ГЛАВА 5

Методы опорных векторов

Метод опорных векторов (***Support Vector Machine*** — *SVM*) — это очень мощная и универсальная модель машинного обучения, способная выполнять линейную или нелинейную классификацию, регрессию и даже выявление выбросов. Она является одной из самых популярных моделей в МО, и любой интересующийся МО обязан иметь ее в своем инструментальном комплекте. Методы SVM особенно хорошо подходят для классификации сложных, но небольших или средних наборов данных.

В настоящей главе объясняются ключевые концепции методов SVM, способы их использования и особенности их работы.

Линейная классификация SVM

Фундаментальную идею, лежащую в основе методов SVM, лучше раскрывать с применением иллюстраций. На рис. 5.1 показана часть набора данных об ирисах, который был введен в конце главы 4. Два класса могут быть легко и ясно разделены с помощью прямой линии (они *линейно сепарабельные*).

На графике слева приведены границы решений трех возможных линейных классификаторов. Модель, граница решений которой представлена пунктирной линией, до такой степени плоха, что даже не разделяет классы надлежащим образом. Остальные две модели прекрасно работают на данном обучающем наборе, но их границы решений настолько близки к образцам, что эти модели, вероятно, не будут выполняться так же хорошо на новых образцах. По контрасту сплошной линией на графике справа обозначена граница решений классификатора SVM; линия не только разделяет два класса, но также находится максимально возможно далеко от ближайших обучающих образцов. Вы можете считать, что классификатор SVM устанавливает самую широкую, какую только возможно, полосу (представленную параллельными пунктирными линиями) между классами. Это называется *классификацией с широким зазором* (***large margin classification***).

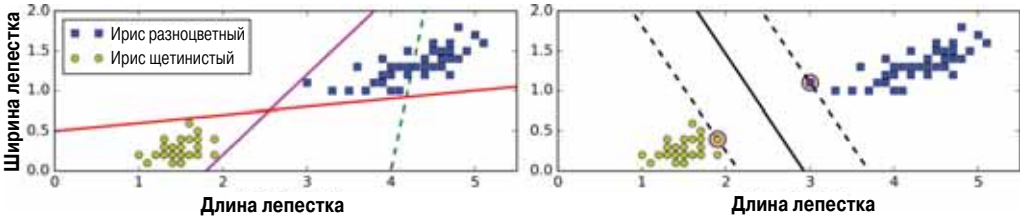


Рис. 5.1. Классификация с широким зазором

Обратите внимание, что добавление дополнительных обучающих образцов "вне полосы" вообще не будет влиять на границу решений: она полностью определяется образцами, расположенными по краям полосы (или "опирается" на них). Такие образцы называются *опорными векторами* (***support vector***); на рис. 5.1 они обведены окружностями.

Методы SVM чувствительны к масштабам признаков, как можно видеть на рис. 5.2: график слева имеет масштаб по вертикали, намного превышающий масштаб по горизонтали, поэтому самая широкая полоса близка к горизонтали. После масштабирования признаков (например, с использованием класса `StandardScaler` из Scikit-Learn) граница решений выглядит гораздо лучше (на графике справа).

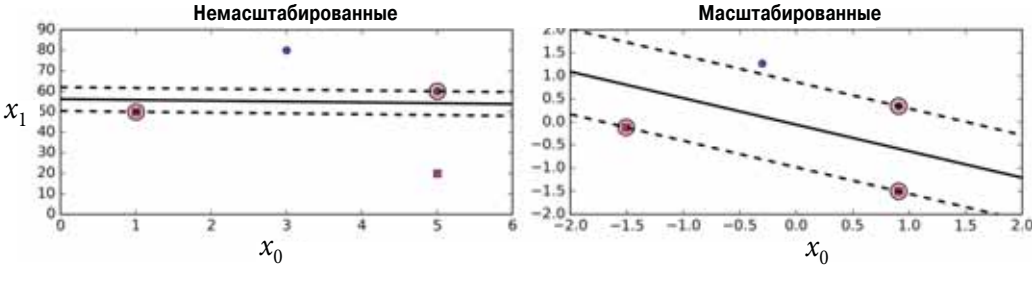


Рис. 5.2. Чувствительность к масштабам признаков

Классификация с мягким зазором

Если мы строго зафиксируем, что все образцы находятся вне полосы и на правой стороне, то получим *классификацию с жестким зазором* (***hard margin classification***). Классификации с жестким зазором присущи две главные проблемы. Во-первых, она работает, только если данные являются линейно сепарабельными. Во-вторых, она довольно чувствительна к выбросам.

На рис. 5.3 приведен набор данных об ирисах с только одним дополнительным выбросом: слева невозможно найти жесткий зазор, а справа граница решений сильно отличается от той, которую мы видели на рис. 5.1 без выброса, и модель, вероятно, не будет обобщаться с тем же успехом.

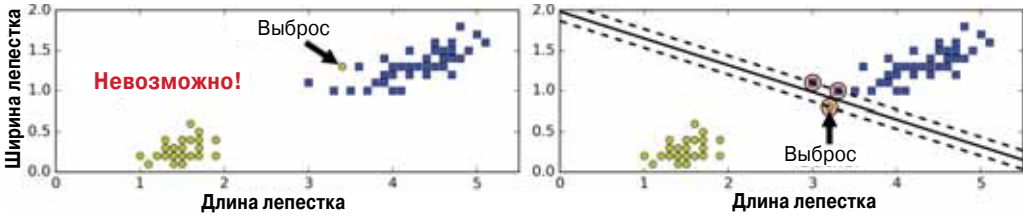


Рис. 5.3. Чувствительность к выбросам жесткого зазора

Чтобы избежать таких проблем, предпочтительнее применять более гибкую модель. Цель заключается в том, чтобы отыскать хороший баланс между удержанием полосы как можно более широкой и ограничением количества *нарушений зазора* (т.е. появления экземпляров, которые оказываются посредине полосы или даже на неправильной стороне). Это называется *классификацией с мягким зазором* (***soft margin classification***).

В классах SVM библиотеки Scikit-Learn вы можете управлять упомянутым балансом, используя гиперпараметр `C`: меньшее значение `C` ведет к более широкой полосе, но большему числу нарушений зазора. На рис. 5.4 показаны границы решений и зазоры двух классификаторов SVM с мягким зазором на нелинейно сепарабельном наборе данных. Слева за счет применения высокого значения `C` классификатор делает меньше нарушений зазора, но имеет меньший зазор. Справа из-за использования низкого значения `C` зазор гораздо больше, но многие образцы попадают на полосу. Однако, похоже, что второй классификатор будете лучше обобщаться: фактически даже на этом обучающем наборе он делает совсем немного ошибок в прогнозах, т.к. большинство нарушений зазора фактически находятся на корректной стороне границы решений.

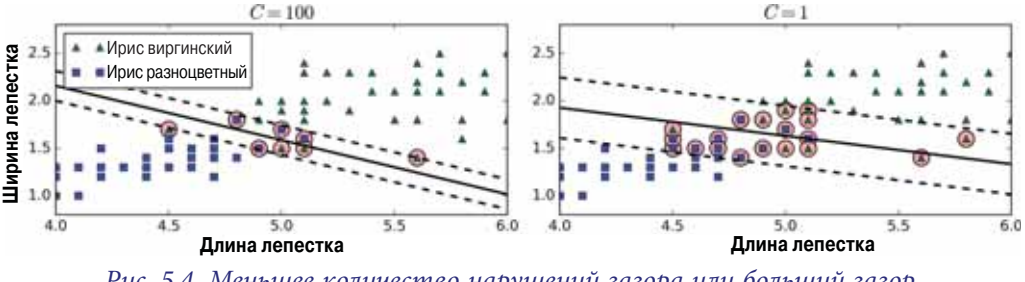


Рис. 5.4. Меньшее количество нарушений зазора или больший зазор

Если ваша модель SVM переобучается, тогда можете попробовать ее регуляризировать путем сокращения `C`.

Следующий код Scikit-Learn загружает набор данных об ирисах, масштабирует признаки и обучает линейную модель SVM (применяя класс `LinearSVC` с `C=1` и *петлевой* (*hinge loss*) функцией, которая вскоре будет описана) для выявления цветков ириса виргинского. Результирующая модель представлена справа на рис. 5.4.

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # длина лепестка, ширина лепестка
y = (iris["target"] == 2).astype(np.float64)  # ирис виргинский

svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge")),
    ])

svm_clf.fit(X, y)
```

Затем, как обычно, вы можете использовать модель для выработки прогнозов:

```
>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```

В отличие от классификаторов, основанных на логистической регрессии, классификаторы SVM не выдают вероятности для каждого класса.

В качестве альтернативы вы могли бы задействовать класс `SVC`, применив `SVC(kernel="linear", C=1)`, но он намного медленнее, особенно с крупными обучающими наборами, а потому не рекомендуется. Еще один вариант — воспользоваться классом `SGDClassifier` в форме `SGDClassifier(loss="hinge", alpha=1/(m*C))`. Здесь для обучения линейного классификатора SVM применяется стохастический градиентный спуск (см. главу 4). Он не сходится настолько быстро, как класс `LinearSVC`,

но может быть полезным для обработки гигантских наборов данных, которые не умещаются в памяти (внешнее обучение), или для решения задач динамической классификации.

Класс `LinearSVC` регуляризирует член смещения, так что вы обязаны сначала центрировать обучающий набор, вычтя его среднее значение. Если вы масштабируете данные с использованием класса `StandardScaler`, то это делается автоматически. Вдобавок удостоверьтесь в том, что установили гиперпараметр `loss` в `"hinge"`, т.к. указанное значение не выбирается по умолчанию. Наконец, для лучшей производительности вы должны установить гиперпараметр `dual` в `False`, если только не существуют дополнительные признаки кроме тех, что есть у обучающих образцов (мы обсудим двойственность позже в настоящей главе).

Нелинейная классификация SVM

Хотя линейные классификаторы SVM эффективны и работают на удивление хорошо в многочисленных случаях, многие наборы данных далеки от того, чтобы быть линейно сепарабельными. Один из подходов к обработке нелинейных наборов данных предусматривает добавление дополнительных признаков, таких как полиномиальные признаки (как делалось в главе 4); в ряде ситуаций результатом может оказаться линейно сепарабельный набор данных. Рассмотрим график слева на рис. 5.5: он представляет простой набор данных с только одним признаком x_1. Как видите, этот набор данных не является линейно сепарабельным. Но если вы добавите второй признак $x_2 = (x_1)^2$, тогда результирующий двумерный набор данных станет полностью линейно сепарабельным.

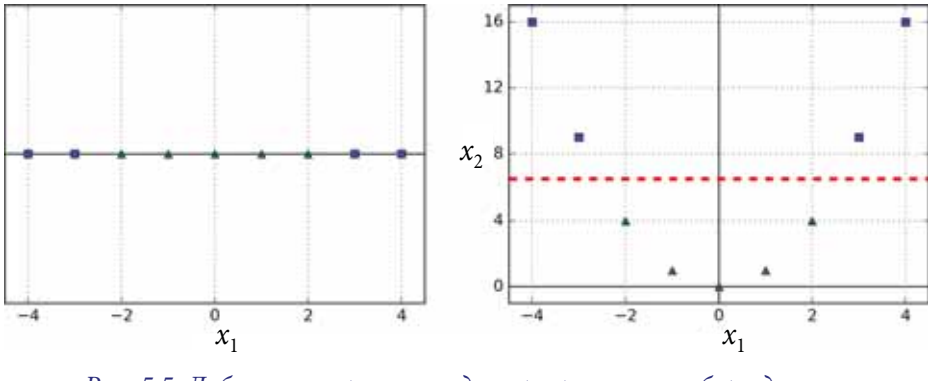


Рис. 5.5. Добавление признаков для превращения набора данных в линейно сепарабельный

Чтобы воплотить указанную идею с применением Scikit-Learn, вы можете создать экземпляр `Pipeline`, содержащий трансформатор `PolynomialFeatures` (как обсуждалось в разделе "Полиномиальная регрессия" главы 4), за которым следуют экземпляры `StandardScaler` и `LinearSVC`. Давайте проверим это на наборе данных `moons` (рис. 5.6):

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge"))
    ])

polynomial_svm_clf.fit(X, y)
```

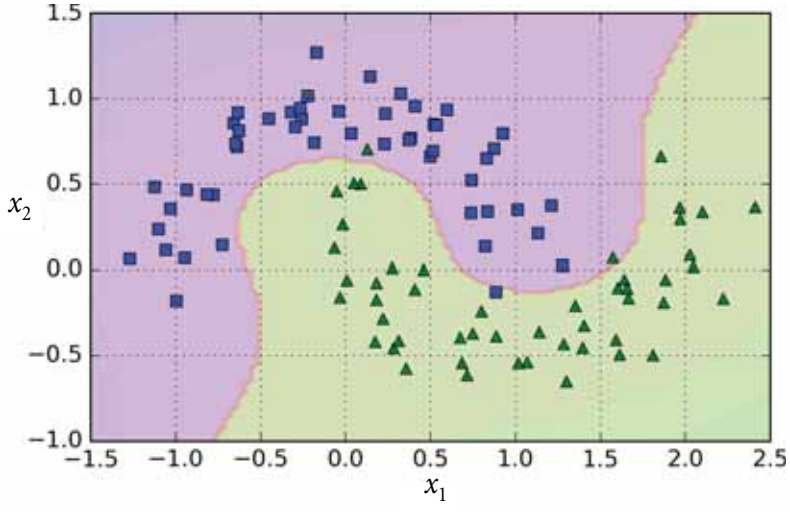


Рис. 5.6. Линейный классификатор SVM, использующий полиномиальные признаки

Полиномиальное ядро

Добавление полиномиальных признаков просто в реализации и может великолепно работать со всеми видами алгоритмов МО (не только с методами SVM), но при низкой полиномиальной степени оно не способно справляться с очень сложными наборами данных, а при высокой полиномиальной степени оно создает огромное количество признаков, делая модель крайне медленной.

К счастью, когда используются методы SVM, вы можете применить почти чудодейственный математический прием, называемый *ядерным трюком* (***kernel trick***), который вскоре будет объяснен. Он позволяет получить тот же самый результат, как если бы вы добавили много полиномиальных признаков, даже при полиномах очень высокой степени, без фактического их добавления. Таким образом, не происходит комбинаторного бурного роста количества признаков, поскольку в действительности вы не добавляете никаких признаков. Ядерный трюк выполняется классом `SVC`. Давайте проверим его на наборе данных `moons`:

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

Этот код обучает классификатор SVM, использующий полиномиальное ядро 3-й степени. Он представлен слева на рис. 5.7. Справа показан еще один классификатор SVM, который применяет полиномиальное ядро 10-й степени. Понятно, что если ваша модель переобучается, то вы можете сократить полиномиальную степень. И наоборот, если модель недообучается, тогда вы можете попробовать увеличить полиномиальную степень. Гиперпараметр `coef0` управляет тем, насколько сильно полиномы высокой степени влияют на модель в сравнении с полиномами низкой степени.

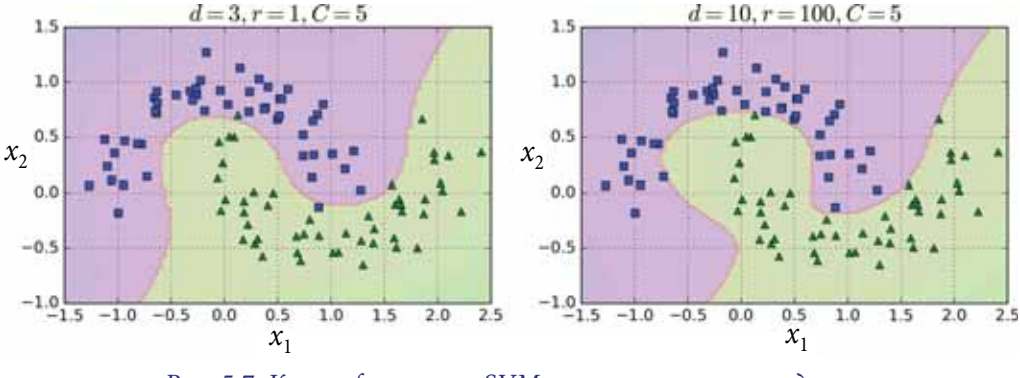


Рис. 5.7. Классификаторы SVM с полиномиальным ядром

Распространенный подход к поиску правильных значений гиперпараметров заключается в использовании решетчатого поиска (см. главу 2). Часто быстрее сначала сделать очень грубый решетчатый поиск, а затем провести более точный решетчатый поиск вокруг найденных лучших значений. Наличие хорошего представления о том, что фактически делает каждый гиперпараметр, также может помочь производить поиск в правильной части пространства гиперпараметров.

Добавление признаков близости

Еще одна методика решения нелинейных задач предусматривает добавление признаков, подсчитанных с применением *функции близости* (***similarity function***), которая измеряет, сколько сходства каждый образец имеет с отдельным *ориентиром* (***landmark***). Например, возьмем обсуждаемый ранее одномерный набор данных и добавим к нему два ориентира в $x_1 = -2$ и $x_1 = 1$ (график слева на рис. 5.8). Затем определим функцию близости как гауссову *радиальную базисную функцию* (***Radial Basis Function*** — ***RBF***) с $\gamma = 0.3$ (уравнение 5.1).

Уравнение 5.1. Гауссова функция RBF

$$\phi_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

Это колоколообразная функция, изменяющаяся от 0 (очень далеко от ориентира) до 1 (на ориентире). Теперь мы готовы вычислить новые признаки. Взглянем на образец $x_1 = -1$: он находится на расстоянии 1 от первого ориентира и расстоянии 2 от второго ориентира. Следовательно, его новыми признаками будут $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ и $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$. График справа на рис. 5.8 показывает трансформированный набор данных (с отбрасыванием первоначальных признаков). Как видите, он теперь линейно сепарабельный.

Вас может интересовать, как выбираются ориентиры. Простейший подход заключается в создании ориентира по местоположению каждого образца в наборе данных. При таком подходе создается много измерений и тем самым растут шансы того, что трансформированный набор данных будет линейно сепарабельным. Недостаток подхода в том, что обучающий набор с m образцами и n признаками трансформируется в обучающий набор с m образцами и m признаками (предполагая отбрасывание первоначальных признаков). Если обучающий набор очень крупный, тогда вы получите в равной степени большое количество признаков.

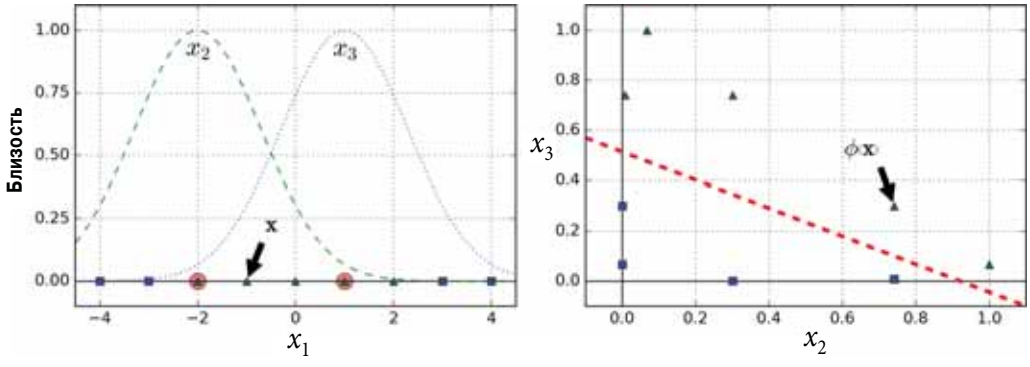


Рис. 5.8. Признаки близости, использующие гауссову функцию RBF

Гауссово ядро RBF

Подобно методу полиномиальных признаков метод признаков близости способен принести пользу любому алгоритму МО, но он может быть вычислительно затратным при подсчете всех дополнительных признаков, особенно в крупных обучающих наборах. Тем не менее, ядерный трюк снова делает свою "магию" SVM: он позволяет получить похожий результат, как если бы добавлялись многочисленные признаки близости, без фактического их добавления. Давайте испытаем *гауссово ядро RBF* (***Gaussian RBF kernel***) с применением класса `SVC`:

```
rbf_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ])
rbf_kernel_svm_clf.fit(X, y)
```

Модель представлена слева внизу на рис. 5.9. На других графиках изображены модели, обученные с разными значениями гиперпараметров `gamma` (γ) и `C`. Увеличение `gamma` приводит к сужению колоколообразной кривой (см. график слева на рис. 5.8), в результате чего сфера влияния каждого образца уменьшается: граница решений становится более неравномерной, извивающейся вблизи индивидуальных образцов. И наоборот, небольшое значение `gamma` делает колоколообразную кривую шире, поэтому образцы имеют бо́льшую сферу влияния, а граница решений оказывается более гладкой. Таким образом, γ действует аналогично гиперпараметру регуляризации: если ваша модель переобучается, тогда вы должны уменьшить значение `gamma`, а если недообучается — то увеличить его (подобно гиперпараметру `C`).

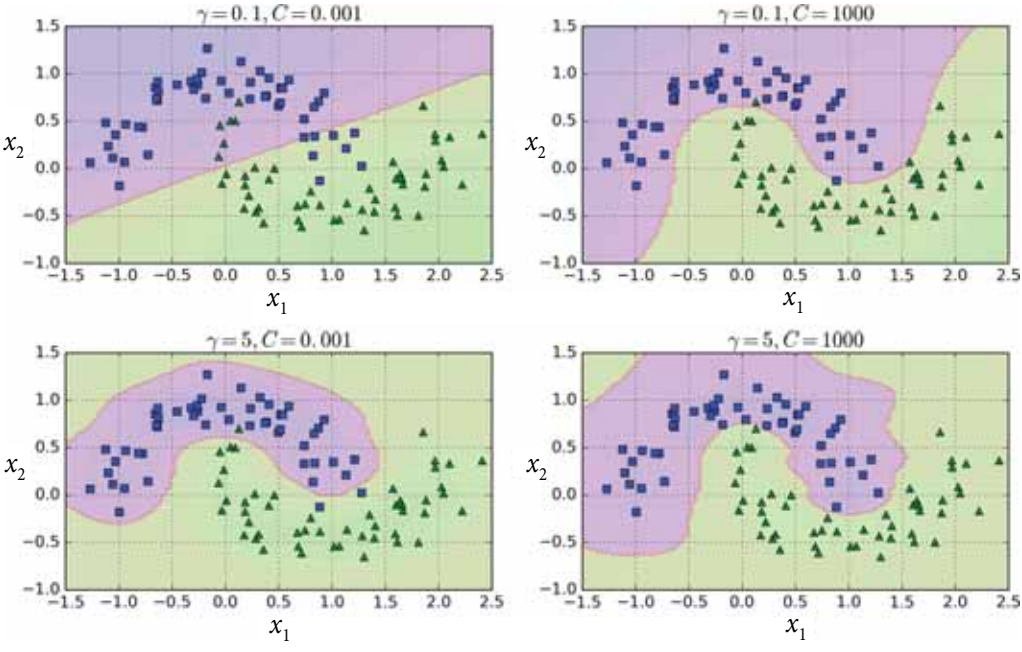


Рис. 5.9. Классификаторы SVM, использующие ядро RBF

Существуют и другие ядра, но они применяются гораздо реже. Например, некоторые ядра приспособлены к специфическим структурам данных. *Строковые ядра* (***string kernel***) иногда используются при классификации текстовых документов или цепочек ДНК (например, с применением *ядра строковых подпоследовательностей* (***string subsequence kernel***) или ядер на основе *расстояния Левенштейна* (***Levenshtein distance***)).

Имея на выбор так много ядер, как принять решение, какое ядро использовать? Примите в качестве эмпирического правила: вы должны всегда первым пробовать линейное ядро (помните, что `LinearSVC` гораздо быстрее `SVC(kernel="linear")`), особенно если обучающий набор очень большой либо изобилует признаками. Если обучающий набор не слишком большой, тогда вы должны испытать также гауссово ядро RBF; оно работает хорошо в большинстве случаев. При наличии свободного времени и вычислительной мощности вы также можете поэкспериментировать с рядом других ядер, применяя перекрестную проверку и решетчатый поиск, в особенности, когда существуют ядра, которые приспособлены к структуре данных вашего обучающего набора.

Вычислительная сложность

Класс `LinearSVC` основан на библиотеке *liblinear*, которая реализует оптимизированный алгоритм (`http://goo.gl/R635CH`) для линейных методов SVM[1]. Он не поддерживает ядерный трюк, но масштабируется почти линейно с количеством обучающих образцов и количеством признаков: сложность его времени обучения составляет ориентировочно $O(m \times n)$.

Алгоритм занимает больше времени, когда требуется очень высокая точность. Точность управляется гиперпараметром допуска ε (называется `tol` в Scikit-Learn). Большинству задач классификации подходит стандартный допуск.

Класс `SVC` основан на библиотеке *libsvm*, которая реализует алгоритм (`http://goo.gl/a8HkE3`), поддерживающий ядерный трюк[2]. Сложность времени обучения обычно находится между $O(m^2 \times n)$ и $O(m^3 \times n)$. К сожалению, это означает, что он становится невероятно медленным при большом количестве обучающих образцов (скажем, в случае сотен тысяч образцов). Такой алгоритм идеален для сложных, но небольших или средних обучающих наборов. Тем не менее, он хорошо масштабируется с количеством признаков, особенно *разреженных (sparse) признаков* (т.е. когда каждый образец имеет лишь немного ненулевых признаков). В этом случае алгоритм масштабируется приблизительно со средним числом ненулевых признаков на образец. В табл. 5.1 сравниваются классы классификации SVM из Scikit-Learn.

Таблица 5.1. Сравнение классов Scikit-Learn для классификации SVM

Класс	Сложность времени обучения	Поддержка внешнего обучения	Требуется ли масштабирование	Ядерный трюк
`LinearSVC`	$O(m \times n)$	Нет	Да	Нет
`SGDClassifier`	$O(m \times n)$	Да	Да	Нет
`SVC`	от $O(m^2 \times n)$ до $O(m^3 \times n)$	Нет	Да	Да

[1] “A Dual Coordinate Descent Method for Large-scale Linear SVM” (“Метод двойного покоординатного спуска для крупномасштабного линейного метода опорных векторов”), Лин и др. (2008 год).

[2] “Sequential Minimal Optimization (SMO)” (“Последовательная минимальная оптимизация”), Дж. Платт (1998 год).

Регрессия SVM

Как упоминалось ранее, алгоритм SVM довольно универсален: он поддерживает не только линейную и нелинейную классификацию, но также линейную и нелинейную регрессию. Прием заключается в инвертировании цели: вместо попытки приспособиться к самой широкой из возможных полосе между двумя классами, одновременно ограничивая нарушения зазора, регрессия SVM пробует уместить как можно больше образцов на полосе наряду с ограничением нарушений зазора (т.е. образцов *вне* полосы). Ширина полосы управляется гиперпараметром ε. На рис. 5.10 показаны две модели линейной регрессии SVM, обученные на случайных линейных данных, одна с широким зазором ($\varepsilon = 1.5$) и одна с узким зазором ($\varepsilon = 0.5$).

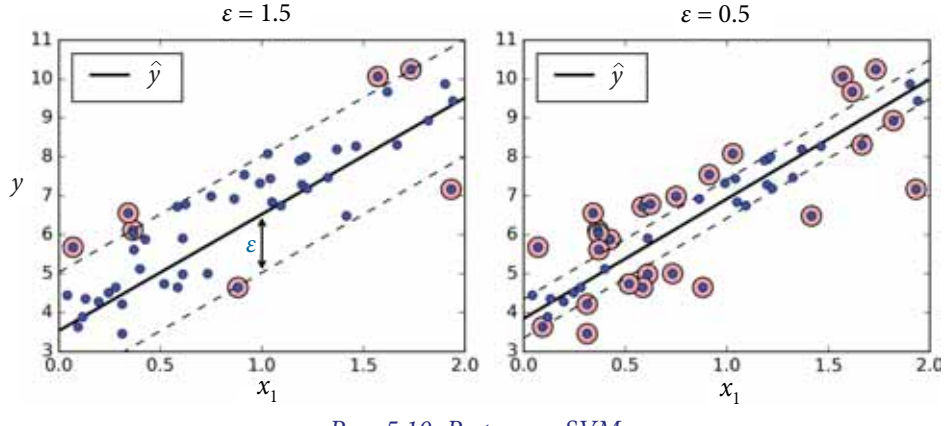


Рис. 5.10. Регрессия SVM

Добавление дополнительных обучающих образцов внутри зазора не влияет на прогнозы модели; соответственно, говорят, что модель *нечувствительна к ε*.

Для выполнения линейной регрессии SVM можно использовать класс `LinearSVR` из Scikit-Learn. Следующий код производит модель, представленную слева на рис. 5.10 (обучающие данные должны быть предварительно масштабированы и отцентрированы):

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

Для решения задач нелинейной регрессии можно применять *параметрически редуцированную (kernelized) модель SVM* (“kernelization” иногда пере-

водят как “кернелизация” — *примеч. пер.*). Например, на рис. 5.11 демонстрируется регрессия SVM на случайном квадратичном обучающем наборе, использующая полиномиальное ядро 2-го порядка. На графике слева производилось немного регуляризации (т.е. крупное значение C), а на графике справа — гораздо больше регуляризации (т.е. небольшое значение C).

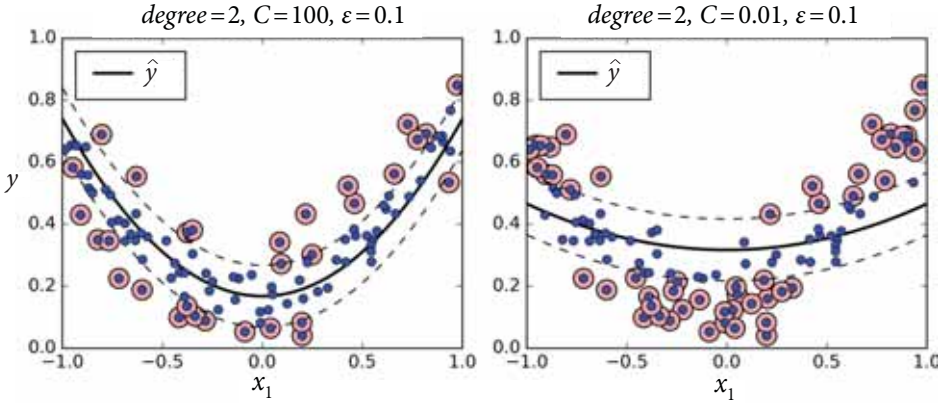


Рис. 5.11. Регрессия SVM, применяющая полиномиальное ядро 2-го порядка

Показанный ниже код порождает модель, представленную слева на рис. 5.11, с использованием класса SVR из Scikit-Learn (который поддерживает ядерный трюк). Класс SVR — это регрессионный эквивалент класса SVC, а класс LinearSVR — регрессионный эквивалент класса LinearSVC.

Класс LinearSVR масштабируется линейно с размером обучающего набора (подобно классу LinearSVC), в то время как класс SVR становится не в меру медленным, когда обучающий набор вырастает до крупного (подобно классу SVC).

```
from sklearn.svm import SVR
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

Методы SVM также могут применяться для выявления выбросов; за дополнительными сведениями обращайтесь в документацию Scikit-Learn.

Внутренняя кухня

В настоящем разделе объясняется, каким образом методы SVM вырабатывают прогнозы и как работают их обучающие алгоритмы, начиная с линейных классификаторов SVM. Если вы только начали изучать МО, тогда можете благополучно пропустить раздел и перейти прямо к упражнениям в конце главы. Позже, когда возникнет желание глубже понять методы SVM, вы всегда сможете сюда вернуться.

Прежде всего, пару слов об обозначениях: в главе 4 мы пользовались соглашением, которое предусматривало помещение всех параметров модели в один вектор θ, включающий член смещения θ_0 и исходные веса признаков от θ_1 до θ_n, а затем добавление ко всем образцам входного смещения $x_0 = 1$.

В этой главе мы будем применять другое соглашение, которое более удобно (и распространено), когда приходится иметь дело с методами SVM: член смещения будет называться b, а вектор весов признаков — $\mathbf{w}$. Никакое смещение к вектору исходных признаков добавляться не будет.

Функция решения и прогнозы

Модель линейной классификации SVM прогнозирует класс нового образца $\mathbf{x}$, просто вычисляя функцию решения $\mathbf{w}^T \cdot \mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b$: если результат положительный, то спрогнозированный класс $\hat{y}$ является положительным (1), а иначе — отрицательным (0); см. уравнение 5.2.

Уравнение 5.2. Прогноз линейного классификатора SVM

$$\hat{y} = \begin{cases} 0, \text{ если } \mathbf{w}^T \cdot \mathbf{x} + b < 0, \\ 1, \text{ если } \mathbf{w}^T \cdot \mathbf{x} + b \geq 0 \end{cases}$$

На рис. 5.12 показана функция решения, которая соответствует модели справа на рис. 5.4: это двумерная плоскость, т.к. набор данных имеет два признака (ширина лепестка и длина лепестка). Граница решений — множество точек, где функция решения равна 0: это пересечение двух плоскостей, которое является прямой (представленной толстой сплошной линией)[3].

[3] В более общих чертах, когда имеется n признаков, функция решения представляет собой n-мерную *гиперплоскость*, а граница решений — $(n-1)$-мерную гиперплоскость.

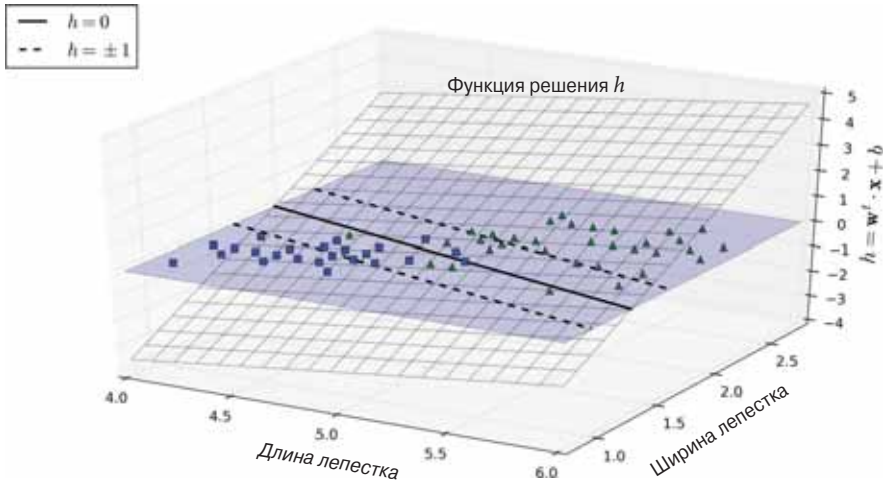


Рис. 5.12. Функция решения для набора данных об ирисах

Пунктирные линии представляют точки, где функция решения равна 1 или –1: они параллельны и находятся на одинаковом расстоянии от границы решений, формируя вокруг нее зазор. Обучение линейного классификатора SVM означает нахождение таких значений **w** и b, которые делают этот зазор как можно более широким, одновременно избегая нарушений зазора (жесткий зазор) или ограничивая их (мягкий зазор).

Цель обучения

Рассмотрим наклон функции решения: он тождественен норме вектора весов, $\| \mathbf{w} \|$. Если мы разделим наклон на 2, тогда точки, в которых функция решения равна ±1, будут в два раза дальше от границы решений. Другими словами, деление наклона на 2 умножит зазор на 2. Возможно, это легче представить себе в двумерном виде на рис. 5.13. Чем меньше вектор весов **w**, тем шире зазор.

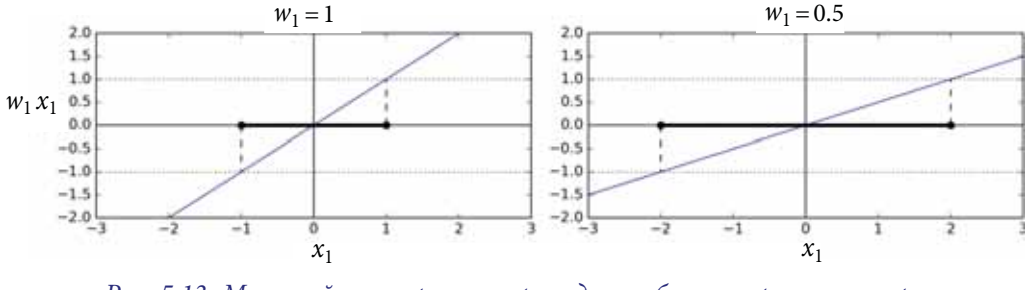


Рис. 5.13. Меньший вектор весов приводит к более широкому зазору

Итак, мы хотим довести до максимума $\|\mathbf{w}\|$, чтобы получить широкий зазор. Однако если мы также хотим избежать любых нарушений зазора (иметь жесткий зазор), то нужно, чтобы функция решения была больше 1 для всех положительных обучающих образцов и меньше –1 для отрицательных обучающих образцов. Если мы определим $t^{(i)} = -1$ для отрицательных образцов (когда $y^{(i)} = 0$) и $t^{(i)} = 1$ для положительных образцов (когда $y^{(i)} = 1$), тогда можем выразить такое ограничение как $t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1$ для всех образцов.

Следовательно, мы можем выразить цель линейного классификатора SVM с жестким зазором как задачу *условной оптимизации* (***constrained optimization***) в уравнении 5.3.

Уравнение 5.3. Цель линейного классификатора SVM с жестким зазором

$$\underset{\mathbf{w}, b}{\text{минимизировать}} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$$

$$\text{при условии} \quad t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{для} \quad i = 1, 2, \cdots, m$$

Мы минимизируем $\frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$, что равносильно $\frac{1}{2}\|\mathbf{w}\|^2$, вместо минимизации $\|\mathbf{w}\|$. Причина в том, что это даст тот же самый результат (поскольку значения $\mathbf{w}$ и b, которые минимизируют какое-то значение, также минимизируют половину его квадрата), но $\frac{1}{2}\|\mathbf{w}\|^2$ имеет подходящую и простую производную (именно $\mathbf{w}$), в то время как $\|\mathbf{w}\|$ не дифференцируется для $\mathbf{w} = 0$. Алгоритмы оптимизации гораздо лучше работают с дифференцируемыми функциями.

Чтобы достичь цели мягкого зазора, нам необходимо ввести *фиктивную переменную* (***slack variable***) $\zeta^{(i)} \geq 0$ для каждого образца[4]: $\zeta^{(i)}$ измеряет, насколько i-тому образцу разрешено нарушать зазор. Теперь мы имеем две противоречивые цели: делать фиктивные переменные как можно меньшими, чтобы сократить нарушения зазора, и делать $\frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$ как можно меньшим, чтобы расширить зазор. Именно здесь в игру вступает гиперпараметр C: он позволяет нам определить компромисс между указанными двумя целями. Мы получаем задачу условной оптимизации, представленную в уравнении 5.4.

[4] Дзета (ζ) — шестая буква греческого алфавита.

Уравнение 5.4. Цель линейного классификатора SVM с мягким зазором

$$\underset{\mathbf{w}, b, \zeta}{\text{минимизировать}} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^{m} \zeta^{(i)}$$

$$\text{при условии} \quad t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right) \geq 1 - \zeta^{(i)} \quad \text{и} \quad \zeta^{(i)} \geq 0 \quad \text{для} \quad i = 1, 2, \ldots, m$$

Квадратичное программирование

Задачи жесткого и мягкого зазора являются задачами выпуклой квадратичной оптимизации с линейными ограничениями. Такие задачи известны как задачи *квадратичного программирования* (*Quadratic Programming* — *QP*). Для решения задач QP доступны многие готовые решатели, использующие разнообразные методики, исследование которых выходит за рамки настоящей книги[5]. В уравнении 5.5 дана общая постановка задачи.

Уравнение 5.5. Задача квадратичного программирования

$$\underset{\mathbf{p}}{\text{минимизировать}} \quad \frac{1}{2}\mathbf{p}^T \cdot \mathbf{H} \cdot \mathbf{p} + \mathbf{f}^T \cdot \mathbf{p}$$

$$\text{при условии} \quad \mathbf{A} \cdot \mathbf{p} \leq \mathbf{b}$$

где

- $\mathbf{p}$ — n_p-размерный вектор (n_p = количество параметров),
- $\mathbf{H}$ — матрица $n_p \times n_p$,
- $\mathbf{f}$ — n_p-размерный вектор,
- $\mathbf{A}$ — матрица $n_c \times n_p$ (n_c = количество ограничений),
- $\mathbf{b}$ — n_c-размерный вектор.

Обратите внимание, что выражение $\mathbf{A} \cdot \mathbf{p} \leq \mathbf{b}$ фактически определяет n_c ограничений: $\mathbf{p}^T \cdot \mathbf{a}^{(i)} \leq b^{(i)}$ для $i = 1, 2, \ldots, n_c$, где $\mathbf{a}^{(i)}$ — вектор, содержащий элементы i-той строки $\mathbf{A}$, а $b^{(i)}$ — i-тый элемент $\mathbf{b}$.

Вы можете легко проверить, что если установить параметры QP следующим образом, то достигается цель линейного классификатора SVM с жестким зазором:

---

[5] Чтобы узнать больше о квадратичном программировании, можете начать с чтения книги Стивена Бойда и Ливена Ванденберга *Convex Optimization* (Cambridge University Press, 2004 год) (`http://goo.gl/FGXuLw`) или просмотреть курс видеолекций (на английском языке) от Ричарда Брауна (`http://goo.gl/rTo3Af`).

- $n_p = n + 1$, где n — количество признаков (+1 предназначено для члена смещения);
- $n_c = m$, где m — количество обучающих образцов;
- $\mathbf{H}$ — единичная матрица $n_p \times n_p$ кроме нуля в левой верхней ячейке (чтобы проигнорировать член смещения);
- $\mathbf{f} = \mathbf{0}$, n_p-размерный вектор, заполненный нулями;
- $\mathbf{b} = \mathbf{1}$, n_c-размерный вектор, заполненный единицами;
- $\mathbf{a}^{(i)} = -t^{(i)} \dot{\mathbf{X}}^{(i)}$, где $\dot{\mathbf{X}}^{(i)}$ тождественно $\mathbf{x}^{(i)}$ с добавочным признаком смещения $\dot{\mathbf{x}}_0$.

Таким образом, один из способов обучения линейного классификатора SVM с жестким зазором предусматривает просто применение готового решателя QP с передачей ему предшествующих параметров. Результирующий вектор $\mathbf{p}$ будет содержать член смещения $b = p_0$ и веса признаков $w_i = p_i$ для $i = 1, 2, \ldots, m$. Аналогично вы можете использовать решатель QP для решения задачи мягкого зазора (взгляните на упражнения в конце главы).

Тем не менее, чтобы применить ядерный трюк, мы собираемся рассмотреть другую задачу условной оптимизации.

Двойственная задача

Имея задачу условной оптимизации, известную как *прямая задача* (***primal problem***), можно выразить отличающуюся, но тесно связанную задачу, которая называется *двойственной задачей* (***dual problem***). Решение двойственной задачи обычно дает нижнюю границу решения прямой задачи, но при некоторых условиях двойственная задача может даже иметь те же самые решения, что и прямая задача. К счастью, задача SVM удовлетворяет этим условиям[6], так что вы можете выбирать, решать прямую задачу или двойственную задачу; обе они будут иметь то же самое решение. В уравнении 5.6 показана двойственная форма цели линейного классификатора SVM (если вас интересует, как выводить двойственную задачу из прямой задачи, тогда обратитесь в приложение B).

[6] Целевая функция является выпуклой, а ограничения неравенства представляют собой непрерывно дифференцируемые и выпуклые функции.

Уравнение 5.6. Двойственная форма цели линейного классификатора SVM

$$\underset{\alpha}{\text{минимизировать}} \quad \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\mathbf{x}^{(i)T}\cdot\mathbf{x}^{(j)} - \sum_{i=1}^{m}\alpha^{(i)}$$

$$\text{при условии} \quad \alpha^{(i)} \geq 0 \quad \text{для} \quad i = 1, 2, \ldots, m$$

После нахождения вектора $\hat{\alpha}$, сводящего к минимуму это уравнение (используя решатель QP), с применением уравнения 5.7 вы можете вычислить $\widehat{\mathbf{w}}$ и $\hat{b}$, которые минимизируют прямую задачу.

Уравнение 5.7. От решения двойственной задачи к решению прямой задачи

$$\widehat{\mathbf{w}} = \sum_{i=1}^{m}\hat{\alpha}^{(i)}t^{(i)}\mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \widehat{\mathbf{w}}^{T}\cdot\mathbf{x}^{(i)}\right)$$

Двойственная задача решается быстрее прямой, когда количество обучающих образцов меньше количества признаков. Что более важно, становится возможным ядерный трюк, в то время как при решении прямой задачи он невозможен. Итак, что же собой представляет этот самый ядерный трюк?

Параметрически редуцированные методы SVM

Предположим, что вы хотите применять полиномиальную трансформацию второй степени к двумерному обучающему набору (такому как `moons`) и затем обучать линейный классификатор SVM на трансформированном обучающем наборе. В уравнении 5.8 показана *полиномиальная отображающая функция* (*mapping function*) второй степени ϕ, которую желательно применять.

Уравнение 5.8. Полиномиальное отображение второй степени

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} {x_1}^2 \\ \sqrt{2}\,x_1x_2 \\ {x_2}^2 \end{pmatrix}$$

Обратите внимание, что трансформированный вектор является трехмерным, а не двумерным. Давайте посмотрим, что получится в результате

применения такого полиномиального отображения второй степени к паре двумерных векторов, **a** и **b**, и вычисления скалярного произведения трансформированных векторов (уравнение 5.9).

Уравнение 5.9. Ядерный трюк для полиномиального отображения второй степени

$$\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = \begin{pmatrix} a_1^2 \\ \sqrt{2}\, a_1 a_2 \\ a_2^2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1^2 \\ \sqrt{2}\, b_1 b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2 a_1 b_1 a_2 b_2 + a_2^2 b_2^2$$

$$= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = \left(\mathbf{a}^T \cdot \mathbf{b}\right)^2$$

Как вам это? Скалярное произведение трансформированных векторов равно квадрату скалярного произведения исходных векторов:

$$\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2.$$

Теперь о сути: если вы примените трансформацию ϕ ко всем обучающим образцам, тогда двойственная задача (см. уравнение 5.6) будет содержать скалярное произведение $\phi(\mathrm{x}^{(i)})^T \cdot \phi(\mathrm{x}^{(j)})$. Но если ϕ — полиномиальная трансформация второй степени, определенная в уравнении 5.8, то вы можете заменить это скалярное произведение трансформированных векторов просто на $\left(\mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)}\right)^2$. Таким образом, в действительности вы вообще не нуждаетесь в трансформации обучающих образцов: всего лишь замените скалярное произведение в уравнении 5.6 его квадратом. Результат будет абсолютно тем же самым, как если бы вы не поленились фактически трансформировать обучающий набор и затем подогнали какой-то линейный алгоритм SVM, но такой трюк делает весь процесс гораздо более эффективным с вычислительной точки зрения. В этом заключается сущность ядерного трюка.

Функция $K(\mathbf{a}, \mathbf{b}) = (\mathrm{a}^T \cdot \mathrm{b})^2$ называется *полиномиальным ядром* второй степени. В машинном обучении *ядро* (***kernel***) — это функция, которая способна вычислять скалярное произведение $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$, базируясь только на исходных векторах **a** и **b**, без необходимости в вычислении трансформации ϕ (или даже знании о ней).

В уравнении 5.10 приведен список самых распространенных ядер.

Уравнение 5.10. Распространенные ядра

$$\text{Линейное: } K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathrm{b}$$

$$\text{Полиномиальное: } K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \cdot \mathbf{b} + r)^d$$

$$\text{Гауссово RBF: } K(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma \| \mathrm{a} - \mathrm{b} \|^2\right)$$

$$\text{Сигмоидальное: } K(\mathbf{a}, \mathbf{b}) = \tanh\left(\gamma \mathbf{a}^T \cdot \mathbf{b} + r\right)$$

Теорема Мерсера

Согласно *теореме Мерсера* (*Mercer's theorem*), если функция $K(\mathbf{a}, \mathbf{b})$ соблюдает несколько математических условий, называемых *условиями Мерсера* (K должна быть непрерывной, симметричной в своих аргументах, так что $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, и т.д.), тогда существует функция ϕ, которая отображает **a** и **b** на другое пространство (возможно, с гораздо большей размерностью), такое что $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$. Таким образом, вы можете использовать K как ядро, поскольку знаете, что ϕ существует, даже если вам неизвестно, что собой представляет ϕ. В случае гауссова ядра RBF можно показать, что ϕ фактически отображает каждый обучающий образец на бесконечномерное пространство, поэтому хорошо, что вам не придется действительно выполнять отображение!

Следует отметить, что некоторые часто употребляемые ядра (такие как сигмоидальное ядро) не соблюдают все условия Мерсера, но на практике в целом работают нормально.

По-прежнему осталась одна загвоздка, которую нужно уладить. Уравнение 5.7 показывает, как перейти от двойственного решения к прямому решению в случае линейного классификатора SVM, но если вы применяете ядерный трюк, то оказываетесь с уравнениями, которые включают $\phi(\mathrm{x}(i))$. На самом деле $\widehat{\mathbf{w}}$ обязано иметь то же количество измерений, что и $\phi(\mathrm{x}(i))$, которое может быть гигантским или даже бесконечным, поэтому вы не будете в состоянии вычислить его. Но как можно вырабатывать прогнозы, не зная $\widehat{\mathbf{w}}$? Хорошая новость в том, что вы можете включить формулу для $\widehat{\mathbf{w}}$ из уравнения 5.7 в функцию решения для нового образца $\mathbf{x}^{(n)}$ и получить уравнение с только скалярными произведениями между входными векторами. Это снова делает возможным использование ядерного трюка (уравнение 5.11).

Уравнение 5.11. Выработка прогнозов с помощью параметрически редуцированного метода SVM

$$h_{\widehat{\mathbf{W}}, \hat{b}}\left(\phi\left(\mathbf{x}^{(n)}\right)\right) = \widehat{\mathbf{W}}^T \cdot \phi\left(\mathbf{x}^{(n)}\right) + \hat{b} = \left(\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \phi\left(\mathbf{x}^{(i)}\right)\right)^T \cdot \phi\left(\mathbf{x}^{(n)}\right) + \hat{b}$$

$$= \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \left(\phi\left(\mathbf{x}^{(i)}\right)^T \cdot \phi\left(\mathbf{x}^{(n)}\right)\right) + \hat{b}$$

$$= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \hat{\alpha}^{(i)} t^{(i)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}\right) + \hat{b}$$

Обратите внимание, что поскольку $\alpha^{(i)} \neq 0$ лишь для опорных векторов, выработка прогнозов включает в себя вычисление скалярного произведения нового входного вектора $\mathbf{x}^{(n)}$ только с опорными векторами, а не со всеми обучающими образцами. Конечно, вам также необходимо подсчитать член смещения $\hat{b}$, применяя тот же трюк (уравнение 5.12).

Уравнение 5.12. Вычисление члена смещения с использованием ядерного трюка

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(1 - t^{(i)} \widehat{\mathbf{w}}^T \cdot \phi\left(\mathbf{x}^{(i)}\right)\right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(1 - t^{(i)} \left(\sum_{j=1}^{m} \hat{\alpha}^{(j)} t^{(j)} \phi\left(\mathbf{x}^{(j)}\right)\right)^T \cdot \phi\left(\mathbf{x}^{(i)}\right)\right)$$

$$= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(1 - t^{(i)} \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^{m} \hat{\alpha}^{(j)} t^{(j)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)\right)$$

Если у вас разболелась голова, то это совершенно нормально: таков печальный побочный эффект от ядерного трюка.

Динамические методы SVM

Прежде чем завершить главу, давайте мельком взглянем на динамические классификаторы SVM (вспомните, что динамическое обучение означает постепенное обучение, обычно по мере поступления новых образцов).

Для линейных классификаторов SVM один из методов предусматривает применение градиентного спуска (например, используя `SGDClassifier`) для сведения к минимуму функции издержек в уравнении 5.13, которое является производным от прямой задачи. К сожалению, она сходится намного медленнее, чем методы, основанные на QP.

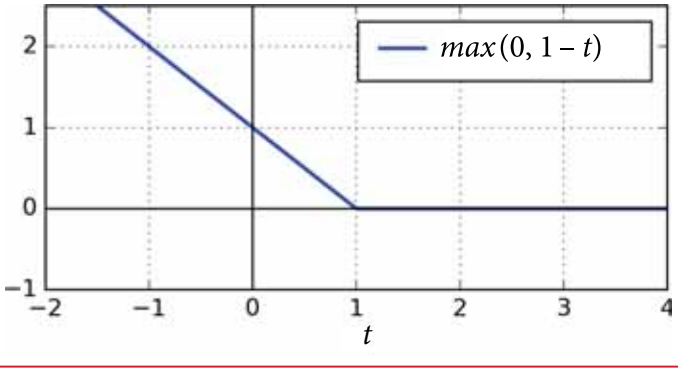
Уравнение 5.13. Функция издержек линейного классификатора SVM

$$J(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^{m} max\left(0, 1 - t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right)\right)$$

Первая сумма в функции издержек вынудит модель иметь небольшой вектор весов **w**, приводя к более широкому зазору. Вторая сумма подсчитывает общее количество нарушений зазора. Нарушение зазора образца равно 0, если он расположен вне полосы на корректной стороне, либо иначе пропорционально расстоянию до корректной стороны полосы. Сведение к минимуму этого члена гарантирует, что модель будет нарушать зазор мало и редко.

Петлевая функция

Функция $max(0, 1-t)$ называется *петлевой* (***hinge loss***) и представлена ниже. Она равна 0, когда $t \geq 1$. Ее производная (наклон) равна –1, если $t < 1$, и 0, если $t > 1$. Петлевая функция не является дифференцируемой при $t = 1$, но подобно лассо-регрессии (см. раздел "Лассо-регрессия" в главе 4) вы по-прежнему можете применять градиентный спуск, используя любой *субдифференциал* при $t = 1$ (т.е. любое значение между –1 и 0).



Также возможно реализовать динамические параметрически редуцированные методы SVM — например, руководствуясь работами "Инкрементное и декрементное обучение SVM" (`http://goo.gl/JEqVui`)[7] или "Быстрые параметрически редуцированные классификаторы с динамическим

[7] "Incremental and Decremental Support Vector Machine Learning" ("Инкрементное и декрементное обучение методами опорных векторов"), Ж. Кофенбергс, Т. Поджо (2001 год).

и активным обучением" (https://goo.gl/hsoUHA)[8]. Однако они реализованы в Matlab и C++. Для крупномасштабных нелинейных задач вы можете обдумать применение нейронных сетей (см. часть II).

Упражнения

1. Какая фундаментальная идея лежит в основе методов опорных векторов?
2. Что такое опорный вектор?
3. Почему важно масштабировать входные образцы при использовании методов SVM?
4. Может ли классификатор SVM выдать меру доверия, когда он классифицирует образец? Как насчет вероятности?
5. Какую форму задачи SVM — прямую или двойственную — вы должны применять для обучения модели на обучающем наборе с миллионами образцов и сотнями признаков?
6. Пусть вы обучаете классификатор SVM с ядром RBF. Кажется, он недообучается на обучающем наборе: вам следует увеличить или же уменьшить γ (`gamma`)? Что скажете о `C`?
7. Как вы должны установить параметры QP (**H**, **f**, **A** и **b**), чтобы решить задачу линейного классификатора SVM с мягким зазором, используя готовый решатель QP?
8. Обучите классификатор `LinearSVC` на линейно сепарабельном наборе данных. Затем обучите на том же наборе данных классификаторы `SVC` и `SGDClassifier`. Посмотрите, можете ли вы заставить их выдавать примерно одинаковые модели.
9. Обучите классификатор SVM на наборе данных MNIST. Поскольку классификаторы SVM являются двоичными, для классификации всех 10 цифр вам придется применять стратегию "один против всех". Вы можете решить отрегулировать гиперпараметры с использованием небольшого проверочного набора, чтобы ускорить процесс. Какой правильности вы можете достичь?
10. Обучите регрессор SVM с помощью набора данных, содержащего цены на жилье в Калифорнии.

Решения приведенных упражнений доступны в приложении A.

[8] "Fast Kernel Classifiers with Online and Active Learning" ("Быстрые параметрически редуцированные классификаторы с динамическим и активным обучением") А. Борд, С. Эртекин, Д. Вестон, Л. Ботту (2005 год).