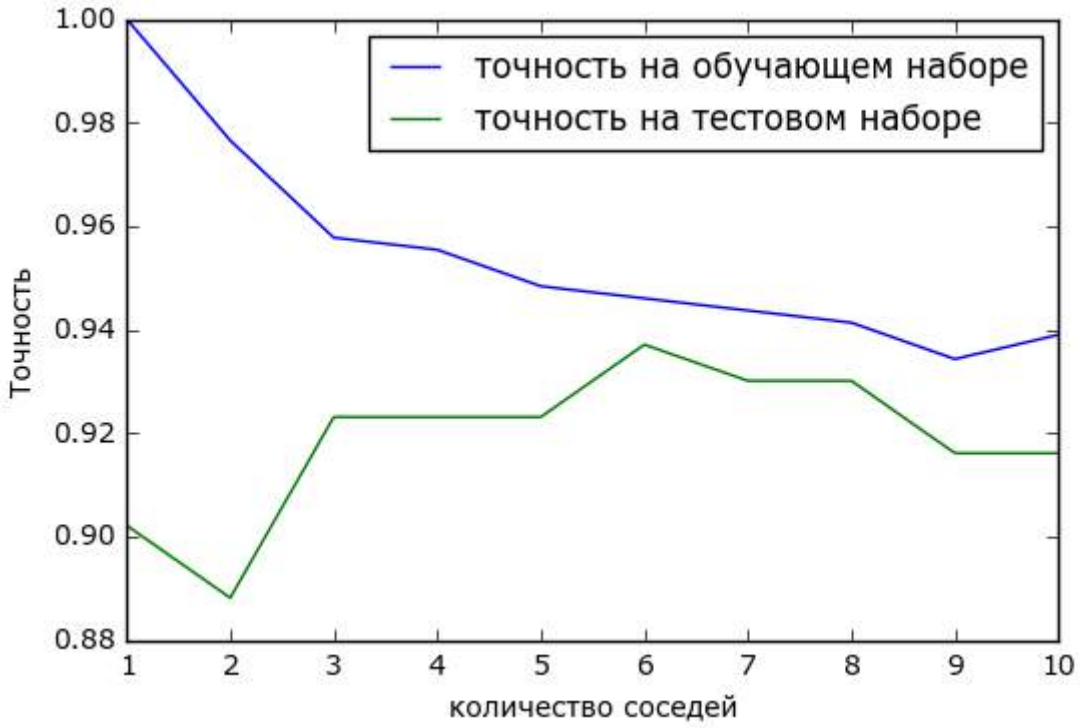

Рис. 2.7 Сравнение правильности на обучающем и тестовом наборах как функции от количества соседей

Регрессия k ближайших соседей

Существует также регрессионный вариант алгоритма *k* ближайших соседей. Опять же, давайте начнем с рассмотрения одного ближайшего соседа, на этот раз воспользуемся набором данных `wave`. Мы добавили три точки тестового набора в виде зеленых звездочек по оси x. Прогноз с использованием одного соседа – это целевое значение ближайшего соседа. На рис. 2.8 прогнозы показаны в виде синих звездочек:

```
In[19]:
mglearn.plots.plot_knn_regression(n_neighbors=1)
```

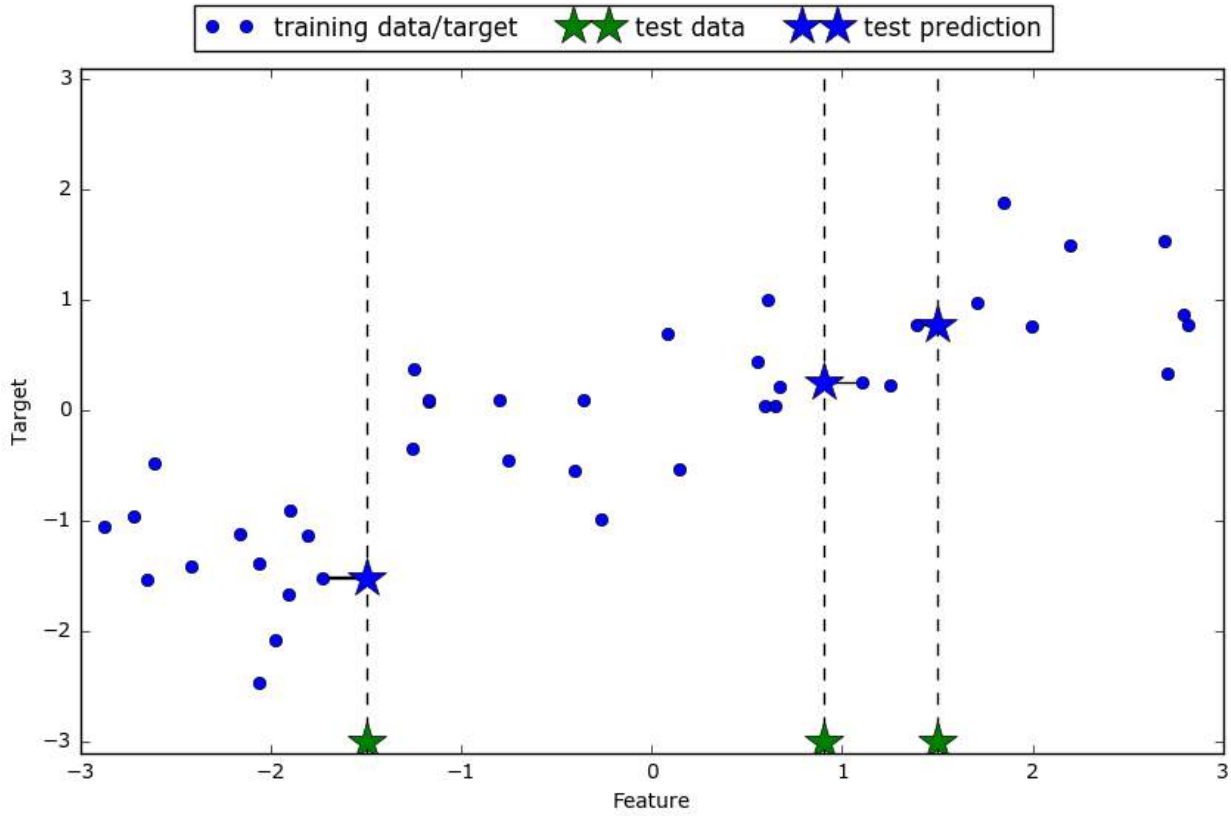

Рис. 2.8 Прогнозы, полученные с помощью регрессионной модели одного ближайшего соседа для набора данных wave

И снова для регрессии мы можем использовать большее количество ближайших соседей. При использовании нескольких ближайших соседей прогнозом становится среднее значение соответствующих соседей (рис. 2.9):

In[20]:
```
mglearn.plots.plot_knn_regression(n_neighbors=3)
```

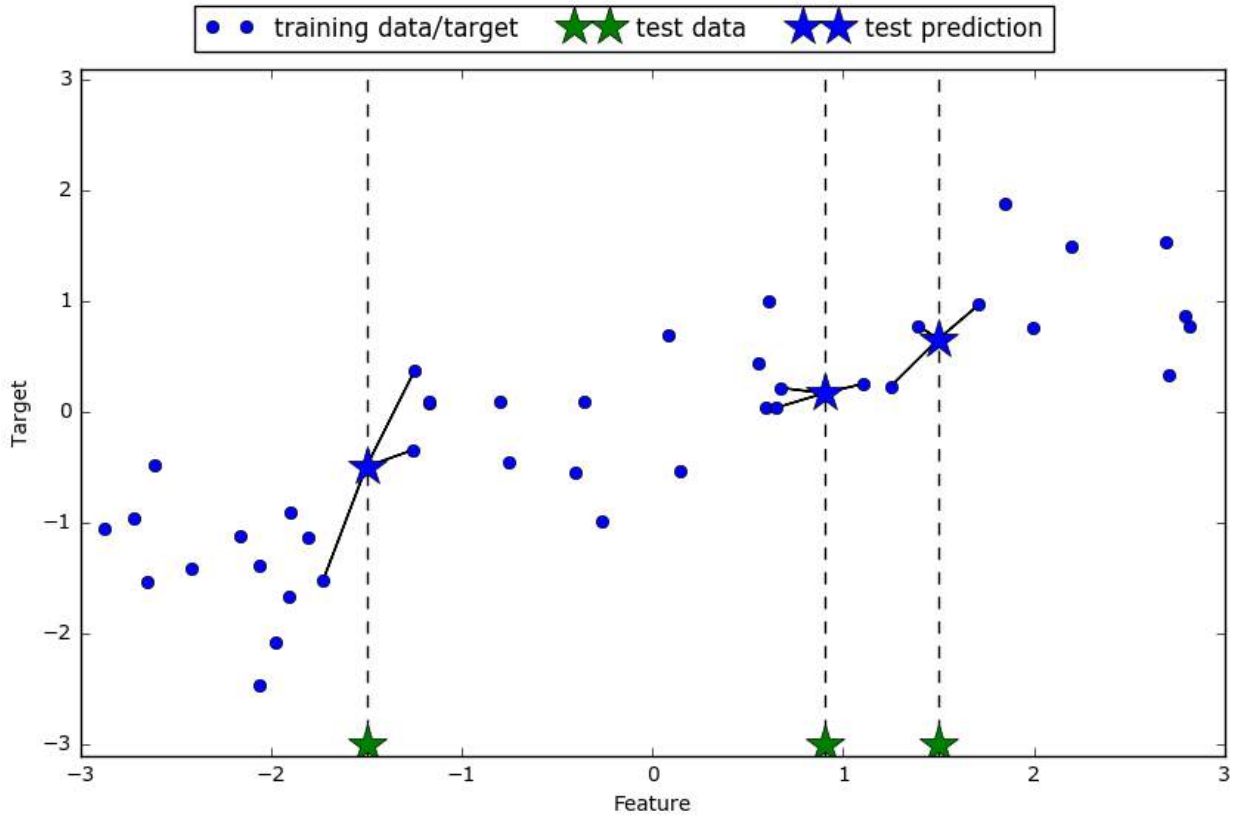

Рис. 2.8 Прогнозы, полученные с помощью регрессионной модели трех ближайших соседей для набора данных wave

Алгоритм регрессии k ближайших соседей реализован в классе `KNeighborsRegressor`. Он используется точно так же, как `KNeighborsClassifier`:

```
In[21]:
from sklearn.neighbors import KNeighborsRegressor

X, y = mglearn.datasets.make_wave(n_samples=40)

# разбиваем набор данных wave на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# создаем экземпляр модели и устанавливаем количество соседей равным 3
reg = KNeighborsRegressor(n_neighbors=3)
# подгоняем модель с использованием обучающих данных и обучающих ответов
reg.fit(X_train, y_train)
```

А теперь получим прогнозы для тестового набора.

```
In[22]:
print("Прогнозы для тестового набора:\n{}".format(reg.predict(X_test)))

Out[22]:
Прогнозы для тестового набора:
[-0.054 0.357 1.137 -1.894 -1.139 -1.631 0.357 0.912 -0.447 -1.139]
```

Кроме того, мы можем оценить качество модели с помощью метода `score`, который для регрессионных моделей возвращает значение R^2. R^2, также известный как коэффициент детерминации, является показателем

качества регрессионной модели и принимает значения от 0 до 1. Значение 1 соответствует идеальной прогнозирующей способности, а значение 0 соответствует константе модели, которая лишь предсказывает среднее значение ответов в обучающем наборе, `y_train`:

```
In[23]:
print("R^2 на тестовом наборе: {:.2f}".format(reg.score(X_test, y_test)))

Out[23]:
R^2 на тестовом наборе: 0.83
```

В данном случае значение R^2 составляет 0.83, что указывает на относительно хорошее качество подгонки модели.

Анализ модели KNeighborsRegressor

Применительно к нашему одномерному массиву данных мы можем увидеть прогнозы для всех возможных значений признаков (рис. 2.10). Для этого мы создаем тестовый набор данных и визуализируем полученные линии прогнозов:

```
In[24]:
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
# создаем 1000 точек данных, равномерно распределенных между -3 и 3
line = np.linspace(-3, 3, 1000).reshape(-1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
    # получаем прогнозы, используя 1, 3, и 9 соседей
    reg = KNeighborsRegressor(n_neighbors=n_neighbors)
    reg.fit(X_train, y_train)
    ax.plot(line, reg.predict(line))
    ax.plot(X_train, y_train, '^', c=mglearn.cm2(0), markersize=8)
    ax.plot(X_test, y_test, 'v', c=mglearn.cm2(1), markersize=8)

    ax.set_title(
        "{} neighbor(s)\n train score: {:.2f} test score: {:.2f}".format(
            n_neighbors, reg.score(X_train, y_train),
            reg.score(X_test, y_test)))
    ax.set_xlabel("Признак")
    ax.set_ylabel("Целевая переменная")
axes[0].legend(["Прогнозы модели", "Обучающие данные/ответы",
                "Тестовые данные/ответы"], loc="best")
```

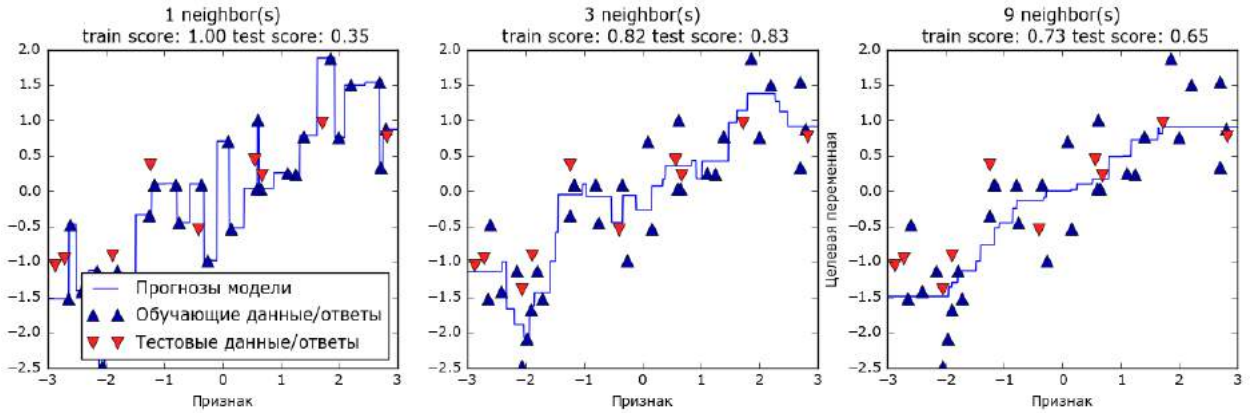

Рис. 2.10 Сравнение прогнозов, полученных с помощью регрессии ближайших соседей для различных значений n_neighbors

Как видно на графике, при использовании лишь одного соседа каждая точка обучающего набора имеет очевидное влияние на прогнозы, и предсказанные значения проходят через все точки данных. Это приводит к очень неустойчивым прогнозам. Увеличение числа соседей приводит к получению более сглаженных прогнозов, но при этом снижается правильность подгонки к обучающим данным.

Преимущества, недостатки и параметры

В принципе, в классификаторе `KNeighbors` есть два важных параметра: количество соседей и мера расстояния между точками данных. На практике использование небольшого числа соседей (например, 3-5) часто работает хорошо, но вы, конечно, можете самостоятельно настроить этот параметр. Вопрос, связанный с выбором правильной меры расстояния, выходит за рамки этой книги. По умолчанию используется евклидово расстояние, которое хорошо работает во многих ситуациях.

Одним из преимуществ метода ближайших соседей является то, что эту модель очень легко интерпретировать и, как правило, этот метод дает приемлемое качество без необходимости использования большого количества настроек. Он является хорошим базовым алгоритмом, который нужно попробовать в первую очередь, прежде чем рассматривать более сложные методы. Как правило, построение модели ближайших соседей происходит очень быстро, но, когда ваш обучающий набор очень большой (с точки зрения количества характеристик или количества наблюдений) получение прогнозов может занять некоторое время. При использовании алгоритма ближайших соседей важно выполнить предварительную обработку данных (смотрите главу 3). Данный метод не так хорошо работает, когда речь идет о наборах данных с большим количеством признаков (сотни и более), и особенно плохо работает в ситуации, когда подавляющее число признаков в большей части наблюдений имеют нулевые значения (так называемые *разреженные наборы данных* или *sparse datasets*).

Таким образом, несмотря на то что алгоритм ближайших соседей легко интерпретировать, на практике он не часто используется из-за скорости вычислений и его неспособности обрабатывать большое количество признаков. Метод, который мы обсудим ниже, лишен этих недостатков.

Линейные модели

Линейные модели представляют собой класс моделей, которые широко используются на практике и были предметом детального изучения в течение последних нескольких десятилетий, а их история насчитывает

более ста лет. Линейные модели дают прогноз, используя *линейную функцию* (*linear function*) входных признаков, о которой мы расскажем ниже.

Линейные модели для регрессии

Для регрессии общая прогнозная формула линейной модели выглядит следующим образом:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b$$

Здесь $x[0]$ по $x[p]$ обозначают признаки (в данном примере число характеристик равно $p+1$) для отдельной точки данных, w и b – параметры модели, оцениваемые в ходе обучения, и $\hat{y}$ – прогноз, выдаваемый моделью. Для набора данных с одним признаком эта формула имеет вид:

$$\hat{y} = w[0] * x[0] + b$$

Возможно из школьного курса математики вы вспомните, что эта формула – уравнение прямой. Здесь $x[0]$ является наклоном, а b – сдвигом по оси y. [9] Когда используется несколько признаков, регрессионное уравнение содержит параметры наклона для каждого признака. Как вариант, прогнозируемый ответ можно представить в виде взвешенной суммы входных признаков, где веса (которые могут быть отрицательными) задаются элементами w.

Попробуем вычислить параметры $w[0]$ и b для нашего одномерного набора данных `wave` с помощью следующей строки программного кода (см. рис. 2.11):

```
In[25]:
mglearn.plots.plot_linear_regression_wave()

Out[25]:
w[0]: 0.393906 b: -0.031804
```

[9] Более точно наклон представляет собой тангенс угла наклона линии регрессии и называется регрессионным коэффициентом, а сдвиг определяет точку пересечения линии регрессии с осью ординат и называется свободным членом или константой. – *Прим. пер.*

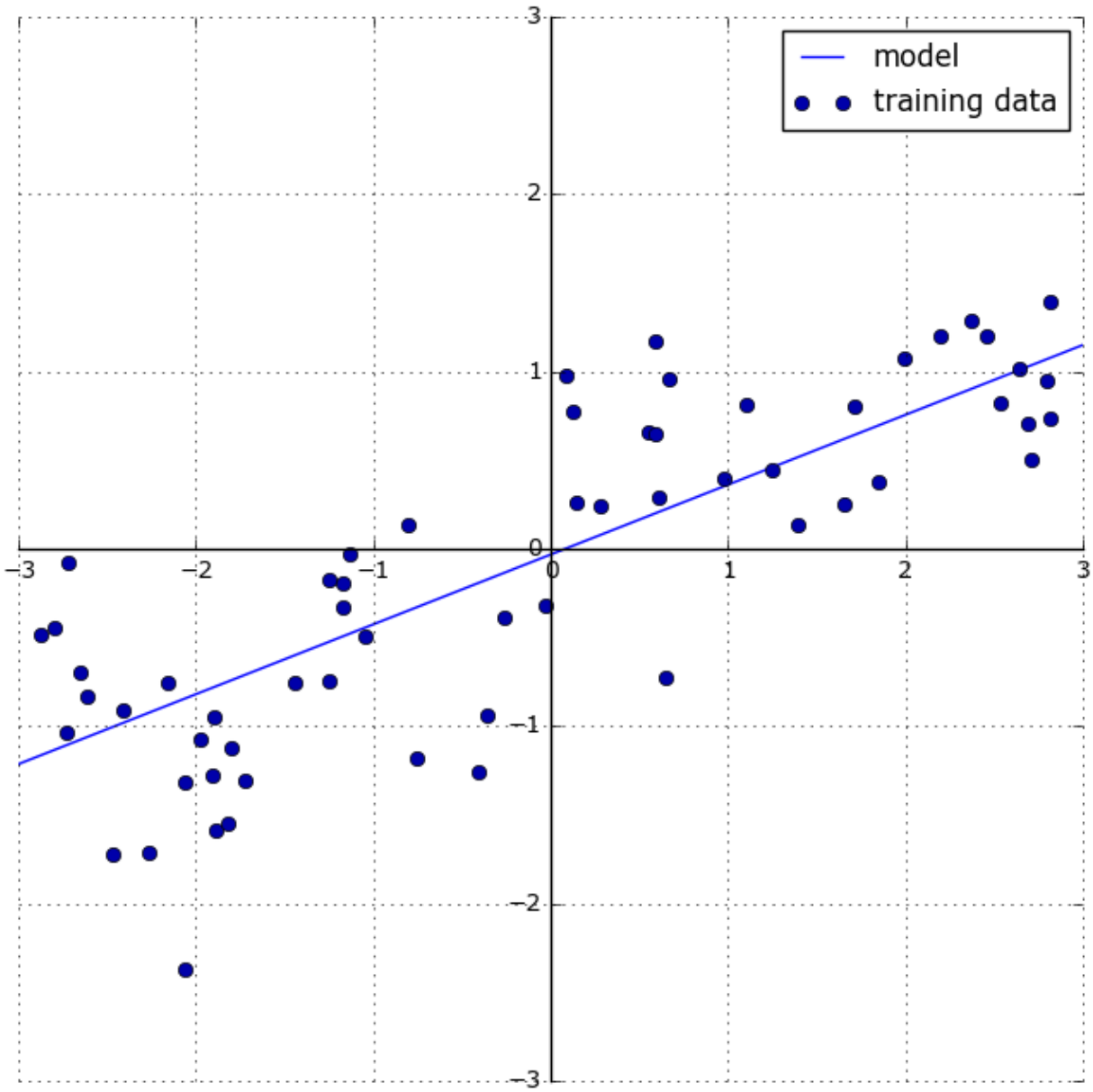

Рис. 2.11 Прогнозы линейной модели для набора данных wave

Мы добавили координатный крест на график, чтобы прямую было проще интерпретировать. Взглянув на значение `w[0]`, мы видим, что наклон должен быть около 0.4 и это визуально подтверждается на графике. Константа (место пересечения линии прогнозов с осью ординат) чуть меньше нуля, что также подтверждается графиком.

Линейные модели для регрессии можно охарактеризовать как регрессионные модели, в которых прогнозом является прямая линия для одного признака, плоскость, когда используем два признака, или гиперплоскость для большего количества измерений (то есть, когда используем много признаков).

Если прогнозы, полученные с помощью прямой линии, сравнить с прогнозами `KNeighborsRegressor` (рис. 2.10), использование линии регрессии для получения прогнозов кажется очень строгим. Похоже, что

все мелкие детали данных не учитываются. В некотором смысле это верно. Мы видвигаем сильное (и в некоторой степени нереальное) предположение, что наша целевая переменная *y* является линейной комбинацией признаков. Однако анализ одномерных данных дает несколько искаженную картину. Для наборов данных с большим количеством признаков линейные модели могут быть очень полезны. В частности, если у вас количество признаков превышает количество точек данных для обучения, любую целевую переменную *y* можно прекрасно смоделировать (на обучающей выборке) в виде линейной функции.[10]

Существует различные виды линейных моделей для регрессии. Различие между этими моделями заключается в способе оценивания параметров модели *w* и *b* по обучающим данным и контроле сложности модели. Теперь мы рассмотрим наиболее популярные линейные модели для регрессии.

Линейная регрессия (обычный метод наименьших квадратов)

Линейная регрессия или *обычный метод наименьших квадратов* (*ordinary least squares*, *OLS*) – это самый простой и наиболее традиционный метод регрессии. Линейная регрессия находит параметры *w* и *b*, которые минимизируют *среднеквадратическую ошибку* (*mean squared error*) между спрогнозированными и фактическими ответами *y* в обучающем наборе. Среднеквадратичная ошибка равна сумме квадратов разностей между спрогнозированными и фактическими значениями. Линейная регрессия проста, что является преимуществом, но в то же время у нее нет инструментов, позволяющих контролировать сложность модели.

Ниже приводится программный код, который строит модель, приведенную на рис. 2.11:

```
In[26]:
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

lr = LinearRegression().fit(X_train, y_train)
```

Параметры «наклона» (*w*), также называемые весами или *коэффициентами* (*coefficients*), хранятся в атрибуте `coef_`, тогда как *сдвиг* (*offset*) или *константа* (*intercept*), обозначаемая как *b*, хранится в атрибуте `intercept_`:

```
In[27]:
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
```

[10] Это легко увидеть, если вы немного знакомы с линейной алгеброй.

```
Out[27]:
lr.coef_: [ 0.394]
lr.intercept_: -0.031804343026759746
```

Вы можете заметить странный символ подчеркивания в конце названий атрибутов `coef_` и `intercept_`. Библиотека `scikit-learn` всегда хранит все, что является производным от обучающих данных, в атрибутах, которые заканчиваются символом подчеркивания. Это делается для того, чтобы не спутать их с пользовательскими параметрами.

Атрибут `intercept_` - это всегда отдельное число с плавающей точкой, тогда как атрибут `coef_` - это массив NumPy, в котором каждому элементу соответствует входной признак. Поскольку в наборе данных `wave` используется только один входной признак, `lr.coef_` содержит только один элемент.

Давайте посмотрим правильность модели на обучающем и тестовом наборах:

```
In[28]:
print("Правильность на обучающем наборе: {:.2f}".format(lr.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(lr.score(X_test, y_test)))

Out[28]:
Правильность на обучающем наборе: 0.67
Правильность на тестовом наборе: 0.66
```

Значение R^2 в районе 0.66 указывает на не очень хорошее качество модели, однако можно увидеть, что результаты на обучающем и тестовом наборах очень схожи между собой. Возможно, это указывает на недообучение, а не переобучение. Для этого одномерного массива данных опасность переобучения невелика, поскольку модель очень проста (или строга). Однако для высокоразмерных наборов данных (наборов данных с большим количеством признаков) линейные модели становятся более сложными и существует более высокая вероятность переобучения. Давайте посмотрим, как `LinearRegression` сработает на более сложном наборе данных, например, на наборе Boston Housing. Вспомним, что этот набор данных имеет 506 примеров (наблюдений) и 105 производных признаков. Во-первых, мы загрузим набор данных и разобьем его на обучающий и тестовый наборы. Затем построим модель линейной регрессии:

```
In[29]:
X, y = mglearn.datasets.load_extended_boston()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

При сравнении правильности на обучающем и тестовом наборах выясняется, что мы очень точно предсказываем на обучающем наборе, однако R^2 на тестовом наборе имеет довольно низкое значение:

```
In[30]:
print("Правильность на обучающем наборе: {:.2f}".format(lr.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(lr.score(X_test, y_test)))

Out[30]:
Правильность на обучающем наборе: 0.95
Правильность на тестовом наборе: 0.61
```

Это несоответствие между правильностью на обучающем наборе и правильностью на тестовом наборе является явным признаком переобучения и поэтому мы должны попытаться найти модель, которая позволит нам контролировать сложность. Одна из наиболее часто используемых альтернатив стандартной линейной регрессии – гребневая регрессия, которую мы рассмотрим ниже.

Гребневая регрессия

Гребневая регрессия[11] также является линейной моделью регрессии, поэтому ее формула аналогична той, что используется в обычном методе наименьших квадратов. В гребневой регрессии коэффициенты (w) выбираются не только с точки зрения того, насколько хорошо они позволяют предсказывать на обучающих данных, они еще подгоняются в соответствии с дополнительным ограничением. Нам нужно, чтобы величина коэффициентов была как можно меньше. Другими словами, все элементы w должны быть близки к нулю. Это означает, что каждый признак должен иметь как можно меньшее влияние на результат (то есть каждый признак должен иметь небольшой регрессионный коэффициент) и в то же время он должен по-прежнему обладать хорошей прогнозной силой. Это ограничение является примером *регуляризации* (*regularization*). Регуляризация означает явное ограничение модели для предотвращения переобучения. Регуляризация, использующаяся в гребневой регрессии, известна как L2 регуляризация.[12]

Гребневая регрессии реализована в классе `linear_model.Ridge`. Давайте посмотрим, насколько хорошо она работает на расширенном наборе данных Boston Housing:

```
In[31]:
from sklearn.linear_model import Ridge

ridge = Ridge().fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(ridge.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(ridge.score(X_test, y_test)))
```

[11] В качестве синонима использует термин «ридж-регрессия». – *Прим. пер.*

[12] С математической точки зрения `Ridge` штрафует L2 норму коэффициентов или евклидову длину w.

```
Out[31]:
Правильность на обучающем наборе: 0.89
Правильность на тестовом наборе: 0.75
```

Здесь мы видим, что на обучающем наборе модель `Ridge` дает *меньшую* правильность, чем модель `LinearRegression`, тогда как правильность на тестовом наборе в случае применения гребневой регрессии *выше*. Это согласуется с нашими ожиданиями. При использовании линейной регрессии мы получили переобучение. `Ridge` – модель с более строгим ограничением, поэтому меньше вероятность переобучения. Менее сложная модель означает меньшую правильность на обучающем наборе, но лучшую обобщающую способность. Поскольку нас интересует только обобщающая способность, мы должны выбрать модель `Ridge` вместо модели `LinearRegression`.

Модель `Ridge` позволяет найти компромисс между простотой модели (получением коэффициентов, близких к нулю) и качеством ее работы на обучающем наборе. Компромисс между простотой модели и качеством работы на обучающем наборе может быть задан пользователем при помощи параметра `alpha`. В предыдущем примере мы использовали значение параметра по умолчанию `alpha=1.0`. Впрочем, нет никаких причин считать, что это даст нам оптимальный компромиссный вариант. Оптимальное значение `alpha` зависит от конкретного используемого набора данных. Увеличение `alpha` заставляет коэффициенты сжиматься до близких к нулю значений, что снижает качество работы модели на обучающем наборе, но может улучшить ее обобщающую способность. Например:

```
In[32]:
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(ridge10.score(X_test, y_test)))

Out[32]:
Правильность на обучающем наборе: 0.79
Правильность на тестовом наборе: 0.64
```

Уменьшая `alpha`, мы сжимаем коэффициенты в меньшей степени, что означает движение вправо на рис. 2.1. При очень малых значениях `alpha`, ограничение на коэффициенты практически не накладывается и мы в конечном итоге получаем модель, напоминающую линейную регрессию:

```
In[33]:
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(ridge01.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(ridge01.score(X_test, y_test)))

Out[33]:
Правильность на обучающем наборе: 0.93
Правильность на тестовом наборе: 0.77
```

Похоже, что здесь параметр `alpha=0.1` сработал хорошо. Мы могли бы попробовать уменьшить `alpha` еще больше, чтобы улучшить обобщающую способность. Сейчас обратите внимание на то, как параметр `alpha` соотносится со сложностью модели (рис. 2.1). В главе 5 мы рассмотрим методы правильного подбора параметров.

Кроме того, мы можем лучше понять, как параметр `alpha` меняет модель, использовав атрибут `coef_` с разными значениями `alpha`. Чем выше `alpha`, тем более жесткое ограничение накладывается на коэффициенты, поэтому следует ожидать меньшие значения элементов `coef_` для высокого значения `alpha`. Это подтверждается графиком на рис. 2.12:

In[34]:
```
plt.plot(ridge.coef_, 's', label="Гребневая регрессия alpha=1")
plt.plot(ridge10.coef_, '^', label="Гребневая регрессия alpha=10")
plt.plot(ridge01.coef_, 'v', label="Гребневая регрессия alpha=0.1")

plt.plot(lr.coef_, 'o', label="Линейная регрессия")
plt.xlabel("Индекс коэффициента")
plt.ylabel("Оценка коэффициента")
plt.hlines(0, 0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()
```

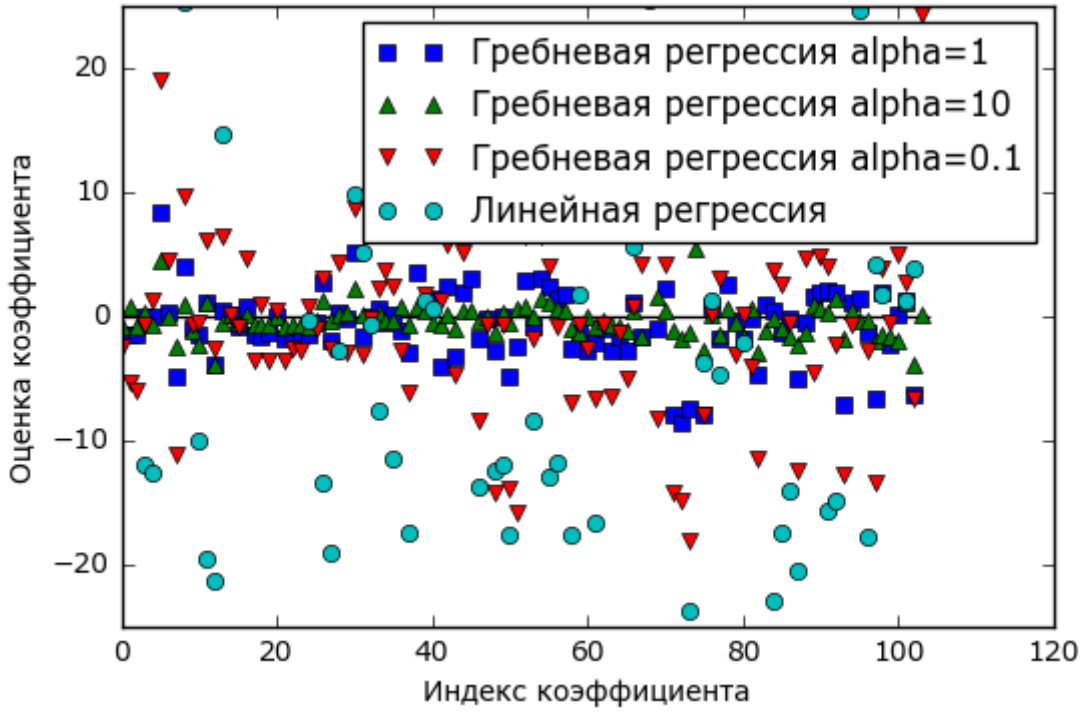

Рис. 2.12 Сравнение оценок коэффициентов гребневой регрессии с разными значениями alpha и линейной регрессии

Здесь ось x соответствует элементам атрибута `coef_`:`x=0` показывает коэффициент, связанный с первым признаком, `x=1` – коэффициент, связанный со вторым признаком и так далее, вплоть до `x=100`. Ось y

показывает числовые значения соответствующих коэффициентов. Ключевой информацией здесь является то, что для `alpha=10` коэффициенты главным образом расположены в диапазоне от -3 до 3. Коэффициенты для модели `Ridge` с `alpha=1` несколько больше. Точки, соответствующие `alpha=0.1`, имеют более высокие значения, а большинство точек, соответствующих линейной регрессии без регуляризации (что соответствует `alpha=0`), настолько велики, что находятся за пределами диаграммы.

Еще один способ понять влияние регуляризации заключается в том, чтобы зафиксировать значение `alpha` и при этом менять доступный объем обучающих данных. Мы сформировали выборки разного объема на основе набора данных Boston Housing и затем построили `LinearRegression` и `Ridge(alpha=1)` на полученных подмножествах, увеличивая объем. На рис. 2.13 приводятся графики, которые показывают качество работы модели в виде функции от объема набора данных, их еще называют *кривыми обучения* (*learning curves*):

```
In[35]:
mglearn.plots.plot_ridge_n_samples()
```

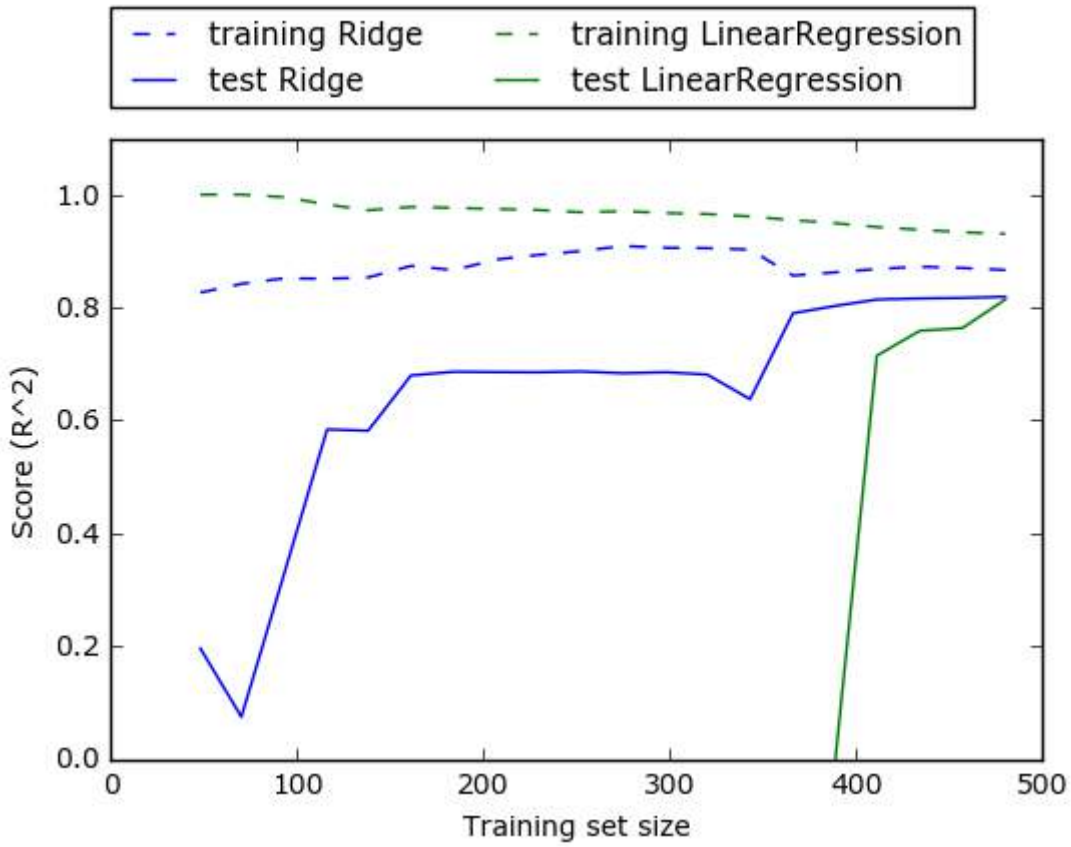

Рис. 2.13 Кривые обучения гребневой регрессии и линейной регрессии для набора данных Boston Housing

Как и следовало ожидать, независимо от объема данных правильность на обучающем наборе всегда выше правильности на тестовом наборе, как в случае использования гребневой регрессии, так и в случае использования линейной регрессии. Поскольку гребневая регрессия – регуляризированная модель, во всех случаях на обучающем наборе правильность гребневой регрессии ниже правильности линейной регрессии. Однако правильность на тестовом наборе у гребневой регрессии выше, особенно для небольших подмножеств данных. При объеме данных менее 400 наблюдений линейная регрессия не способна обучиться чему-либо. По мере возрастания объема данных, доступного для моделирования, обе модели становятся лучше и в итоге линейная регрессия догоняет гребневую регрессию. Урок здесь состоит в том, что при достаточном объеме обучающих данных регуляризация становится менее важной и при удовлетворительном объеме данных гребневая и линейная регрессии будут демонстрировать одинаковое качество работы (тот факт, что в данном случае это происходит при использовании полного набора данных, является просто случайностью). Еще одна интересная деталь рис. 2.13 – это снижение правильности линейной регрессии на обучающем наборе. С возрастанием объема данных модели становится все сложнее переобучиться или запомнить данные.

Лассо

Альтернативой `Ridge` как метода регуляризации линейной регрессии является `Lasso`. Как и гребневая регрессия, лассо также сжимает коэффициенты до близких к нулю значений, но несколько иным способом, называемым L1 регуляризацией.[13] Результат L1 регуляризации заключается в том, что при использовании лассо некоторые коэффициенты становятся равны *точно нулю*. Получается, что некоторые признаки полностью исключаются из модели. Это можно рассматривать как один из видов автоматического отбора признаков. Получение нулевых значений для некоторых коэффициентов часто упрощает интерпретацию модели и может выявить наиболее важные признаки вашей модели.

Давайте применим метод лассо к расширенному набору данных Boston Housing:

```
In[36]:
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(lasso.score(X_train, y_train)))
print("Правильность на контрольном наборе: {:.2f}".format(lasso.score(X_test, y_test)))
print("Количество использованных признаков: {}".format(np.sum(lasso.coef_ != 0)))
```

[13] Лассо штрафует L1 норму вектора коэффициентов или, другими словами, сумму абсолютных значений коэффициентов.

```
Out[36]:
Правильность на обучающем наборе: 0.29
Правильность на контрольном наборе: 0.21
Количество использованных признаков: 4
```

Как видно из сводки, `Lasso` дает низкую правильность как на обучающем, так и на тестовом наборе. Это указывает на недообучение и мы видим, что из 105 признаков используются только 4. Как и `Ridge`, `Lasso` также имеет параметр регуляризации `alpha`, который определяет степень сжатия коэффициентов до нулевых значений. В предыдущем примере мы использовали значение по умолчанию `alpha=1.0`. Чтобы снизить недообучение, давайте попробуем уменьшить `alpha`. При этом нам нужно увеличить значение `max_iter` (максимальное количество итераций):

```
In[37]:
# мы увеличиваем значение "max_iter",
# иначе модель выдаст предупреждение, что нужно увеличить max_iter.
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(lasso001.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(lasso001.score(X_test, y_test)))
print("Количество использованных признаков: {}".format(np.sum(lasso001.coef_ != 0)))

Out[37]:
Правильность на обучающем наборе: 0.90
Правильность на контрольном наборе: 0.77
Количество использованных признаков: 33
```

Более низкое значение `alpha` позволило нам получить более сложную модель, которая продемонстрировала более высокую правильность на обучающем и тестовом наборах. Лассо работает немного лучше, чем гребневая регрессия, и мы используем лишь 33 признака из 105. Это делает данную модель более легкой с точки зрения интерпретации.

Однако, если мы установим слишком низкое значение `alpha`, мы снова нивелируем эффект регуляризации и получим в конечном итоге переобучение, придя к результатам, аналогичным результатам линейной регрессии:

```
In[38]:
lasso00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(lasso00001.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(lasso00001.score(X_test, y_test)))
print("Количество использованных признаков: {}".format(np.sum(lasso00001.coef_ != 0)))

Out[38]:
Правильность на обучающем наборе: 0.95
Правильность на контрольном наборе: 0.64
Количество использованных признаков: 94
```

Опять же, мы можем построить графики для коэффициентов различных моделей, аналогичные рис. 2.12. Результат приведен на рис. 2.14:

```
In[39]:
plt.plot(lasso.coef_, 's', label="Лассо alpha=1")
plt.plot(lasso001.coef_, '^', label="Лассо alpha=0.01")
plt.plot(lasso00001.coef_, 'v', label="Лассо alpha=0.0001")

plt.plot(ridge01.coef_, 'o', label="Гребневая регрессия alpha=0.1")
plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-25, 25)
plt.xlabel("Индекс коэффициента")
plt.ylabel("Оценка коэффициента")
```

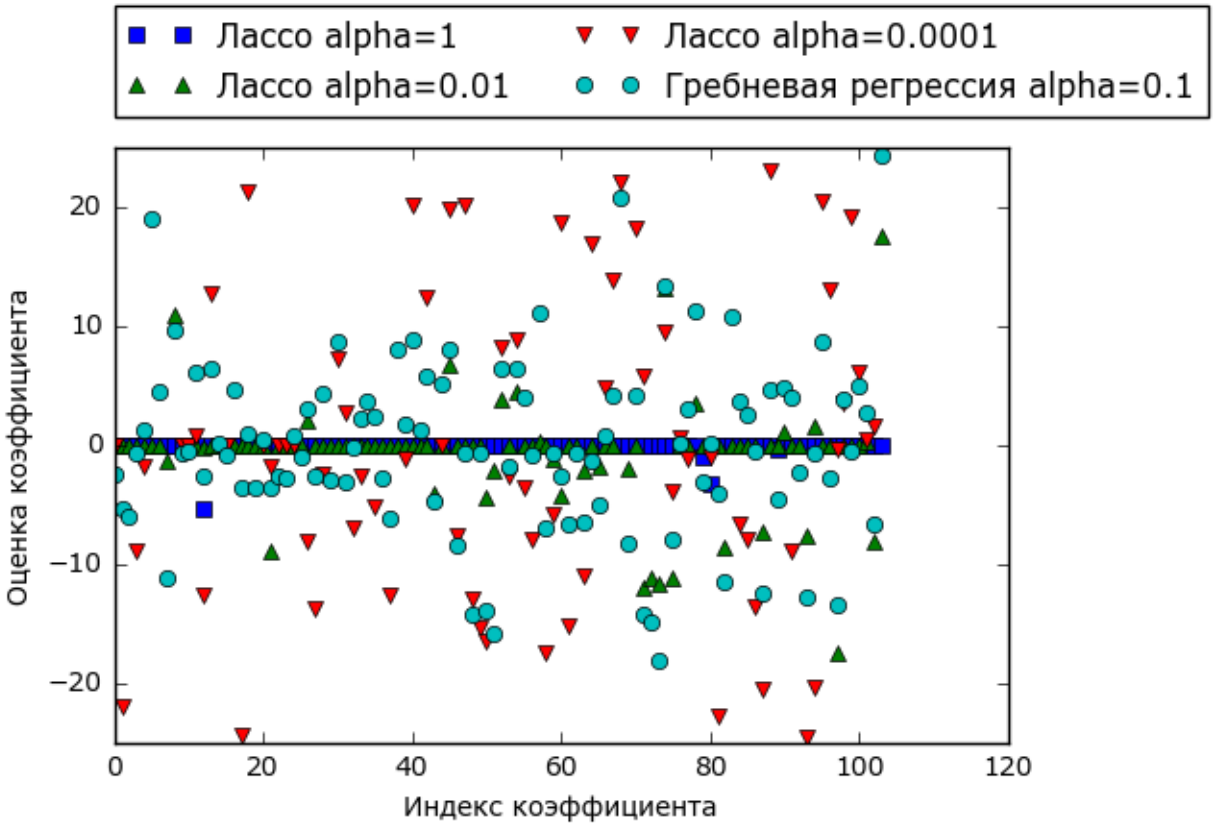

Рис. 2.14 Сравнение коэффициентов для лассо-регрессии с разными значениями alpha и гребневой регрессии

Для `alpha=1` мы видим, что не только большинство коэффициентов равны нулю (что мы уже знали), но и остальные коэффициенты также малы по величине. Уменьшив `alpha` до `0.01`, получаем решение, показанное в виде зеленых треугольников, большая часть коэффициентов для признаков становятся в точности равными нулю. При `alpha=0.0001` мы получаем практически нерегуляризированную модель, у которой большинство коэффициентов отличны от нуля и имеют большие значения. Для сравнения приводится наилучшее решение, полученное с помощью гребневой регрессии. Модель `Ridge` с `alpha=0.1` имеет такую же прогностическую способность, что и модель лассо с `alpha=0.01`, однако при использовании гребневой регрессии все коэффициенты отличны от нуля.

На практике, когда стоит выбор между гребневой регрессией и лассо, предпочтение, как правило, отдается гребневой регрессии. Однако, если у вас есть большое количество признаков и есть основания считать, что лишь некоторые из них важны, `Lasso` может быть оптимальным выбором. Аналогично, если вам нужна легко интерпретируемая модель, `Lasso` поможет получить такую модель, так как она выберет лишь подмножество входных признаков. В библиотеке `scikit-learn` также имеется класс `ElasticNet`, который сочетает в себе штрафы `Lasso` и `Ridge`. На практике эта комбинация работает лучше всего, впрочем, это достигается за счет двух корректируемых параметров: один для L1 регуляризации, а другой – для L2 регуляризации.

Линейные модели для классификации

Линейные модели также широко используются в задачах классификации. Давайте посмотрим сначала на бинарную классификацию. В этом случае прогноз получают с помощью следующей формулы:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b > 0$$

Формула очень похожа на формулу линейной регрессии, но теперь вместо того, чтобы просто возвратить взвешенную сумму признаков, мы задаем для прогнозируемого значения порог, равный нулю. Если функция меньше нуля, мы прогнозируем класс −1, если она больше нуля, мы прогнозируем класс +1. Это прогнозное правило является общим для всех линейных моделей классификации. Опять же, есть много различных способов найти коэффициенты (w) и константу (b).

Для линейных моделей регрессии выход $\hat{y}$ является линейной функцией признаков: линией, плоскостью или гиперплоскостью (для большого количества измерений). Для линейных моделей классификации *граница принятия решений* (*decision boundary*) является линейной функцией аргумента. Другими словами, (бинарный) линейный классификатор – это классификатор, который разделяет два класса с помощью линии, плоскости или гиперплоскости. В этом разделе мы приведем кокретные примеры.

Существует масса алгоритмов обучения линейных моделей. Два критерия задают различия между алгоритмами:

- Измеряемые метрики качества подгонки обучающих данных;
- Факт использования регуляризации и вид регуляризации, если она используется.

Различные алгоритмы по-разному определяют, что значит «хорошая подгонка обучающих данных». В силу технико-математических причин

невозможно скорректировать *w* и *b*, чтобы минимизировать количество неверно классифицированных случаев, выдаваемое алгоритмами, как можно было бы надеяться. С точки зрения поставленных нами целей и различных сфер применения различные варианты метрик качества подгонки (так называемые *функции потерь* или *loss functions*) не имеют большого значения.

Двумя наиболее распространенными алгоритмами линейной классификации являются *логистическая регрессия (logistic regression)*, реализованная в классе `linear_model.LogisticRegression`, и *линейный метод опорных векторов* (*linear support vector machines*) или линейный SVM, реализованный в классе `svm.LinearSVC` (SVC расшифровывается как *support vector classifier* – *классификатор опорных векторов*). Несмотря на свое название, логистическая регрессия является алгоритмом классификации, а не алгоритмом регрессии, и его не следует путать с линейной регрессией.

Мы можем применить модели `LogisticRegression` и `LinearSVC` к набору данных `forge` и визуализировать границу принятия решений, найденную линейными моделями (рис. 2.15):

```
In[40]:
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

X, y = mglearn.datasets.make_forge()

fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5,
                                    ax=ax, alpha=.7)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{}".format(clf.__class__.__name__))
    ax.set_xlabel("Признак 0")
    ax.set_ylabel("Признак 1")
axes[0].legend()
```

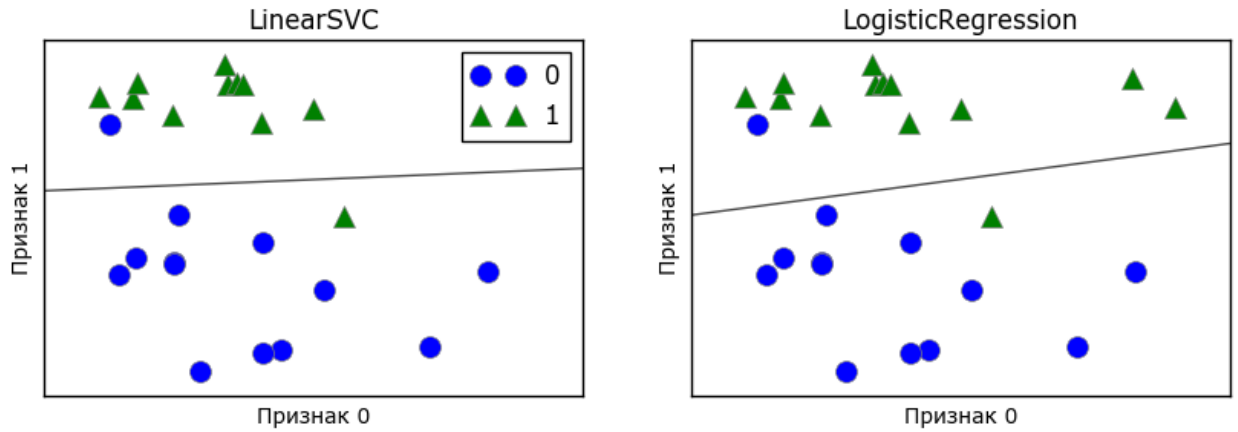

Рис. 2.15 Границы принятия решений линейного SVM и логистической регрессии для набора данных forge (использовались значения параметров по умолчанию)

На этом рисунке, как и раньше, первый признак набора данных `forge` отложен по оси x, а второй признак – по оси y. Здесь показаны границы принятия решений, найденные `LinearSVC` и `LogisticRegression` соответственно. Они представлены в виде прямых линий, отделяющих область значений, классифицированных как класс 1 (в верхней части графика) от области значений, классифицированных как класс 0 (в нижней части графика). Другими словами, любая новая точка данных, которая лежит выше черной линии будет отнесена соответствующей моделью к классу 1, тогда как любая точка, лежащая ниже черной линии, будет отнесена к классу 0.

Обе модели имеют схожие границы принятия решений. Обратите внимание, что обе модели неправильно классифицировали две точки. По умолчанию обе модели используют L2 регуляризацию, тот же самый метод, который используется в гребневой регрессии.

Для `LogisticRegression` и `LinearSVC` компромиссный параметр, который определяет степень регуляризации, называется `C`, и более высокие значения `C` соответствуют *меньшей* регуляризации. Другими словами, когда вы используете высокое значение параметра `C`, `LogisticRegression` и `LinearSVC` пытаются подогнать модель к обучающим данным как можно лучше, тогда как при низких значениях параметра `C` модели делают б*о*льший акцент на поиске вектора коэффициентов (w), близкого к нулю.

Существует еще одна интересная деталь, связанная с работой параметра `C`. Использование низких значений `C` приводит к тому, что алгоритмы пытаются подстроиться под «большинство» точек данных, тогда как использование более высоких значений `C` подчеркивает важность того, чтобы каждая отдельная точка данных была классифицирована правильно. Ниже приводится иллюстрация использования `LinearSVC` (рис. 2.16):

```
In[41]:
mglearn.plots.plot_linear_svc_regularization()
```

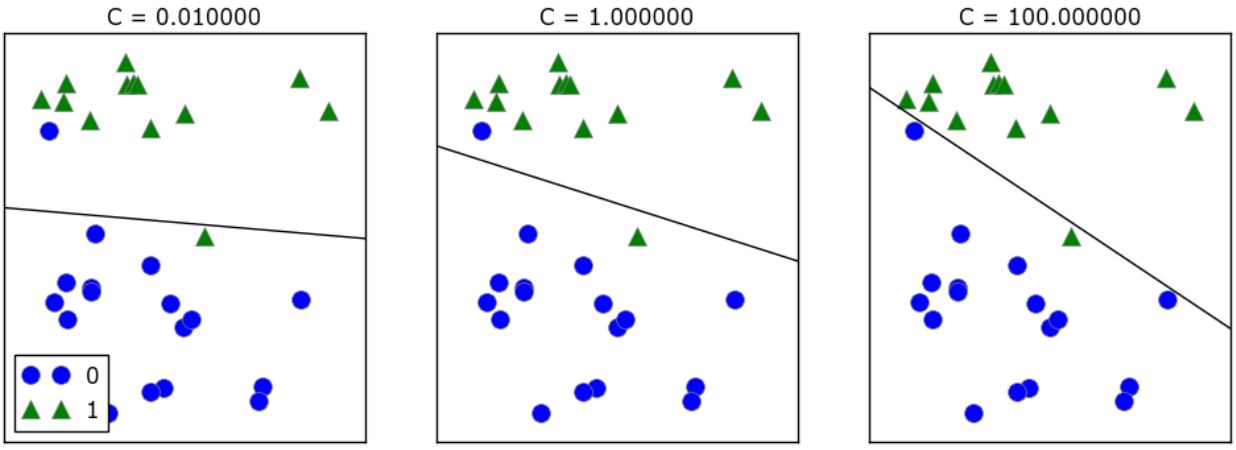

Рис. 2.16 Границы принятия решений линейного SVM с различными значениями C для набора данных forge

На графике слева показана модель с очень маленьким значением `C`, соответствующим большой степени регуляризации. Большая часть точек класса 0 находятся в нижней части графика, а большинство точек класса 1 находятся в верхней части. Сильно регуляризованная модель дает относительно горизонтальную линию, неправильно классифицируя две точки. На центральном графике значение `C` немного выше и модель в большей степени фокусируется на двух неправильно классифицированных примерах, наклоняя границу принятия решений. Наконец, на графике справа очень высокое значение `C` модели наклоняет границу принятия решений еще сильнее, теперь правильно классифицируя все точки класса 0. Одна из точек класса 1 по-прежнему неправильно классифицирована, поскольку невозможно правильно классифицировать все наблюдения этого набора данных с помощью прямой линии. Модель на графике справа старается изо всех сил правильно классифицировать все точки, но не может дать хорошего обобщения сразу для обоих классов. Другими словами, эта модель скорее всего переобучена.

Как и в случае с регрессией, линейные модели классификации могут показаться слишком строгими в условиях низкоразмерного пространства, предлагая границы принятия решений в виде прямых линий или плоскостей. Опять же, при наличии большого числа измерений линейные модели классификации приобретают высокую прогнозную силу и с увеличением числа признаков защита от переобучения становится все более важной.

Давайте более подробно разберем работу `LogisticRegression` на наборе данных Breast Cancer:

```
In[42]:
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, stratify=cancer.target, random_state=42)
logreg = LogisticRegression().fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.3f}".format(logreg.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(logreg.score(X_test, y_test)))

Out[42]:
Правильность на обучающем наборе: 0.953
Правильность на тестовом наборе: 0.958
```

Значение по умолчанию `C=1` обеспечивает неплохое качество модели, правильность на обучающем и тестовом наборах составляет 95%. Однако поскольку качество модели на обучающем и тестовом наборах примерно одинако, вполне вероятно, что мы недообучили модель. Давайте попробуем увеличить `C`, чтобы подогнать более гибкую модель:

```
In[43]:
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(logreg100.score(X_test, y_test)))

Out[43]:
Правильность на обучающем наборе: 0.972
Правильность на тестовом наборе: 0.965
```

Использование `C=100` привело к более высокой правильности на обучающей выборке, а также немного увеличилась правильность на тестовой выборке, что подтверждает наш довод о том, что более сложная модель должна сработать лучше.

Кроме того, мы можем выяснить, что произойдет, если мы воспользуемся более регуляризованной моделью (установив `C=0.01` вместо значения по умолчанию `C=1`):

```
In[44]:
logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.3f}".format(logreg001.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(logreg001.score(X_test, y_test)))

Out[44]:
Правильность на обучающем наборе: 0.934
Правильность на тестовом наборе: 0.930
```

Как и следовало ожидать, когда мы получили недообученную модель и переместились влево по шкале, показанной на рис. 2.1, правильность как на обучающем, так и на тестовом наборах снизилась по сравнению с правильностью, которую мы получили, использовав параметры по умолчанию.

И, наконец, давайте посмотрим на коэффициенты логистической регрессии, полученные с использованием трех различных значений параметра регуляризации `C` (рис. 2.17):

```
In[45]:
plt.plot(logreg.coef_.T, 'o', label="C=1")
plt.plot(logreg100.coef_.T, '^', label="C=100")
plt.plot(logreg001.coef_.T, 'v', label="C=0.001")
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.ylim(-5, 5)
plt.xlabel("Индекс коэффициента")
plt.ylabel("Оценка коэффициента")
plt.legend()
```

Поскольку `LogisticRegression` по умолчанию использует L2 регуляризацию, результат похож на результат, полученный при использовании модели `Ridge` (рис. 2.12). Бо́льшая степень регуляризации сильнее сжимает коэффициенты к нулю, хотя коэффициенты никогда не станут в точности равными нулю. Изучив график более внимательно, можно увидеть интересный эффект, произошедший с третьим коэффициентом, коэффициентом признака «mean perimeter». При `C=100` и `C=1` коэффициент отрицателен, тогда как при `C=0.001` коэффициент положителен, при этом его оценка больше, чем оценка коэффициента при `C=1`. Когда мы интерпретируем данную модель, коэффициент говорит нам, какой класс связан с этим признаком. Возможно, что высокое значение признака «texture error» связано с примером, классифицированным как «злокачественный». Однако изменение знака коэффициента для признака «mean perimeter» означает, что в зависимости от рассматриваемой модели высокое значение «mean perimeter» может указывать либо на доброкачественную, либо на злокачественную опухоль. Приведенный пример показывает, что интерпретировать коэффициенты линейных моделей всегда нужно с осторожностью и скептицизмом.

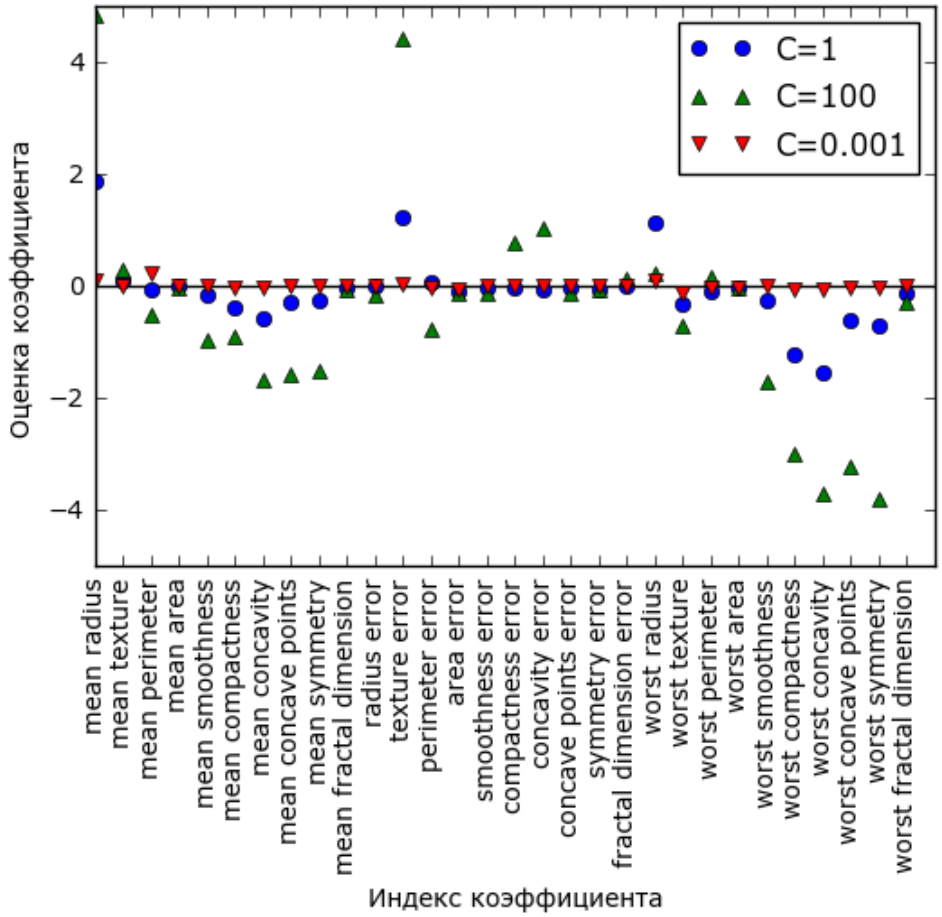

Рис. 2.17 Коэффициенты, полученные с помощью логистической регрессии с разными значениями C для набора данных Breast Cancer

Если мы хотим получить более интерпретабельную модель, нам может помочь L1 регуляризация, поскольку она ограничивает модель использованием лишь нескольких признаков. Ниже приводится график с коэффициентами и оценками правильности для L1 регуляризации (рис. 2.18):

```
In[46]:
for C, marker in zip([0.001, 1, 100], ['o', '^', 'v']):
    lr_l1 = LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)
    print("Правильность на обучении для логрегрессии l1 с C={:.3f}: {:.2f}".format(
          C, lr_l1.score(X_train, y_train)))
    print("Правильность на тесте для логрегрессии l1 с C={:.3f}: {:.2f}".format(
          C, lr_l1.score(X_test, y_test)))

plt.plot(lr_l1.coef_.T, marker, label="C={:.3f}".format(C))
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.xlabel("Индекс коэффициента")
plt.ylabel("Оценка коэффициента")

plt.ylim(-5, 5)
plt.legend(loc=3)

Out[46]:
Правильность на обучении для логрегрессии l1 с C=0.001: 0.91
Правильность на тесте для логрегрессии l1 с C=0.001: 0.92
Правильность на обучении для логрегрессии l1 с C=1.000: 0.96
Правильность на тесте для логрегрессии l1 с C=1.000: 0.96
```

```
Правильность на обучении для логрегрессии l1 с C=100.000: 0.99
Правильность на тесте для логрегрессии l1 с C=100.000: 0.98
```

Видно, что существует много параллелей между линейными моделями для бинарной классификации и линейными моделями для регрессии. Как и в регрессии, основное различие между моделями – в параметре `penalty`, который влияет на регуляризацию и определяет, будет ли модель использовать все доступные признаки или выберет лишь подмножество признаков.

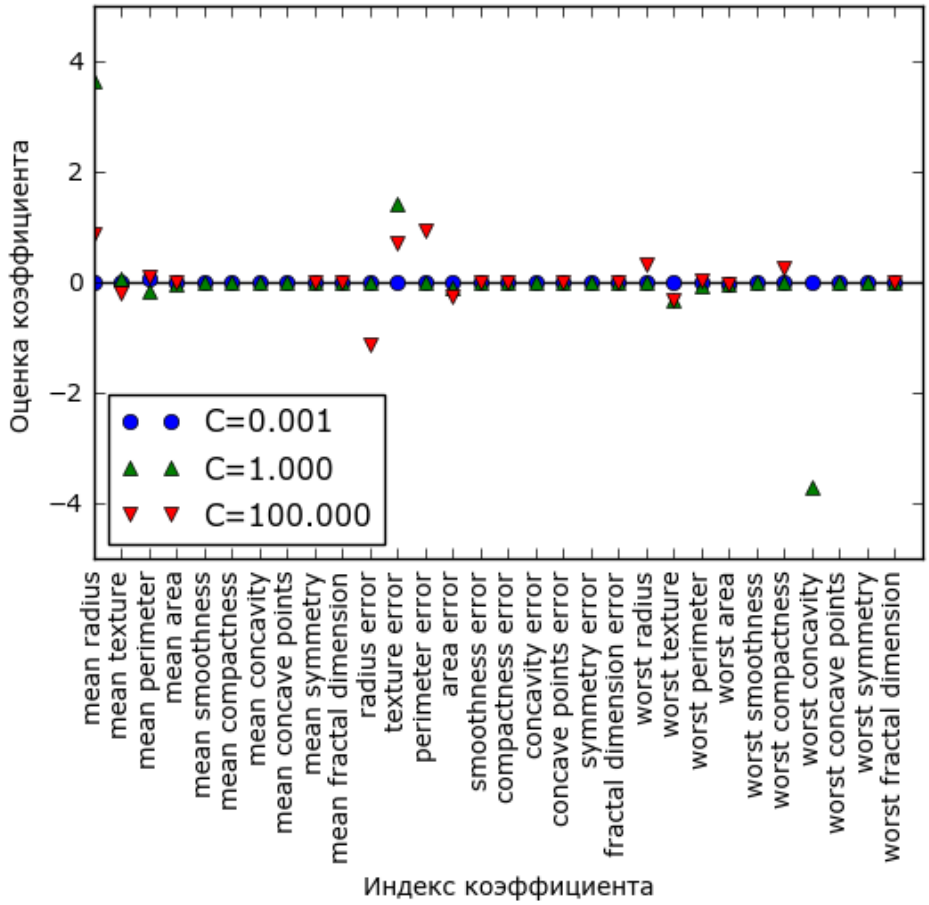

Рис. 2.18 Коэффициенты логистической регрессии с L1 штрафом для набора данных Breast Cancer (использовались различные значения C)

Линейные модели для мультиклассовой классификации

Многие линейные модели классификации предназначены лишь для бинарной классификации и не распространяются на случай мультиклассовой классификации (за исключением логистической регрессии). Общераспространенный подход, позволяющий распространить алгоритм бинарной классификации на случай мультиклассовой классификации называет подходом «*один против остальных*» (*one-vs.-rest*).[14] В подходе «один против остальных» для каждого класса строится бинарная модель, которая пытается отделить

[14] Также используется название «один против всех» (one-vs.-all). – *Прим. пер.*

этот класс от всех остальных, в результате чего количество моделей определяется количеством классов. Для получения прогноза точка тестового набора подается на все бинарные классификаторы. Классификатор, который выдает по своему классу наибольшее значение, «побеждает» и метка этого класса возвращается в качестве прогноза.

Используя бинарный классификатор для каждого класса, мы получаем один вектор коэффициентов (w) и одну константу (b) по каждому классу. Класс, который получает наибольшее значение согласно нижеприведенной формуле, становится присвоенной меткой класса:

$$w[0]*x[0]+w[1]*x[1]+...+w[p]*x[p]+b$$

Математический аппарат мультиклассовой логистической регрессии несколько отличается от подхода «один против остальных», однако он также дает один вектор коэффициентов и константу для каждого класса и использует тот же самый способ получения прогнозов.

Давайте применим метод «один против остальных» к простому набору данных с 3-классовой классификацией. Мы используем двумерный массив данных, где каждый класс задается данными, полученными из гауссовского распределения (см. рис. 2.19):

```
In[47]:
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=42)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
plt.legend(["Класс 0", "Класс 1", "Класс 2"])
```

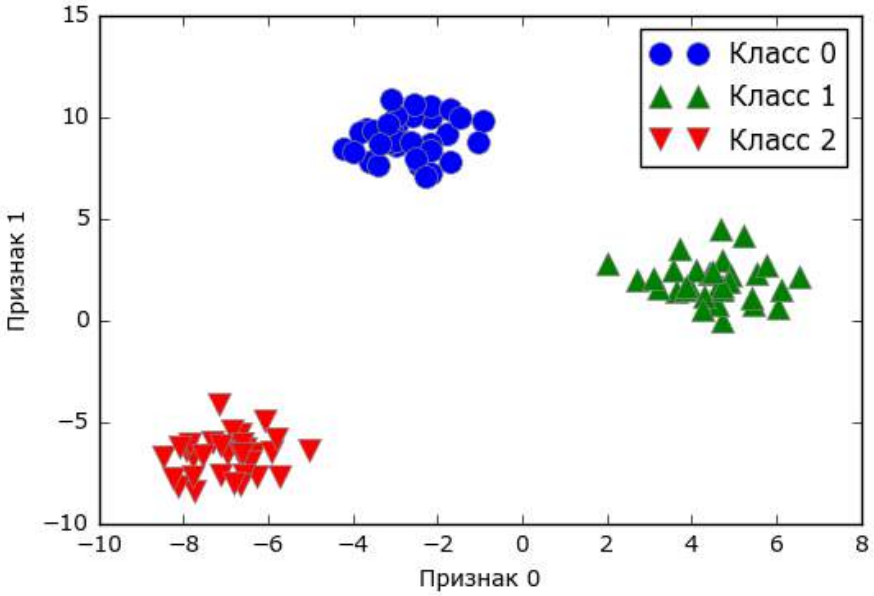

Рис. 2.19 Двумерный синтетический набор данных, содержащий три класса

Теперь обучаем классификатор `LinearSVC` на этом наборе данных:

```
In[48]:
linear_svm = LinearSVC().fit(X, y)
print("Форма коэффициента: ", linear_svm.coef_.shape)
print("Форма константы: ", linear_svm.intercept_.shape)

Out[48]:
Форма коэффициента: (3, 2)
Форма константы: (3,)
```

Мы видим, что атрибут `coef_` имеет форму `(3, 2)`, это означает, что каждая строка `coef_` содержит вектор коэффициентов для каждого из трех классов, а каждый столбец содержит значение коэффициента для конкретного признака (в этом наборе данных их два). Атрибут `intercept_` теперь является одномерным массивом, в котором записаны константы классов.

Давайте визуализируем линии (границы принятия решений), полученные с помощью трех бинарных классификаторов (рис. 2.20):

```
In[49]:
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
                                  ['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
plt.legend(['Класс 0', 'Класс 1', 'Класс 2', 'Линия класса 0', 'Линия класса 1',
            'Линия класса 2'], loc=(1.01, 0.3))
```

Видно, что все точки, принадлежащие классу 0 в обучающих данных, находятся выше линии, соответствующей классу 0. Это означает, что они отнесены к «классу 0» данного бинарного классификатора. Точки класса 0 находятся выше линии, соответствующей классу 2. Это означает, что они классифицируются бинарным классификатором для класса 2 как «остальные». Точки, принадлежащие классу 0, находятся слева от линии, соответствующей классу 1. Это означает, что бинарный классификатор для класса 1 также классифицирует их как «остальные». Таким образом, в итоге любая точка в этой области будет отнесена к классу 0 (результат, получаемый по формуле для классификатора 0, больше нуля, тогда как для двух остальных классов он меньше нуля).

Однако что насчет треугольника в середине графика? Все три бинарных классификатора относят точки, расположенные там, к «остальным». Какой класс будет присвоен точке, расположенной в треугольнике? Ответ – класс, получивший наибольшее значение по формуле классификации, то есть класс ближайшей линии.

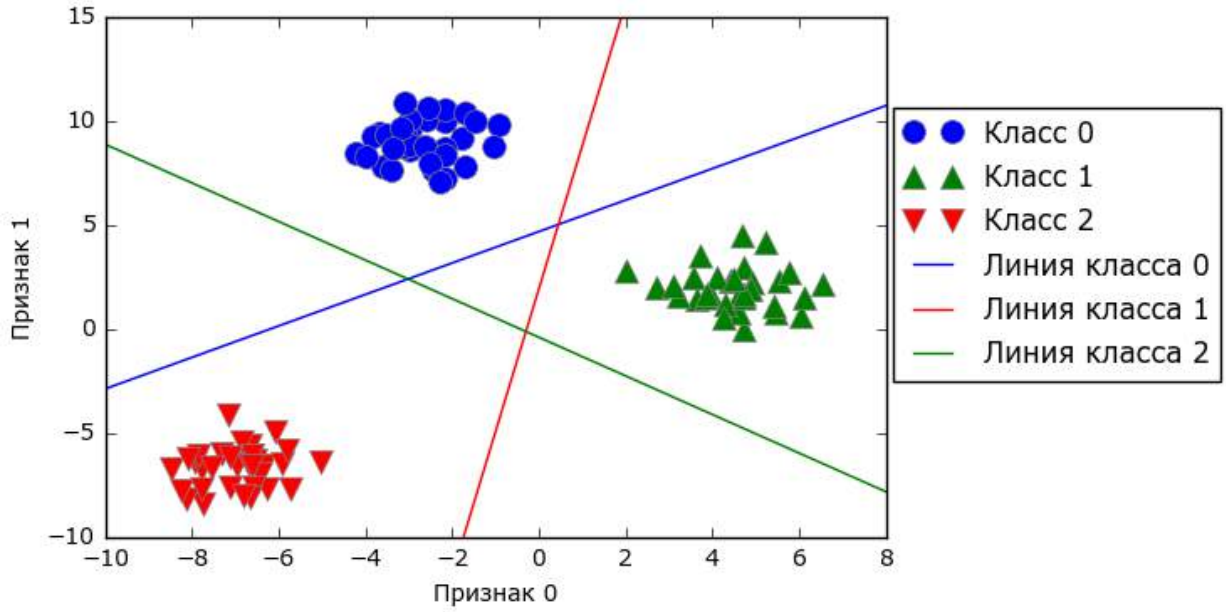

Рис. 2.20 Границы принятия решений, полученные с помощью трех бинарных классификаторов в рамках подхода «один против остальных»

Следующий пример (рис. 2.21) показывает прогнозы для всех областей двумерного пространства:

```
In[50]:
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
                                  ['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.legend(['Класс 0', 'Класс 1', 'Класс 2', 'Линия класса 0', 'Линия класса 1',
            'Линия класса 2'], loc=(1.01, 0.3))
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

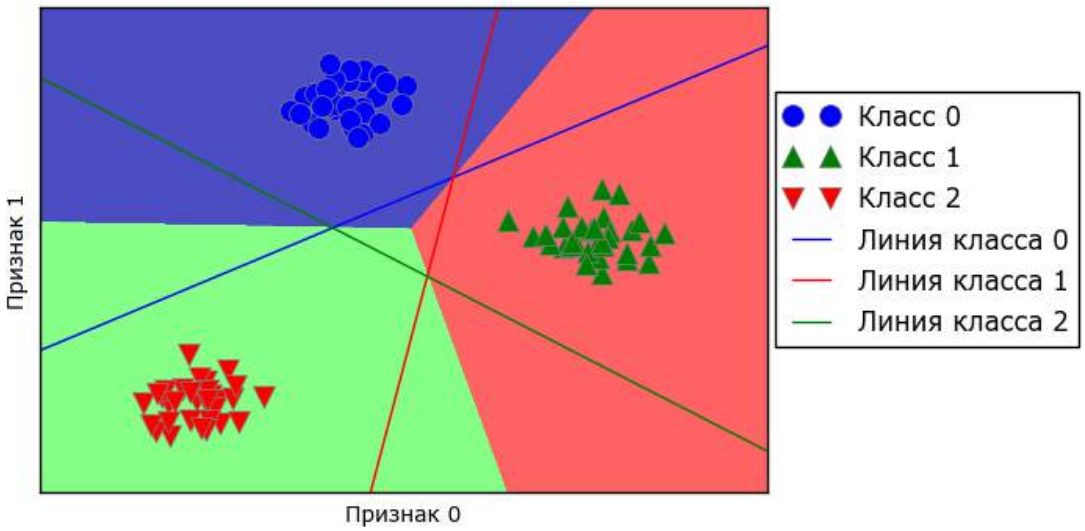

Рис. 2.21 Мультиклассовые границы принятия решений, полученные с помощью трех бинарных классификаторов в рамках подхода «один против остальных»

Преимущества, недостатки и параметры

Основной параметр линейных моделей – параметр регуляризации, называемый `alpha` в моделях регрессии и `C` в `LinearSVC` и `LogisticRegression`. Большие значения `alpha` или маленькие значения `C` означают простые модели. Конкретно для регрессионных моделей настройка этих параметров имеет весьма важное значение. Как правило, поиск `C` и `alpha` осуществляется по логарифмической шкале. Кроме того вы должны решить, какой вид регуляризации нужно использовать: L1 или L2. Если вы полагаете, что на самом деле важны лишь некоторые признаки, следует использовать L1. В противном случае используйте установленную по умолчанию L2 регуляризацию. Еще L1 регуляризация может быть полезна, если интерпретируемость модели имеет важное значение. Поскольку L1 регуляризация будет использовать лишь несколько признаков, легче будет объяснить, какие признаки важны для модели и каковы эффекты этих признаков.

Линейные модели очень быстро обучаются, а также быстро прогнозируют. Они масштабируются на очень большие наборы данных, а также хорошо работают с разреженными данными. При работе с данными, состоящими из сотен тысяч или миллионов примеров, вас, возможно, заинтересует опция `solver='sag'` в `LogisticRegression` и `Ridge`, которая позволяет получить результаты быстрее, чем настройки по умолчанию. Еще пара опций – это класс `SGDClassifier` и класс `SGDRegressor`, реализующие более масштабируемые версии описанных здесь линейных моделей.

Еще одно преимущество линейных моделей заключается в том, что они позволяют относительно легко понять, как был получен прогноз, при помощи формул, которые мы видели ранее для регрессии и классификации. К сожалению, часто бывает совершенно не понятно, почему были получены именно такие коэффициенты. Это особенно актуально, если ваш набор данных содержит высоко коррелированные признаки, в таких случаях коэффициенты сложно интерпретировать.

Как правило, линейные модели хорошо работают, когда количество признаков превышает количество наблюдений. Кроме того, они часто используются на очень больших наборах данных, просто потому, что не представляется возможным обучить другие модели. Вместе с тем в низкоразмерном пространстве альтернативные модели могут показать более высокую обобщающую способность. В разделе «Ядерные машины опорных векторов» мы рассмотрим несколько примеров, в которых использование линейных моделей не увенчалось успехом.

Цепочка методов (method chaining)

Во всех моделях `scikit-learn` метод `fit` возвращает `self`. Это позволяет писать код, приведенный ниже и уже широко использованный нами в этой главе:

```
In[51]:
# создаем экземпляр модели и подгоняем его в одной строке
logreg = LogisticRegression().fit(X_train, y_train)
```

Здесь мы использовали значение, возвращаемое методом `fit` (`self`), чтобы присвоить обученную модель переменной `logreg`. Эта конкатенация вызовов методов (в данном случае `_init_`, а затем `fit`) известна как *цепочка методов* (*method chaining*). Еще одно обшераспространенное применение цепочки методов в `scikit-learn` – это связывание методов `fit` и `predict` в одной строке:

```
In[52]:
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

Наконец, вы можете создать экземпляр модели, подогнать модель и получить прогнозы в одной строке:

```
In[53]:
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

Однако этот очень короткий вариант не идеален. В одной строке происходит масса всего, что может сделать код трудночитаемым. Кроме того, подогнанная модель логистической регрессии не сохранена в какой-то определенной переменной, поэтому мы не можем проверить или использовать ее, чтобы получить прогнозы для других данных.

Наивные байесовские классификаторы

Наивные байесовские классификаторы представляют собой семейство классификаторов, которые очень схожи с линейными моделями, рассмотренными в предыдущем разделе. Однако они имеют тенденцию обучаться быстрее. Цена, которую приходится платить за такую эффективность – немного более низкая обобщающая способность моделей Байеса по сравнению с линейными классификаторами типа `LogisticRegression` и `LinearSVC`.

Причина, по которой наивные байесовские модели столь эффективны, заключается в том, что они оценивают параметры, рассматривая каждый признак отдельно и по каждому признаку собирают простые статистики классов. В `scikit-learn` реализованы три вида наивных байесовских классификаторов: `GaussianNB`, `BernoulliNB` и `MultinomialNB`. `GaussianNB` можно применить к любым непрерывным данным, в то время как `BernoulliNB` принимает бинарные данные, `MultinomialNB` принимает счетные или дискретные данные (то есть каждый признак представляет собой подсчет целочисленных значений какой-то характеристики,

например, речь может идти о частоте встречаемости слова в предложении). `BernoulliNB` и `MultinomialNB` в основном используются для классификации текстовых данных.

Классификатор `BernoulliNB` подсчитывает ненулевые частоты признаков по каждому классу. Это легче всего понять на примере:

```
In[54]:
X = np.array([[0, 1, 0, 1],
              [1, 0, 1, 1],
              [0, 0, 0, 1],
              [1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])
```

Здесь у нас есть четыре точки данных с четырьмя бинарными признаками. Есть два класса 0 и 1. Для класса 0 (первая и третья точки данных) первый признак равен нулю два раза и отличен от нуля ноль раз, второй признак равен нулю один раз и отличен от нуля один раз и так далее. Те же самые частоты затем подсчитываются для точек данных во втором классе. Подсчет ненулевых элементов в каждом классе по сути выглядит следующим образом:

```
In[55]:
counts = {}
for label in np.unique(y):
    # итерируем по каждому классу
    # подсчитываем (суммируем) элементы 1 по признаку
    counts[label] = X[y == label].sum(axis=0)
print("Частоты признаков:\n{}".format(counts))

Out[55]:
Частоты признаков:
{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

Две другие наивные байесовские модели `MultinomialNB` и `GaussianNB`, немного отличаются с точки зрения вычисляемых статистик. `MultinomialNB` принимает в расчет среднее значение каждого признака для каждого класса, в то время как `GaussianNB` записывает среднее значение, а также стандартное отклонение каждого признака для каждого класса.

Для получения прогноза точка данных сравнивается со статистиками для каждого класса и прогнозируется наиболее подходящий класс. Интересно отметить, что для `MultinomialNB` и `BernoulliNB` это приводит к прогнозной формуле, которая имеет точно такой же вид, что и формула для линейных моделей (см. «Линейные модели классификации»). К сожалению, `coef_` для наивных байесовских моделей имеет несколько иной смысл, чем `coef_` для линейных моделей, здесь `coef_` не тождественен *и.*

Преимущества, недостатки и параметры

`MultinomialNB` и `BernoulliNB` имеют один параметр `alpha`, который контролирует сложность модели. Параметр `alpha` работает следующим образом: алгоритм добавляет к данным зависящее от `alpha` определенное количество искусственных наблюдений с положительными значениями для всех признаков. Это приводит к «сглаживанию» статистик. Большее значение `alpha` означает более высокую степень сглаживания, что приводит к построению менее сложных моделей. Алгоритм относительно устойчив к разным значениям `alpha`. Это означает, что значение `alpha` не оказывает значительного влияния на хорошую работу модели. Вместе с тем тонкая настройка этого параметра обычно немного увеличивает правильность.

`GaussianNB` в основном используется для данных с очень высокой размерностью, тогда как остальные наивные байесовские модели широко используются для разреженных дискретных данных, например, для текста. `MultinomialNB` обычно работает лучше, чем `BernoulliNB`, особенно на наборах данных с относительно большим количеством признаков, имеющих ненулевые частоты (т.е. на больших документах).

Наивные байесовские модели разделяют многие преимущества и недостатки линейных моделей. Они очень быстро обучаются и прогнозируют, а процесс обучения легко интерпретировать. Модели очень хорошо работают с высокоразмерными разреженными данными и относительно устойчивы к изменениям параметров. Наивные байесовские модели являются замечательными базовыми моделями и часто используются на очень больших наборах данных, где обучение даже линейной модели может занять слишком много времени.

Деревья решений

Деревья решений являются моделями, широко используемыми для решения задач классификации и регрессии. По сути они задают вопросы и выстраивают иерархию правил «если... то», приводящую к решению.

Эти вопросы похожи на вопросы, которые вы можете спросить в игре «20 Questions». Представьте, вам нужно научиться отличать друг от друга четыре вида животных: медведей, ястребов, пингвинов и дельфинов. Ваша цель состоит в том, чтобы получить правильный ответ, задав несколько вопросов. Вы могли бы начать с вопроса, есть ли у этих видов животных перья, вопроса, который сужает количество возможных видов животных до двух. Если получен ответ «да», вы можете задать еще один вопрос, который может помочь вам различать ястребов и пингвинов. Например, вы могли бы спросить, может ли данный вид животных летать. Если у этого вида животных нет перьев, ваши возможные

варианты – дельфины и медведи, и вам нужно задать вопрос, чтобы провести различие между этими двумя видами животных, например, спросить, есть ли плавники у этого вида животных.

Эти вопросы можно выразить в виде дерева решений, как это показано на рис. 2.22.

```
In[56]:
mglearn.plots.plot_animal_tree()
```

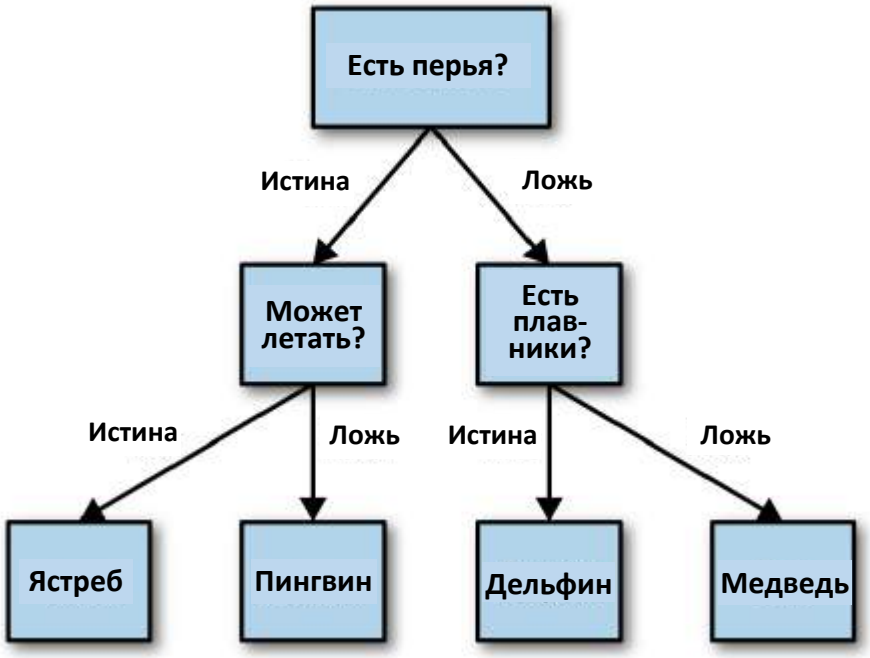

Рис. 2.22 Дерево решений, различающее несколько видов животных

На этом рисунке каждый узел дерева либо представляет собой либо вопрос, либо терминальный узел (его еще называют *листом* или *leaf*), который содержит ответ. Ребра соединяют вышестоящие узлы с нижестоящими.

Говоря языком машинного обучения, мы построили модель, различающую четыре класса животных (ястребов, пингвинов, дельфинов и медведей), используя три признака «есть перья», «может летать» и «имеет плавники». Вместо того, чтобы строить эти модели вручную, мы можем построить их с помощью контролируемого обучения.

Построение деревьев решений

Давайте рассмотрим процесс построения дерева решений для двумерного классификационного набора данных, показанного на рис. 2.23. Набор данных состоит из точек, обозначаемых маркерами двух типов. Каждому типу маркера соответствует свой класс, на каждый класс приходится по 75 точек данных. Назовем этот набор данных `two_moons`.

Построение дерева решений означает построение последовательности правил «если... то...», которая приводит нас к истинному ответу максимально коротким путем. В машинном обучении эти правила называются *тестами* (*tests*). Не путайте их с тестовым набором, который мы используем для проверки обобщающей способности нашей модели. Как правило, данные бывают представлены не только в виде бинарных признаков да/нет, как в примере с животными, но и в виде непрерывных признаков, как в двумерном наборе данных, показанном на рис. 2.23. Тесты, которые используются для непрерывных данных имеют вид «Признак *i* больше значения *a*?"

```
In[57]:
mglearn.plots.plot_tree_progressive()
```

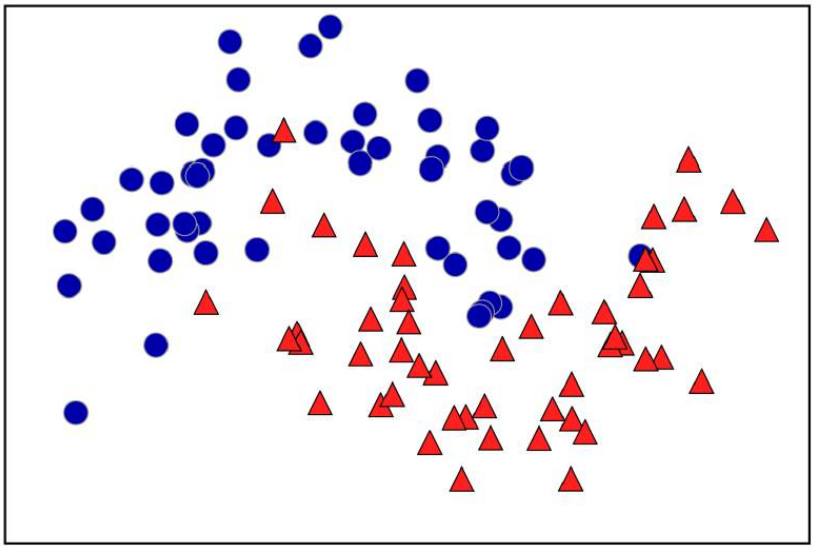

Рис. 2.23 Набор данных two_moons, по которому будет построено дерево решений

Чтобы построить дерево, алгоритм перебирает все возможные тесты и находит тот, который является наиболее информативным с точки зрения прогнозирования значений целевой переменной. Рис. 2.24 показывает первый выбранный тест. Разделение набора данных по горизонтали в точке `x[1]=0.0596` дает наиболее полную информацию. Оно лучше всего разделяет точки класса 0 от точек класса 1. Верхний узел, также называемый *корнем* (*root*), представляет собой весь набор данных, состоящий из 50 точек, принадлежащих к классу 0, и 50 точек, принадлежащих к классу 1. Разделение выполняется путем тестирования `x[1]<=0.0596`, обозначенного черной линией. Если тест верен, точка назначается левому узлу, который содержит 2 точки, принадлежащие классу 0, и 32 точки, принадлежащие классу 1. В противном случае точка будет присвоена правому узлу, который содержит 48 точек, принадлежащих классу 0, и 18 точек, принадлежащих классу 1. Эти два

узла соответствуют верхней и нижней областям, показанным на рис. 2-24. Несмотря на то что первое разбиение довольно хорошо разделило два класса, нижняя область по-прежнему содержит точки, принадлежащие к классу 0, а верхняя область по-прежнему содержит точки, принадлежащие к классу 1. Мы можем построить более точную модель, повторяя процесс поиска наилучшего теста в обоих областях. Рис. 2.25 показывает, что следующее наиболее информативное разбиение для левой и правой областей основывается на `x[0]`.

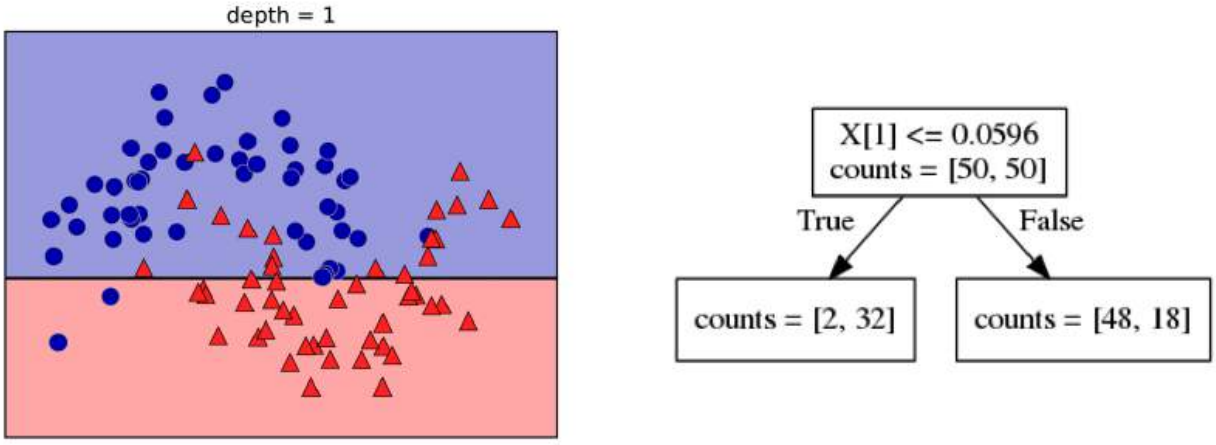

Рис. 2.24 Граница принятия решений, полученная с помощью дерева глубиной 1 (слева) и соответствующее дерево решений (справа)

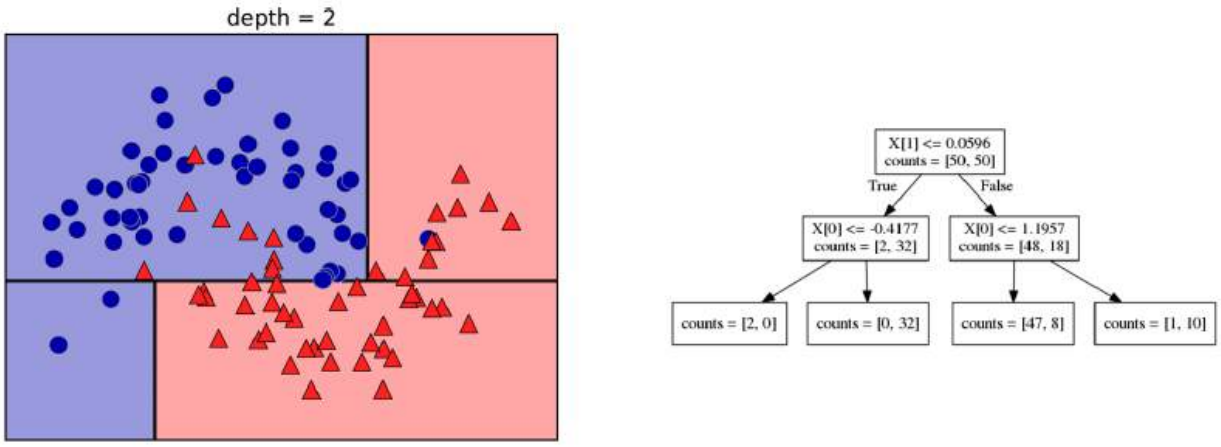

Рис. 2.25 Граница принятия решений, полученная с помощью дерева глубиной 2 (слева) и соответствующее дерево решений (справа)

Этот рекурсивный процесс строит в итоге бинарное дерево решений, в котором каждый узел соответствует определенному тесту. Кроме того, вы можете интерпретировать тест как разбиение части данных, рассматриваемое в данном случае вдоль одной оси. Это позволяет составить представление об алгоритме как способе выстроить иерархию разбиений. Поскольку каждый тест рассматривает только один признак,

области, получающиеся в результате разбиения, всегда имеют границы, параллельные осям.

Рекурсивное разбиение данных повторяется до тех пор, пока все точки данных в каждой области разбиения (каждом листе дерева решений) не будут принадлежать одному и тому же значению целевой переменной (классу или количественному значению). Лист дерева, который содержит точки данных, относящиеся к одному и тому же значению целевой переменной, называется *чистым* (*pure*). Итоговое разбиение для нашего набора данных показано на рис. 2.26.

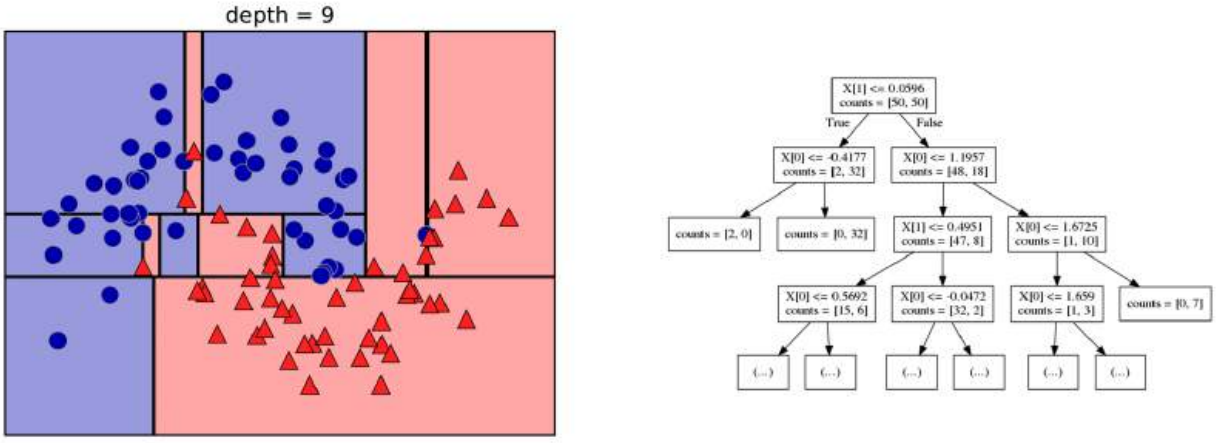

Рис. 2.26 Граница принятия решений, полученная с помощью дерева глубиной 9 (слева) и фрагмент соответствующего дерева (справа), полное дерево имеет довольно большой размер и его сложно визуализировать

Прогноз для новой точки данных получают следующим образом: сначала выясняют, в какой области разбиения пространства признаков находится данная точка, а затем определяют класс, к которому относится большинство точек в этой области (либо единственный класс в области, если лист является чистым). Область может быть найдена с помощью обхода дерева, начиная с корневого узла и путем перемещения влево или вправо, в зависимости от того, выполняется ли тест или нет.

Кроме того, можно использовать деревья для решения задач регрессии, используя точно такой же подход. Для получения прогноза мы обходим дерево на основе тестов в каждом узле и находим лист, в который попадает новая точка данных. Выходом для этой точки данных будет значение целевой переменной, усредненное по всем обучающим точкам в этом листе.

Контроль сложности деревьев решений

Как правило, построение дерева, описанное здесь и продолжающееся до тех пор, пока все листья не станут чистыми, приводит к получению моделей, которые являются очень сложными и характеризуются сильным переобучением на обучающих данных. Наличие чистых листьев

означает, что дерево имеет 100%-ную правильность на обучающей выборке. Каждая точка обучающего набора находится в листе, который имеет правильный мажоритарный класс. Переобучение можно увидеть в левой части рис. 2.26. Видно, что точки, определяемые как точки класса 1, находятся посреди точек, принадлежащих к классу 0. С другой стороны, мы видим ряд точек, спрогнозированных как класс 1, вокруг точки, отнесенной к классу 0. Это не та граница принятия решений, которую мы могли бы себе представить. Здесь граница принятия решений фокусируется больше на отдельных точках-выбросах, которые находятся слишком далеко от остальных точек данного класса.

Есть две общераспространенные стратегии, позволяющие предотвратить переобучение. Первая стратегия – ранняя остановка построения дерева, называемая *предварительной обрезкой* (*pre-pruning*). Вторая стратегия – построение дерева с последующим удалением или сокращением малоинформативных узлов, называемое *пост-обрезкой* (*post-pruning*) или просто *обрезкой* (*pruning*). Возможные критерии предварительной обрезки включают в себя ограничение максимальной глубины дерева, ограничение максимального количества листьев или минимальное количество наблюдений в узле, необходимое для разбиения.

В библиотеке `scikit-learn` деревья решений реализованы в классах `DecisionTreeRegressor` и `DecisionTreeClassifier`. Обратите внимание, в `scikit-learn` реализована лишь предварительная обрезка.

Давайте более детально посмотрим, как работает предварительная обрезка на примере набора данных Breast Cancer. Как всегда, мы импортируем набор данных и разбиваем его на обучающую и тестовую части. Затем мы строим модель, используя настройки по умолчанию для построения полного дерева (выращиваем дерево до тех пор, пока все листья не станут чистыми). Зафиксируем `random_state` для воспроизводимости результатов:

```
In[58]:
from sklearn.tree import DecisionTreeClassifier

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.3f}".format(tree.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(tree.score(X_test, y_test)))

Out[58]:
Правильность на обучающем наборе: 1.000
Правильность на тестовом наборе: 0.937
```

Как и следовало ожидать, правильность на обучающем наборе составляет 100%, поскольку листья являются чистыми. Дерево имеет

глубину, как раз достаточную для того, чтобы прекрасно запомнить все метки обучающих данных. Правильность на тестовом наборе немного хуже, чем при использовании ранее рассмотренных линейных моделей, правильность которых составляла около 95%.

Если не ограничить глубину, дерево может быть сколь угодно глубоким и сложным. Поэтому необрезанные деревья склонны к переобучению и плохо обобщают результат на новые данные. Теперь давайте применим к дереву предварительную обрезку, которая остановит процесс построения дерева до того, как мы идеально подгоним модель к обучающим данным. Один из вариантов – остановка процесса построения дерева по достижении определенной глубины. Здесь мы установим `max_depth=4`, то есть можно задать только четыре последовательных вопроса (см. рис. 2.24 и 2.26). Ограничение глубины дерева уменьшает переобучение. Это приводит к более низкой правильности на обучающем наборе, но улучшает правильность на тестовом наборе:

```
In[59]:
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

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

Out[59]:
Правильность на обучающем наборе: 0.988
Правильность на тестовом наборе: 0.951
```

Анализ деревьев решений

Мы можем визуализировать дерево, используя функцию `export_graphviz` из модуля `tree`. Она записывает файл в формате *.dot*, который является форматом текстового файла, предназначенным для описания графиков. Мы можем задать цвет узлам, чтобы выделить класс, набравший большинство в каждом узле, и передать имена классов и признаков, чтобы дерево было правильно размечено:

```
In[60]:
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot", class_names=["malignant", "benign"],
                feature_names=cancer.feature_names, impurity=False, filled=True)
```

Мы можем прочитать этот файл и визуализировать его, как показано на рис. 2.27, используя модуль `graphviz`[15] (или любую другую программу, которая может читать файлы с расширением *.dot*):

[15] Если вы используете Anaconda под Windows, то необходимо установить conda-пакет `graphviz` и pip-пакет `graphviz`:

```
conda install -c anaconda graphviz=2.38.0
```

```
In[61]:
import graphviz

with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

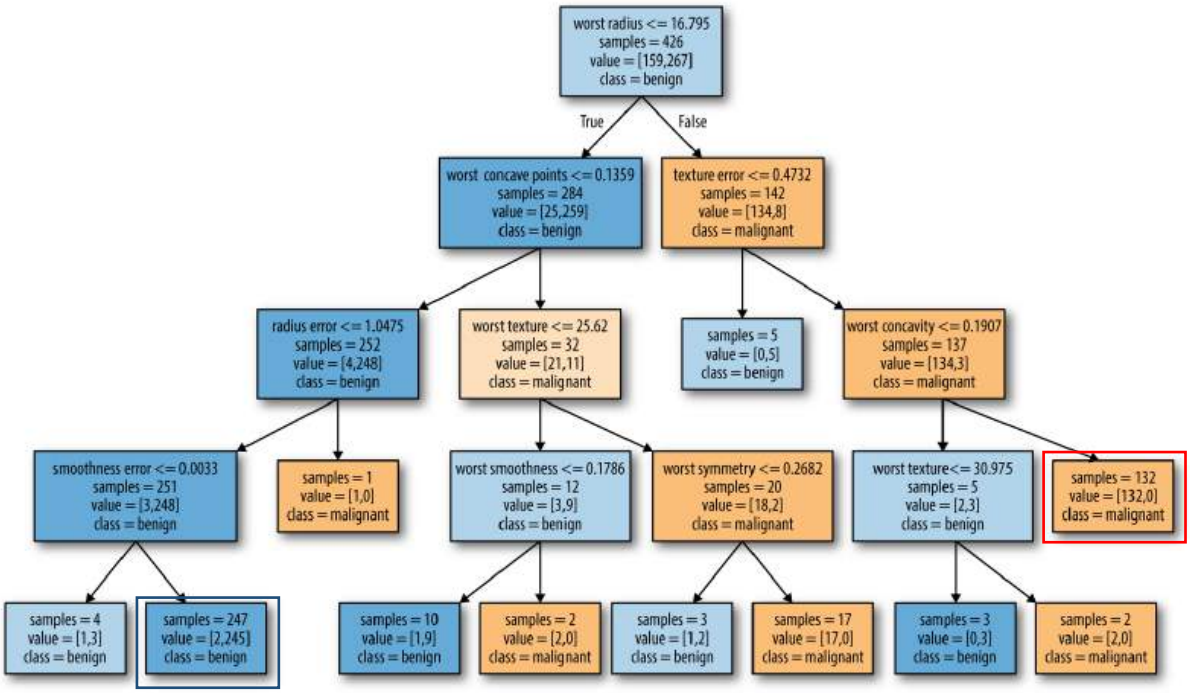

Рис. 2.27 Визуализация дерева решений, построенного на наборе данных Breast Cancer

Как вариант, можно построить диаграмму дерева и записать ее в файл *.pdf*. Дополнительно нам потребуется модуль `pydotplus`.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import mglearn
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn import tree
from sklearn.tree import export_graphviz
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
clf = tree.DecisionTreeClassifier(max_depth=4, random_state=0)
clf = clf.fit(X_train, y_train)

import pydotplus
dot_data = tree.export_graphviz(clf, out_file=None)
```

```
pip install graphviz
```

Затем в переменной окружения PATH необходимо прописать полный путь к установленной папке `graphviz`. В Windows 7 для этого нажмите кнопку **Пуск**, выберите **Панель управления**. Дважды нажмите на **Система**, затем выберите **Дополнительные параметры системы**. Во вкладке **Дополнительно** нажмите на **Переменные среды**. Выберите **Path** и нажмите на **Изменить**. В поле **Значение переменной** введите путь к папке `graphviz` (например, C:\Anaconda3\Library\bin\graphviz). – *Прим. пер.*

```
graph = pydotplus.graph_from_dot_data(dot_data)
graph.write_pdf("cancer.pdf")
```

Можно построить визуализацию дерева с помощью функции `Image` интерактивной оболочки `IPython`:

```
from IPython.display import Image
dot_data = tree.export_graphviz(clf, out_file=None,
                      feature_names=cancer.feature_names,
                      class_names=cancer.target_names,
                      filled=True, rounded=True,
                      special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data)
Image(graph.create_png())
```

Визуализация дерева дает более глубокое представление о том, как алгоритм делает прогнозы и является хорошим примером алгоритма машинного обучения, который легко объяснить неспециалистам. Однако, как показано здесь, даже при глубине 4 дерево может стать немного громоздким. Деревья с бо́льшим значением глубины (деревья глубиной 10 – не редкость) еще труднее понять. Один из полезных способов исследования дерева заключается в том, чтобы выяснить, какие узлы содержат наибольшее количество данных. Параметр `samples`, выводимый в каждом узле на рис. 2.27, показывает общее количество примеров в узле, тогда как параметр `value` показывает количество примеров в каждом классе. Проследовав по правой ветви, отходящей от корневого узла, мы видим, что правилу `worst radius > 16.795` соответствует узел, который содержит 134 случая злокачественной опухоли и лишь 8 случаев доброкачественной опухоли. Далее дерево выполняет серию более точных разбиений оставшихся 142 случаев. Из 142 случаев, которые при первоначальном разбиении были записаны в правый узел, почти все (132) в конечном итоге попали в правый лист (для удобства выделен красной рамкой).

Проследовав по левой ветви, отходящей от корневого узла, мы видим, что правилу `worst radius <= 16.795` соответствует узел, который содержит 25 случаев злокачественной опухоли и 259 случаев доброкачественной опухоли. Почти все случаи доброкачественной опухоли попадают во второй лист справа (для удобства выделен синей рамкой), остальные случаи распределяются по нескольким листьям, содержащим очень мало наблюдений.

Важность признаков в деревьях

Вместо того, чтобы просматривать все дерево, что может быть обременительно, есть некоторые полезные параметры, которые мы можем использовать как итоговые показатели работы дерева. Наиболее часто используемым показателем является *важность признаков* (*feature importance*), которая оценивает, насколько важен каждый признак с

точки зрения получения решений. Это число варьирует в диапазоне от 0 до 1 для каждого признака, где 0 означает «не используется вообще», а 1 означает, что «отлично предсказывает целевую переменную». Важности признаков в сумме всегда дают 1:

```
In[62]:
print("Важности признаков:\n{}".format(tree.feature_importances_))
```

```
Out[62]:
Важности признаков
[ 0.          0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.01019737  0.04839825  0.          0.
  0.0024156   0.          0.          0.          0.          0.
  0.72682851  0.0458159   0.          0.          0.0141577   0.          0.018188
  0.1221132   0.01188548  0.        ]
```

Приведенная сводка не совсем удобна, поскольку мы не знаем, каким именно признакам соответствуют приведенные важности. Чтобы исправить это, воспрользуемся программным кодом, приведенным ниже:

```
for name, score in zip(cancer["feature_names"], tree.feature_importances_):
    print(name, score)
```

```
mean radius 0.0
mean texture 0.0
mean perimeter 0.0
mean area 0.0
mean smoothness 0.0
mean compactness 0.0
mean concavity 0.0
mean concave points 0.0
mean symmetry 0.0
mean fractal dimension 0.0
radius error 0.0101973682021
texture error 0.0483982536186
perimeter error 0.0
area error 0.0
smoothness error 0.00241559508532
compactness error 0.0
concavity error 0.0
concave points error 0.0
symmetry error 0.0
fractal dimension error 0.0
worst radius 0.72682850946
worst texture 0.0458158970889
worst perimeter 0.0
worst area 0.0
worst smoothness 0.0141577021047
worst compactness 0.0
worst concavity 0.0181879968645
worst concave points 0.122113199265
worst symmetry 0.0118854783101
worst fractal dimension 0.0
```

Мы можем визуализировать важности признаков аналогично тому, как мы визуализируем коэффициенты линейной модели (рис. 2.28):

```
In[63]:
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(tree)
```

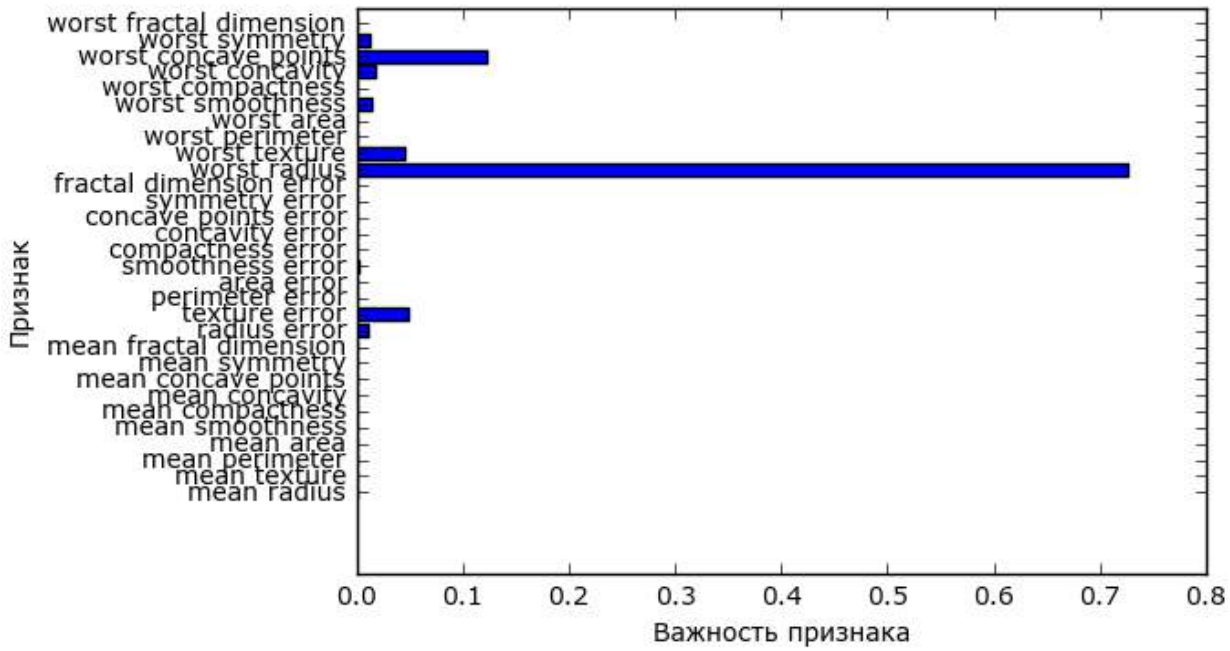

Рис. 2.28 Важности признаков, вычисленные с помощью дерева решений для набора данных Breast Cancer

Здесь мы видим, что признак, использованный в самом верхнем разбиении («worst radius»), на данный момент является наиболее важным. Это подтверждает наш вывод о том, что уже на первом уровне два класса достаточно хорошо разделены.

Однако, если признак имеет низкое значение `feature_importance_`, это не значит, что он неинформативен. Это означает только то, что данный признак не был выбран деревом, поскольку, вероятно, другой признак содержит ту же самую информацию.

В отличие от коэффициентов линейных моделей важности признаков всегда положительны и они не указывают на взаимосвязь с каким-то конкретным классом. Важности признаков говорят нам, что «worst radius» важен, но мы не знаем, является ли высокое значение радиуса признаком доброкачественной или злокачественной опухоли. На самом деле, найти такую очевидную взаимосвязь между признаками и классом невозможно, что можно проиллюстрировать на следующем примере (рис. 2.29 и 2.30):

```
In[64]:
tree = mglearn.plots.plot_tree_not_monotone()
display(tree)

Out[64]:
Feature importances: [ 0. 1.]
```

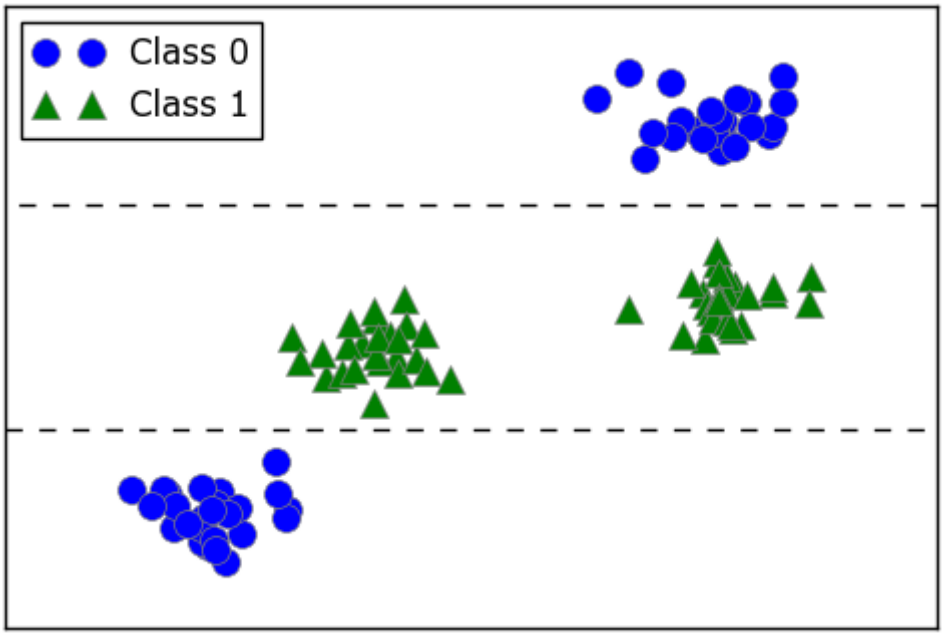

Рис. 2.29 Двумерный массив данных, в котором признак имеет немонотонную взаимосвязь с меткой класса, и границы принятия решений, найденные с помощью дерева

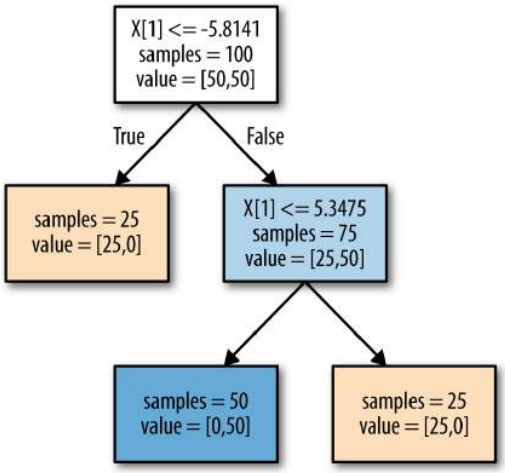

Рис. 2.30 Дерево решений для набора данных, показанном на рис. 2.29

График показывает набор данных с двумя признаками и двумя классами. Здесь вся информация содержится в `X[1]`, а `X[0]` не используется вообще. Но взаимосвязь между `X[1]` и целевым классом не является монотонной, то есть мы не можем сказать, что «высокое

значение `X[0]` означает класс 0, а низкое значение означает класс 1» (или наоборот).

Несмотря на то что мы сосредоточились здесь на деревьях классификации, все вышесказанное верно и для деревьев регрессии, которые реализованы в `DecisionTreeRegressor`. Применение и анализ деревьев регрессии очень схожи с применением и анализом деревьев классификации. Однако существует одна особенность использования деревьев регрессии, на которую нужно указать. `DecisionTreeRegressor` (и все остальные регрессионные модели на основе дерева) не умеет *экстраполировать* или делать прогнозы вне диапазона значений обучающих данных.

Давайте детальнее рассмотрим это, воспользовавшись набором данных RAM Price (содержит исторические данные о ценах на компьютерную память). Рис. 2.31 визуализирует этот набор данных[16], дата отложена по оси x, а цена одного мегабайта оперативной памяти в соответствующем году – по оси y:

```
In[65]:
import pandas as pd
ram_prices = pd.read_csv("C:/Data/ram_price.csv")

plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Год")
plt.ylabel("Цена $/Мбайт")
```

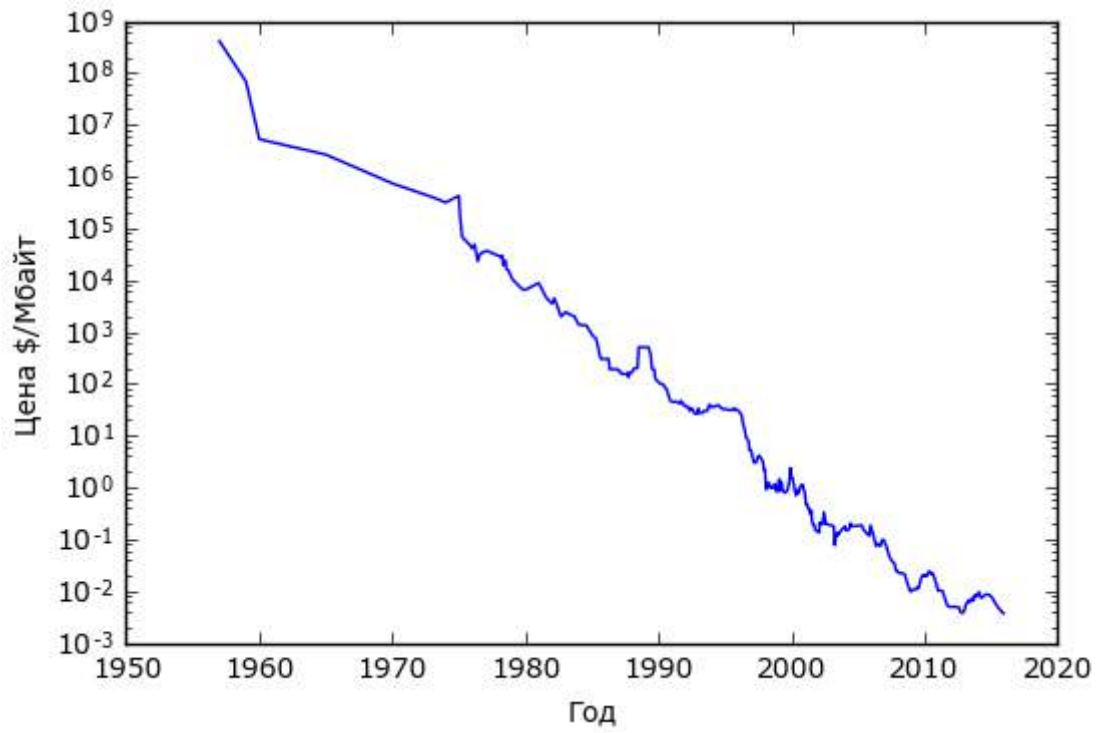

Рис. 2.31 Историческое развитие цен на RAM по логарифмической шкале

[16] Скачайте набор данных ram_price.csv по ссылке и перенесите его в папку Data на диске C. – *Прим. пер.*

Обратите внимание на логарифмическую шкалу оси y. При логарифмическом преобразовании взаимосвязь выглядит вполне линейной и таким образом становится легко прогнозируемой, за исключением некоторых всплесков.

Мы будем прогнозировать цены на период после 2000 года, используя исторические данные до этого момента, единственным признаком будут даты. Мы сравним две простые модели: `DecisionTreeRegressor` и `LinearRegression`. Мы отмасштабируем цены, используя логарифм, таким образом, взаимосвязь будет относительно линейной. Это несущественно для `DecisionTreeRegressor`, однако существенно для `LinearRegression` (мы рассмотрм ее более подробно в главе 4). После обучения модели и получения прогнозов мы применим экспоненцирование, чтобы обратить логарифмическое преобразование. Мы получим и визуализируем прогнозы для всего набора данных, но для количественной оценки мы будем рассматривать только тестовый набор:

```
In[66]:
from sklearn.tree import DecisionTreeRegressor
# используем исторические данные для прогнозирования цен после 2000 года
data_train = ram_prices[ram_prices.date < 2000]
data_test = ram_prices[ram_prices.date >= 2000]

# прогнозируем цены по датам
X_train = data_train.date[:, np.newaxis]
# мы используем логпреобразование, что получить простую взаимосвязь между данными и откликом
y_train = np.log(data_train.price)

tree = DecisionTreeRegressor().fit(X_train, y_train)
linear_reg = LinearRegression().fit(X_train, y_train)

# прогнозируем по всем данным
X_all = ram_prices.date[:, np.newaxis]

pred_tree = tree.predict(X_all)
pred_lr = linear_reg.predict(X_all)

# экспоненцируем, чтобы обратить логарифмическое преобразование
price_tree = np.exp(pred_tree)
price_lr = np.exp(pred_lr)
```

Рис. 2.32, созданный здесь, сравнивает прогнозы дерева решений и линейной регрессии с реальными.

```
In[67]:
plt.semilogy(data_train.date, data_train.price, label="Обучающие данные")
plt.semilogy(data_test.date, data_test.price, label="Тестовые данные")
plt.semilogy(ram_prices.date, price_tree, label="Прогнозы дерева")
plt.semilogy(ram_prices.date, price_lr, label="Прогнозы линейной регрессии")
plt.legend()
```

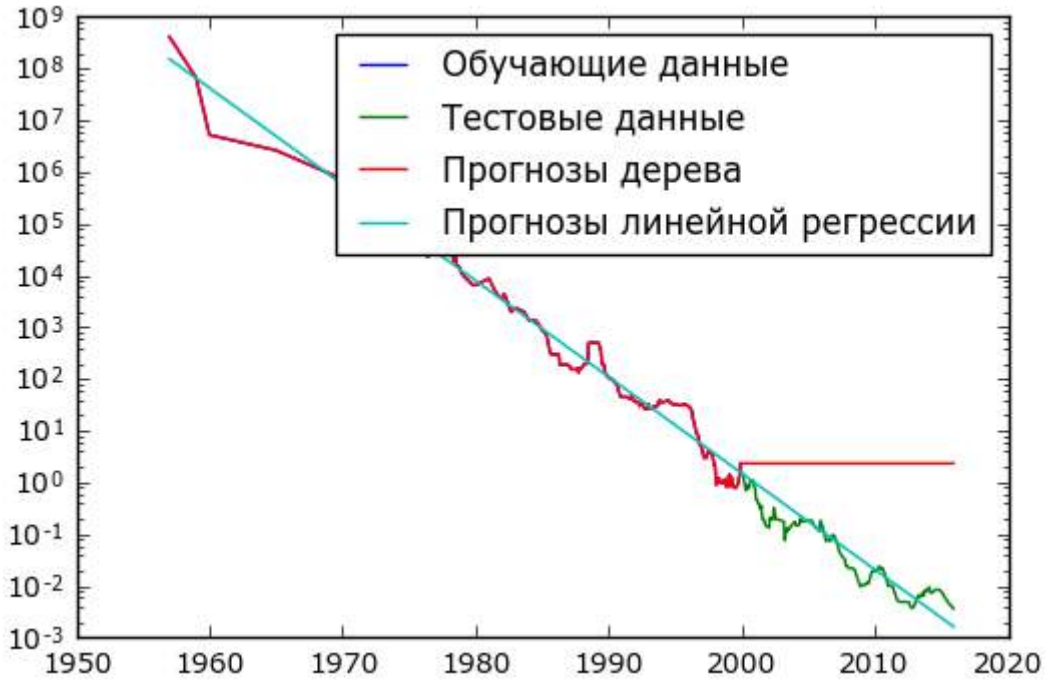

Рис. 2.32 Сравнение прогнозов линейной модели и прогнозов дерева регрессии для набора данных RAM price

Разница между моделями получилась весьма впечатляющая. Линейная модель аппроксимирует данные с помощью уже известной нам прямой линии. Эта линия дает достаточно хороший прогноз для тестовых данных (период после 2000 года), при этом сглаживая некоторые всплески в обучающих и тестовых данных. С другой стороны, модель дерева прекрасно прогнозирует на обучающих данных. Здесь мы не ограничивали сложность дерева, поэтому она полностью запомнила весь набор данных. Однако, как только мы выходим из диапазона значений, известных модели, модель просто продолжает предсказывать последнюю известную точку. Дерево не способно генерировать «новые» ответы, выходящие за пределы значений обучающих данных. Этот недостаток относится ко всем моделям на основе деревьев решений.[17]

Преимущества, недостатки и параметры

Как уже говорилось выше, параметры, которые контролируют сложность модели в деревьях решений – это параметрами предварительной обрезки дерева, которые останавливают построение дерева, прежде чем оно достигнет максимального размера. Обычно, чтобы предотвратить

[17] На самом деле с помощью деревьев решений можно получать очень точные прогнозы (например, предсказать, будет ли цена повышаться или понижаться). Суть этого примера была не в том, чтобы показать, что деревья являются плохой моделью для временных рядов, а в том, чтобы конкретно показать, как деревья делают прогнозы.

переобучение, достаточно выбрать одну из стратегий предварительной обрезки – настроить `max_depth`, `max_leaf_nodes` или `min_samples_leaf`.

По сравнению со многими алгоритмами, обсуждавшимися до сих пор, деревья решений обладают двумя преимуществами: полученная модель может быть легко визуализирована и понята неспециалистами (по крайней мере это верно для небольших деревьев) и деревья не требуют масштабирования данных. Поскольку каждый признак обрабатывается отдельно, а возможные разбиения данных не зависят от масштабирования, алгоритмы деревьев решений не нуждаются в таких процедурах предварительной обработки, как нормализация или стандартизация признаков. Деревья решений хорошо работают, когда у вас есть признаки, измеренные в совершенно разных шкалах, или когда ваши данные представляют смесь бинарных и непрерывных признаков.

Основным недостатком деревьев решений является то, что даже при использовании предварительной обрезки, они склонны к переобучению и имеют низкую обобщающую способность. Поэтому в большинстве случаев, как правило, вместо одиночного дерева решений используются ансамбли деревьев, которые мы обсудим далее.

Ансамбли деревьев решений

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, и 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:

```
In[68]:
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. 2.33):

```
In[69]:
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.

В 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.

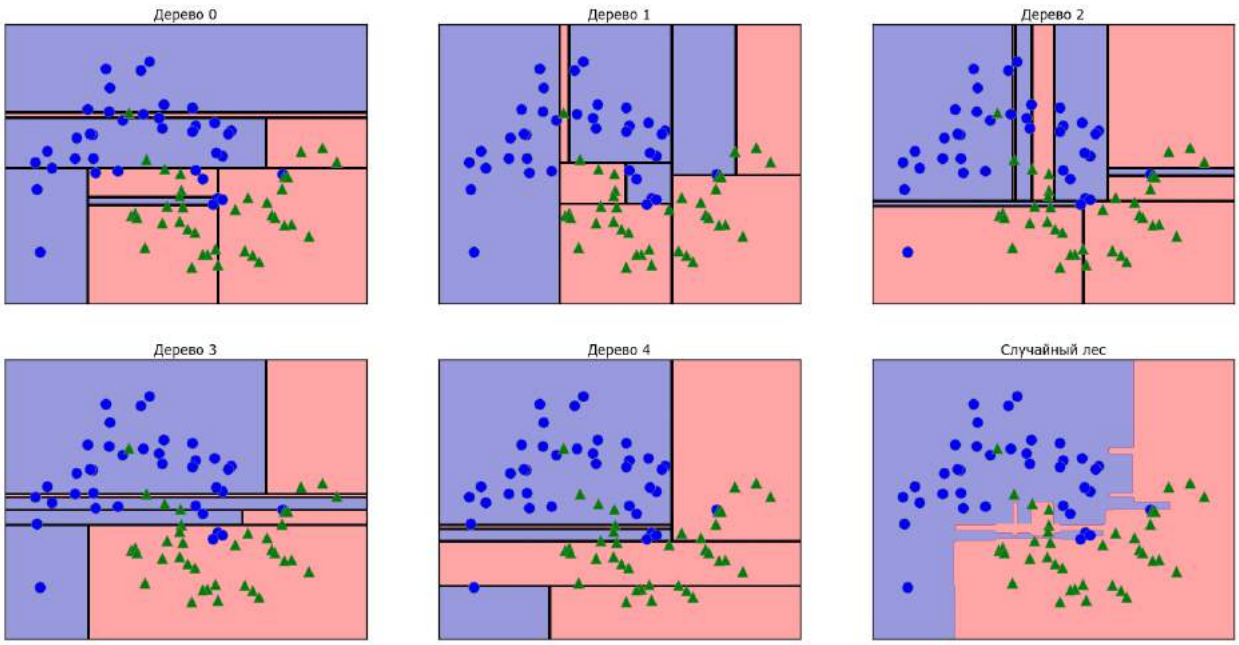

Рис. 2.33 Границы принятия решений, найденные пятью рандомизированными деревьями решений, и граница принятия решений, полученная путем усреднения их спрогнозированных вероятностей

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

```
In[70]:
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)))
```

```
Out[70]:
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. 2.34.

```
In[71]:
```

```
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)
```

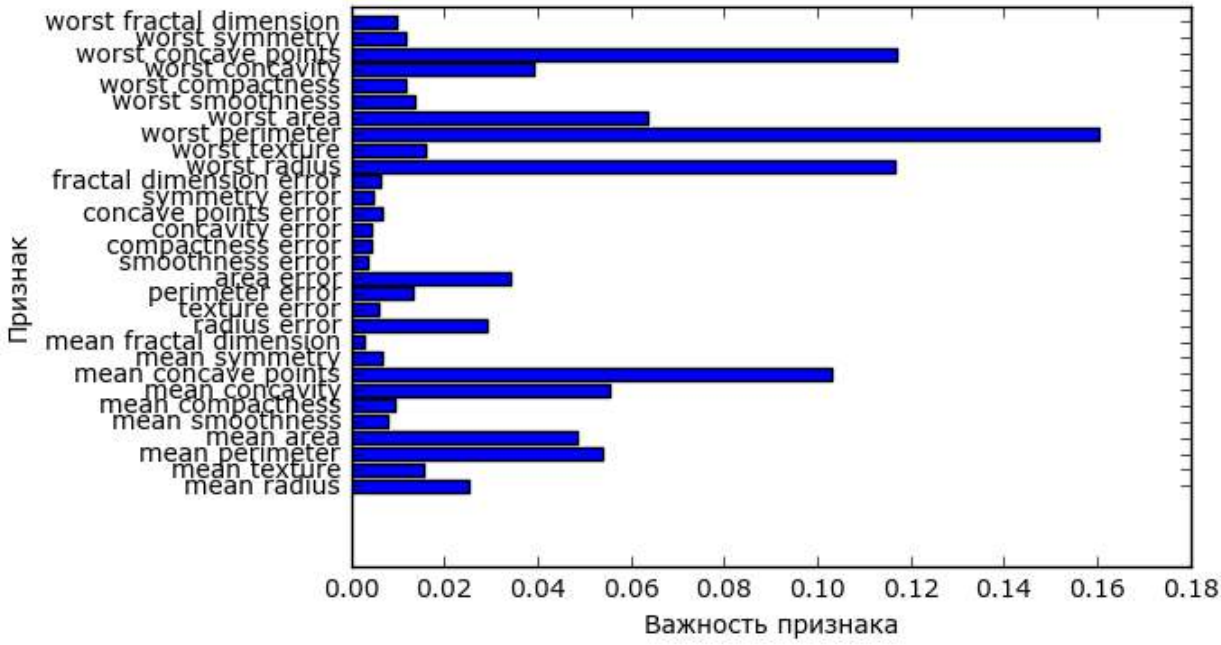

Рис. 2.34 Важности признаков, вычисленные случайным лесом для набора данных Breast Cancer

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 в 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

В 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 регрессии (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:

```
In[72]:
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)))

Out[72]:
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:

```
In[73]:
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)))

Out[73]:
Правильность на обучающем наборе: 0.991
Правильность на тестовом наборе: 0.972

In[74]:
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)))

Out[74]:
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.

И 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. 2.35). Поскольку мы использовали 100 деревьев, вряд ли целесообразно проверять все деревья, даже если все они имеют глубину 1:

```
In[75]:
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)
```

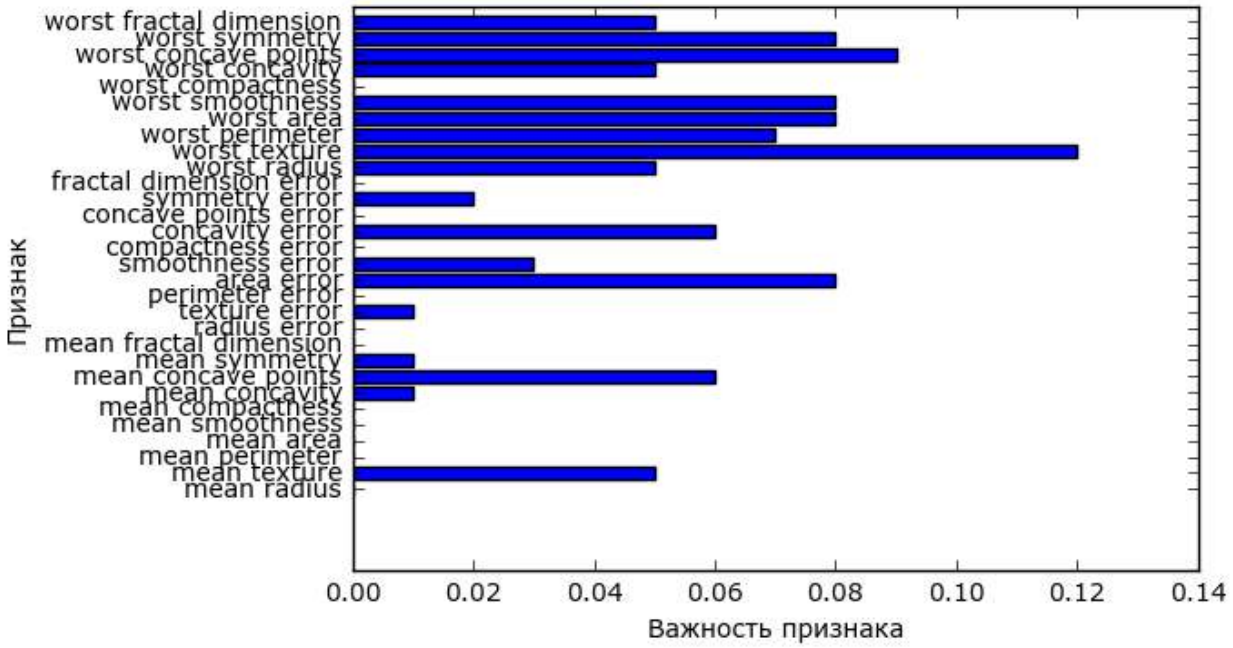

Рис. 2.35 Важности признаков, вычисленные случайным лесом для набора данных Breast Cancer

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` и 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.

Ядерный метод опорных векторов

Следующий тип обучения с учителем, который мы обсудим, – это метод опорных векторов. Мы рассматривали использование линейного метода опорных векторов для задач классификации в разделе «Линейные модели для классификации». Ядерный метод опорных векторов (часто его просто называют SVM) – это расширение метода опорных векторов, оно позволяет получать более сложные модели, которые не сводятся к построению простых гиперплоскостей в пространстве. Несмотря на то что метод опорных векторов можно применять для задач классификации и регрессии, мы ограничимся классификацией, реализованной в `SVC`. Аналогичные принципы применяются в опорных векторах для регрессии и реализованы в `SVR`.

Математический аппарат ядерного метода опорных векторов сложен и выходит за рамки данной книги. Вы можете подробнее прочитать о нем в главе 12 книги Хасти, Тибширани и Фридмана «Элементы статистического обучения». Однако мы попытаемся дать вам некоторое представление об идеях, лежащих в основе этого метода.

Линейные модели и нелинейные признаки

На рис. 2.15 было видно, что в низкоразмерных пространствах линейные модели накладывают весьма жесткие ограничения, поскольку линии и гиперплоскости имеют ограниченную гибкость. Один из способов сделать линейную модель более гибкой – добавить новые признаки, например, добавить взаимодействия или полиномы входных признаков.

Давайте взглянем на синтетический набор данных, который мы использовали в разделе «Важность признаков в деревьях» (см. рис. 2.29):

```
In[76]:
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")
```

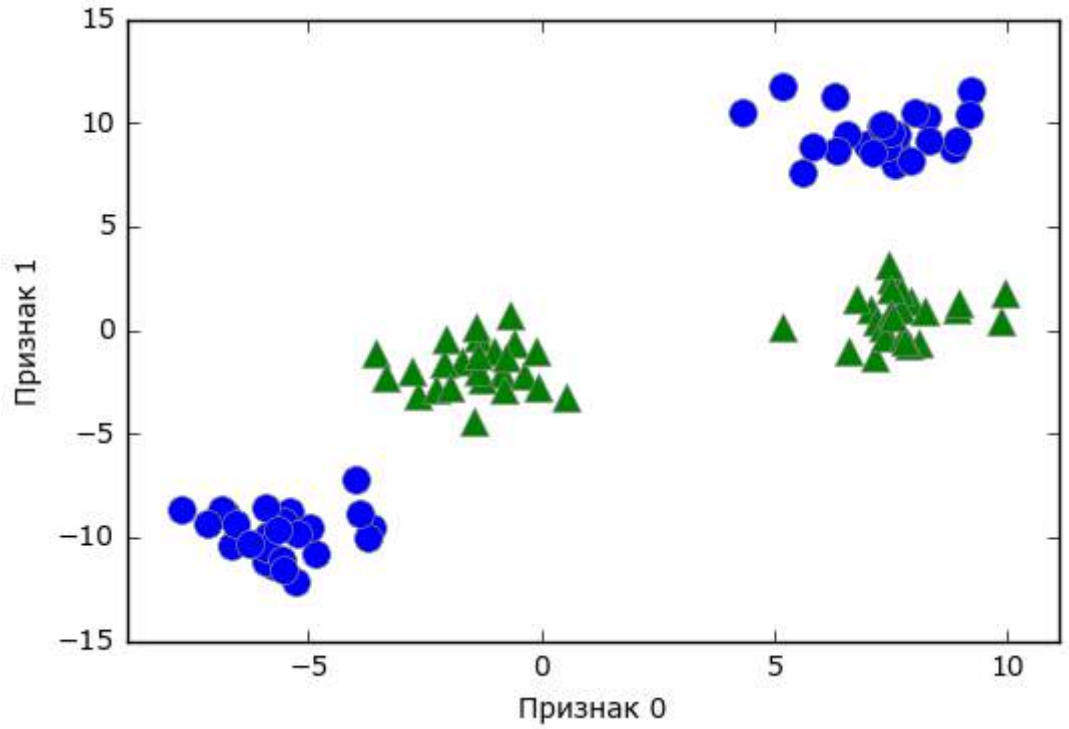

Рис. 2.36 Набор данных с двухклассовой классификацией, в котором классы линейно неразделимы

Линейная модель классификации может отделить точки только с помощью прямой линии и не может дать хорошее качество для этого набора данных (см. рис. 2.37):

```
In[77]:
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")
```

Теперь давайте расширим набор входных признаков, скажем, добавим в качестве нового признака `feature1 ** 2`, квадрат второго признака. Теперь каждую точку данных мы представим не в виде точки двумерного пространства (`feature0`, `feature1`), а виде точки трехмерного пространства (`feature0`, `feature1`, `feature1 ** 2`).[18] Новое пространство признаков показано на рис. 2.38 в виде трехмерной диаграммы рассеяния:

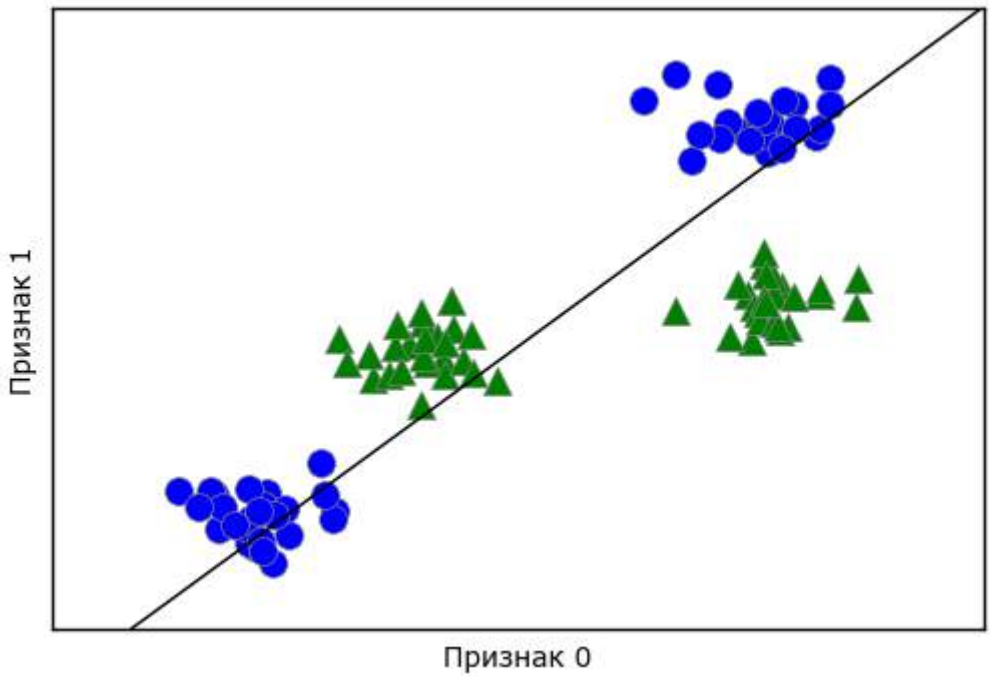

Рис. 2.37 Граница принятия решений, найденная с помощью линейного SVM

```
In[78]:
# добавляем второй признак, возведенный в квадрат
X_new = np.hstack([X, X[:, 1:] ** 2])

from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
# визуализируем в 3D
ax = Axes3D(figure, elev=-152, azim=-26)
# сначала размещаем на графике все точки с y == 0, затем с y == 1
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")
```

[18] Мы добавили этот признак в иллюстративных целях. Этот выбор не является принципиально важным.

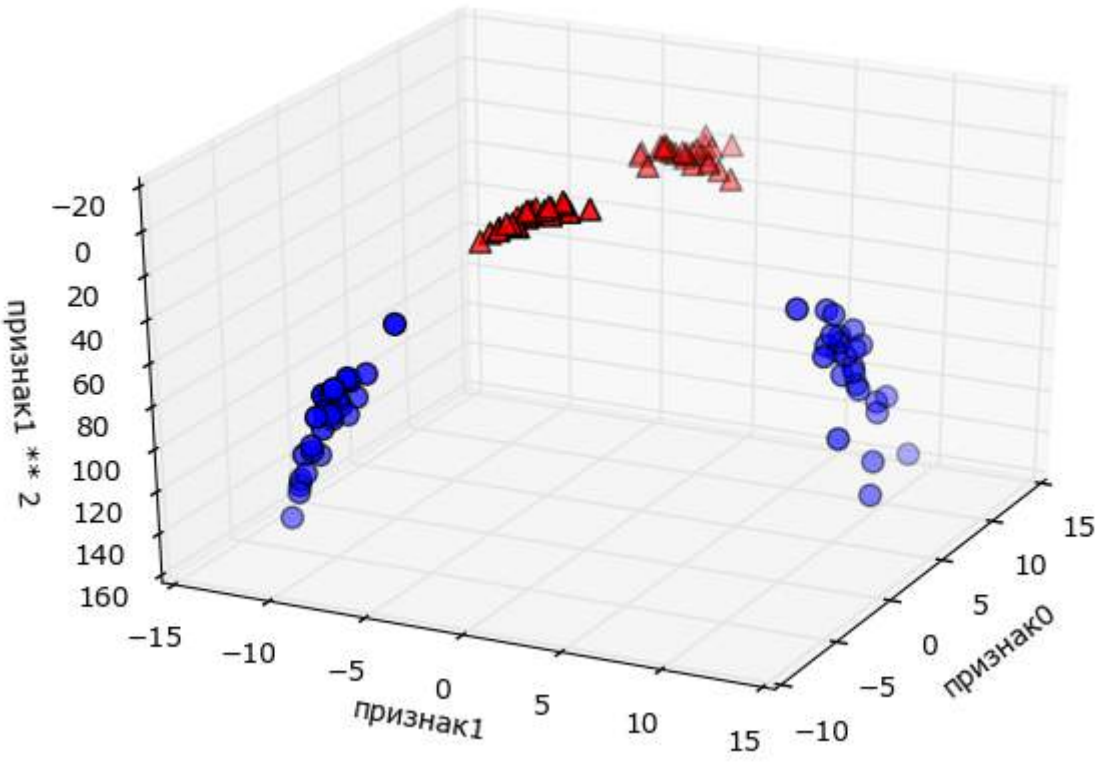

Рис. 2.38 Расширение набора данных, показанного на рис. 2.37, за счет добавления третьего признака, полученного на основе признака 1

В новом представлении данных уже можно отделить два класса друг от друга, используя линейную модель, плоскость в трехмерном пространстве. Мы можем убедиться в этом, подогнав линейную модель к дополненным данным (см. рис. 2.39):

In[79]:

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

# показать границу принятия решений линейной модели
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")
```

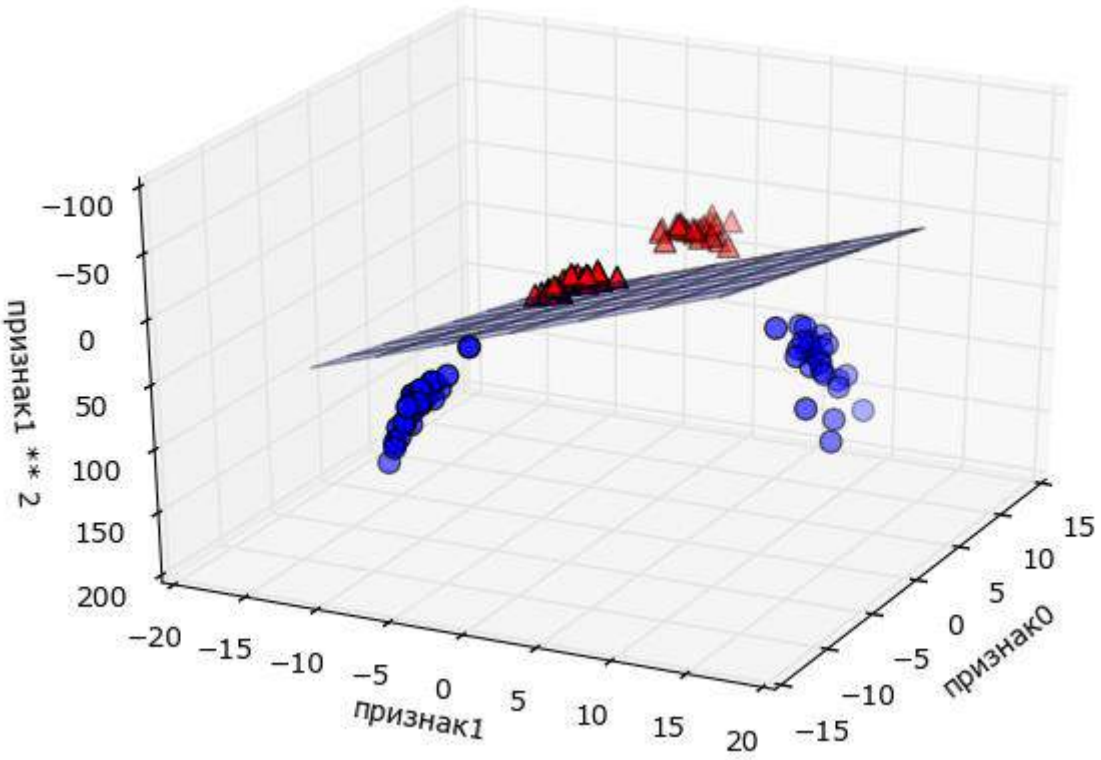

Рис. 2.39 Граница принятия решений, найденная линейным SVM для расширенного трехмерного набора данных

Фактически модель линейного SVM как функция исходных признаков не является больше линейной. Это не линия, а скорее эллипс, как можно увидеть на графике, построенном ниже (рис. 2.40):

In[80]:

```
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")
```

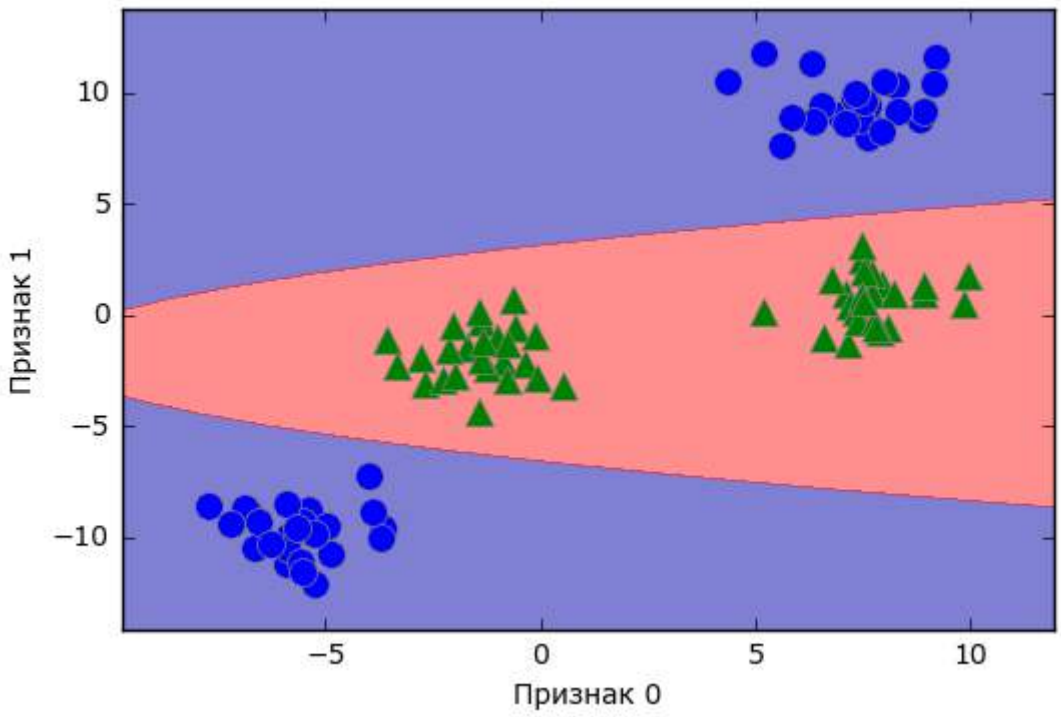

Рис. 2.40 Граница принятия решений для рис. 2.39 как функция от двух исходных признаков

«Ядерный трюк» (kernel trick)

Из вышесказанного можно сделать вывод, что добавление нелинейных признаков может улучшить прогнозную силу линейной модели. Однако часто мы не знаем, какие признаки необходимо добавить, и добавление большего числа признаков (например, рассмотрение всех возможных взаимодействий в 100-мерном пространстве признаков) может очень сильно увеличить стоимость вычислений. К счастью, есть хитрый математический трюк, который позволяет нам обучить классификатор в многомерном пространстве, фактически не прибегая к вычислению нового, возможно, очень высокоразмерного пространства. Этот трюк известен под названием *«ядерный трюк»* (*kernel trick*) и он непосредственно вычисляет евклидовы расстояния (более точно, скалярные произведения точек данных), чтобы получить расширенное пространство признаков без фактического добавления новых признаков.

Существуют два способа поместить данные в высокоразмерное пространство, которые чаще всего используются методом опорных векторов: полиномиальное ядро, которое вычисляет все возможные полиномиальные комбинации исходных признаков до определенной степени, и ядро RBF (радиальная базисная функция), также известное как гауссовское ядро. Гауссовское ядро немного сложнее объяснить, поскольку оно соответствует бесконечному пространству признаков. Объяснить гауссовское ядро можно так: оно рассматривает все

возможные полиномы всех степеней, однако важность признаков снижается с возрастанием степени.[19]

И хотя на практике математические детали ядерного SVM не столь важны и можно легко понять, каким образом SVM с помощью ядра RBF делает прогнозы, мы рассмотрим их в следующем разделе.

Понимание принципов работы SVM

В ходе обучения SVM вычисляет важность каждой точки обучающих данных с точки зрения определения решающей границы между двумя классами. Обычно лишь часть точек обучающего набора важна для определения границы принятия решений: точки, которые лежат на границе между классами. Они называются *опорными векторами* (*support vectors*) и дали свое название машине опорных векторов.

Чтобы получить прогноз для новой точки, измеряется расстояние до каждого опорного вектора. Классификационное решение принимается, исходя из расстояний до опорных векторов, а также важности опорных векторов, полученных в процессе обучения (хранятся в атрибуте `dual_coef_` класса `SVC`).

Расстояние между точками данных измеряется с помощью гауссовского ядра:

$$k_{\text{rbf}}(x_1, x_2) = \exp(-\gamma \| x_1 - x_2 \|^2)$$

Здесь x_1 и x_2 – точки данных, $\| x_1 - x_2 \|$ обозначает евклидово расстояние, а γ (гамма) – параметр, который регулирует ширину гауссовского ядра.

Рис. 2-41 показывает результат обучения машины опорных векторов на двумерном 2-классовом наборе данных. Граница принятия решений показана черным цветом, а опорные векторы – это точки большего размера, обведенные широким контуром. Программный код строит этот график, обучая SVM на наборе данных `forge`:

```
In[81]:
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)
# размещаем на графике опорные векторы
sv = svm.support_vectors_
# метки классов опорных векторов определяются знаком дуальных коэффициентов
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")
```

[19] Это следует из ряда Тейлора для экспоненциальной функции.

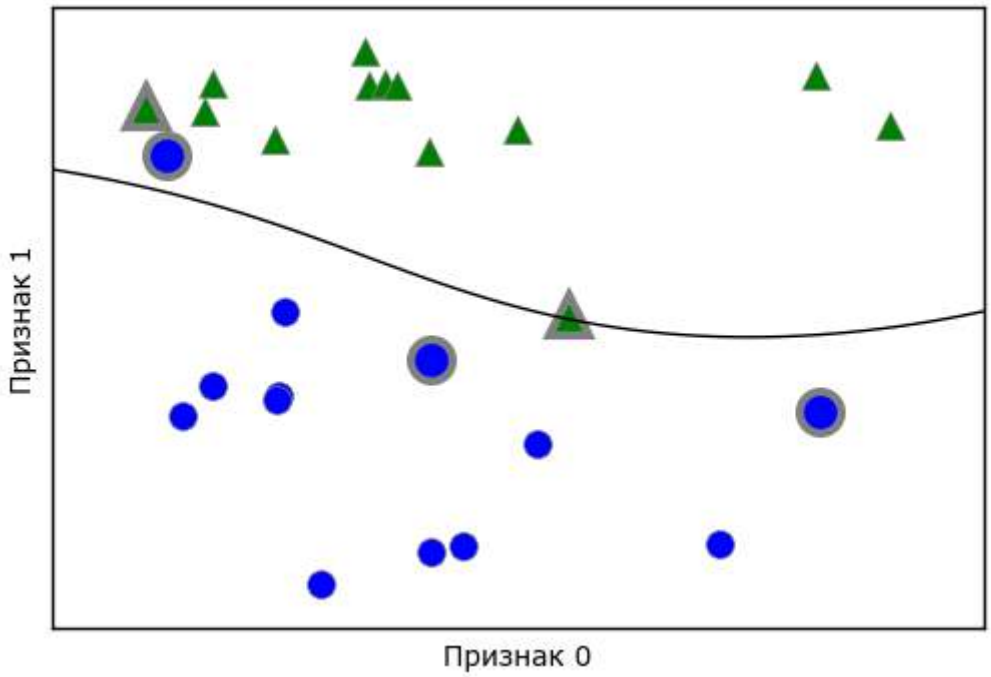

Рис. 2.40 Граница принятия решений и опорные векторы, найденные SVM с помощью ядра RBF

В данном случае SVM дает очень гладкую и нелинейную (непрямую) границу. Здесь мы скорректировали параметр `C` и параметр `gamma`, которые сейчас подробно обсудим.

Настройка параметров SVM

Параметр `gamma` – это параметр формулы, приведенной в предыдущем разделе. Он регулирует ширину гауссовского ядра. Параметр `gamma` задает степень близости расположения точек. Параметр `C` представляет собой параметр регуляризации, аналогичный тому, что использовался в линейных моделях. Он ограничивает важность каждой точки (точнее, ее `dual_coef_`).

Давайте посмотрим, что происходит при изменении этих параметров (рис. 2.42):

```
In[82]:
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))
```

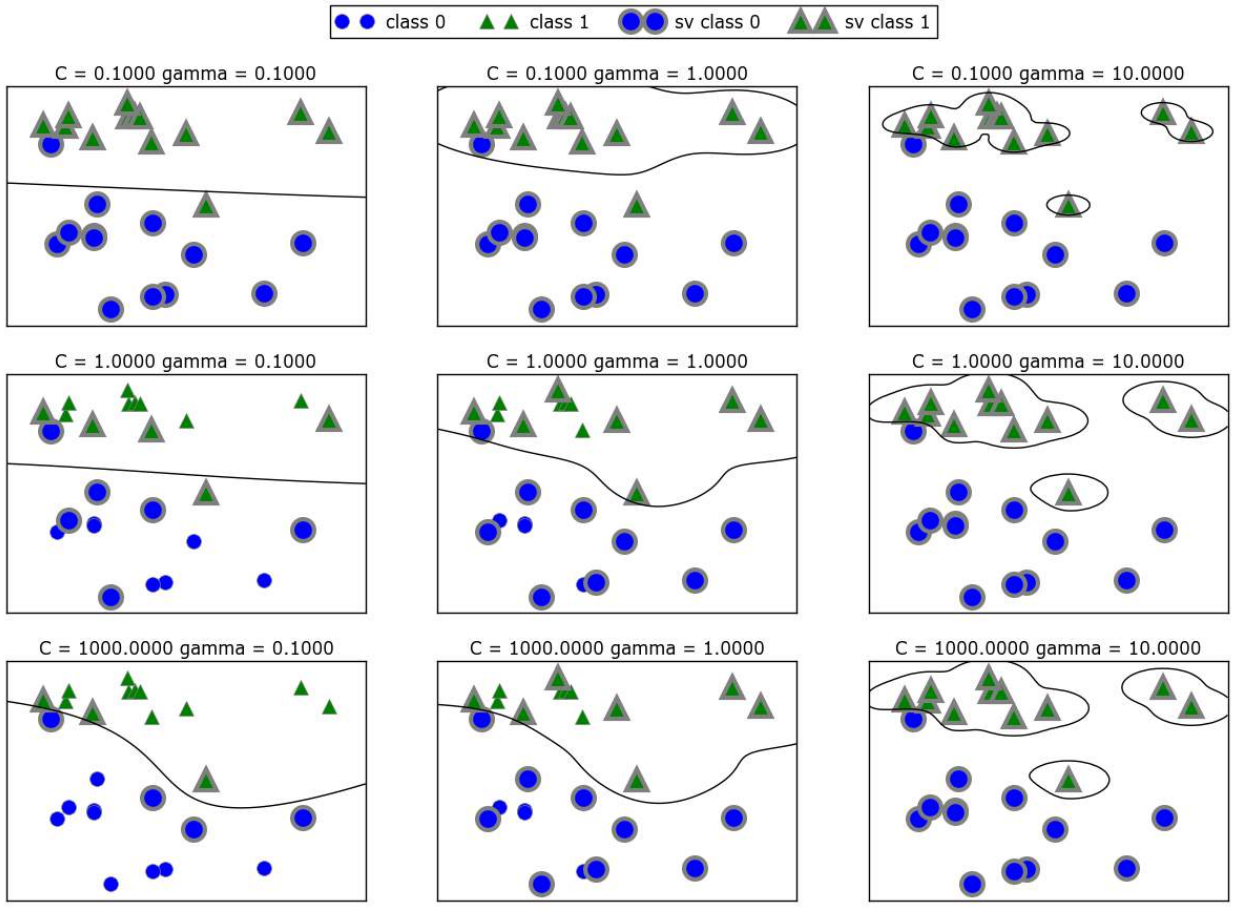

Рис. 2.42 Границы принятия решений и опорные векторы для различных значений параметров C и gamma

Перемещаясь слева направо, мы увеличиваем значение параметра `gamma` с `0.1` до `10`. Небольшое значение `gamma` соответствует большому радиусу гауссовского ядра, это означает, что многие точки рассматриваются как расположенные поблизости. Это приводит к получению очень гладких границ принятия решений, показанных в левой части графика, а границы, которые больше фокусируются на отдельных точках, расположились в правой части графика. Низкое значение `gamma` означает медленное изменение решающей границы, которое дает модель низкой сложности, в то время как высокое значение `gamma` дает более сложную модель.

Перемещаясь сверху вниз, мы увеличиваем параметр `C` с `0.1` до `1000`. Как и в случае с линейными моделями, небольшое значение `C` соответствует модели с весьма жесткими ограничениями, в которой каждая точка данных может иметь лишь очень ограниченное влияние. В левом верхнем углу рис. можно увидеть, что граница принятия решений выглядит как почти линейная, неправильно классифицированные точки почти не влияют на линию. Увеличение значения `C`, как показано в левом нижнем углу, позволяет этим точкам оказывать более сильное влияние

на модель и делает решающую границу изогнутой, позволяя правильно классифицировать данные точки.

Давайте применим SVM с RBF-ядром к набору данных Breast Cancer. По умолчанию используются `C=1` и `gamma=1/n_features`:

```
In[83]:
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)))

Out[83]:
Правильность на обучающем наборе: 1.00
Правильность на тестовом наборе: 0.63
```

Переобучение модели весьма существенно: идеальная правильность на обучающем наборе и лишь 63%-ная правильность на тестовом наборе. Хотя SVM часто дает хорошее качество модели, он очень чувствителен к настройкам параметров и масштабированию данных. В частности, SVM требует, чтобы все признаки были измерены в одном и том же масштабе. Давайте посмотрим на минимальное и максимальное значения каждого признака в log-пространстве (рис. 2.43):

```
In[84]:
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")
```

Исходя из этого графика, мы можем заключить, что признаки в наборе данных Breast Cancer имеют совершенно различные порядки величин. Для ряда моделей (например, для линейных моделей) данный факт может быть в некоторой степени проблемой, однако для ядерного SVM он будет иметь разрушительные последствия. Давайте рассмотрим некоторые способы решения этой проблемы.

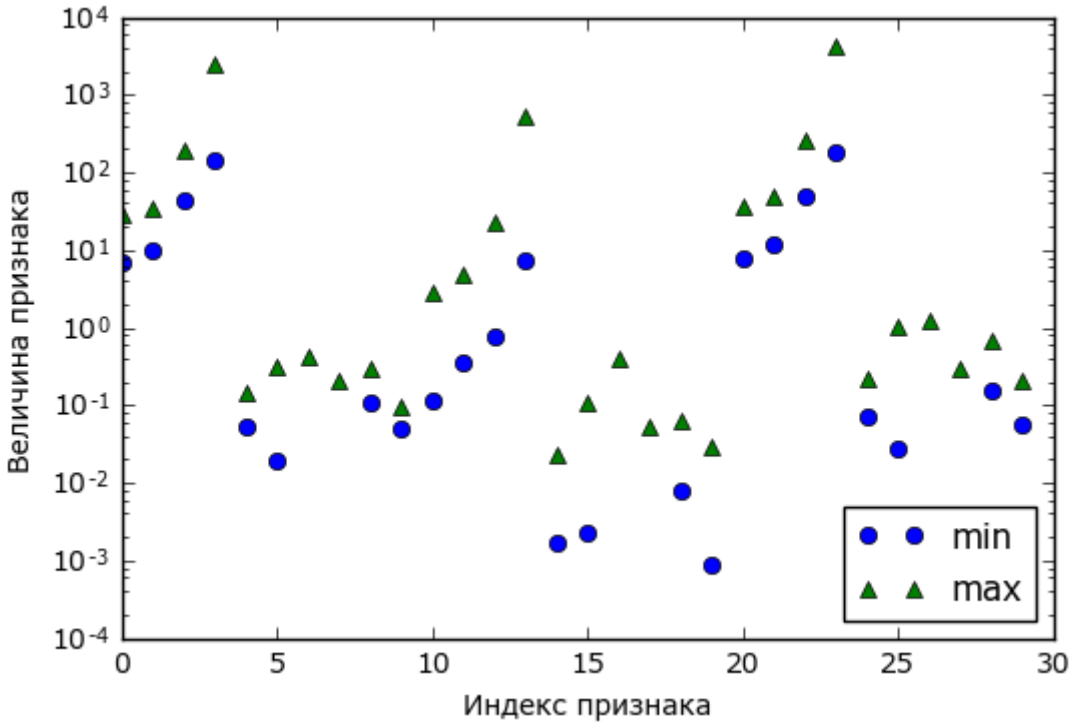

Рис. 2.43 Диапазоны значений признаков для набора данных Breast Cancer (обратите внимание, что ось y имеет логарифмическую шкалу)

Предварительная обработка данных для SVM

Один из способов решения этой проблемы – масштабирование всех признаков таким образом, чтобы все они имели примерно один и тот же масштаб. Общераспространенный метод масштабирования для ядерного SVM заключается в масштабировании данных так, чтобы все признаки принимали значения от 0 до 1. Мы увидим, как это делается с помощью метода предварительной обработки `MinMaxScaler` в главе 3, в которой дадим более подробную информацию. А сейчас давайте сделаем это «вручную»:

```
In[85]:
# вычисляем минимальное значение для каждого признака обучающего набора
min_on_training = X_train.min(axis=0)
# вычисляем ширину диапазона для каждого признака (max - min) обучающего набора
range_on_training = (X_train - min_on_training).max(axis=0)

# вычитаем минимальное значение и затем делим на ширину диапазона
# min=0 и max=1 для каждого признака
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)))
```

```
Out[85]:
Минимальное значение для каждого признака
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
Максимальное значение для каждого признака
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

```
In[86]:
# используем ТО ЖЕ САМОЕ преобразование для тестового набора,
# используя минимум и ширину диапазона из обучающего набора (см. главу 3)
X_test_scaled = (X_test - min_on_training) / range_on_training

In[87]:
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)))

Out[87]:
Правильность на обучающем наборе: 0.948
Правильность на тестовом наборе: 0.951
```

Масштабирование данных привело к огромной разнице! На самом деле наша модель имеет признаки недообучения, когда качество модели на обучающем и тестовом наборе весьма схоже, но все еще далеко от 100%-ной правильности. Исходя из этого, мы можем попытаться увеличить `C` или `gamma`, чтобы подогнать более сложную модель. Например:

```
In[88]:
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)))

Out[88]:
Правильность на обучающем наборе: 0.988
Правильность на тестовом наборе: 0.972
```

В данном случае увеличение `C` позволяет значительно улучшить модель, в результате правильность на тестовом наборе составляет 97.2%.

Премущества, недостатки и параметры

Ядерный метод опорных векторов – это модели, обладающие мощной прогнозной силой и хорошо работающие на различных наборах данных. SVM позволяет строить сложные решающие границы, даже если данные содержат лишь несколько признаков. Они хорошо работают на низкоразмерных и высокоразмерных данных (то есть когда у нас мало или, наоборот, много признаков), однако плохо масштабируются с ростом объема данных. Запуск SVM на наборе данных объемом 10000 наблюдений не составляет проблем, однако работа с наборами данных объемом 100000 наблюдений и больше может стать сложной задачей с точки зрения времени вычислений и использования памяти.

Другим недостатком является то, что SVM требует тщательной предварительной обработки данных и настройки параметров. Именно поэтому сейчас многие специалисты в различных сферах вместо SVM

используют модели на основе дерева, например, случайные леса или градиентный бустинг (который практически не требуют предварительную обработки данных). Кроме того, модели SVM трудноисследуемы, тяжело понять, почему был сделан именно такой прогноз и довольно сложно объяснить модель неспециалисту.

Однако все же стоит попробовать SVM, особенно в тех случаях, когда все ваши признаки имеют одинаковые единицы измерения (например, все признаки являются интенсивностями пикселей) и измерены в одном и том же масштабе.

Важными параметрами ядерного SVM являются параметр регуляризации `C`, тип ядра, а также параметры, определяемые ядром. Хотя мы в основном сосредоточились на ядре RBF, в `scikit-learn` доступны и другие типы ядер. Ядро RBF имеет лишь один параметр `gamma`, который является обратной величиной ширины гауссовского ядра. `gamma` и `C` регулируют сложность модели, более высокие значения этих параметров дают более сложную модель. Таким образом, оптимальные настройки обоих параметров, как правило, сильно взаимосвязаны между собой и поэтому `C` и `gamma` должны быть отрегулированы вместе.

Нейронные сети (глубокое обучение)

Семейство алгоритмов, известное как нейронные сети, недавно пережило свое возрождение под названием «глубокое обучение». Несмотря на то что глубокое обучение сулит большие перспективы в различных сферах применения машинного обучения, алгоритмы глубоко обучения, как правило, жестко привязаны к конкретным случаям использования. В данном разделе мы рассмотрим лишь некоторые относительно простые методы, а именно многослойные персептроны для классификации и регрессии, которые могут служить отправной точкой в изучении более сложных методов машинного обучения. Многослойные персептроны (MLP) также называют *простыми* (*vanilla*) нейронными сетями прямого распространения, а иногда и просто нейронными сетями.

Модель нейронной сети

MLP можно рассматривать как обобщение линейных моделей, которое прежде чем прийти к решению выполняет несколько этапов обработки данных.

Вспомним, что в линейной регрессии прогноз получают с помощью следующей формулы:

$$\hat{y} = w[0]*x[0]+w[1]*x[1]+...+w[p]*x[p]+b$$

Говоря простым языком, $\hat{y}$ – это взвешенная сумма входных признаков $x[0]$ ... $x[p]$. Входные признаки взвешены по вычисленным в ходе обучения коэффициентам $w[0]$ ... $w[p]$. Мы может представить их графически, как показано на рис. 2-44:

```
In[89]:
display(mglearn.plots.plot_logistic_regression_graph())
```

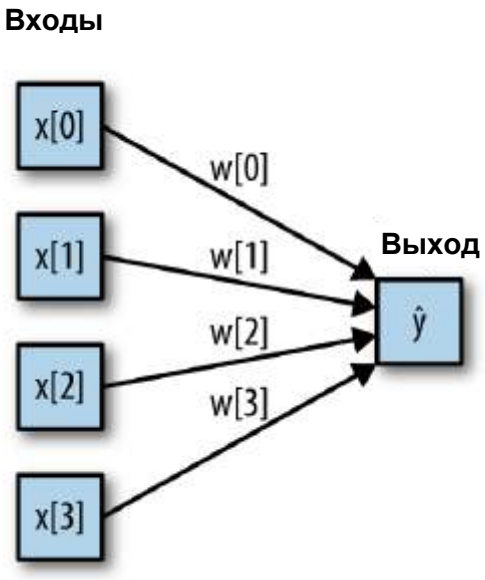

Рис. 2.44 Визуализация логистической регрессии, в которой входные признаки и прогнозы показаны в виде узлов, а коэффициенты – в виде соединений между узлами

Здесь каждый узел, показанный слева, представляет собой входной признак, соединительные линии – коэффициенты, а узел справа – это выход, который является взвешенной суммой входов.

В MLP процесс вычисления взвешенных сумм повторяется несколько раз. Сначала вычисляются *скрытые элементы* (*hidden units*), которые представляют собой промежуточный этап обработки. Они вновь объединяются с помощью взвешенных сумм для получения конечного результата (рис. 2.45):

```
In[90]:
display(mglearn.plots.plot_single_hidden_layer_graph())
```

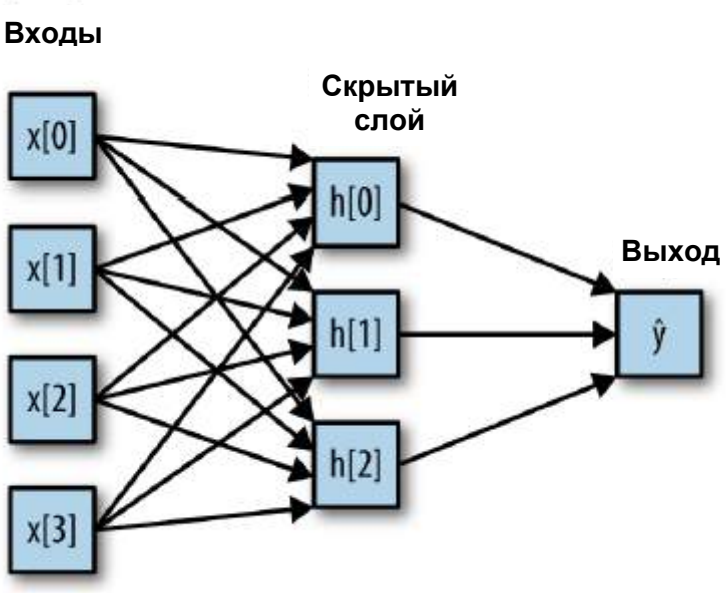

Рис. 2.45 Иллюстрация многослойного персептрона с одним скрытым слоем

У этой модели гораздо больше вычисляемых коэффициентов (также называемых весами): коэффициент между каждым входом и каждым скрытым элементом (которые образуют *скрытый слой* или *hidden layer*) и коэффициент между каждым элементом скрытого слоя и выходом.
С математической точки зрения вычисление серии взвешенных сумм – это то же самое, что вычисление лишь одной взвешенной суммы, таким образом, чтобы эта модель обладала более мощной прогнозной силой, чем линейная модель, нам нужен один дополнительный трюк. Поясним его на примере нейронной сети с одним скрытым слоем. Входной слой просто передает входы скрытому слою сети, либо без преобразования, либо выполнив сначала стандартизацию входов. Затем происходит вычисление взвешенной суммы входов для каждого элемента скрытого слоя, к ней применяется функция активации – обычно используются нелинейные функции *выпрямленный линейный элемент* (*rectified linear unit* или *relu*) или *гиперболический тангенс* (*hyperbolic tangent* или *tanh*). В итоге получаем выходы нейронов скрытого слоя. Эти промежуточные выходы могут считаться нелинейными преобразованиями и комбинациями первоначальных входов. Они становятся входами выходного слоя. Снова вычисляем взвешенную сумму входов, применяем функцию активации и получаем итоговые значения целевой переменной. Функции активации relu и tanh показаны на рис. 2.46. Relu отсекает значения ниже нуля, в то время как tanh принимает значения от –1 до 1 (соответственно для минимального и максимального значений входов). Любая из этих двух нелинейных функций позволяет нейронной сети в

отличие от линейной модели вычислять гораздо более сложные зависимости.

```
In[91]:
line = np.linspace(-3, 3, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.legend(loc="best")
plt.xlabel("x")
plt.ylabel("relu(x), tanh(x)")
```

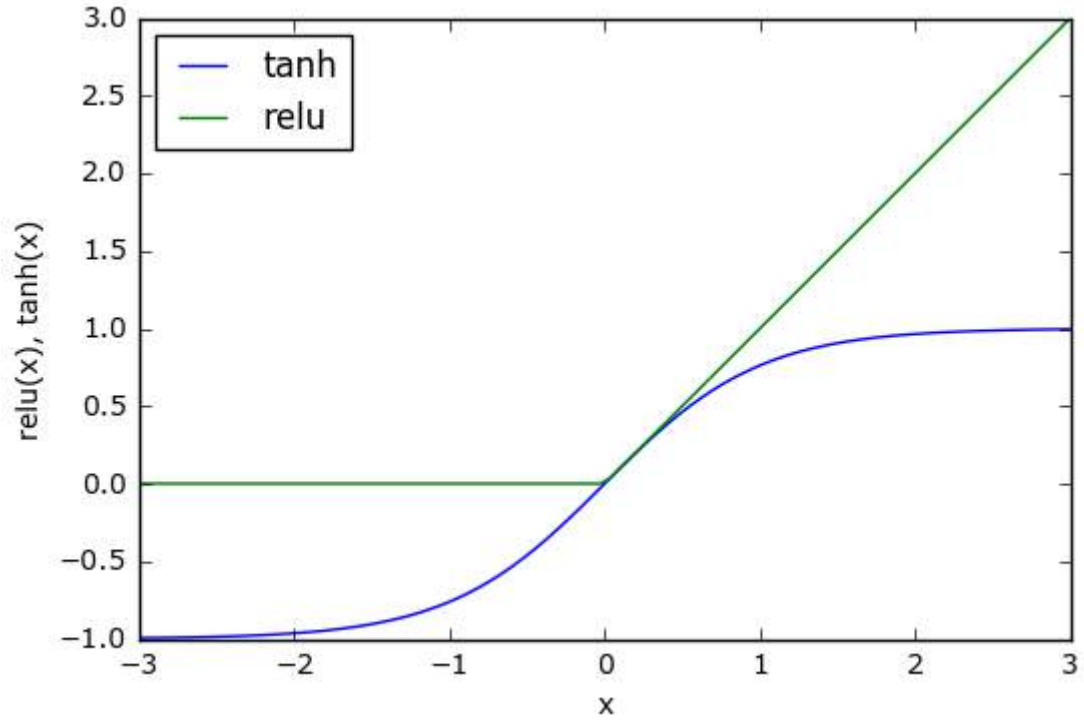

Рис. 2.46 Функция активации гиперболический тангенс и функция активации выпрямленного линейного элемента

Для небольшой нейронной сети, изображенной на рис. 2.45, полная формула вычисления $\hat{y}$ в случае регрессии будет выглядеть так (при использовании tanh):

$$h[0] = \tanh(w[0,0]*x[0] + w[1,0]*x[1] + w[2,0]*x[2] + w[3,0]*x[3])$$
$$h[1] = \tanh(w[0,0]*x[0] + w[1,0]*x[1] + w[2,0]*x[2] + w[3,0]*x[3])$$
$$h[2] = \tanh(w[0,0]*x[0] + w[1,0]*x[1] + w[2,0]*x[2] + w[3,0]*x[3])$$
$$\hat{y} = v[0]*h[0] + v[1]*h[1] + v[2]*h[2]$$

Здесь w – веса между входом x и скрытом слоем h, а v – весовые коэффициенты между скрытым слоем h и выходом $\hat{y}$. Веса v и w вычисляются по данным, x являются входными признаками, $\hat{y}$ - вычисленный выход, а h – промежуточные вычисления. Важный параметр, который должен задать пользователь – количество узлов в скрытом слое. Его значение может быть маленьким, например, 10 для очень маленьких или простых наборов данных или же большим,

например, 10000 для очень сложных данных. Кроме того, можно добавить дополнительные скрытые слои, как показано на рис. 2.47:

```
In[92]:
mglearn.plots.plot_two_hidden_layer_graph()
```

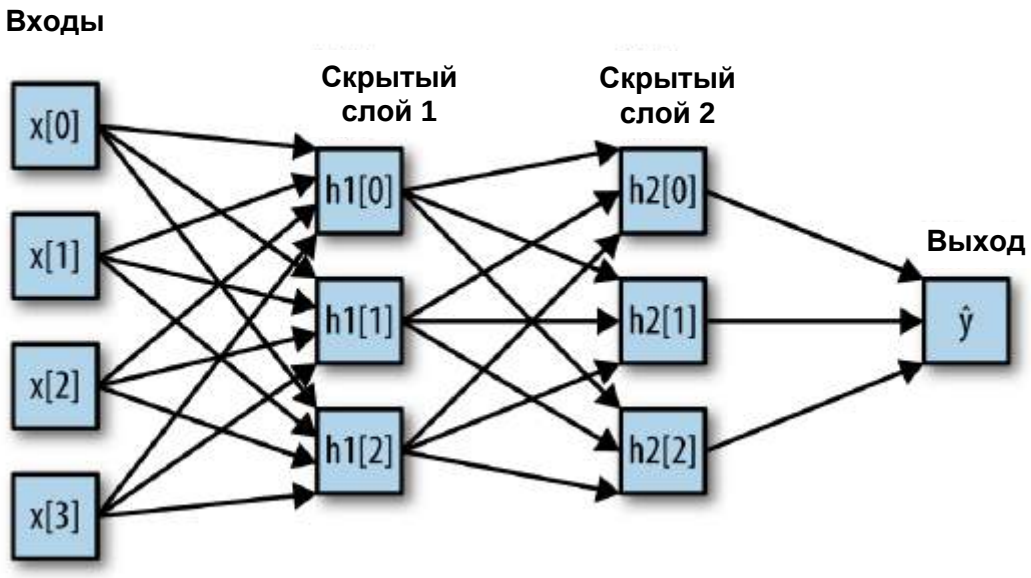

Рис. 2.47 Многослойный персептрон с двумя скрытыми слоями

Построение больших нейронных сетей, состоящих из множества слоев вычислений, вдохновило специалистов ввести в обиход термин «глубокое обучение» («deep learning»).

Настройка нейронных сетей

Давайте посмотрим, как работает MLP, применив `MLPClassifier` к набору данных `two_moons`, который мы использовали ранее в этой главе. Результаты показаны на рис. 2.48:

```
In[93]:
from sklearn.neural_network import MLPClassifier
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)

mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

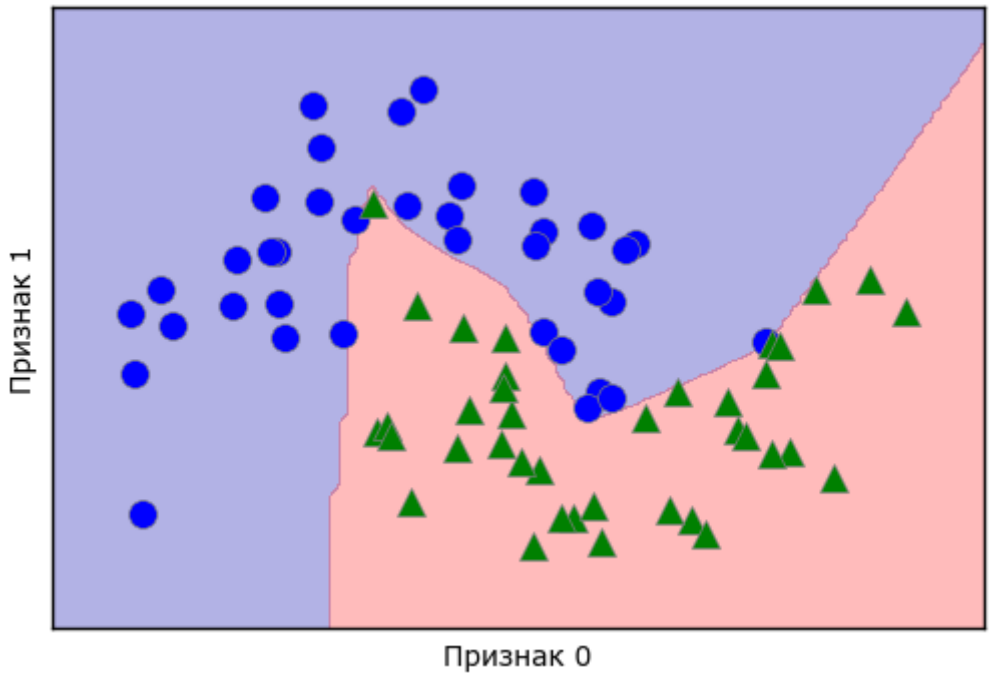

Рис. 2.48 Граница принятия решений, построенная нейронной сетью со 100 скрытыми элементами на наборе данных two_moons

Как видно из рис., нейронная сеть построила нелинейную, но относительно гладкую границу принятия решений. Мы использовали `solver='lbfgs'`, который рассмотрим позднее.

По умолчанию MLP использует 100 скрытых узлов, что довольно много для этого небольшого набора данных. Мы можем уменьшить число (что снизит сложность модели) и снова получить хороший результат (рис. 2.49):

```
In[94]:
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

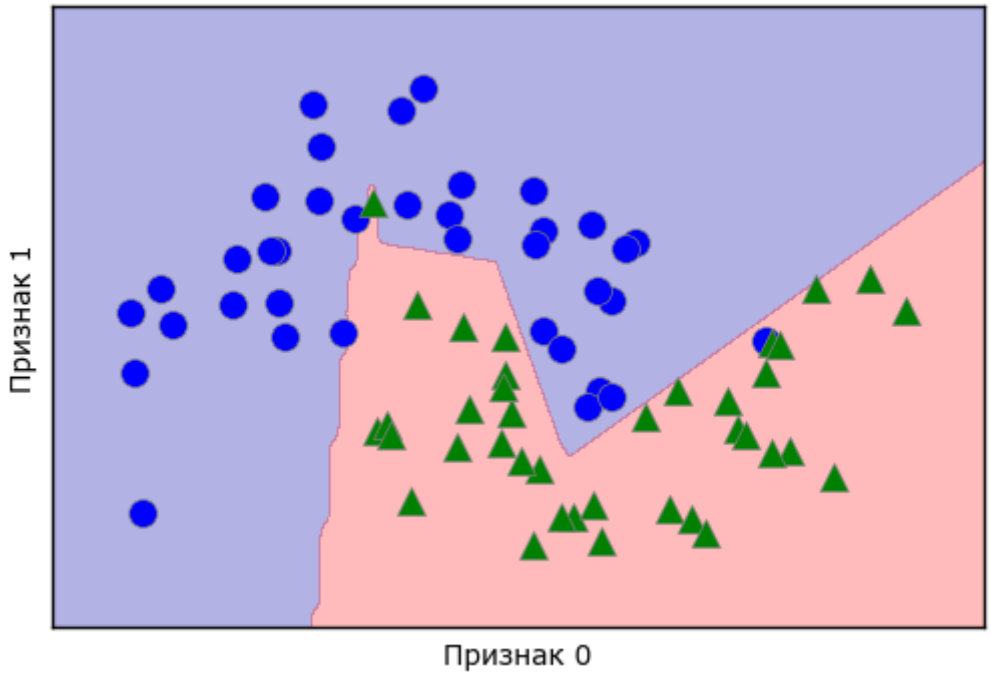

Рис. 2.49 Граница принятия решений, построенная нейронной сетью с 10 скрытыми элементами на наборе данных two_moons

При использовании лишь 10 скрытых элементов граница принятия решений становится более неровной. По умолчанию используется функция активации relu, показанная на рис. 2.46. При использовании одного скрытого слоя решающая функция будет состоять из 10 прямолинейных отрезков. Если необходимо получить более гладкую решающую границу, можно добавить большее количество скрытых элементов (как показано на рис. 2.49), добавить второй скрытый слой (рис. 2.50), или использовать функцию активации tanh (рис. 2.51):

In[95]:
```
# использование двух скрытых слоев по 10 элементов в каждом
mlp = MLPClassifier(solver='lbfgs', random_state=0,
                    hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

In[96]:
```
# использование двух скрытых слоев по 10 элементов в каждом, на этот раз с функцией tanh
mlp = MLPClassifier(solver='lbfgs', activation='tanh',
                    random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

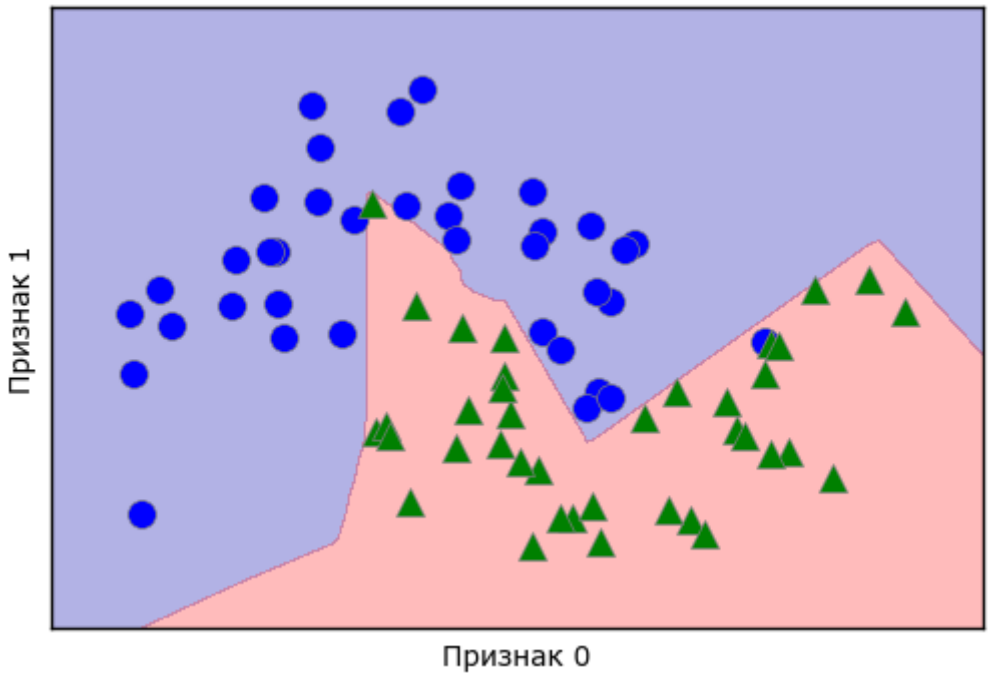

Рис. 2.50 Граница принятия решений, построенная нейронной сетью с 2 скрытыми слоями по 10 элементов в каждом и функцией активации выпрямленный линейный элемент

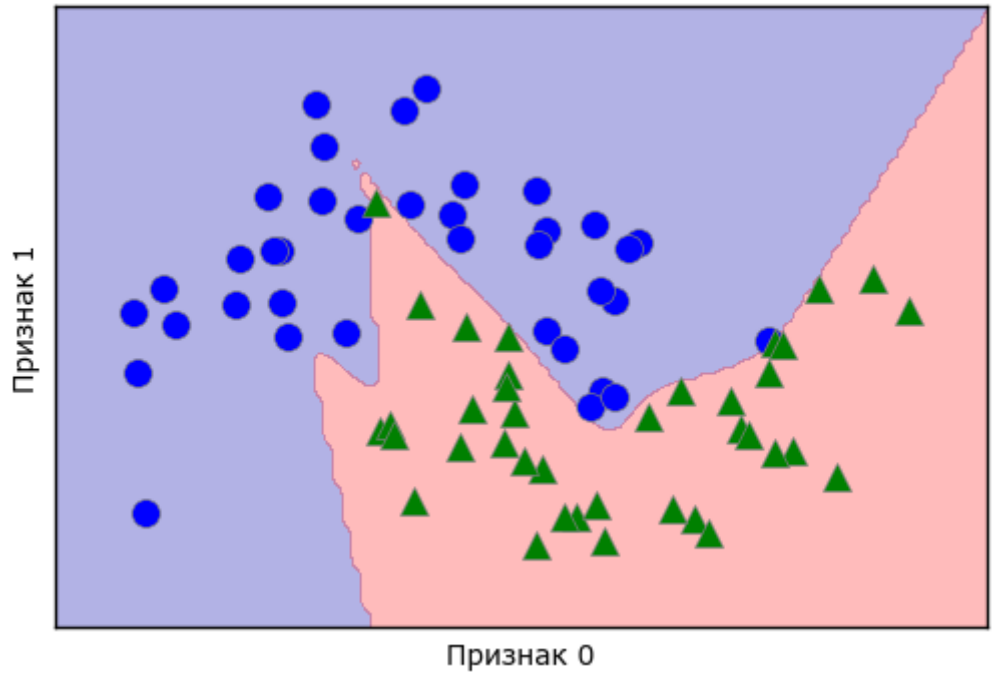

Рис. 2.51 Граница принятия решений, построенная нейронной сетью с 2 скрытыми слоями по 10 элементов в каждом и функцией активации гиперболический тангенс

И, наконец, мы можем дополнительно настроить сложность нейронной сети с помощью `l2` штрафа, чтобы сжать весовые коэффициенты до близких к нулю значений, как мы это делали в гребневой регрессии и линейных классификаторов. В `MLPClassifier` за это отвечает параметр `alpha` (как и в моделях линейной регрессии), и по

умолчанию установлено очень низкое значение (небольшая регуляризация). На рис. 2.52 показаны результаты применения к набору данных `two_moons` различных значений `alpha` с использованием двух скрытых слоев с 10 или 100 элементами в каждом:

```
In[97]:
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for axx, n_hidden_nodes in zip(axes, [10, 100]):
    for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):
        mlp = MLPClassifier(solver='lbfgs', random_state=0,
                            hidden_layer_sizes=[n_hidden_nodes, n_hidden_nodes],
                            alpha=alpha)
        mlp.fit(X_train, y_train)
        mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
        mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
        ax.set_title("n_hidden=[{}, {}]\nalpha={:.4f}".format(
                     n_hidden_nodes, n_hidden_nodes, alpha))
```

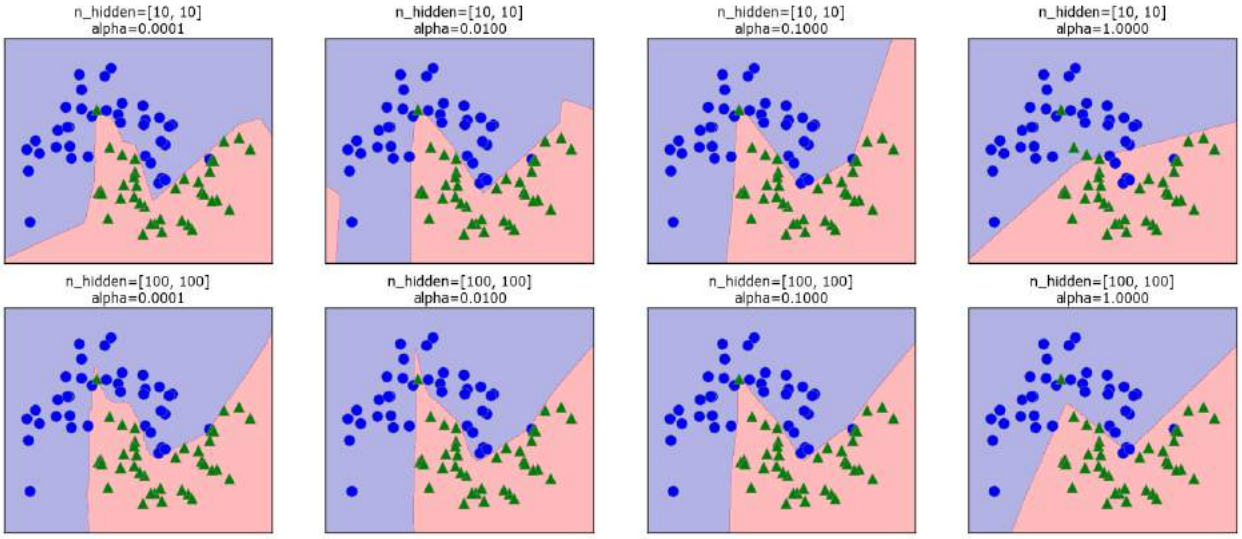

Рис. 2.52 Границы принятия решений для различного количества скрытых элементов и разных значений параметра alpha

Как вы, наверное, уже поняли, существуют различные способы регулировать сложность нейронной сети: количество скрытых слоев, количество элементов в каждом скрытом слое и регуляризация (`alpha`). На самом деле их гораздо больше, но мы не будем здесь вдаваться в подробности.

Важным свойством нейронных сетей является то, что их веса задаются случайным образом перед началом обучения и случайная инициализация влияет на процесс обучения модели. Это означает, что даже при использовании одних и тех же параметров мы можем получить очень разные модели, задавая разные стартовые значения генератора псевдослучайных чисел. При условии, что сеть имеет большой размер и ее сложность настроена правильно, данный факт не должен сильно влиять на правильность, однако о нем стоит помнить (особенно при работе с небольшими сетями). На рис. 2-53 представлены графики

нескольких моделей, обученных с использованием тех же самых значений параметров:

```
In[98]:
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(solver='lbfgs', random_state=i,
                        hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
```

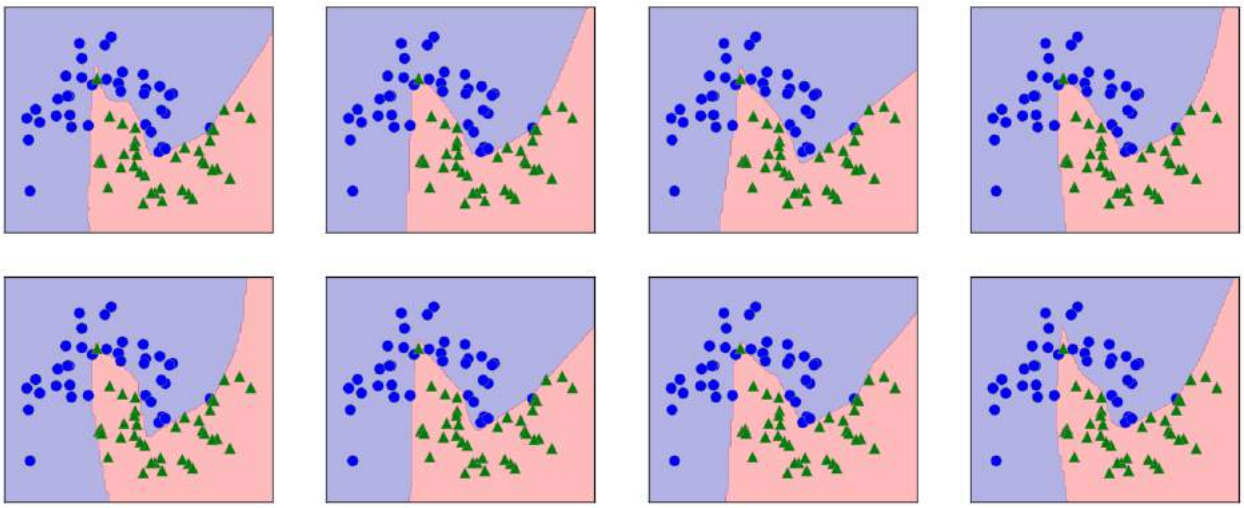

Рис. 2.53 Границы принятия решений, полученные с использованием тех же самых параметров, но разных стартовых значений

Чтобы лучше понять, как нейронная сеть работает на реальных данных, давайте применим `MLPClassifier` к набору данных Breast Cancer. Мы начнем с параметров по умолчанию:

```
In[99]:
print("Максимальные значения характеристик:\n{}".format(cancer.data.max(axis=0)))

Out[99]:
Максимальные значения характеристик:
[   28.110    39.280    188.500    2501.000    0.163    0.345     0.427
     0.201     0.304      0.097       2.873    4.885   21.980   542.200
     0.031     0.135      0.396       0.053    0.079    0.030    36.040
    49.540   251.200   4254.000       0.223    1.058    1.252     0.291
     0.664     0.207]

In[100]:
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.2f}".format(mlp.score(X_train, y_train)))
print("Правильности на тестовом наборе: {:.2f}".format(mlp.score(X_test, y_test)))

Out[100]:
Правильность на обучающем наборе: 0.92
Правильность на тестовом наборе: 0.90
```

MLP демонстрирует довольно неплохую правильность, однако не столь хорошую, если сравнивать с другими моделями. Как и в предыдущем примере с SVC, это, вероятно, обусловлено масштабом данных. Нейронные сети также требуют того, чтобы все входные признаки были измерены в одном и том же масштабе, в идеале они должны иметь среднее 0 и дисперсию 1. Мы должны отмасштабировать наши данные так, чтобы они отвечали этим требованиям. Опять же, мы будем делать это вручную, однако в главе 3 расскажем, как это делать автоматически с помощью `StandardScaler`.

```
In[101]:
# вычисляем среднее для каждого признака обучающего набора
mean_on_train = X_train.mean(axis=0)
# вычисляем стандартное отклонение для каждого признака обучающего набора
std_on_train = X_train.std(axis=0)

# вычитаем среднее и затем умножаем на обратную величину стандартного отклонения
# mean=0 и std=1
X_train_scaled = (X_train - mean_on_train) / std_on_train
# используем ТО ЖЕ САМОЕ преобразование (используем среднее и стандартное отклонение
# обучающего набора) для тестового набора
X_test_scaled = (X_test - mean_on_train) / std_on_train

mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(mlp.score(X_test_scaled, y_test)))

Out[101]:
Правильность на обучающем наборе: 0.991
Правильность на тестовом наборе: 0.965

ConvergenceWarning:
    Stochastic Optimizer: Maximum iterations reached and the optimization
    hasn't converged yet.
```

После масштабирования результаты стали намного лучше и теперь уже вполне могут конкурировать с результатами остальных моделей. Впрочем, мы получили предупреждение о том, что достигнуто максимальное число итераций. Оно является неотъемлемой частью алгоритма `adam` и сообщает нам о том, что мы должны увеличить число итераций:

```
In[102]:
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(mlp.score(X_test_scaled, y_test)))

Out[102]:
Правильность на обучающем наборе: 0.995
Правильность на тестовом наборе: 0.965
```

Увеличение количества итераций повысило правильность лишь на обучающем наборе. Тем не менее модель имеет достаточно высокое значение правильности. Поскольку существует определенный разрыв между правильностью на обучающем наборе и правильностью на тестовом наборе, мы можем попытаться уменьшить сложность модели, чтобы улучшить обобщающую способность. В данном случае мы увеличим параметр `alpha` (довольно сильно с `0.0001` до `1`), чтобы применить к весам более строгую регуляризацию:

```
In[103]:
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(mlp.score(X_test_scaled, y_test)))

Out[103]:
Правильность на обучающем наборе: 0.988
Правильность на тестовом наборе: 0.972
```

Это дает правильность, сопоставимую с правильностью лучших моделей.[20]

Несмотря на то что анализ нейронной сети возможен, он, как правило, гораздо сложнее анализа линейной модели или модели на основе дерева. Один из способов анализа нейронной сети заключается в том, чтобы исследовать веса модели. Образец такого анализа вы можете увидеть в галерее примеров `scikit-learn`. Применительно к набору данных Breast Cancer такой анализ может быть немного сложен. Следующий график (рис. 2.54) показывает весовые коэффициенты, которые были вычислены при подключении входного слоя к первому скрытому слою. Строки в этом графике соответствуют 30 входным признакам, а столбцы – 100 скрытым элементам. Светлые цвета соответствуют высоким положительным значениям, в то время как темные цвета соответствуют отрицательным значениям:

```
In[104]:
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Столбцы матрицы весов")
plt.ylabel("Входная характеристика")
plt.colorbar()
```

[20] Здесь вы могли заметить, что большинство моделей, давших наилучший прогноз, достигают одинаковой правильности на тестовом наборе 0.972. Это означает, что все эти модели дают одинаковое количество ошибочных прогнозов, равное четырем. Если сравнить полученные прогнозы с фактическими, вы увидите, что все эти модели неправильно прогнозируют одни и те же наблюдения. Это может быть обусловлено очень маленьким размером набора данных или же тем, что эти наблюдения сильно отличаются от остальных.

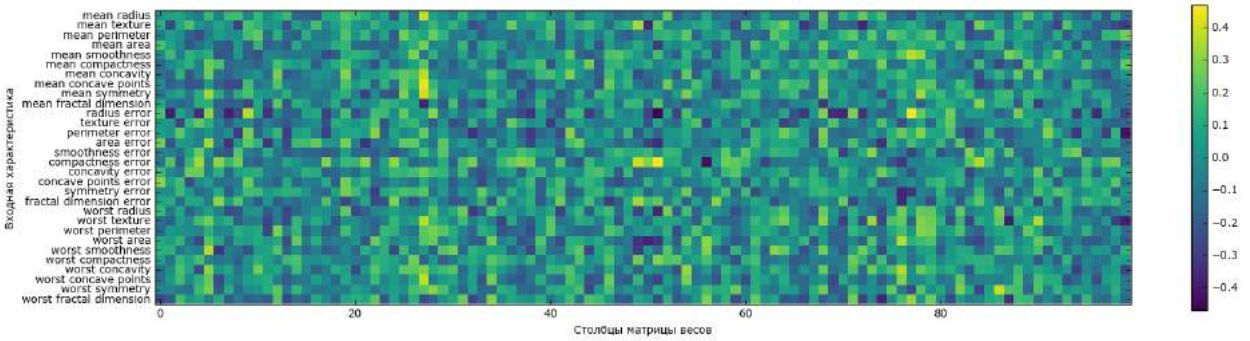

Рис. 2.54 Теплокарта для весов первого слоя нейронной сети, обученной на наборе данных Breast Cancer

Один из возможных выводов, который мы можем сделать, заключается в том, что признаки с небольшими весами скрытых элементов «менее важны» в модели. Мы можем увидеть, что «mean smoothness» и «mean compactness» наряду с признаками, расположенными между «smothness error» и «fractal dimension error», имеют относительно низкие веса по сравнению с другими признаками. Это может означать, что эти признаки являются менее важными или, возможно, мы не преобразовали их таким способом, чтобы их могла использовать нейронная сеть.

Кроме того, мы можем визуализировать веса, соединяющие скрытый слой с выходным слоем, но их еще труднее интерпретировать.

Несмотря на то что для наболее распространенных архитектур нейронных сетей `MLPClassifier` и `MLPRegressor` предлагают легкий в использовании интерфейс, они представляют лишь небольшой набор возможных средств, позволяющих строить нейронные сети. Если вас интересует работа с более гибкими или более масштабными моделями, мы рекомендуем вам не ограничиваться возможностями библиотеки `scikit-learn` и обратиться к фантастическим по своими возможностями библиотекам глубокого обучения. Для пользователей Python наиболее устоявшимися являются `keras`, `lasagna` и `tenzor-flow`. `lasagna` построена на основе библиотеки `theano`, тогда как `keras` может использовать либо `tensor-flow`, либо `theano`. Эти библиотеки предлагают гораздо более гибкий интерфейс для построения нейронных сетей и обновляются в соответствии с последними достижениями в области глубокого обучения. Кроме того, все популярные библиотеки глубокого обучения позволяют использовать высокопроизводительные графические процессоры (GPU), которые в `scikit-learn` не поддерживаются. Использование графических процессоров позволяет ускорить вычисления от 10 до 100 раз, и они

имеют важное значение для применения методов глубого обучения для крупномасштабных наборов данных.

Преимущества, недостатки и параметры

Нейронные сети вновь «вышли на сцену» во многих сферах применения машинного обучения в качестве передовых методов. Одно из их главных преимуществ заключается в том, что они способны обрабатывать информацию, содержащуюся в больших объемах данных, и строить невероятно сложные модели. При наличии достаточного времени вычислений, данных и тщательной настройки параметров нейронные сети часто превосходят другие алгоритмы машинного обучения (для задач классификации и регрессия).

Это дает и свои минусы. Нейронные сети, особенно крупные нейронные сети, как правило, требуют длительного времени обучения. Как мы видели здесь, они также требуют тщательной предварительной обработки данных. Аналогично SVM, нейронные сети лучше всего работают с «однородными» данными, где все признаки измерены в одном и том же масштабе. Что касается данных, в которых признаки имеют разный масштаб, модели на основе дерева могут дать лучший результат. Кроме того, настройка параметров нейронной сети – это само по себе искусство. В наших экспериментах мы едва коснулись возможных способов настройки и обучения нейросетевых моделей.

Оценка сложности в нейронных сетях

Наиболее важными параметрами являются ряде слоев и число скрытых блоков в одном слое. Вы должны начать с одной или двумя скрытыми слоями, и, возможно, расширить оттуда. Количество узлов на скрытом уровне часто аналогично числу входных функций, но редко выше, чем в низких до средних тысяч.

Полезным показателем, позволяющим судить о сложности нейронной сети, является количество вычисляемых в ходе обучения весов или коэффициентов. Если вы работаете с 2-классовым набором данных, содержащим 100 признаков, и используете нейронную сеть, состоящую из 100 скрытых элементов, то между входным и первым скрытым слоем будет 100 * 100 = 10000 весов. Кроме того, между скрытым слоем и выходным слоем будет 100 * 1 = 100 весов, что в итоге даст около 10100 весов. Если использовать второй скрытый слой размером 100 скрытых элементов, то 100 * 100 = 10000 весов из первого скрытого слоя добавятся ко второму скрытому слою, что в итоге составит 20100 весов. Если вместо этого вы будете использовать один слой из 1000 скрытых элементов, вы вычислите 100*1000 = 100000 весов на пути от входного слоя к скрытому слою и 1000*1 весов на пути из скрытого слоя к выходному слою, всего

101000 весов. Если использовать второй скрытый слой, вы добавите еще 1000*1000 = 1000000 весов, получив колоссальную цифру 1101000, что в 50 раз больше количества весов, вычисленных для модели с двумя скрытыми слоями по 100 элементов в каждом.

Общераспространенный способ настройки параметров в нейронной сети – сначала построить сеть достаточно большого размера, чтобы она обучилась. Затем, убедившись в том, что сеть может обучаться, сжимаете веса сети или увеличиваете `alpha`, чтобы добавить регуляризацию, которая улучшит обобщающую способность.

В наших экспериментах мы сосредоточились главным образом на спецификации модели: количестве слоев и узлов в слое, регуляризации и нелинейных функциях активации. Эти параметры задают модель, которую мы хотим обучить. Кроме того, встает вопрос о том, *как* обучить модель или алгоритм, который используется для вычисления весов и задается с помощью параметра `solver`. Существует два простых в использовании алгоритма. Алгоритм `'adam'`, выставленный по умолчанию, дает хорошее качество в большинстве ситуаций, но весьма чувствителен к масштабированию данных (поэтому важно отмасштабировать ваши данные так, чтобы каждая характиристика имела среднее 0 и дисперсию 1). Другой алгоритм `'lbfgs'` вполне надежен, но может занять много времени в случае больших моделей или больших массивов данных. Существует также более продвинутая опция `'sgd'`, которая используется многими специалистами по глубокому обучению. Опция`'sgd'` имеет большее количество дополнительных параметров, которые нужно настроить для получения наилучших результатов. Вы можете найти описания всех этих параметров в руководстве пользователя. Начиная работать с MLP, придерживайтесь алгоритмов `'adam'` и `'l-bfgs'`.

Каждый раз метод fit строит модель заново

Важное свойство моделей `scikit-learn` заключается в том, что вызов метода `fit` всегда будет сбрасывать все, чему модель обучилась ранее. Таким образом, если вы построите модель на одном наборе данных, а затем вызовете метод `fit` снова для другого набора данных, модель «забудет» все, чему обучилась на первом наборе данных. Вы можете без ограничений вызывать метод `fit` для модели и результат будет таким, как если бы вы вызвали его для «новой» модели.

Оценки неопределенности для классификаторов

Еще одна полезная деталь интерфейса `scikit-learn`, о которой мы еще не говорили – это возможность вычислить оценки неопределенности

прогнозов. Часто вас интересует не только класс, спрогнозированный моделью для определенной точки тестового набора, но и степень уверенности модели в правильности прогноза. В реальной практике различные виды ошибок приводят к очень разным результатам. Представьте себе медицинский тест для определения рака. Ложно положительный прогноз может привести к проведению дополнительных исследований, тогда как ложно отрицательный прогноз может привести к пропуску серьезной болезни. Мы подробнее разберем эту тему в главе 6.

В `scikit-learn` существует две различные функции, с помощью которых можно оценить неопределенность прогнозов: `decision_function` и `predict_proba`. Большая часть классификаторов (но не все) позволяет использовать по крайней мере одну из этих функций. Давайте применим эти две функции к синтетическому двумерному набору данных, построив классификатор `GradientBoostingClassifier`, который позволяет использовать как метод `decision_function`, так и метод `predict_proba`:

```
In[105]:
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_blobs, make_circles
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)

# мы переименовываем классы в «blue» и «red» для удобства
y_named = np.array(["blue", "red"])[y]

# мы можем вызвать train_test_split с любым количеством массивов,
# все будут разбиты одинаковым образом
X_train, X_test, y_train_named, y_test_named, y_train, y_test = \
train_test_split(X, y_named, y, random_state=0)

# строим модель градиентного бустинга
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train_named)
```

Решающая функция

В бинарной классификации возвращаемое значение `decision_function` имеет форму `(n_samples)`:

```
In[106]:
print("Форма массива X_test: {}".format(X_test.shape))
print("Форма решающей функции: {}".format(
     gbrt.decision_function(X_test).shape))

Out[106]:
Форма массива X_test: (25, 2)
Форма решающей функции: (25,)
```

Возвращаемое значение представляет собой число с плавающей точкой для каждого примера:

```
In[107]:
# выведем несколько первых элементов решающей функции
print("Решающая функция:\n{}".format(gbrt.decision_function(X_test)[:6]))

Out[107]:
Решающая функция:
[ 4.136 -1.683 -3.951 -3.626 4.29 3.662]
```

Значение показывает, насколько сильно модель уверена в том, что точка данных принадлежит «положительному» классу, в данном случае, классу 1. Положительное значение указывает на предпочтение в пользу позиционного класса, а отрицательное значение – на предпочтение в пользу «отрицательного» (другого) класса.

Мы можем судить о прогнозах, лишь взглянув на знак решающей функции.

```
In[108]:
print("Решающая функция с порогом отсечения:\n{}".format(
    gbrt.decision_function(X_test) > 0))
print("Прогнозы:\n{}".format(gbrt.predict(X_test)))

Out[108]:
Решающая функция с порогом отсечения:
[ True False False False True True False True True True False True
  True False True False False False True True True True True False
  False]
Прогнозы:
['red' 'blue' 'blue' 'blue' 'red' 'red' 'blue' 'red' 'red' 'red' 'blue'
 'red' 'red' 'blue' 'red' 'blue' 'blue' 'blue' 'red' 'red' 'red' 'red'
 'red' 'blue' 'blue']
```

Для бинарной классификации «отрицательный» класс – это всегда первый элемент атрибута `classes_`, а «положительный» класс – второй элемент атрибута `classes_`. Таким образом, если вы хотите полностью просмотреть вывод метода `predict`, вам нужно воспользоваться атрибутом `classes_`:

```
In[109]:
# переделаем булевы значения True/False в 0 и 1
greater_zero = (gbrt.decision_function(X_test) > 0).astype(int)
# используем 0 и 1 в качестве индексов атрибута classes_
pred = gbrt.classes_[greater_zero]
# pred идентичен выводу gbrt.predict
print("pred идентичен прогнозам: {}".format(
    np.all(pred == gbrt.predict(X_test))))

Out[109]:
pred идентичен прогнозам: True
```

Диапазон значений `decision_function` может быть произвольным и зависит от данных и параметров модели:

```
In[110]:
decision_function = gbrt.decision_function(X_test)
print("Решающая функция минимум: {:.2f} максимум: {:.2f}".format(
    np.min(decision_function), np.max(decision_function)))

Out[110]:
Решающая функция минимум: -7.69 максимум: 4.29
```

Это произвольное масштабирование часто затрудняет интерпретацию вывода `decision_function`.

В следующем примере мы построим `decision_function` для всех точек в двумерной плоскости, используя цветовую кодировку и уже знакомую визуализацию решающей границы. Мы представим точки обучающего набора в виже кружков, а тестовые данные – в виде треугольников (рис. 2.55):

```
In[111]:
fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4,
                                fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1],
                                            alpha=.4, cm=mglearn.ReBl)

for ax in axes:
    # размещаем на графике точки обучающего и тестового наборов
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Характеристика 0")
    ax.set_ylabel("Характеристика 1")
    cbar = plt.colorbar(scores_image, ax=axes.tolist())
    axes[0].legend(["Тест класс 0", "Тест класс 1", "Обучение класс 0",
                    "Обучение класс 1"], ncol=4, loc=(.1, 1.1))
```

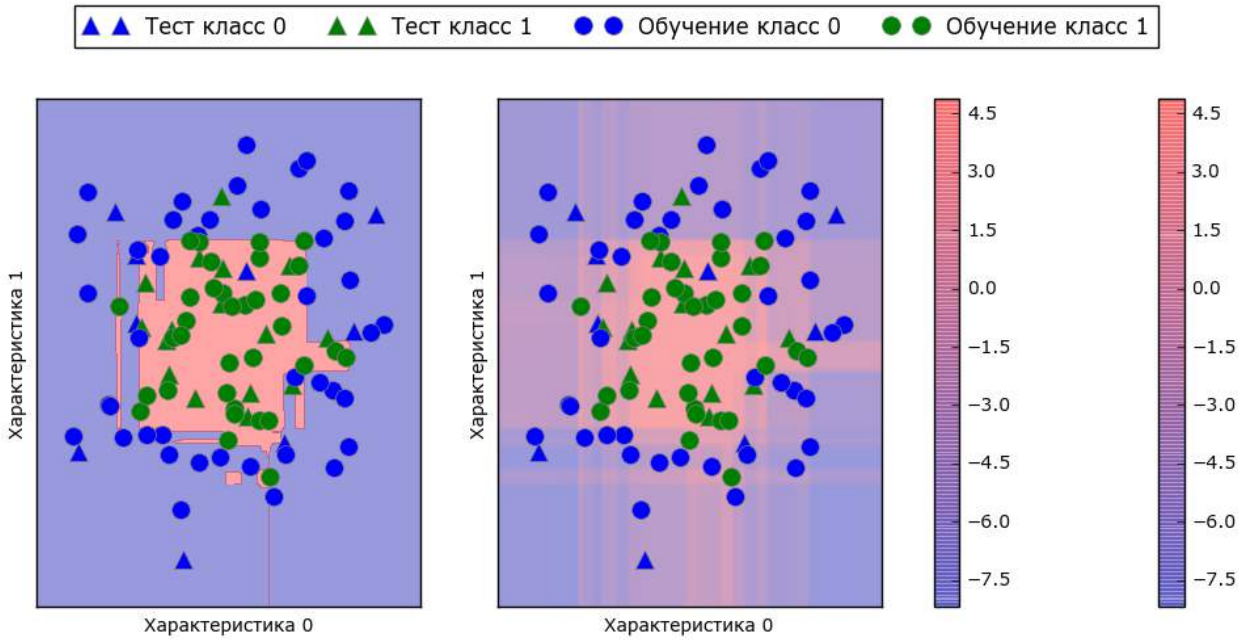

Рис. 2.55 Граница принятия решений (слева) и решающая функция (справа) модели градиентного бустинга, построенной на двумерном синтетическом наборе данных

Цветовая кодировка не только спрогнозированного результата, но степени определенности прогноза дает дополнительную информацию.

Однако в этой визуализации трудно разглядеть границу между двумя классами.

Прогнозирование вероятностей

Вывод метода `predict_proba` – это вероятность каждого класса и часто его легче понять, чем вывод метода `decision_function`. Для бинарной классификации он всегда имеет форму `(n_samples, 2)`:

```
In[112]:
print("Форма вероятностей: {}".format(gbrt.predict_proba(X_test).shape))

Out[112]:
Форма вероятностей: (25, 2)
```

Первый элемент строки – это оценка вероятности первого класса, а второй элемент строки – это оценка вероятности второго класса. Поскольку речь идет о вероятности, то значения в выводе `predict_proba` всегда находятся в диапазоне между 0 и 1, а сумма значений для обоих классов всегда равна 1:

```
In[113]:
# выведем первые несколько элементов predict_proba
print("Спрогнозированные вероятности:\n{}".format(
    gbrt.predict_proba(X_test[:6])))

Out[113]:

Спрогнозированные вероятности:
[[ 0.016 0.984]
 [ 0.843 0.157]
 [ 0.981 0.019]
 [ 0.974 0.026]
 [ 0.014 0.986]
 [ 0.025 0.975]]
```

Поскольку вероятности обоих классов в сумме дают 1, один из классов всегда будет иметь определенность, превышающую 50%. Этот класс и будет спрогнозирован.[21]

В предыдущем выводе видно, что большинство точек отнесены к тому или иному классу с высокой долей определенности. Соответствие спрогнозированной неопределенности фактической зависит от модели и параметров. Для переобученной модели характерна высокая доля определенности прогнозов, даже если они и ошибочные. Модель с меньшей сложностью обычно характеризуется высокой долей неопределенности своих прогнозов. Модель называется *калиброванной* (*calibrated*), если вычисленная неопределенность соответствует

[21] Поскольку вероятности – это числа с плавающей точкой, то маловероятно, что они обе будут точно равны 0.500. Однако, если это произойдет, то прогноз будет осуществлен случайным образом.

фактической: в калиброванной модели прогноз, полученный с 70%-ной определенностью, будет правильным в 70% случаев.

В следующем примере (рис. 2.56) мы снова покажем границу принятия решения для набора данных, а также вероятности для класса 1:

```
In[114]:
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

mglearn.tools.plot_2d_separator(
    gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(
    gbrt, X, ax=axes[1], alpha=.5, cm=mglearn.ReBl, function='predict_proba')

for ax in axes:
    # размещаем на графике точки обучающего и тестового наборов
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Характеристика 0")
    ax.set_ylabel("Характеристика 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Тест класс 0", "Тест класс 1", "Обуч класс 0",
                "Обуч класс 1"], ncol=4, loc=(.1, 1.1))
```

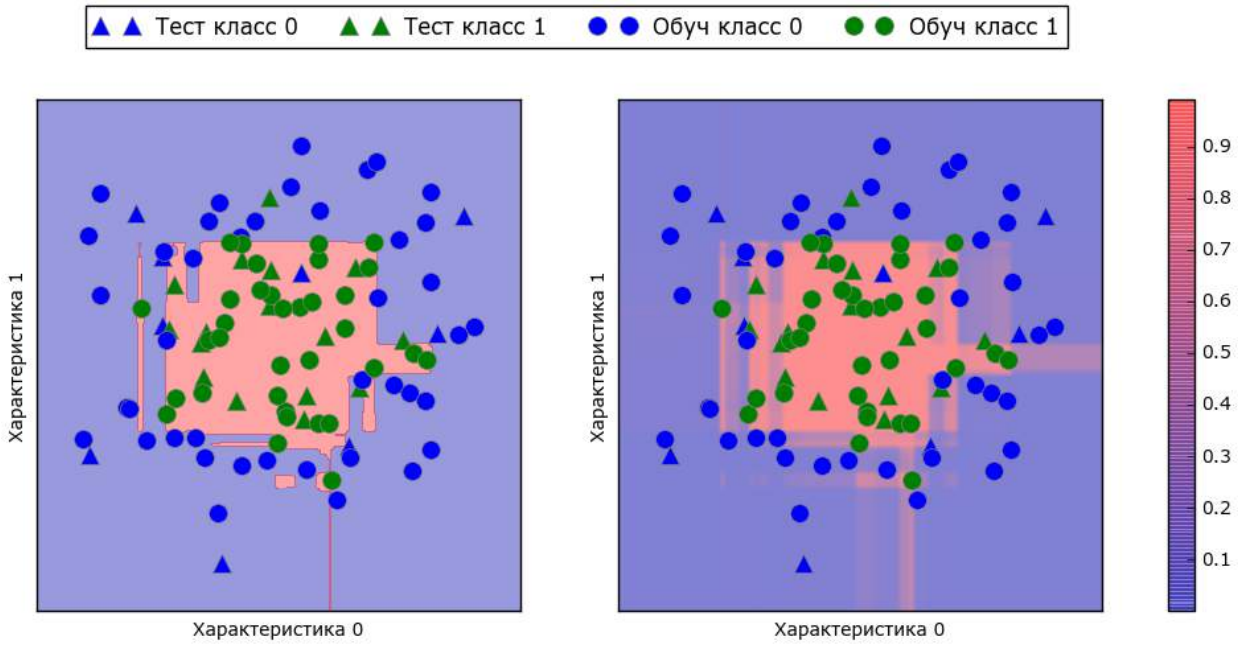

Рис. 2.56 Граница принятия решений (слева) спрогнозированные вероятности для модели градиентного бустинга, показанной на рис. 2.55

Границы на этом рисунке определены гораздо более четко, а небольшие участки неопределенности отчетливо видны.

На сайте `scikit-learn` дается сравнение различных моделей и визуализации оценок неопределенности для этих моделей. Мы воспроизвели их на рис. 2.57 и рекомендуем ознакомиться с ними.

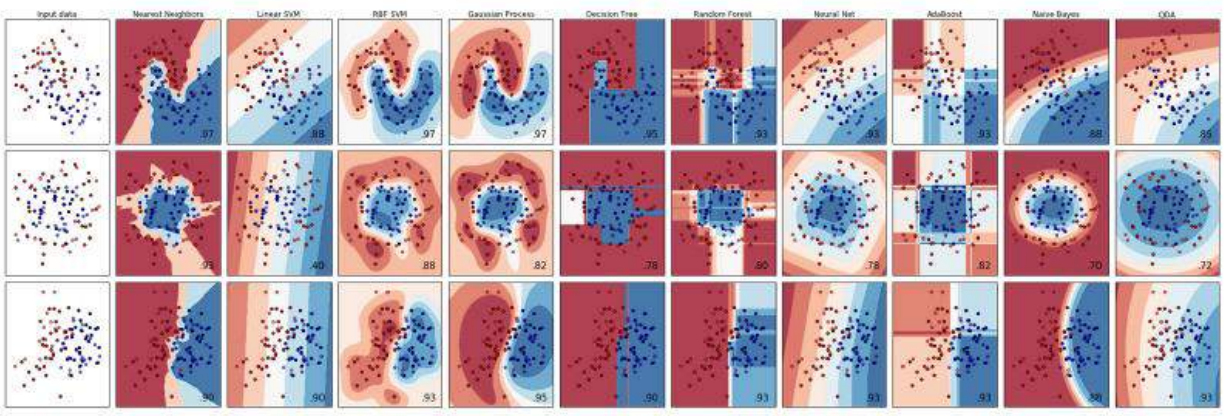

Рис. 2.57 Сравнение нескольких классификаторов scikit-learn, построенных на синтетических наборах данных (изображение предоставлено http://scikit-learn.org)

Неопределенность в мультиклассовой классификации

До сих пор мы говорили только об оценках неопределенности в бинарной классификации. Однако методы `decision_function` и `predict_proba` также можно применять в мультиклассовой классификации. Давайте применим их к набору данных Iris, который представляет собой пример 3-классовой классификации:

```
In[115]:
from sklearn.datasets import load_iris

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=42)

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

In[116]:
print("Форма решающей функции: {}".format(gbrt.decision_function(X_test).shape))
# выведем первые несколько элементов решающей функции
print("Решающая функция:\n{}".format(gbrt.decision_function(X_test)[:6, :]))

Out[116]:
Форма решающей функции: (38, 3)
Решающая функция:
[[-0.529  1.466 -0.504]
 [ 1.512 -0.496 -0.503]
 [-0.524 -0.468  1.52 ]
 [-0.529  1.466 -0.504]
 [-0.531  1.282  0.215]
 [ 1.512 -0.496 -0.503]]
```

В мультиклассовой классификации `decision_function` имеет форму `(n_samples, n_classes)` и каждый столбец показывает «оценку определенности» для каждого класса, где высокая оценка означает бо́льшую вероятность данного класса, а низкая оценка означает меньшую вероятность этого класса. Вы можете получить прогнозы, исходя из этих

оценок, с помощью функции `np.argmax`. Она возвращает индекс максимального элемента массива для каждой точки данных:

```
In[117]:
print("Argmax решающей функции:\n{}".format(
      np.argmax(gbrt.decision_function(X_test), axis=1)))
print("Прогнозы:\n{}".format(gbrt.predict(X_test)))
```

```
Out[117]:
Argmax решающей функции:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Прогнозы:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

Вывод `predict_proba` имеет точно такую же форму `(n_samples, n_classes)`. И снова вероятности возможных классов для каждой точки данных дают в сумме 1:

```
In[118]:
# выведем первые несколько элементов predict_proba
print("Спрогнозированные вероятности:\n{}".format(gbrt.predict_proba(X_test)[:6]))
# покажем, что сумма значений в каждой строке равна 1
print("Суммы: {}".format(gbrt.predict_proba(X_test)[:6].sum(axis=1)))
```

```
Out[118]:
Спрогнозированные вероятности:
[[ 0.107 0.784 0.109]
 [ 0.789 0.106 0.105]
 [ 0.102 0.108 0.789]
 [ 0.107 0.784 0.109]
 [ 0.108 0.663 0.228]
 [ 0.789 0.106 0.105]]
Суммы: [ 1. 1. 1. 1. 1. 1.]
```

Мы вновь можем получить прогнозы, вычислив `argmax` для `predict_proba`:

```
In[119]:
print("Argmax спрогнозированных вероятностей:\n{}".format(
    np.argmax(gbrt.predict_proba(X_test), axis=1)))
print("Прогнозы:\n{}".format(gbrt.predict(X_test)))
```

```
Out[119]:
Argmax спрогнозированных вероятностей:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Прогнозы:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

Подводя итог, отметим, что `predict_proba` и `decision_function` всегда имеют форму `(n_samples, n_classes)`, за исключением `decision_function` в случае бинарной классификации. В бинарной классификации `decision_function` имеет только один столбец, соответствующий «положительному» классу `classes_[1]`.

Для количества столбцов, равного `n_classes`, вы можете получить прогноз, вычислив `argmax` по столбцам. Однако будьте осторожны, если ваши классы – строки или вы используете целые числа, которые не являются последовательными и начинаются не с 0. Если вы хотите

сравнить результаты, полученные с помощью `predict`, с результатами `decision_function` или `predict_proba`, убедитесь, что используете атрибут `classes_` для получения фактических названий классов:

```
In[120]:
logreg = LogisticRegression()

# представим каждое целевое значение названием класса в наборе iris
named_target = iris.target_names[y_train]
logreg.fit(X_train, named_target)
print("уникальные классы в обучающем наборе: {}".format(logreg.classes_))
print("прогнозы: {}".format(logreg.predict(X_test)[:10]))
argmax_dec_func = np.argmax(logreg.decision_function(X_test), axis=1)
print("argmax решающей функции: {}".format(argmax_dec_func[:10]))
print("argmax объединенный с классами_: {}".format(
        logreg.classes_[argmax_dec_func][:10]))

Out[120]:
уникальные классы в обучающем наборе: ['setosa' 'versicolor' 'virginica']
прогнозы: ['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor'
'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
argmax решающей функции: [1 0 2 1 1 0 1 2 1 1]
argmax объединенный с классами_: ['versicolor' 'setosa' 'virginica' 'versicolor'
'versicolor' 'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
```

Выводы и перспективы

Мы начали эту главу с обсуждения такого понятия, как сложность модели, а затем рассказали об *обобщающей способности* (*generalization*), то есть о построении такой модели, которая может хорошо работать на новых, ранее неизвестных данных. Это привело нас к понятиям «недообучение», когда модель не может описать изменчивость обучающих данных, и «переобучение», когда модель слишком много внимания уделяет обучающим данным[22] и не способна хорошо обобщить новые данные.

Затем мы рассмотрели широкий спектр моделей машинного обучения для классификации и регрессии, их преимущества и недостатки, настройки сложности для каждой модели. Мы увидели, что для достижения хорошего качества работы во многих алгоритмах важное значение имеет установка правильных параметров. Кроме того, некоторые алгоритмы чувствительны к типу входных данных, и, в частности, к тому, как масштабированы признаки. Поэтому слепое применение алгоритма к данным без понимания исходных предположений модели и принципов работы параметров редко приводит к построению точной модели.

Эта глава содержит много информации об алгоритмах, но вам необязательно помнить все эти детали, чтобы понимать содержание следующих глав. Тем не менее некоторая информация о моделях,

[22] То есть модель стремится описать даже незначительные колебания в обучающих данных, которые могут иметь случайный характер (принимает шум за сигнал). – *Прим. пер.*

упомянутых здесь, и контексте использования этих моделей, имеет важное значение для успешного применения машинного обучения на практике. Ниже дается краткий обзор случаев использования той или иной модели:

Ближайшие соседи

Подходит для небольших наборов данных, хорош в качестве базовой модели, прост в объяснении.

Линейные модели

Считается первым алгоритмом, который нужно попробовать, хорош для очень больших наборов данных, подходит для данных с очень высокой размерностью.

Наивный байесовский классификатор

Подходит только для классификации. Работает даже быстрее, чем линейные модели, хорош для очень больших наборов данных и высокоразмерных данных. Часто менее точен, чем линейные модели.

Деревья решений

Очень быстрый метод, не нужно масштабировать данные, результаты можно визуализировать и легко объяснить.

Случайные леса

Почти всегда работают лучше, чем одно дерево решений, очень устойчивый и мощный метод. Не нужно масштабировать данные. Плохо работает с данными очень высокой размерности и разреженными данными.

Градиентный бустинг деревьев решений

Как правило, немного более точен, чем случайный лес. В отличие от случайного леса медленнее обучается, но быстрее предсказывает и требует меньше памяти. По сравнению со случайным лесом требует настройки большего числа параметров.

Машины опорных векторов

Мощный метод для работы с наборами данных среднего размера и признаками, измеренными в едином масштабе. Требует масштабирования данных, чувствителен к изменению параметров.

Нейронные сети

Можно построить очень сложные модели, особенно для больших наборов данных. Чувствительны к масштабированию данных и выбору параметров. Большим моделям требуется много времени для обучения.

При работе с новым набором данных лучше начать с простой модели, например, с линейной модели, наивного байесовского классификатора или классификатора ближайших соседей, и посмотреть, как далеко можно продвинуться с точки зрения качества модели. Лучше изучив данные, вы можете выбрать алгоритм, который может строить более сложные модели, например, случайный лес, градиентный бустинг деревьев решений, SVM или нейронную сеть.

Теперь у вас уже есть некоторое представление о том, как применять, настраивать и анализировать модели, которые мы здесь рассмотрели. В этой главе мы сосредоточились на бинарном классификации, поскольку ее, как правило, легче всего интерпретировать. Большинство представленных алгоритмов могут решать задачи регрессии и классификации вариантов, при этом все алгоритмы классификации поддерживают как бинарную, так и мультиклассовую классификацию. Попробуйте применить любой из этих алгоритмов к наборам данных, включенным в `scikit-learn`, например, к наборам для регрессии `boston_housing` или `diabetes`, или к набору `digits` для мультиклассовой классификации. Экспериментирование с алгоритмами на различных наборах данных позволит вам лучше понять, насколько быстро обучаются различные алгоритмы, насколько легко анализировать построенные с их помощью модели и насколько эти алгоритмы чувствительны к типу данных.

Несмотря на то что мы проанализировали результаты применения различных параметров для исследованных нами алгоритмов, процесс построения модели, которая будет хорошо обобщать новые данные, выглядит немного сложнее. В главе 6 мы увидим, как правильно настраивать параметры и как автоматически найти оптимальные параметры.

Но для начала в следующей главе мы более детально рассмотрим обучение без учителя и предварительную обработку данных.