**Тезисы главы 3 «Машинного обучения без учителя»**

**Два вида машинного обучения без учителя**: неконтролируемые преобразования, алгоритмы кластеризации.

**Неконтролируемые преобразования**

Применение неконтролируемых преобразований:

1. Сокращение размерности;
2. Поиск компонент из которых состоят данные (выделение тем из коллекций текстовых документов).

Алгоритмы машинного обучения без учителя часто используются:

1. в разведочных целях, когда специалист хочет лучше изучить сами данные
2. они служат этапом предварительной обработки данных для алгоритмов машинного обучения с учителем

Одним из самых простых и наиболее широко используемых алгоритмов не контролируемого обучения является **анализ главных компонентов** (principal component analysis, **PCA**).

*Анализ главных компонент (PCA) используется для* ***сокращения размерности*** *и* ***выделение признаков***.

Работа алгоритма **PCA** основана на вращение данных, а затем удаление направлений, объясняющих незначительную дисперсию данных. Важно отметить, что РСА является методом машинного обучения без учителя и не использует какой-либо информации о классах при поиске поворота. Он просто анализирует корреляционные связи в данных. Корреляционная связь говорит о взаимосвязанности данных параметров.

Для **сокращения размерности** до 2-х измерений, используют PCA c первыми двумя главными компонентами. Идея, лежащая в основе выделения признаков, заключается в поиске нового представления данных, которое в отличие от исходного лучше подходит для анализа.

**Выделение признаков** методом «**Собственных лиц**» заключается в использовании алгоритма PCA как преобразования способом поворота данных с последующим удалением компонент («выбеливание»), имеющих **низкую дисперсию**.

**Факторизация неотрицательных матриц (NMF)** – еще один алгоритм машинного обучения без учителя, цель которого – выделить полезные характеристики. Он работает так же, как PCA, а также его можно использовать для уменьшения размерности.

Факторизация — это операция разложения объекта (число, матрица) на его простые составляющие.

**t-SNE** -- ещё один алгоритм неконтролируемого преобразования – это множественное обучение. Существует класс алгоритмов визуализации, называемых алгоритмами множественного обучения (manifold learning algorithms), которые используют гораздо более сложные графические представления данных и позволяют получить визуализации лучшего качества. Особенно полезным является алгоритм t-SNE.

Алгоритмы множественного обучения в основном направлены на визуализацию и поэтому редко используются для получения более двух новых характеристик. Некоторые из них, в том числе t-SNE, создают новое представление обучающих данных, но при этом не осуществляют преобразования новых данных. Это означает, что данные алгоритмы **нельзя применить к тестовому набору**, они могут преобразовать лишь те данные, на которых они были обучены. Множественное обучение может использоваться для разведочного анализа данных, но редко используется в тех случаях, когда конечной целью является применение модели машинного обучения с учителем. Идея, лежащая в основе алгоритма t-SNE, заключается в том, чтобы найти двумерное представление данных, сохраняющее расстояния между точками наилучшим образом. t-SNE начинает свою работу со случайного двумерного представления каждой точки данных, а затем пытается сблизить точки, которые в пространстве исходных признаков находятся близко друг к другу, и отдаляет друг от друга точки, которые находятся далеко друг от друга. При этом t-SNE уделяет большее внимание сохранению расстояний между точками, близко расположенными друг к другу. Иными словами, он пытается сохранить информацию, указывающую на то, какие точки являются соседями друг другу.

Имейте в виду, что этот метод не использует информацию о метках классов: он является полностью неконтролируемым. Тем не менее он может найти двумерное представление данных, которое четко разграничивает классы, используя лишь информацию о расстояниях между точками данных в исходном пространстве.

**Кластеризация**

**Кластеризация k-средних** – один из самых простых и наиболее часто используемых алгоритмов кластеризации. Сначала выбирается число кластеров k. После выбора значения k алгоритм k-средних отбирает точки, которые будут представлять центры кластеров (cluster centers). Затем для каждой точки данных вычисляется его евклидово расстояние до каждого центра кластера. Каждая точка назначается ближайшему центру кластера. Алгоритм вычисляет центроиды (centroids) – центры тяжести кластеров. Каждый центроид – это вектор, элементы которого представляют собой средние значения характеристик, вычисленные по всем точкам кластера. Центр кластера смещается в его центроид. Точки заново назначаются ближайшему центру кластера. Этапы изменения центров кластеров и переназначения точек итеративно повторяются до тех пор, пока границы кластеров и расположение центроидов не перестанут изменяться, т.е. на каждой итерации в каждый кластер будут попадать одни и те же точки данных.

**Недостатки алгоритма** **k-средних.**

Каждый кластер определяется исключительно его центром, это означает, что каждый кластер имеет выпуклую форму. В результате этого алгоритм k-средних может описать относительно простые формы. Кроме того, алгоритм k-средних предполагает, что все кластеры в определенном смысле имеют одинаковый «диаметр», он всегда проводит границу между кластерами так, чтобы она проходила точно посередине между центрами кластеров.

Кроме того, алгоритм k-средних плохо работает, когда кластеры имеют более сложную форму. Например, случай двух кластеров в форме полумесяцев.

**Алгомеративная кластеризация** (agglomerative clustering) относится к семейству алгоритмов кластеризации, в основе которых лежат одинаковые принципы: алгоритм начинает свою работу с того, что каждую точку данных заносит в свой собственный кластер и по мере выполнения объединяет два наиболее схожих между собой кластера до тех пор, пока не будет удовлетворен определенный критерий остановки.

Критерий остановки, реализованный в scikit-learn – это количество кластеров, поэтому схожие между собой кластеры объединяются до тех пор, пока не останется заданное число кластеров. Есть несколько критериев связи (linkage), которые задают точный способ измерения «наиболее схожего кластера». В основе этих критериев лежит расстояние между двумя существующими кластерами.

Изначально количество кластеров равно количеству точек данных. Затем на каждом шаге объединяются два ближайших друг к другу кластера.

Из-за своего способа работы алгоритм агломеративной кластеризации **не может вычислить прогнозы для новых точек** данных. Поэтому алгоритм агломеративной кластеризации не имеет метода predict.

**Иерархическая кластеризация.**

Результатом агломеративной кластеризации является иерархическая кластеризация (hierarchical clustering). Кластеризация выполняется итеративно, и каждая точка совершает путь от отдельной точки-кластера до участника итогового кластера. На каждом промежуточном шаге происходит кластеризация данных (с разным количеством кластеров).

К сожалению, алгоритм агломеративной кластеризации по-прежнему не в состоянии обработать сложные данные типа набора two\_moons. Чего нельзя сказать о DBSCAN

**DBSCAN** – плотностный алгоритм кластеризации пространственных данных с присутствием шума.

Основные преимущества алгоритма DBSCAN заключаются в том, что пользователю *не нужно заранее задавать количество кластеров*, алгоритм может выделить кластеры сложной формы и способен определить точки, которые не принадлежат какому-либо кластеру. DBSCAN работает немного медленнее, чем алгоритм агломеративной кластеризации и алгоритм k-средних, но также может масштабироваться на относительно большие наборы данных.

DBSCAN определяет точки, расположенные в «густонаселенных» областях пространства характеристик, когда многие точки данных расположены близко друг к другу. Эти области называются плотными (dense) областями пространства характеристик. Идея алгоритма DBSCAN заключается в том, что *кластеры образуют плотные области данных, которые отделены друг от друга относительно пустыми областями*.

Параметр **eps** чуть более важен, поскольку он определяет, что подразумевается под «близостью» точек друг к другу. Очень маленькое значение eps будет означать отсутствие ядерных точек и может привести к тому, что все точки будут помечены как шумовые. Очень большое значение eps приведет к тому, что все точки сформируют один кластер.

Значение **min\_samples** главным образом определяет, будут ли точки, расположенные в менее плотных областях, помечены как выбросы или как кластеры. Если увеличить значение min\_samples, все, что могло бы стать кластером с количеством точек, не превышающим min\_samples, будет помечено как шум. Поэтому значение min\_samples задает минимальный размер кластера.

Несмотря на то, что в алгоритме **DBSCAN** не нужно явно указывать количество кластеров, значение eps неявно задает количество выделяемых кластеров. Иногда подобрать оптимальное значение eps становится проще после масштабирования данных с помощью StandardScaler или MinMaxScaler, так как использование этих методов масштабирования гарантирует, что все характеристики будут иметь одинаковый масштаб.

**Тезисы главы 4 «Типы данных и конструирование признаков»**

Все признаки делятся на **непрерывные** и **категорийные**. Непрерывные признаки представлены числом с плавающей точкой. Категорийные признаки известны как дискретные, поскольку не имеют числовых значений.

Вопрос оптимальной подготовки данных для конкретного прикладного применения известен под названием feature engineering (**конструирование признаков**) и является одной из главных задач для специалистов по машинному обучению, пытающихся решить реальные проблемы.

**Прямое кодирование (дамми-переменные).**

На сегодняшний момент наиболее распространенным способом представления категориальных переменных является прямое кодирование или, если перевести дословно, кодирование с одним горячим состоянием (one-hot-encoding или one-out-of-N encoding). Идея, лежащая в основе прямого кодирования, заключается в том, чтобы заменить категориальную переменную одной или несколькими новыми признаками, которые могут принимать значения 0 и 1. Значения 0 и 1 придают смысл формуле линейной бинарной классификации (а также всем остальным моделям в scikit-learn) и с помощью дамми-переменных мы можем выразить любое количество категорий, вводя по одному новому признаку для каждой категории.

Существует два способа выполнить прямое кодирование категориальных переменных, используя либо **pandas**, либо **scikit-learn**. На момент написания книги использование pandas выглядело немного проще.

При работе с данными, которые были введены людьми (например, пользователями на сайте), получить фиксированный набор категорий невозможно, наличие различных вариантов написания слов может потребовать предварительной обработки.

*Хороший способ проверить содержимое столбца – воспользоваться функцией* **value\_counts** для пандасовского типа данных Series (каждый столбец DataFrame является структурой Series), чтобы посмотреть, что представляют из себя уникальные значения и как часто они встречаются:

In[3]:

print(data.gender.value\_counts())

Out[3]:

Male 21790

Female 10771

Name: gender, dtype: int64

Библиотека **pandas** предлагает очень простой способ кодирования данных с помощью функции **get\_dummies**. Функция **get\_dummies** автоматически преобразует все столбцы, которые содержат объектные типы (например, строки) или являются категориальными данными

data\_dummies = pd.get\_dummies(data) .values

или

features = data\_dummies.ix[:, 'age':'occupation\_ Transport-moving'] .values

Категориальные признаки часто кодируются с помощью целых чисел. Тот факт, что для кодировки используются числа, вовсе не означает, что они должны обрабатываться как непрерывные признаки.

Функция **get\_dummies** в **pandas** обрабатывает все числа как непрерывные значения и не будет создавать дамми-переменные для них. Чтобы обойти эту проблему, вы можете либо воспользоваться OneHotEncoder в scikit-learn (указав, какие переменные являются непрерывными, а какие – дискретными), либо преобразовать столбцы с числами, содержащиеся в дата-фрейме, в строки либо явно указать параметр **colum** в функции **get\_dummies**.

data\_dummies = pd.get\_dummies(data, columns=['spec', 'fin'])

**Биннинг и дискретизация для линейных моделей и деревьев.**

Оптимальный способ представления данных зависит не только от содержательного смысла данных, но и от вида используемой модели. Линейные модели и модели на основе дерева (например, деревья решений, градиентный бустинг деревьев решений и случайный лес), представляющие собой две большие и наиболее часто используемые группы методов, сильно отличаются друг от друга с точки зрения обработки признаков различных типов.

Одним из способов повысить прогнозную силу линейных моделей при работе с непрерывными данными является **биннинг** характеристик (binning), также известный как дискретизация (discretization), который *разбивает исходный признак на несколько категорий*.

Для этого мы используем функцию **np.linspace**, создаем 11 элементов, которые дадут 10 категорий – интервалов, ограниченных двумя границами.

Функция **np.linspace** возвращает равномерно распределенные числа за указанный интервал.

bins = np.linspace(-3, 3, 11)

Out[12]:

категории: [-3. -2.4 -1.8 -1.2 -0.6 0. 0.6 1.2 1.8 2.4 3. ]

Далее мы записываем для каждой точки данных категорию, в которую она попадает. Это можно легко вычислить с помощью функции **np.digitize**:

Функция **np.digitize** возвращает индексы интервалов, которым принадлежит каждое значение во входном массиве

which\_bin = np.digitize(X, bins=bins)

Out[13]:

Точки данных:

[[-0.753]

[ 2.704]

[ 1.392]

[ 0.592]

[-2.064]]

Категории для точек данных:

[[ 4]

[10]

[ 8]

[ 6]

[ 2]]

То, что мы сделали здесь, называется ***преобразованием непрерывного входного признака набора данных в категориальный признак***. С помощью категориального признака мы задаем категорию для каждой точки данных.

Чтобы запустить модель **scikit-learn** на этих данных, мы выполним прямое кодирование этого дискретного признака с помощью функции **OneHotEncoder** из модуля **preprocessing**. Функция **OneHotEncoder** выполняет ту же самую кодировку, что и **pandas.get\_dummies**, хотя в настоящее время она работает только с категориальными переменными, которые принимают целочисленные значения

from sklearn.preprocessing import OneHotEncoder

bins = np.linspace(-3, 3, 11)

which\_bin = np.digitize(X, bins=bins)

encoder = OneHotEncoder(sparse=False)

encoder.fit(which\_bin)

X\_binned = encoder.transform(which\_bin)

print(X\_binned[:5])

Out[14]:

[[ 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]

[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]

[ 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]

[ 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]

[ 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]]

In[15]:

print("форма массива X\_binned: {}".format(X\_binned.shape))

Out[15]:

форма массива X\_binned: (100, 10)

Сравнив модели, обученные до и после биннинга переменных, мы видим, что *линейная модель стала теперь гораздо более гибкой*, потому что теперь оно присваивает различные значения категориям, в то время как модель дерева решений стала существенно менее гибкой. В целом *биннинг признаков не дает положительного эффекта для моделей на основе дерева*, поскольку эти модели **сами** могут научится разбивать данные по любому значению.

Если есть веские причины использовать линейную модель для конкретного набора данных (например, он имеет большой объем и является многомерным), но некоторые признаки имеют нелинейные взаимосвязи с зависимой переменной – биннинг может быть отличным способом увеличить прогнозную силу модели.

***Использование биннинга – это способ увеличения пространства входных признаков.***

**Взаимодействия и полиномы.**

Еще один способ обогатить пространство признаков, в частности, для линейных моделей, заключается в добавлении **взаимодействий признаков** (interaction features) и **полиномиальных признаков** (polynomial features). Конструирование признаков подобного рода получило распространение в статистическом моделировании, а также широко используется во многих практических сферах применения машинного обучения.

Известно, что линейные модели могут вычислить не только значения сдвига, но и значения наклона. Один из способов **добавить наклон в линейную модель**, построенную на основе категоризированных данных, заключается в том, чтобы добавить обратно исходный признак.

Функция **np.hstack** объединяет данные по второй оси

bins = np.linspace(-3, 3, 11)

which\_bin = np.digitize(X, bins=bins)

encoder = OneHotEncoder(sparse=False).fit(which\_bin)

X\_binned = encoder.transform(which\_bin)

X\_combined = np.hstack([X, X\_binned])

print(X\_combined.shape)

Out[17]:

(100, 11)

Вычисленный наклон направлен вниз и он является общим для всех категорий, так как у нас имеется толька один признак по оси x с одним коэффициентом. Поскольку наличие одного наклона для всех категорий не очень сильно поможет с точки зрения моделирования, мы бы хотели вычислить **для каждой категории свой собственный наклон**! Мы можем добиться этого, добавив взаимодействие или произведение признаков, указывающее категорию точки данных и ее расположение на оси х. Данный признак является произведением индикатора категории и исходной переменной

X\_product = np.hstack([X\_binned, X \* X\_binned])

print(X\_product.shape)

Out[19]:

(100, 20)

Теперь набор данных содержит 20 признаков. Теперь каждая категория имеет свое собственное значение сдвига и свое собственное значение наклона.

*Еще один способ увеличения пространства входных признаков заключается в* *использовании полиномов (polynomials) исходных признаков*. Для признака х мы рассмотрим х \*\* 2, х \*\* 3, х \*\* 4 и так далее. Данную операцию можно выполнить с помощью **PolynomialFeatures** модуля preprocessing:

from sklearn.preprocessing import PolynomialFeatures

# задаем степень полинома 10:

# значение по умолчанию "include\_bias=True" добавляет признак-константу 1

poly = PolynomialFeatures(degree=10, include\_bias=False)

poly.fit(X)

X\_poly = poly.transform(X)

Элементы массива X:

[[-0.753]

[ 2.704]

[ 1.392]

[ 0.592]

[-2.064]]

форма массива X\_poly: (100, 10)

Элементы массива X\_poly:

[[ -0.753 0.567 -0.427 0.321 -0.242 0.182 -0.137 0.103 -0.078 0.058]

[ 2.704 7.313 19.777 53.482 144.632 391.125 1057.714 2860.360 7735.232 20918.278]

[ 1.392 1.938 2.697 3.754 5.226 7.274 10.125 14.094 19.618 27.307]

[ 0.592 0.350 0.207 0.123 0.073 0.043 0.025 0.015 0.009 0.005]

[ -2.064 4.260 -8.791 18.144 -37.448 77.289 -159.516 329.222 -679.478 1402.367]]

Вы можете понять содержательный смысл этих признаков, вызвав метод **get\_feature\_names**, который выведет название каждого признака:

print("Имена полиномиальных признаков:\n{}".format(poly.get\_feature\_names()))

Имена полиномиальных признаков:

['x0', 'x0^2', 'x0^3', 'x0^4', 'x0^5', 'x0^6', 'x0^7', 'x0^8', 'x0^9', 'x0^10']

*Используя более сложную модель, модель ядерного SVM, мы можем получить такой же сложный прогноз*, как в случае полиномиальной регрессии, не прибегая к явному преобразованию признаков.

**Автоматический отбор признаков.**

Добавление новых признаков делает модели более сложными и поэтому увеличивает вероятность переобучения. Добавляя новые признаки или работая с высокоразмерными наборами данных, неплохо бы уменьшить количество признаков и оставить только наиболее полезные из них. Это позволит получить более простые модели с лучшей обобщающей способностью. Однако как узнать, насколько полезен каждый признак? *Существуют три основные стратегии*: **одномерные статистики** (univariate statistics), **отбор на основе модели** (model-based selection) и **итеративный отбор** (iterative selection).

*Все эти методы относятся методам машинного обучения с учителем, то есть для подгонки модели им требуется зависимая переменная*.

1. **Одномерные статистики**

С помощью одномерных статистик мы *определяем наличие статистически значимой взаимосвязи между каждым признаком и зависимой переменной*. Затем отбираем признаки, сильнее всего связанные с зависимой переменной (имеющие уровень значимости, не превышающий заданного порогового значения. Ключевым свойством этих тестов является то, что они являются одномерными, то есть они рассматривают каждую характеристику по отдельности.

Отбор признаков в scikit-learn, производится обычно либо **f\_classif** (по умолчанию) для классификации или **f\_regression** для регрессии, а также метод исключения признаков, основанный на р-значениях, вычисленных в ходе теста. Все методы исключения параметров используют пороговое значение. Методы отличаются способами вычисления этого порогового значения, самым простым из которых являются **SelectKB**, выбирающий фиксированное число k признаков, и **SelectPercentile**, выбирающий фиксированный процент признаков.

from sklearn.feature\_selection import SelectPercentile

# используем f\_classif (по умолчанию)

# и SelectPercentile, чтобы выбрать 50% признаков

select = SelectPercentile(percentile=50)

select.fit(X\_train, y\_train)

# преобразовываем обучающий набор

X\_train\_selected = select.transform(X\_train)

# преобразовываем тестовые данные

X\_test\_selected = select.transform(X\_test)

Выяснить, какие функции были отобраны, воспользовавшись методом **get\_support**, который возвращает булевы значения для каждого признака.

mask = select.get\_support()

print(mask)

1. **Отбор признаков на основе модели**

Отбор признаков на основе модели использует модель машинного обучения с учителем, чтобы вычислить важность каждого признака, и оставляет только самые важные из них. Модель машинного обучения с учителем, которая используется для отбора признаков, не должна использоваться для построения итоговой модели. Модель, применяющаяся для отбора признаков, требует вычисления

В отличие от одномерного отбора отбор на основе модели рассматривает все признаки сразу и поэтому может обнаружить взаимодействия (если модель способна выявить их). Чтобы применить отбор на основе модели, мы должны воспользоваться модификатором **SelectFromModel**

from sklearn.feature\_selection import SelectFromModel

from sklearn.ensemble import RandomForestClassifier

select = SelectFromModel(

RandomForestClassifier(n\_estimators=100, random\_state=42), threshold="median")

Класс **SelectFromModel** отбирает все признаки, у которых показатель важности (заданный моделью машинного обучения с учителем) превышает установленное пороговое значение. В данном случае, порог равен **медиане**.

select.fit(X\_train, y\_train)

X\_train\_l1 = select.transform(X\_train)

X\_test\_l1 = select.transform(X\_test)

1. **Итеративный отбор признаков**

В итеративном отборе признаков строится последовательность моделей с различным количеством признаков. *Существует два основных метода*. **Первый метод** начинается шага, когда в модель включена лишь одна константа (входных признаков нет) и затем добавляет признак за признаком до тех пор, пока не будет достигнут критерий остановки. **Второй метод** начинается с шага, когда все признаки включены в модель, и затем начинает удалять признак за признаком, пока не будет достигнут критерий остановки.

Одним из таких методов является **метод рекурсивного исключения признаков** (recursive feature elimination, **RFE**), который начинается с включения всех признаков, строит модель и исключает наименее важный признак с точки зрения модели. Затем строится новая модель с использованием всех признаков, кроме исключенного, и так далее.

from sklearn.feature\_selection import RFE

select = RFE(RandomForestClassifier(n\_estimators=100, random\_state=42),

n\_features\_to\_select=40)

select.fit(X\_train, y\_train)

X\_train\_rfe= select.transform(X\_train)

X\_test\_rfe= select.transform(X\_test)

**Тезисы главы 5 «Оценка и улучшение качества подели»**

*Рассматриваемые темы:*

1. *Перекрестная проверка*
2. *Решетчатый поиск*
3. *Метрики для оценки алгоритмов машинного обучения*
4. *Использование метрик для оценки отбора моделей*

*Описываемые библиотеки и функции:*

**Перекрестная проверка.**

В перекрестной проверке данные разбиваются несколько раз и строится несколько моделей. Наиболее часто используемый вариант перекрестной проверки – **k-блочная кросс-проверка** (k-fold cross-validation), в которой k – это задаваемое пользователем число, как правило, 5 или 10. При выполнении пятиблочной перекрестной проверки данные сначала разбиваются на пять частей (примерно) одинакового размера, называемых блоками (folds) складками. Затем строится последовательность моделей. Первая модель обучается, используя блок 1 в качестве тестового набора, а остальные блоки (2-5) выполняют роль обучающего набора. Модель строится на основе данных, расположенных в блоках 2-5, а затем на данных блока 1 оценивается ее правильность. Затем происходит обучение второй модели, на этот раз в качестве тестового набора используется блок 2, а данные в блоках 1, 3, 4, и 5 служат обучающим набором. Этот процесс повторяется для блоков 3, 4 и 5, выполняющих роль тестовых наборов.

В scikit-learn перекрестная проверка реализована с помощью функции **cross\_val\_score** модуля **model\_selection**. По умолчанию cross\_val\_score выполняет трехблочную перекрестную проверку, возвращая три значения правильности. Мы можем изменить количество блоков, задав другое значение параметра cv.

logreg = LogisticRegression()

scores = cross\_val\_score(logreg, iris.data, iris.target, cv=5)

print("Значения правильности перекрестной проверки: {}".format(scores))

print("Средняя правильность перекрестной проверки: {:.2f}".format(scores.mean()))

Преимущество перекрестной проверки по сравнению с однократным разбиением данных заключается в том, что мы используем наши данные более эффективно. Применяя пятиблочную перекрестную проверку, на каждой итерации для подгонки модели мы можем использовать 4/5 данных (80%). При использовании 10-блочной перекрестной проверки мы можем использовать для подгонки модели 9/10 данных (90%).

Основной недостаток перекрестной проверки – увеличение стоимости вычислений.

*Важно помнить, что кросс-валидация не является способом построения модели, которую можно применить к новым данным. Перекрестная проверка не возвращает модель*. При вызове cross\_val\_score строится несколько внутренних моделей, однако цель перекрестной проверки заключается только в том, чтобы оценить обобщающую способность данного алгоритма, обучив на определенном наборе данных.

**Стратифицированная k-блочная перекрестная проверка и другие стратегии.**

В стратифицированной перекрестной проверке мы разбиваем данные таким образом, чтобы пропорции классов в каждом блоке в точности соответствовали пропорциям классов в наборе данных.

Scikit-learn позволяет значительно точнее настроить процесс перекрестной проверки, используя в качестве параметра cv **генератор разбиений перекрестной проверки** (cross-validation splitter).

Класс **KFold** из модуля model\_selection позволяет передавать генератор разбиений **kfold** в качестве параметра **cv** в функцию **cross\_val\_score**. Вместо стратификации возможно перемешивать данные и тем самым нарушить порядок сортировки примеров, определяемый их метками. Мы можем сделать это, передав генератору **KFold** параметр **shuffle=True**.

kfold = KFold(n\_splits=3, shuffle=True, random\_state=0)

score = cross\_val\_score(logreg, iris.data, iris.target, cv=kfold)

**Перекрёстная проверка с исключением по одному.**

Перекрестную проверку с исключением по одному можно представить в виде k-блочной

перекрестной проверки, в которой каждый блок представляет собой отдельный пример. По каждому разбиению вы выбираете одну точку данных в качестве тестового набора. Этот вид проверки может занимать очень много времени, особенно при работе с большими наборами данных, однако иногда позволяет получить более точные оценки на небольших наборах данных

from sklearn.model\_selection import LeaveOneOut

loo = LeaveOneOut()

scores = cross\_val\_score(logreg, iris.data, iris.target, cv=loo)

**Перекрестная проверка со случайными перестановками при разбиении**

Перекрестная проверка со случайными перестановками при разбиении (shuffle-split cross-validation). В этом виде проверки каждое разбиение выбирает **train\_size** точек для обучающего набора и **test\_size** точек для тестового набора (при этом обучающее и тестовое подмножества не

пересекаются). Точки выбираются с возвращением. Разбиение повторяется **n\_iter** раз.

from sklearn.model\_selection import ShuffleSplit

shuffle\_split = ShuffleSplit(test\_size=.5, train\_size=.5, n\_splits=10)

scores = cross\_val\_score(logreg, iris.data, iris.target, cv=shuffle\_split)

Перекрестная проверка со случайными перестановками при разбиении позволяет задавать количество итераций независимо от размеров обучающего и тестового наборов, что иногда может быть полезно. Кроме того, этот метод позволяет использовать на каждой итерации лишь часть данных (значения train\_size и test\_size необязательно должны в сумме давать 1). Подобное прореживание данных может быть полезно при работе с большими наборами данных.

Существует также стратифицированный вариант **ShuffleSplit**, названный **StratifiedShuffleSplit**, который позволяет получить более надежные результаты при решении задач классификации.

**Перекрёстная проверка с использованием групп.**

При оценки модели изображений 100 человек, в котором каждый человек сфотографирован несколько раз, важно точно оценить способность модели обобщать результат на новые лица. Для этого необходимо убедиться в том, что обучающий и тестовый наборы содержат изображения разных людей.

Для решения этой задачи мы можем воспользоваться **GroupKFold**, принимающий в качестве аргумента массив **groups**. С помощью него мы указываем, какой человек изображен на снимке. В данном случае массив **groups** указывает группы данных, которые не следует разбивать при создании обучающего и тестового наборов, при этом их не следует путать с метками классов.

from sklearn.model\_selection import GroupKFold

# создаем синтетический набор данных

X, y = make\_blobs(n\_samples=12, random\_state=0)

# предположим, что первые три примера относятся к одной и той же группе,

# затем следующие четыре и так далее.

groups = [0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3] scores = cross\_val\_score(logreg, X, y, groups, cv=GroupKFold(n\_splits=3))

**Решётчатый поиск.**

Поскольку поиск оптимальных значений параметров является общераспространенной задачей, библиотека scikit-learn предлагает стандартные методы, позволяющие решить ее. Наиболее часто используемый метод – это *решетчатый поиск (grid search), который по сути является попыткой перебрать все возможные комбинации интересующих параметров*.

Поскольку решетчатый поиск с перекрестной проверкой является весьма распространенным методом настройки параметров, библиотека scikit-learn предлагает класс **GridSearchCV**, в котором решетчатый поиск реализован в виде модели. Чтобы воспользоваться классом **GridSearchCV**, сначала необходимо указать искомые параметры с помощью словаря. GridSearchCV построит все необходимые модели. Ключами словаря являются имена настраиваемых параметров (в данном случае **С** и **gamma**), а значениями – тестируемые настройки параметров. Перебор значений 0.001, 0.01, 0.1, 1, 10 и 100 для C и gamma требует словаря следующего вида:

param\_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}

Теперь мы можем создать экземпляр класса **GridSearchCV**, передав модель (**SVC**), сетку искомых параметров (**param\_grid**), а также стратегию перекрестной проверки, которую мы хотим использовать (допустим, пятиблочную стратифицированную перекрестную проверку):

from sklearn.model\_selection import GridSearchCV

from sklearn.svm import SVC

grid\_search = GridSearchCV(SVC(), param\_grid, cv=5)

Вместо разбиения на обучающий и проверочный набор, использованного нами ранее, GridSearchCV запустит перекрестную проверку. Однако нам по-прежнему нужно разделить данные на обучающий и тестовый наборы

Созданный нами объект grid\_search аналогичен классификатору, мы можем вызвать стандартные методы **fit**, **predict** и **score** от его имени. Однако, когда мы вызываем fit, он запускает перекрестную проверку для каждой комбинации параметров, указанных в **param\_grid**

grid\_search.fit(X\_train, y\_train)

*Процесс подгонки объекта* ***GridSearchCV*** *включает в себя не только поиск лучших параметров, но и автоматическое построение новой модели* на всем обучающем наборе данных. Для ее построения используются параметры, которые дают наилучшее значение правильности перекрестной проверки.

*Главный момент здесь в том, что мы не использовали тестовый набор для отбора параметров*. Найденная комбинация параметров сохраняется в атрибуте **best\_params\_**, а наилучшее значение правильности перекрестной проверки (значение правильности, усредненное по всем разбиениям для данной комбинации параметров) – в атрибуте **best\_score\_**.

print("Наилучшие значения параметров: {}".format(grid\_search.best\_params\_))

print("Наилучшее значение кросс-валидац. правильности:{:.2f}".format(grid\_search.best\_score\_))

В ряде случаев вам необходимо будет ознакомиться с полученной моделью, например, взглянуть на коэффициенты или важности признаков. Посмотреть наилучшую модель, построенную на всем обучающем наборе, вы можете с помощью атрибута best\_estimator\_

print("Наилучшая модель:\n{}".format(grid\_search.best\_estimator\_))

Наилучшая модель:

SVC(C=100, cache\_size=200, class\_weight=None, coef0=0.0, decision\_function\_shape=None, degree=3, gamma=0.01, kernel='rbf', max\_iter=-1, probability=False,

Результаты решетчатого поиска можно найти в атрибуте **cv\_results**, который является словарем, хранящим все настройки поиска. Словарь содержит множество деталей и принимает более привлекательный вид после преобразования в пандасовский DataFrame

import pandas as pd

# преобразуем в DataFrame

results = pd.DataFrame(grid\_search.cv\_results\_)

# показываем первые 5 строк

display(results.head())

**Экономичный решётчатый поиск.**

В случае когда поиск по всем возможным комбинациям C, gamma и kernel не имеет смысла: если kernel='linear', то gamma не используется и перебор различных значений gamma – это пустая трата времени. Чтобы обработать подобные «условные» параметры, GridSearchCV позволяет превратить param\_grid в список словарей.

param\_grid = [{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},

{'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}]

**Применение различных стратегий перекрёстной проверки с помощью решётчатого поиска.**

Как и cross\_val\_score, **GridSearchCV** использует по умолчанию **k- блочную** перекрестную проверку для классификации и **k-блочную** перекрестную проверку для регрессии. Однако при использовании GridSearchCV вы *можете дополнительно передать любой генератор разбиения* в качестве параметра cv. В частности, чтобы получить только одно разбиение на обучащий и проверочный наборы, вы можете воспользоваться ShuffleSplit или StratifiedShuffleSplit с n\_iter=1. Данная настройка может оказаться полезной для очень больших наборов данных или очень медленных моделей.

**Вложенная перекрёстная проверка.**

Мы можем вместо однократного разбиения исходных данных на обучающий и тестовый наборы использовать несколько разбиений перекрестной проверки. В результате мы получим вложенную перекрестную проверкой (nested cross-validation). Во вложенной перекрестной проверке используется внешний цикл по разбиениям данных на обучающий и тестовый наборы. Для каждого из них выполняется решетчатый поиск.

*Результатом этой процедуры является не модель и не настройки параметров, а список значений правильности*. Значения правильности указывают нам на обобщающую способность модели с использованием лучших параметров, найденных в ходе решетчатого поиска.

Поскольку вложенная перекрестная проверка *не дает модель*, которую можно использовать на новых данных, *ее редко используют при поиске прогнозной модели для применения к новым данным*.

scores = cross\_val\_score(GridSearchCV(SVC(), param\_grid, cv=5), iris.data, iris.target, cv=5)

**Распараллеливание перекрёстной проверки и решётчатого поиска.**

В **GridSearchCV** и **cross\_val\_score** вы можете использовать несколько процессорных ядер, задав значение параметра **n\_jobs** равным нужному количеству ядер. Вы можете установить **n\_jobs=-1**, чтобы использовать все доступные ядра.

Имейте в виду, *что scikit-learn не поддерживает вложенность параллельных операций* (nesting of parallel operations). Поэтому, если вы используете опцию n\_jobs для вашей модели (например, для случайного леса), вы не можете использовать ее в GridSearchCV для осуществления поиска по этой модели.

**Метрики для бинарной классификации.**

**Типы ошибок.**

Существует соглашение считать положительным тестом (наличие рака) положительный класс, а отрицательный тест соответствует отрицательному классу.

Пример, неправильно спрогнозированный как положительный, называется ложно положительным (false positive). Пример, неправильно спрогнозированный как отрицательный, называется ложно отрицательным (false negative).

**Несбалансированный набор данных.**

Наборы данных, в которых один класс встречается гораздо чаще, чем остальные, часто называют *несбалансированными наборами данных* (imbalanced datasets) или наборами данных с несбалансированными классами (datasets with imbalanced classes).

Проблема здесь заключается в том, что *для несбалансированных наборов данных правильность не является адекватной метрикой*, позволяющей количественно оценить прогностическую способность модели.

**Матрица ошибок.**

Одним из наиболее развернутых способов, позволяющих оценить качество бинарной классификации, является использование **матрицы ошибок**.

from sklearn.metrics import confusion\_matrix

confusion = confusion\_matrix(y\_test, pred\_logreg)

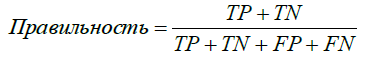
print("Confusion matrix:\n{}".format(confusion))

Вывод **confusion\_matrix** представляет собой массив размером 2x2, где строки соответствуют фактическим классам, а столбцы соответствуют спрогнозированным классам Число в каждой ячейке показывает количество примеров, когда спрогнозированный класс, представленный столбцом, совпадает или не совпадает с фактическим классом, представленным строкой.

*Элементы главной диагонали матрицы ошибок соответствуют правильным прогнозам* (результатам классификации), тогда как остальные элементы показывают, сколько примеров, относящихся к одному классу, были ошибочно классифицированы как другой класс.

**Связь с правильностью.**

Правильность – это количество верно классифицированных примеров (TP и TN), поделенное на общее количество примеров (суммируем все элементы матрицы ошибок).



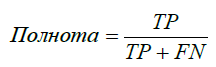
**Плотность, точность, F-мера и classification\_report.**

**Точность** (precision) показывает, сколько из предсказанных положительных примеров оказались действительно положительными.



Точность используется в качестве показателя качества модели, когда *цель состоит в том, чтобы снизить количество ложно положительных примеров*. В качестве примера представьте модель, которая должна спрогнозировать, будет ли эффективен новый лекарственный препарат при лечении болезни.

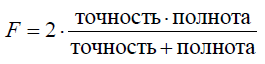
**Полнота** (recall) показывает, сколько от общего числа фактических положительных примеров было предсказано как положительный класс.



*Полнота используется* в качестве показателя качества модели, когда нам необходимо определить все положительные примеры, то есть, *когда важно снизить количество ложно отрицательных примеров*. Пример диагностики рака, приведенный ранее в этой главе, является хорошей иллюстрацией подобной задачи: важно выявить всех больных пациентов, при этом, возможно, включив в их число здоровых пациентов. Другие названия полноты – чувствительность (sensitivity),

**F-мера.**

Точность и полнота являются очень важными метриками, сами по себе они не дадут вам полной картины. Одним из способов подытожить их является **F-мера** (F-measure), которая *представляет собой гармоническое среднее точности и полноты*.



Этот вариант вычисления F-меры еще известен как f1-мера. Поскольку f1-мера учитывает точность и полноту, то *для бинарной классификации несбалансированных данных она может быть более лучшей метрикой, чем правильность*.

from sklearn.metrics import f1\_score

dummy = DummyClassifier().fit(X\_train, y\_train)

pred\_dummy = dummy.predict(X\_test)

print("f1-мера дамми: {:.2f}".format(f1\_score(y\_test, pred\_dummy)))

Вместе с тем ***недостаток f-меры*** заключается в том, что в отличие от правильности *ее труднее интерпретировать и объяснить.*

**Сlassification\_report.**

Если мы хотим получить более развернутый отчет о точности, полноте и f1-мере, можно воспользоваться удобной функцией **classification\_report**.

*Функция* ***classification\_report*** *печатает отчет, в котором выводятся показатели точности, полноты и f-меры для отрицательного и положительного классов*.

В последней строке отчета приводятся средние значения метрик, взвешенные по количеству фактических примеров в каждом классе.

classification\_report(y\_test, pred\_dummy,target\_names=["not nine", "nine"])

**Принимаем во внимание неопределённость.**

Матрица ошибок и отчет о результатах классификации позволяют провести очень детальный анализ полученных прогнозов. Однако *сами по себе прогнозы лишены большого объема информации, которая собрана моделью*.

Большинство классификаторов для оценки степени достоверности прогнозов позволяют использовать методы **decision\_function** или **predict\_proba**. Получить прогнозы можно, установив для **decision\_function** или **predict\_proba** пороговое значение в некоторой фиксированной точке – в

случае бинарной классификации мы используем 0 для решающей функции и 0.5 для метода **predict\_proba**.