**Лаб.5. Заглянем глубже: линейная регрессия**

Аналогично тому, как наивный байесовский классификатор (который мы рассматривали в лабораторной работе: «Заглянем глубже: наивная байесовская классификация») — отличная отправная точка для задач классификации, так и линейные регрессионные модели — хорошая отправная точка для задач регрессии. Подобные модели популярны в силу быстрой обучаемости и возврата очень удобных для интерпретации результатов. Вероятно, вы уже знакомы с простейшей формой линейной регрессионной модели (то есть подбора для данных разделяющей прямой линии), но такие модели можно распространить на моделирование и более сложного поведения данных. В данной лабораторной работе мы начнем с быстрого и интуитивно понятного математического описания известной задачи, а потом перейдем к вопросам обобщенных линейных моделей, учитывающих более сложные паттерны в данных. Начнем с обычных импортов:

In[1]: %matplotlib inline

import matplotlib.pyplot as plt

import seaborn as sns; sns.set()

import numpy as np

**Простая линейная регрессия**

Начнем с линейной регрессии, прямолинейной аппроксимации наших данных. Прямолинейная аппроксимация представляет собой модель вида *y = ax + b*, в которой *a* известна как угловой коэффициент, а *b* — как точка пересечения с осью координат Y. Рассмотрим следующие данные, распределенные около прямой с угловым коэффициентом 2 и точкой пересечения – 5 (рис. 5.1):

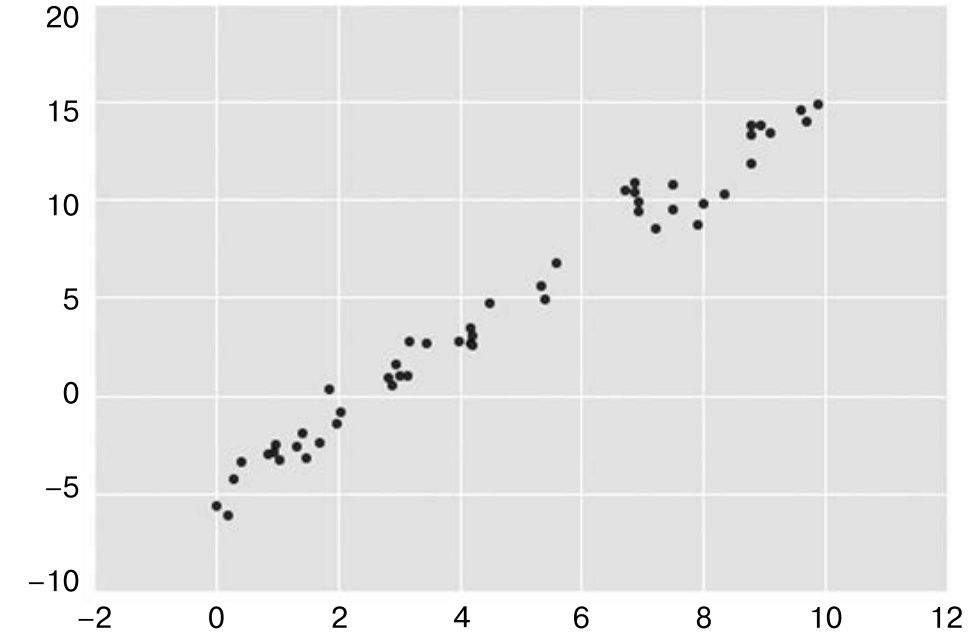


Рис. 5.1. Данные для линейной регрессии

In[2]: rng = np.random.RandomState(1)

x = 10 \* rng.rand(50)

y = 2 \* x - 5 + rng.randn(50)

plt.scatter(x, y);

Воспользуемся оценивателем LinearRegression из библиотеки Scikit-Learn для обучения на этих данных и поиска оптимальной прямой (рис. 5.2):

In[3]: from sklearn.linear\_model import LinearRegression

model = LinearRegression(fit\_intercept=True)

model.fit(x[:, np.newaxis], y)

xfit = np.linspace(0, 10, 1000)

yfit = model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)

plt.plot(xfit, yfit);

Подбираемые параметры модели (в библиотеке Scikit-Learn всегда содержат в конце знак подчеркивания) включают угловой коэффициент и точку пересечения с осью координат. В данном случае соответствующие параметры — coef\_ и intercept\_ :

In[4]: print("Model slope: ", model.coef\_[0])

print("Model intercept:", model.intercept\_)

Model slope: 2.02720881036

Model intercept: -4.99857708555

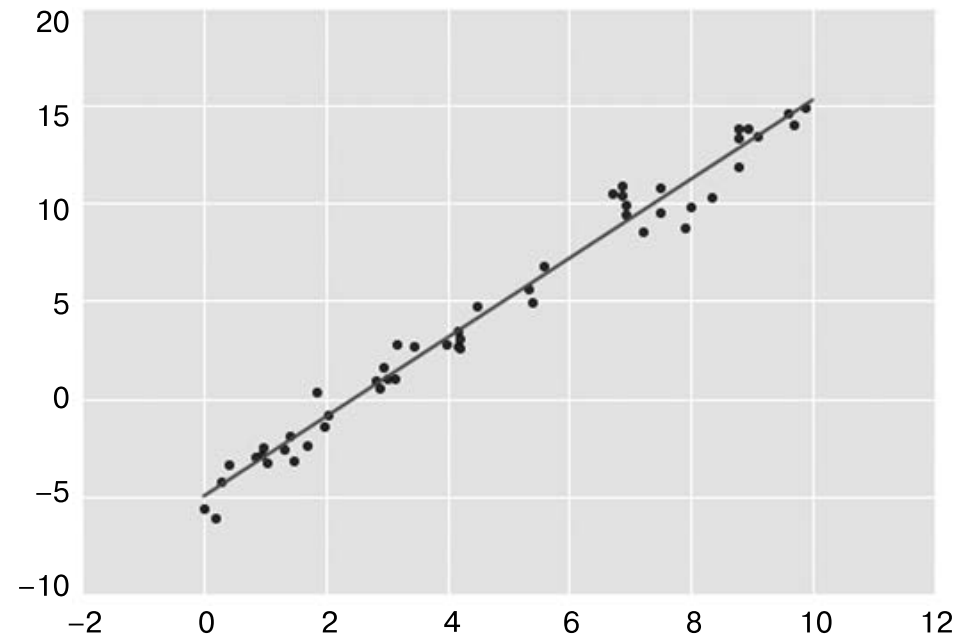


Рис. 5.2. Линейная регрессионная модель

Видим, что результаты очень близки к входным данным, как мы и надеялись.

Однако возможности оценивателя LinearRegression намного шире этого: помимо аппроксимации прямыми линиями, он может также работать с многомерными линейными моделями следующего вида:

*y = a*0 *+ a*1*x*1 *+a*2*x*2 *+...*

с несколькими величинами *x*. Геометрически это подобно подбору плоскости для точек в трех измерениях или гиперплоскости для точек в пространстве с еще большим числом измерений.

Многомерная сущность подобных регрессий усложняет их визуализацию, но мы можем посмотреть на одну из этих аппроксимаций в действии, создав данные для нашего примера с помощью оператора матричного умножения из библиотеки NumPy:

In[5]: rng = np.random.RandomState(1)

X = 10 \* rng.rand(100, 3)

y = 0.5 + np.dot(X, [1.5, -2., 1.])

model.fit(X, y)

print(model.intercept\_)

print(model.coef\_)

0.5

[ 1.5 -2. 1. ]

Здесь данные величины *y* сформированы из трех случайных значений величины *x*, а линейная регрессия восстанавливает использовавшиеся для их формирования коэффициенты.

Аналогичным образом можно использовать оцениватель LinearRegression для аппроксимации наших данных прямыми, плоскостями и гиперплоскостями. По-прежнему складывается впечатление, что этот подход ограничивается лишь строго линейными отношениями между переменными, но оказывается, что ослабление этого требования также возможно.

**Регрессия по комбинации базисных функций**

Один из трюков, позволяющих приспособить линейную регрессию к нелинейным отношениям между переменными, — преобразование данных в соответствии с новыми базисными функциями. Один из вариантов этого трюка мы уже встречали в конвейере PolynomialRegression, который использовался в разделах «Гиперпараметры и проверка модели» и «Проектирование признаков» данной главы. Идея состоит в том, чтобы взять многомерную линейную модель:

и построить *x*1, *x*2, *x*3 и т. д. на основе имеющегося одномерного входного значения *x*. То есть у нас *x*n = fn(*x*), где fn(*x*) — некая функция, выполняющая преобразование данных.

Например, если fn(*x*) = *x*n, то наша модель превращается в полиномиальную регрессию:

Обратите внимание, что модель по-прежнему остается *линейной* — линейность относится к тому, что коэффициенты *a*n никогда не умножаются и не делятся друг на друга. Фактически мы взяли наши одномерные значения *x* и выполнили проекцию их на более многомерное пространство, так что с помощью линейной аппроксимации мы можем теперь отражать более сложные зависимости между *x* и *y*.

**Полиномиальные базисные функции**

Данная полиномиальная проекция настолько удобна, что была встроена в библиотеку **Scikit-Learn** в виде преобразователя PolynomialFeatures :

In[6]: from sklearn.preprocessing import PolynomialFeatures

x = np.array([2, 3, 4])

poly = PolynomialFeatures(3, include\_bias=False)

poly.fit\_transform(x[:, None])

Out[6]: array([[ 2., 4., 8.],

[ 3., 9., 27.],

[ 4., 16., 64.]])

Как видим, преобразователь превратил наш одномерный массив в трехмерный путем возведения каждого из значений в степень. Это новое, более многомерное

представление данных можно далее использовать для линейной регрессии.

Как мы уже видели в разделе «Проектирование признаков» данной главы, самый

изящный способ выполнения этого — воспользоваться конвейером. Создадим,

указанным образом полиномиальную модель седьмого порядка:

In[7]: from sklearn.pipeline import make\_pipeline

poly\_model = make\_pipeline(PolynomialFeatures(7), LinearRegression())

После такого преобразования можно воспользоваться линейной моделью для под-

бора намного более сложных зависимостей между величинами x и y. Например,

рассмотрим зашумленную синусоиду (рис. 5.44):

In[8]: rng = np.random.RandomState(1)

x = 10 \* rng.rand(50)

y = np.sin(x) + 0.1 \* rng.randn(50)

poly\_model.fit(x[:, np.newaxis], y)

yfit = poly\_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)

plt.plot(xfit, yfit);

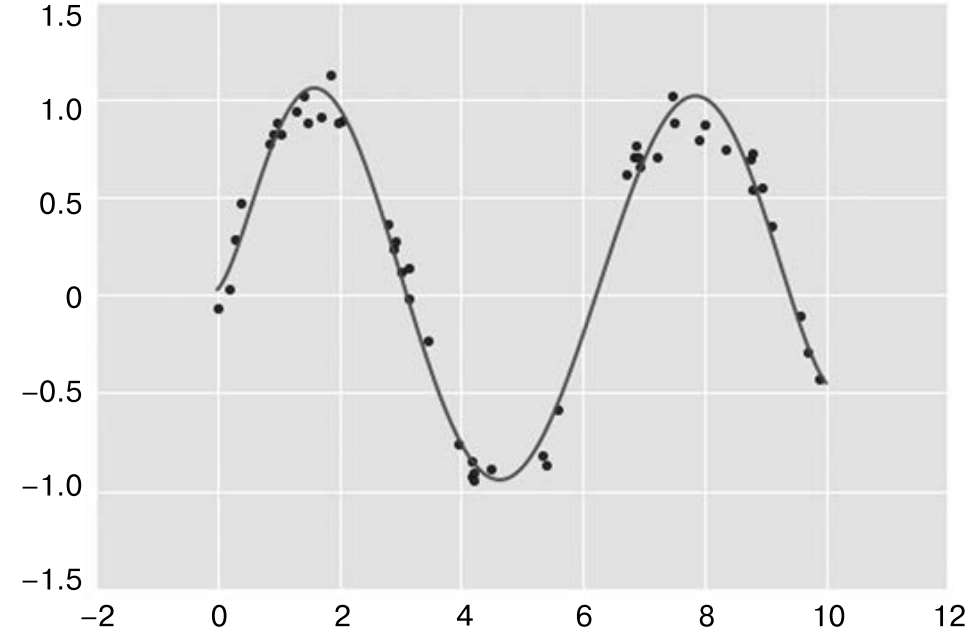


Рис. 5.3. Полиномиальная аппроксимация нелинейной обучающей последовательности

С нашей линейной моделью, используя полиномиальные базисные функции

седьмого порядка, мы получили великолепную аппроксимацию этих нелиней-

ных данных!

**Гауссовы базисные функции**

Можно использовать и другие базисные функции. Например, один из полезных

паттернов — обучение модели, представляющей собой сумму не полиномиальных,

а Гауссовых базисных функций. Результат будет выглядеть следующим образом

(рис. 5.45):

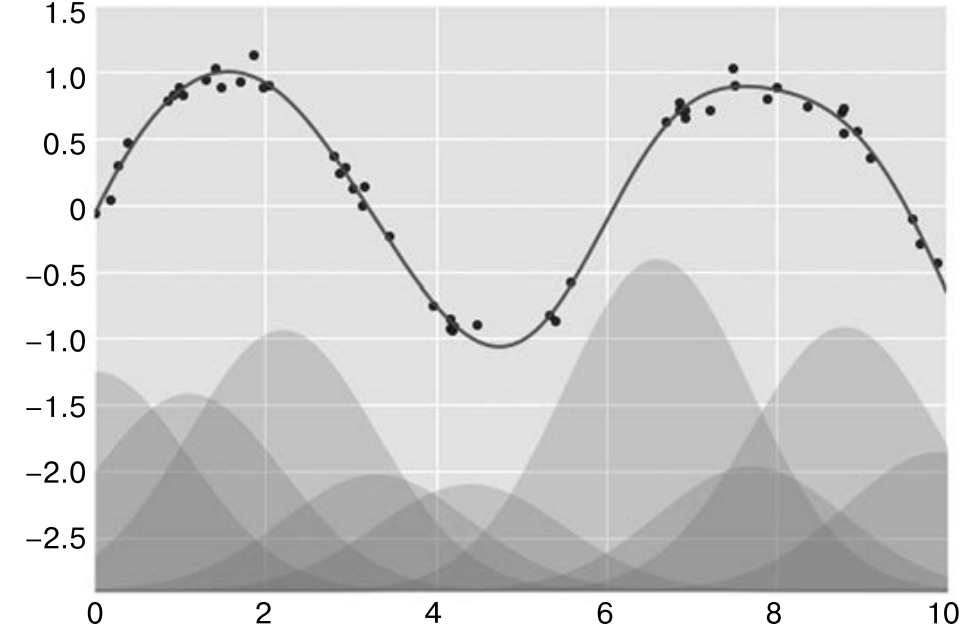


Рис. 5.4. Аппроксимация нелинейных данных с помощью Гауссовых базисных функций

Затененные области на рис. 5.45 — нормированные базисные функции, дающие

при сложении аппроксимирующую данные гладкую кривую. Эти Гауссовы базис-

ные функции не встроены в библиотеку Scikit-Learn, но мы можем написать для их

создания пользовательский преобразователь, как показано ниже и проиллюстри-

ровано на рис. 5.46 (преобразователи библиотеки Scikit-Learn реализованы как

классы языка Python; чтение исходного кода библиотеки Scikit-Learn — отличный

способ разобраться с их созданием):

In[9]:

from sklearn.base import BaseEstimator, TransformerMixin

class GaussianFeatures(BaseEstimator, TransformerMixin):

"""Равномерно распределенные Гауссовы признаки

для одномерных входных данных"""

def \_\_init\_\_(self, N, width\_factor=2.0):

self.N = N

self.width\_factor = width\_factor

@staticmethod

def \_gauss\_basis(x, y, width, axis=None):

arg = (x - y) / width

return np.exp(-0.5 \* np.sum(arg \*\* 2, axis))

def fit(self, X, y=None):

# Создаем N центров, распределенных по всему диапазону данных

self.centers\_ = np.linspace(X.min(), X.max(), self.N)

self.width\_ = self.width\_factor \*

(self.centers\_[1] - self.centers\_[0])

return self

def transform(self, X):

return self.\_gauss\_basis(X[:, :, np.newaxis], self.centers\_,

self.width\_, axis=1)

gauss\_model = make\_pipeline(GaussianFeatures(20),

LinearRegression())

gauss\_model.fit(x[:, np.newaxis], y)

yfit = gauss\_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)

plt.plot(xfit, yfit)

plt.xlim(0, 10);

Мы привели этот пример лишь для того, чтобы подчеркнуть, что в полиноми-

альных базисных функциях нет никакого колдовства. Если у вас есть какие-то

дополнительные сведения о процессе генерации ваших данных, исходя из кото-

рых у вас есть основания полагать, что наиболее подходящим будет тот или иной

базис, — тоже можете его использовать.

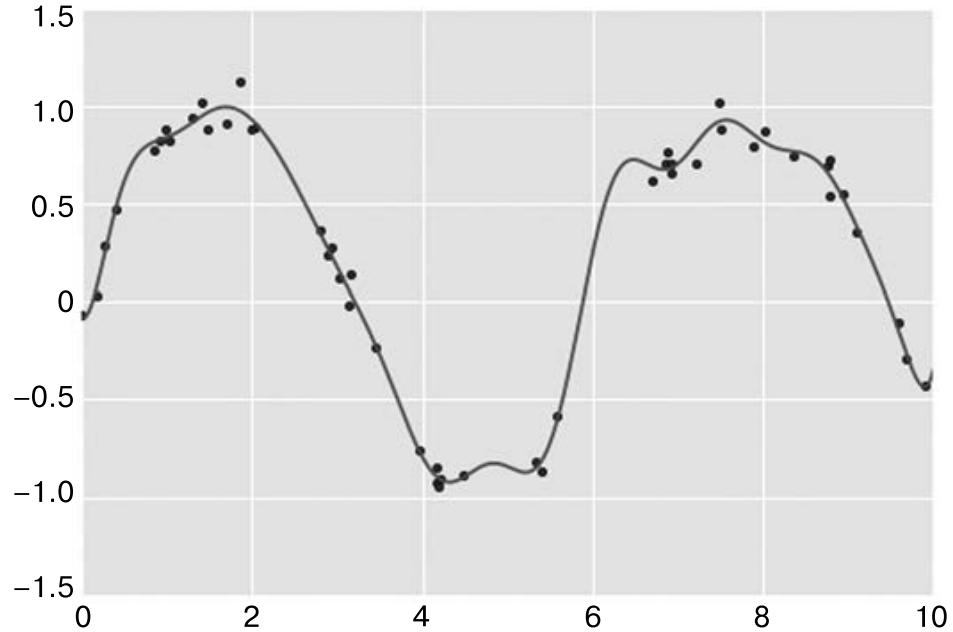


Рис. 5.5. Аппроксимация Гауссовыми базисными функциями, вычисленными с помощью пользовательского преобразователя

**Регуляризация**

Применение базисных функций в нашей линейной модели делает ее намного гибче,

но также и быстро приводит к переобучению (за подробностями обратитесь к раз-

делу «Гиперпараметры и проверка модели» данной главы). Например, если выбрать

слишком много Гауссовых базисных функций, мы в итоге получим не слишком

хорошие результаты (рис. 5.47):

In[10]: model = make\_pipeline(GaussianFeatures(30),

LinearRegression())

model.fit(x[:, np.newaxis], y)

plt.scatter(x, y)

plt.plot(xfit, model.predict(xfit[:, np.newaxis]))

plt.xlim(0, 10)

plt.ylim(-1.5, 1.5);

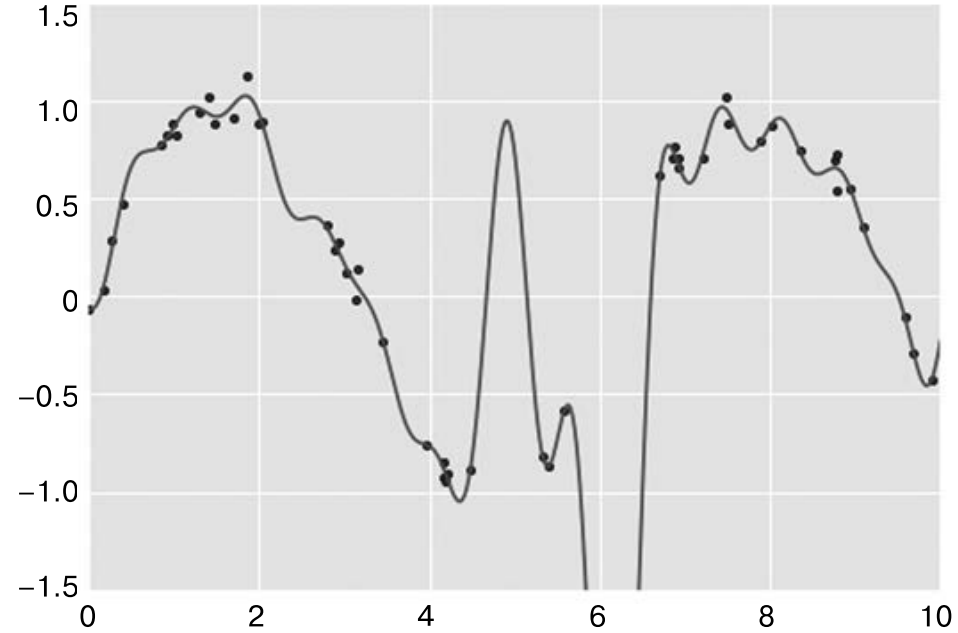


Рис. 5.6. Пример переобучения на данных: слишком сложная модель с базисными функциями

В результате проекции данных на 30-мерный базис модель оказалась слишком уж

гибкой и стремится к экстремальным значениям в промежутках между точками, в ко-

торых она ограничена данными. Причину этого можно понять, построив график коэф-

фициентов Гауссовых базисных функций в соответствии с координатой x (рис. 5.48):

In[11]: def basis\_plot(model, title=None):

fig, ax = plt.subplots(2, sharex=True)

model.fit(x[:, np.newaxis], y)

ax[0].scatter(x, y)

ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))

ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

if title:

ax[0].set\_title(title)

ax[1].plot(model.steps[0][1].centers\_,

model.steps[1][1].coef\_)

ax[1].set(xlabel='basis location', # Базовое местоположение

ylabel='coefficient',

# Коэффициент

xlim=(0, 10))

model = make\_pipeline(GaussianFeatures(30), LinearRegression())

basis\_plot(model)

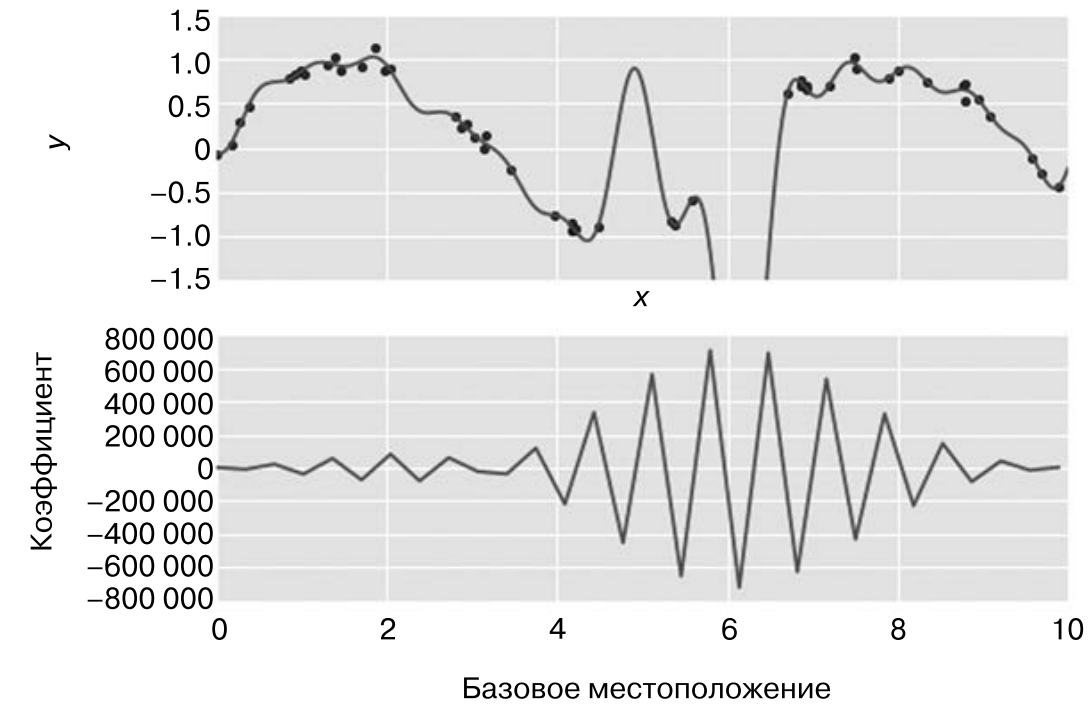


Рис. 5.7. Коэффициенты при Гауссовых базисных функциях в чрезмерно сложной модели

Нижняя часть рис. 5.48 демонстрирует амплитуду базисной функции в каждой

из точек. Это типичное поведение для переобучения с перекрытием областей

определения базисных функций: коэффициенты соседних базисных функций

усиливают и подавляют друг друга. Мы знаем, что подобное поведение при-

водит к проблемам и было бы неплохо ограничивать подобные пики в модели

явным образом, «накладывая штраф» на большие значения параметров модели.

Подобное «штрафование» известно под названием регуляризации и существует

в нескольких вариантах.

Гребневая регрессия (L 2 -регуляризация)

Вероятно, самый часто встречающийся вид регуляризации — гребневая регрессия

(ridge regression), или L 2 -регуляризация (L 2 -regularization), также иногда называ-

емая регуляризацией Тихонова (Tikhonov regularization). Она заключается в нало-

жении штрафа на сумму квадратов (евклидовой нормы) коэффициентов модели.

В данном случае штраф для модели будет равен:

где α — свободный параметр, служащий для управления уровнем штрафа. Этот тип

модели со штрафом встроен в библиотеку Scikit-Learn в виде оценивателя Ridge

(рис. 5.49):

In[12]: from sklearn.linear\_model import Ridge

model = make\_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))

basis\_plot(model, title='Ridge Regression') # Гребневая регрессия

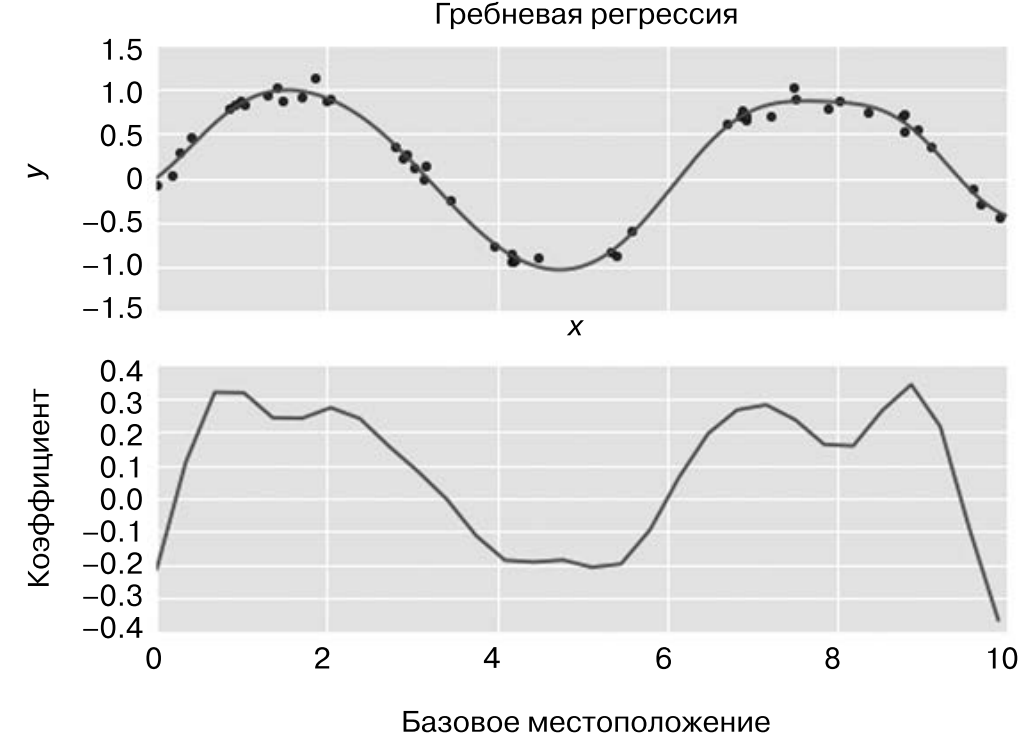


Рис. 5.8. Применение гребневой (L2 ) регуляризации к слишком сложной модели (ср. с рис. 5.7)

Параметр α служит для управления сложностью получаемой в итоге модели.

В предельном случае α → 0 мы получаем результат, соответствующий стандартной

линейной регрессии; в предельном случае α → ∞ будет происходить подавление лю-

бого отклика модели. Достоинства гребневой регрессии включают, помимо прочего,

возможность ее эффективного расчета — вычислительные затраты практически не

превышают затрат на расчет исходной линейной регрессионной модели.

Лассо-регуляризация (L 1 )

Еще один распространенный тип регуляризации — так называемая лассо-регуля-

ризация, включающая штрафование на сумму абсолютных значений (L 1 -норма)

коэффициентов регрессии:

Хотя концептуально эта регрессия очень близка к гребневой, результаты их могут

очень сильно различаться. Например, по геометрическим причинам лассо-регрес-

сия любит разреженные модели, то есть она по возможности делает коэффициенты

модели равными нулю.

Посмотреть на поведение этой регрессии мы можем, воспроизведя показанный на

рис. 5.49 график, но с использованием коэффициентов, нормализованных с по-

мощью нормы L 1 (рис. 5.50):

In[13]: from sklearn.linear\_model import Lasso

model = make\_pipeline(GaussianFeatures(30), Lasso(alpha=0.001))

basis\_plot(model, title='Lasso Regression') # Лассо-регуляризация

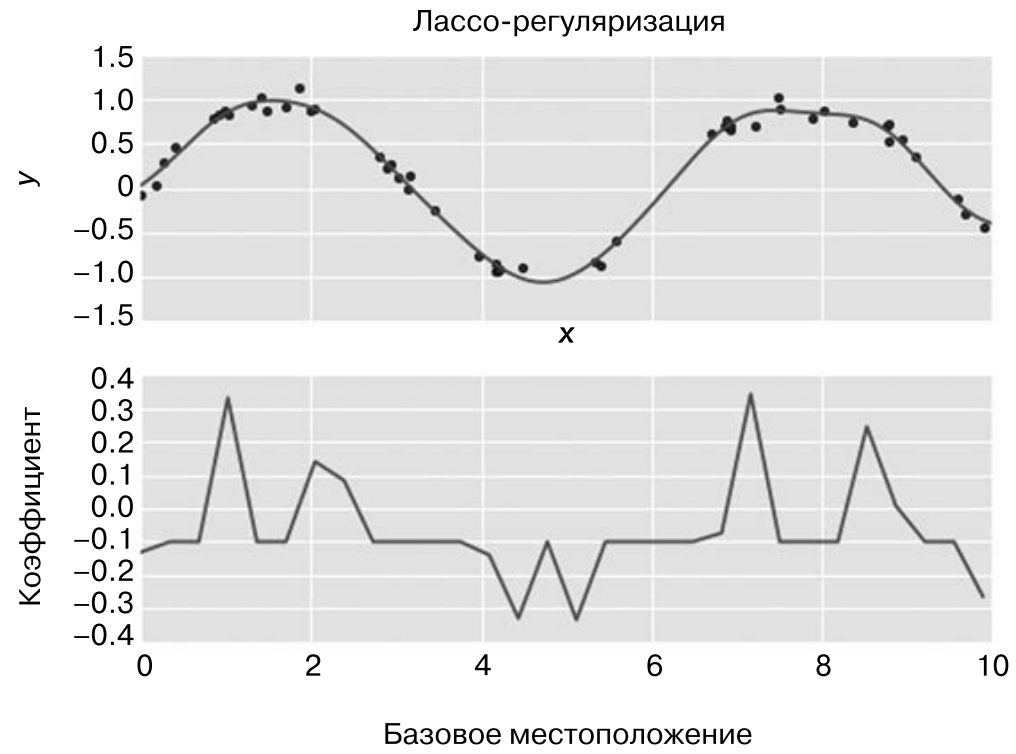


Рис. 5.8. Применение гребневой (L 2 ) регуляризации к слишком сложной модели (ср. с рис. 5.6)

При использовании штрафа лассо-регрессии большинство коэффициентов в точ-

ности равны нулю, а функциональное поведение моделируется небольшим под-

множеством из имеющихся базисных функций. Как и в случае гребневой регуля-

ризации, параметр α управляет уровнем штрафа и его следует определять путем

перекрестной проверки (см. раздел «Гиперпараметры и проверка модели» данной

главы).

Пример: предсказание велосипедного трафика

В качестве примера посмотрим, сможем ли мы предсказать количество пересека-

ющих Фримонтский мост в Сиэтле велосипедов, основываясь на данных о погоде,

времени года и других факторах. Мы уже работали с этими данными в разделе

«Работа с временными рядами» главы 3.