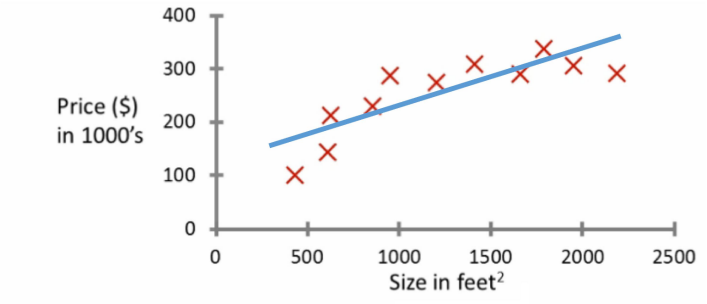
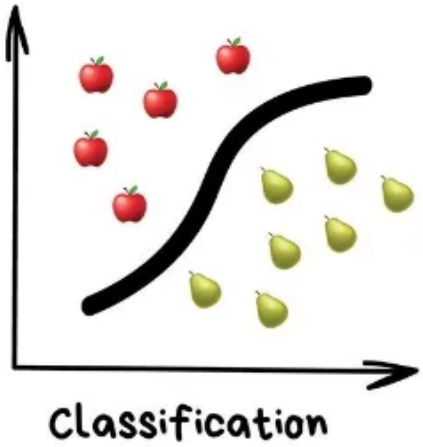
Линейные модели — это модели, отображающие зависимость целевого признака от факторов в виде линейной взаимосвязи.

Линейные модели



Задача классификации

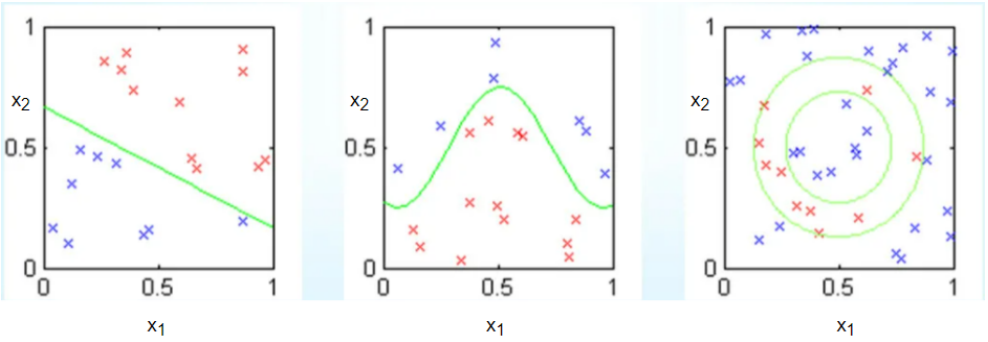


Задача классификации (classification) — это задача, в которой мы пытаемся предсказать класс объекта на основе признаков в наборе данных. То есть задача сводится к предсказанию целевого признака, который является категориальным.

Когда классов, которые мы хотим предсказать, только два, классификация называется **бинарной**.

Когда классов, которые мы хотим предсказать, более двух, классификация называется **мультиклассовой (многоклассовой)**.

Решить задачу классификации — значит построить разделяющую поверхность в пространстве признаков, которая делит пространство на части, каждая из которых соответствует определённому классу.



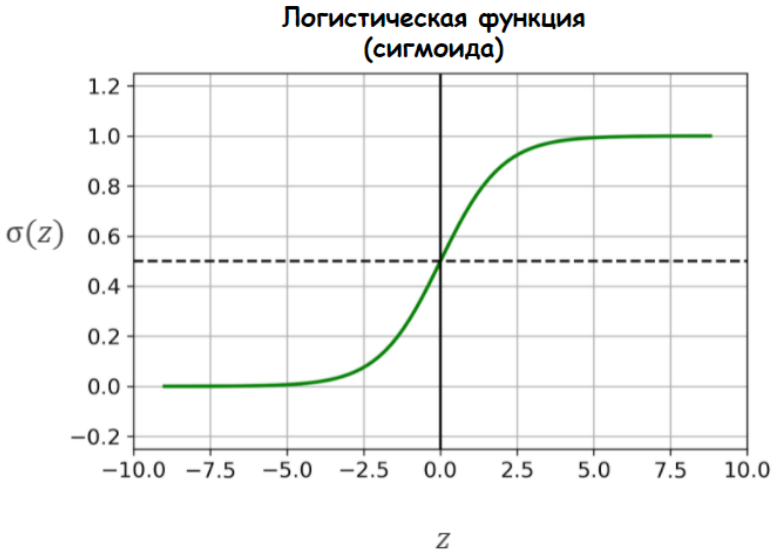
Модели, которые решают задачу классификации, называются **классификаторами (classifier)**.

Логистическая регрессия

Логистическая регрессия (Logistic Regression) — одна из простейших моделей для решения задачи классификации. Несмотря на простоту, модель входит в ТОП самых часто используемых алгоритмов классификации в Data Science.

В основе логистической регрессии лежит логистическая функция $\sigma(z)$ **(logistic function)** (отсюда и название модели), но более распространённое название этой функции — **сигмоида (sigmoid)**. Записывается она следующим образом:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$



Свойства сигмоиды:

- Значения сигмоиды $\sigma(z)$ лежат в диапазоне от 0 до 1 при любых значениях аргумента z. Какой бы z вы ни подставили, число меньше 0 или больше 1 вы не получите.
- Сигмоида выдаёт значения $\sigma(z) > 0.5$ при её аргументе $z > 0$, $\sigma(z) < 0.5$, при $z < 0$ и $\sigma(z) = 0.5$ при $z = 0$.

Основная идея модели логистической регрессии: берём модель линейной регрессии (обозначим её выход за z)

$$z = \omega_0 + \sum_{j=1}^{m} \omega_j x_j$$

и подставляем выход модели z в функцию сигмоиды, чтобы получить искомые ***оценки вероятности***:

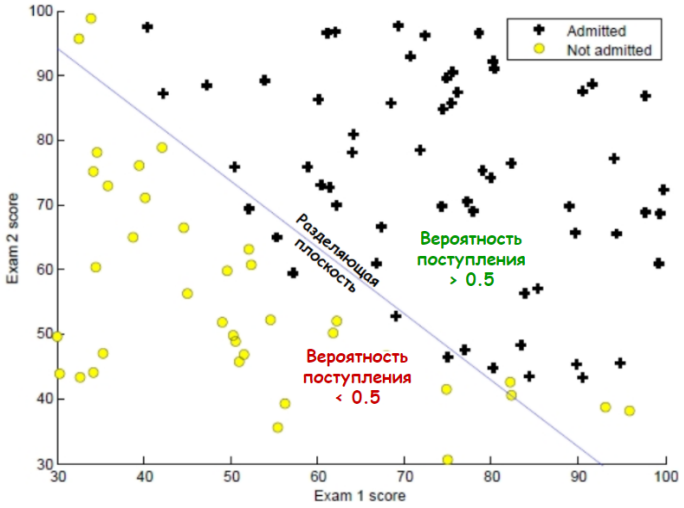
$$\widehat{P} = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-\omega_0 - \sum_{j=1}^{m} \omega_j x_j}}$$

Если вероятность $\widehat{P} > 0.5$, значит относим объект к классу 1, а если $\widehat{P} \leq 0.5$, то относим к классу 0.
То есть предсказание класса формируется следующим образом:

$$\hat{y} = I[\hat{P}] = \begin{cases} 1, \hat{P} > 0.5 \\ 0, \hat{P} \leq 0.5 \end{cases}$$

Геометрически построить логистическую регрессию на основе двух факторов — значит найти такие коэффициенты ω_0, ω_1 и ω_2 уравнения плоскости, при которых наблюдается наилучшее разделение пространства на две части.

$$z = \omega_0 + \omega_1 x_1 + \omega_2 x_2$$

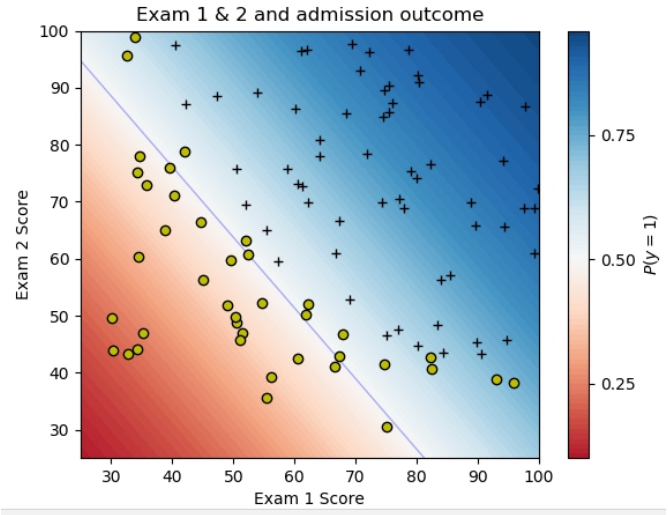


В чём математический секрет?

Математически подстановка в уравнение плоскости точки, которая не принадлежит ей (находится ниже или выше), означает вычисление расстояния от этой точки до плоскости. Если точка находится ниже плоскости, то расстояние будет отрицательным ($z < 0$). Если точка находится выше плоскости, то расстояние будет положительным ($z > 0$). Если точка находится на самой плоскости, то $z = 0$.

Мы знаем, что подстановка отрицательных чисел в сигмоиду приведёт к вероятности $\widehat{P} < 0.5$, а постановка положительных — к вероятности $\widehat{P} > 0.5$.
То есть ключевым моментом в предсказании логистической регрессии является расстояние от точки до разделяющей плоскости в пространстве факторов. Это расстояние в литературе часто называется **отступом (margin)**.

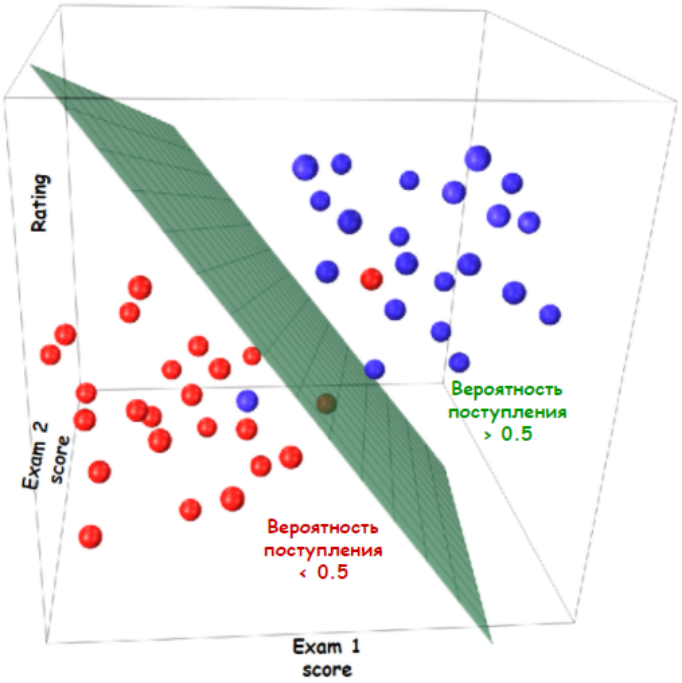
Тепловая карта вероятностей:



Для случая зависимости целевого признака y от трёх факторов x_1, x_2 и x_3, например, баллов за два экзамена и рейтинга университета, из которого поступает абитуриент, будем иметь следующее выражение для z:

$$z = \omega_0 + \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3$$

Уравнение задаёт плоскость в четырёхмерном пространстве. Но если вспомнить, что y категориальный и классы можно обозначить цветом, то получится перейти в трёхмерное пространство, и разделяющая плоскость будет выглядеть следующим образом:

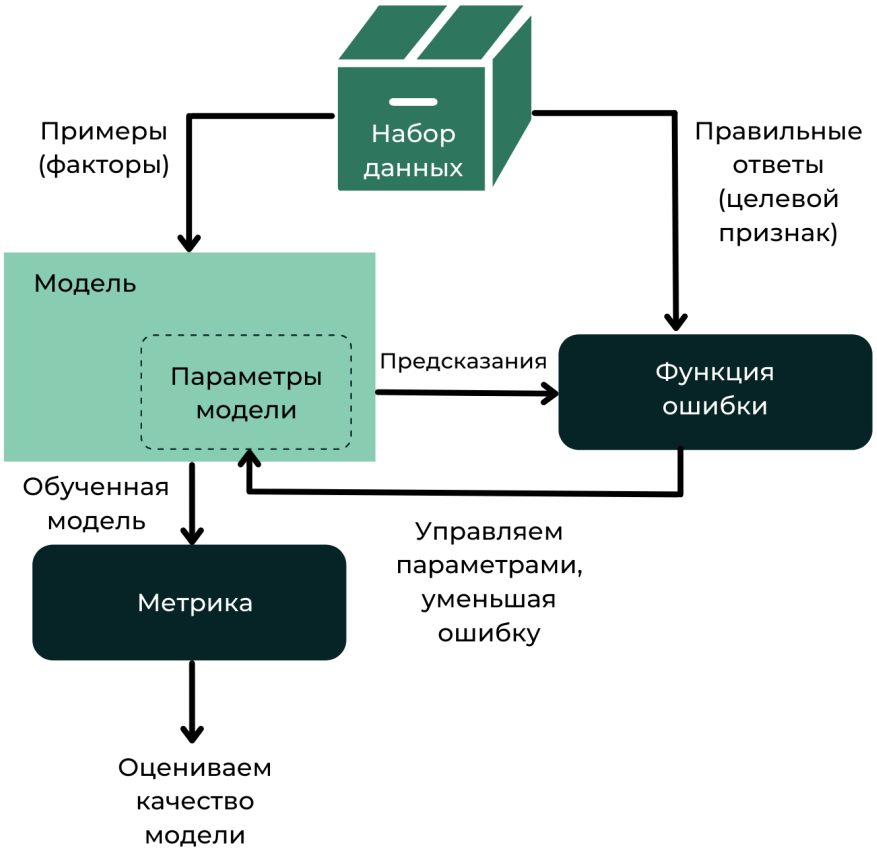


В общем случае, когда у нас зависимость от m факторов, линейное выражение, находящееся под сигмоидой, будет обозначать **разделяющую гиперплоскость**.

$$z = \omega_0 + \omega_1 x_1 + \omega_2 x_2 + \ldots + \omega_m x_m = \omega_0 + \sum_{j=1}^{m} \omega_j x_j$$

Поиск параметров логистической регрессии

Поиск производится по схеме минимизации эмпирического риска:



Функция ошибки, которая минимизируется, выводится из **метода максимального правдоподобия (maximum likelihood estimation — MLE)**.

Данный метод позволяет получить **функцию правдоподобия**.

Цель всего метода — найти такие параметры $\omega = (\omega_0, \omega_1, \omega_2, ..., \omega_m)$, в которых наблюдается максимум функции правдоподобия.

Задача оптимизации правдоподобия:

$$likelihood = \sum_i^n (y_i log(\widehat{P}_i) + (1 - y_i) log(1 - \widehat{P}_i)) \rightarrow max_\omega$$

Умножим правдоподобие на — получим функцию, которая носит название **функция логистических потерь**, или **logloss**. Также часто можно встретить название **кросс-энтропия**, или **cross-entropy loss**:

$$L(\omega) = logloss = -\sum_i^n (y_i log(\widehat{P}_i) + (1 - y_i) log(1 - \widehat{P}_i)) \rightarrow min_\omega$$

$$\widehat{P}_i = \frac{1}{1+e^{-\omega_0 - \sum_{j=1}^{m} \omega_j x_j}}$$

Эту функцию мы и будем минимизировать в рамках поиска параметров логистической регрессии. Мы должны найти такие параметры разделяющей плоскости ω, при которых наблюдается минимум $logloss$.

Минимум ищется с помощью численных методов оптимизации, например градиентного спуска:

$$\omega^{(k+1)} = \omega^{(k)} - \eta \nabla L(\omega^{(k)})$$

Новое значение параметров $\omega^{(k+1)}$ получается путём сдвига текущих $\omega^{(k)}$ в сторону вектора антиградиента $-\nabla L(\omega^{(k)})$, умноженного на темп обучения η.

Во избежание переобучения модели в функцию потерь логистической регрессии традиционно добавляется **регуляризация**.
При **L1-регуляризации** мы добавляем в функцию потерь $L(\omega)$ штраф из суммы модулей параметров, а саму функцию logloss умножаем на коэффициент C:

$$L(\omega) = C \cdot logloss + \sum_{j=1}^{m} |\omega_j| \rightarrow min_\omega$$

А при **L2-регуляризации** — штраф из суммы квадратов параметров:

$$L(\omega) = C \cdot logloss + \sum_{j=1}^{m} (\omega_j)^2 \rightarrow min_\omega$$

Значение коэффициента C — коэффициент, обратный коэффициенту регуляризации. Чем **больше** C, тем **меньше** «сила» регуляризации.

Логистическая регрессия в sklearn

```
from sklearn import linear_model
```

Реализация логистической регрессии в sklearn заложена в классе **LogisticRegression**.

Основные параметры LogisticRegression:

- `random_state` — число, на основе которого происходит генерация случайных чисел.
- `penalty` — метод регуляризации. Возможные значения:
  - `'l1'` — L1-регуляризация;
  - `'l2'` — L2-регуляризация (используется по умолчанию);

  - `'elasticnet'` — эластичная сетка (L1+L2);
  - `'None'` — отсутствие регуляризации.

- `C` — коэффициент, обратный коэффициенту регуляризации, равен $\frac{1}{\alpha}$. Чем больше C, тем меньше регуляризация. По умолчанию C=1, тогда $\alpha = 1$.
- `solver` — численный метод оптимизации функции потерь logloss, может быть:
  - `'sga'` — стохастический градиентный спуск (нужна стандартизация/нормализация).
  - `'saga'` — модификация предыдущего, которая поддерживает работу с негладкими функциями (нужна стандартизация/нормализация).
  - `'newton-cg'` — метод Ньютона с модификацией сопряжённых градиентов (не нужна стандартизация/нормализация);
  - `'lbfgs'` — метод Бройдена – Флетчера – Гольдфарба – Шанно (не нужна стандартизация/нормализация). Используется по умолчанию, так как из всех методов теоретически обеспечивает наилучшую сходимость.
  - `'liblinear'` — метод покоординатного спуска (не нужна стандартизация/нормализация).
- `max_iter` — максимальное количество итераций, выделенных на сходимость.

Обучение (поиск параметров) — метод fit():

```
#Создаём объект класса LogisticRegression
log_reg = linear_model.LogisticRegression(random_state=42)
#Обучаем модель, минизируя logloss
log_reg.fit(X, y)
```

Предсказание класса — метод predict():

```
y_pred = log_reg.predict(X)
```

Предсказание вероятностей принадлежности — метод predict_proba():

```
y_pred_proba = log_reg.predict_proba(X)
```

Посмотреть коэффициенты — атрибуты coef_ и intercept_:

- `log_reg.coef_` — коэффициенты при факторах,
- `log_reg.intercept _` — свободный член.

Метрики классификации

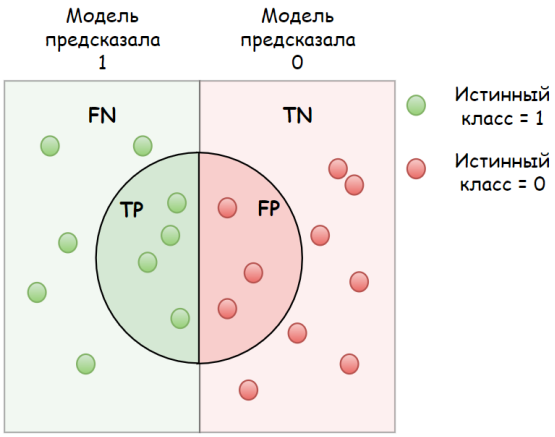
> **Метрика** — это численное выражение качества моделирования.

Матрица ошибок (confusion matrix) показывает все возможные исходы совпадения и несовпадения предсказания модели с действительностью. Используется для расчёта других метрик.

Допустим, у нас есть два класса и алгоритм, предсказывающий принадлежность каждого объекта одному из классов.
Назовём класс 1 **положительным исходом (positive)**, а класс 0 — **отрицательным исходом (negative)**.

Тогда матрица ошибок классификации будет выглядеть следующим образом:

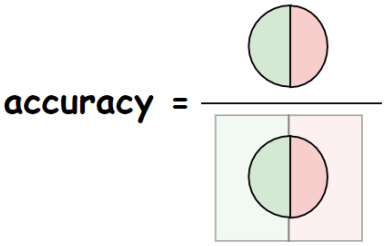
	$y = 1$	$y = 0$
$\hat{y} = 1$	True Positive (TP)	False Positive (FP)
$\hat{y} = 0$	False Negative (FN)	True Negative (TN)



> **Accuracy (достоверность)** — это доля правильных ответов модели среди всех ответов. Правильные ответы — это **истинно положительные (True Positive)** и **истинно отрицательные ответы (True Negative)**:

$$accuracy = \frac{TP+TN}{TP+TN+FN+FP}$$

В виде диаграммы соотношение будет выглядеть так:

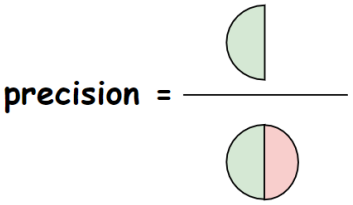


Интерпретация — как много (в долях) модель угадала ответов.

Precision (точность), или **PPV (Positive Predictive Value)** — доля объектов, которые действительно являются положительными, по отношению ко всем объектам, названным моделью положительными.

$$precision = \frac{TP}{TP+FP}$$

Соотношение в виде диаграммы:



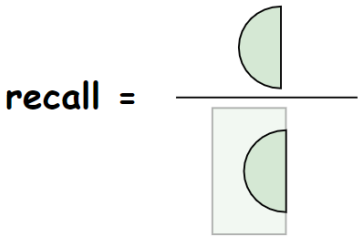
Метрика также изменяется от 0 до 1.

Интерпретация — способность отделить класс 1 от класса 0. Чем больше precision, тем меньше ложных попаданий.

Recall (полнота), или **TPR (True Positive Rate)** — доля объектов, названных классификатором положительными, по отношению ко всем объекта положительного класса.

$$recall = \frac{TP}{TP+FN}$$

Соотношение в виде диаграммы:



Метрика изменяется от 0 до 1.

Интерпретация — способность модели обнаруживать класс 1 вообще, то есть охват класса 1. Заметьте, что ложные срабатывания не влияют на recall.

F_β **(F-мера)** — ***взвешенное среднее гармоническое*** между precision и recall:

$$F_\beta = (1 + \beta^2)\frac{precision \cdot recall}{(\beta^2 precision) + recall}$$

где β — это вес precision в метрике: чем больше β, тем больший вклад.
В частном случае, когда $\beta = 1$, мы получаем равный вклад для precision и recall, а формула будет выражать простое среднее гармоническое или метрику F_1 ($F_1 -$ мера):

$$F_1 = 2\frac{precision \cdot recall}{precision + recall}$$

Название	Формула	Интерпретация и применение	Достоинства	Недостатки	Функция в модуле metrics библиотеки sklearn
Accuracy (достоверность)	$\frac{TP+TN}{TP+TN+FN+FP}$	Доля правильных ответов среди всех ответов модели. Применяется в задачах, где классы сбалансированы.	Доля правильных ответов среди всех ответов модели. Применяется в задачах, где классы сбалансированы. Очень	Плохо показывает себя на сильно несбалансированных классах.	accuracy_score()

			легко интерпретировать. Автоматически можно посчитать процент ошибок модели как 1 - accuracy.		
Precision (точность)	$\frac{TP}{TP+FP}$	Способность модели отделять класс 1 от класса 0. Используется в задачах, где важно минимальное количество ложных срабатываний модели.	Можно использовать на несбалансированных выборках.	Вычисляется только для положительного класса — класса 1. Для класса 0 показатель необходимо вычислять отдельно. Не даёт представления о том, как много объектов положительного класса из общей совокупности нашёл алгоритм.	precision_score()
Recall (полнота)	$\frac{TP}{TP+FN}$	Способность модели находить класс 1. Используется в задачах, где важно охватить как можно больше объектов положительного класса (1).	Можно использовать на несбалансированных выборках.	Вычисляется только для положительного класса — класса 1. Для класса 0 показатель необходимо вычислять отдельно. Не даёт представления о том, насколько	recall_score()

				точно модель находит объекты положительного класса (как много ложных срабатываний).	
F1-мера	$2\frac{precision \cdot recall}{precision + recall}$	Нет бизнес-интерпретации. Используется в задачах, где необходимо балансировать между precision и recall.	Даёт обобщённое представление о точности и полноте. Максимум достигается, когда максимальны обе метрики, минимум — когда хотя бы одна из метрик равна 0. При желании можно использовать обобщённый вариант — F_β, чтобы управлять вкладом precision в общую метрику.	Отсутствие интерпретации не даёт интуитивного понимания человеку, не знакомому с этой метрикой.	`f1_score()`

Расчёт метрик на Python

```
from sklearn import metrics
```

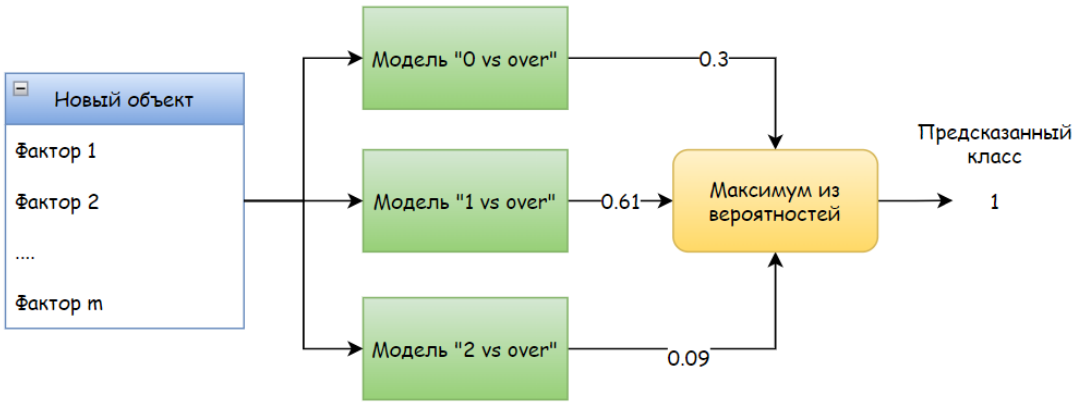
- `confusion_matrix()` — расчёт матрицы ошибок;
- `accuracy_score()` — расчёт accuracy;
- `precision_score()` — расчёт precision;
- `recall_score()` — расчёт recall;
- `f1_score()` — расчёт F_1-меры.

Каждая из этих функций первым аргументом принимает истинные ответы y, а вторым аргументом — предсказанные значения $\hat{y}$.

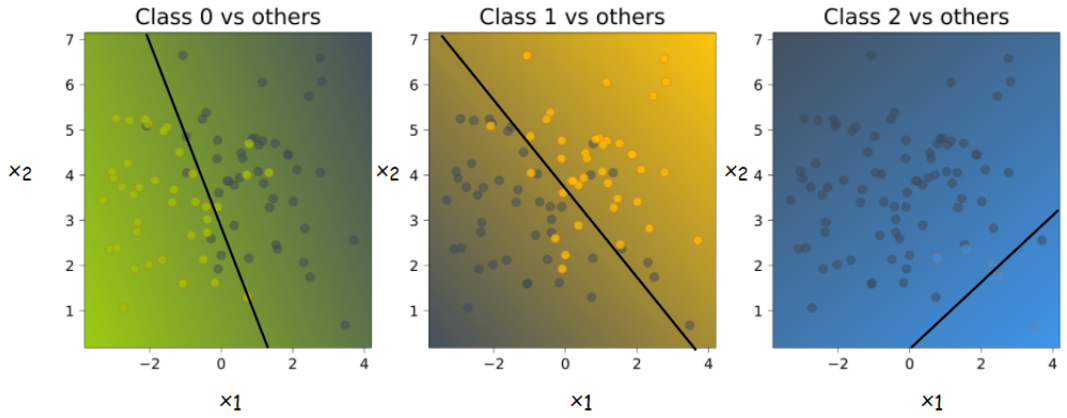
Мультиклассовая классификация

Для мультиклассовой классификации используется подход, который называется **«один против всех» (one-vs-all)**.

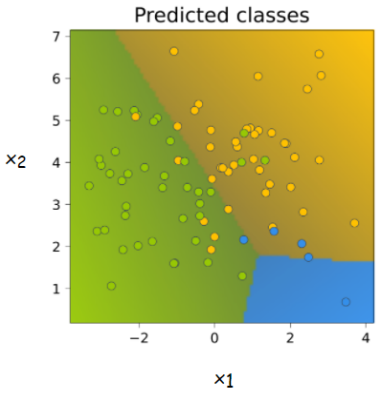
Если у нас есть k различных классов (k>2), давайте обучим k классификаторов, каждый из которых будет предсказывать вероятности принадлежности каждого объекта к определённому классу.



Если мы используем в качестве классификатора логистическую регрессию и количество факторов равно двум (x_1 и x_2), то можно изобразить тепловую карту вероятностей принадлежности к каждому из классов в каждой точке пространства, а также разделяющие плоскости, которые образуются при пороге вероятности в 0.5.



После объединения результатов:



Обобщение логистической регрессии на мультиклассификацию

Модель логистической регрессии легко обобщается на случай мультиклассовой классификации.

Пусть мы построили несколько разделяющих плоскостей с различными наборами параметров ω_k, где k — номер классификатора. То есть имеем K разделяющих плоскостей:

$$z_k = \omega_{0k} + \sum_{j=1}^{m} \omega_{jk} x_j = \omega_k \cdot x$$

Чтобы преобразовать результат каждой из построенных моделей в вероятности в логистической регрессии, используется функция **softmax** — многомерный аналог сигмоиды:

$$\widehat{P}_k = softmax(z_k) = \frac{exp(\widehat{y_k})}{\sum_{k=1}^{K} exp(\widehat{y_{jk}})}$$

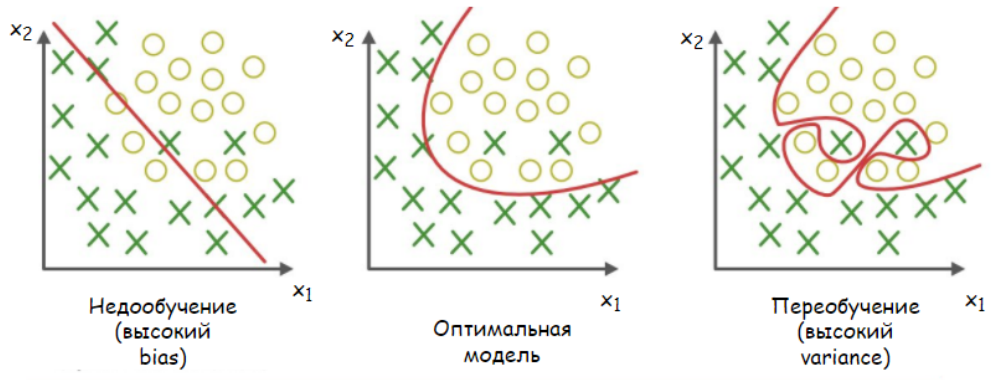
Данная функция выдаёт нормированные вероятности, то есть в сумме для всех классов вероятность будет равна 1.

Достоинства и недостатки логистической регрессии

Достоинства	Недостатки
➔ Простой, интерпретируемый, но в то же время эффективный алгоритм. Благодаря этому он является очень популярным в мире машинного обучения. ➔ Поиск параметров линейный или квадратичный (в зависимости от метода оптимизации), то есть ресурсозатратность алгоритма очень низкая. ➔ Не требует подбора внешних параметров (гиперпараметров), так как практически не зависит от них.	➔ Хорошо работает, только когда классы линейно разделимы, что бывает очень редко в реальных задачах. Поэтому обычно данная модель используется в качестве baseline.

Недостаток с линейной разделимостью классов можно побороть с помощью введения **полиномиальных признаков**, тем самым снизив смещение модели. Тогда логистическая регрессия вместо разделяющей плоскости будет означать выгнутую разделяющую поверхность сложной структуры.

Однако мы знаем, что с этим трюком стоит быть аккуратным, так как можно получить переобученную модель. Поэтому в комбинации с полиномиальными признаками стоит подобрать наилучший параметр регуляризации.



Деревья решений

Дерево решений предсказывает значение целевой переменной с помощью применения последовательности простых решающих правил. Этот процесс в некотором смысле согласуется с естественным для человека процессом принятия решений.

Пример обученного дерева решений:

Успешнее всего деревья применяют **в следующих областях:**

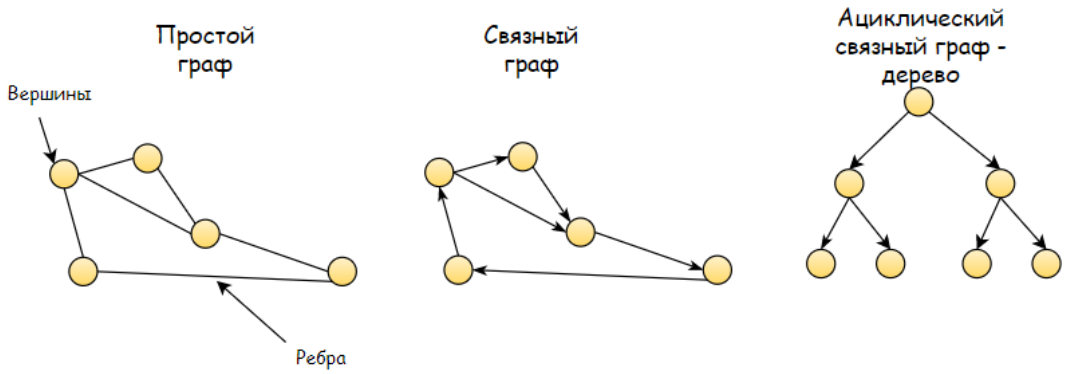
- *Банковское дело.* Оценка кредитоспособности клиентов банка при выдаче кредитов.
- *Промышленность.* Контроль качества продукции (обнаружение дефектов в готовых товарах), испытания без нарушений (например, проверка качества сварки) и т. п.
- *Медицина.* Диагностика заболеваний разной сложности.
- *Молекулярная биология.* Анализ строения аминокислот.
- *Торговля.* Классификация клиентов и товара.

Формально структура дерева решений — **это связный ациклический граф**.

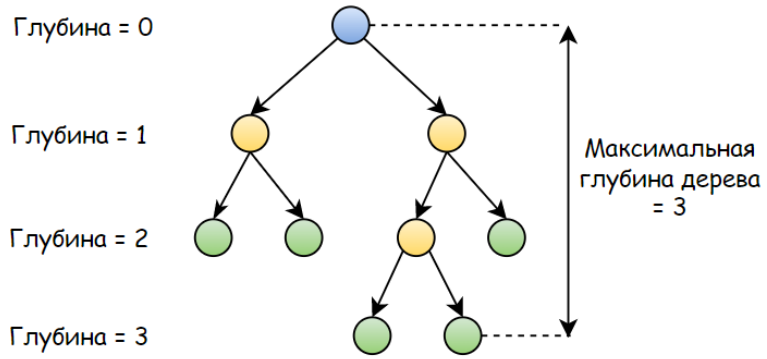
Граф — это абстрактная топологическая модель, которая состоит из вершин и соединяющих их рёбер.

Связный граф — это граф, в котором между любой парой существует направленная связь.

Ациклический граф — это граф, в котором отсутствуют циклы, то есть в графе не существует такого пути, по которому можно вернуться в начальную вершину.



В дереве решений можно выделить **три типа вершин:**



- **Корневая вершина (root node)** — то, откуда всё начинается. Это первый и самый главный вопрос, который дерево задаёт объекту. В примере со страхованием это был «возраст автовладельца > 40».

- **Внутренние вершины (intermediate nodes)**. Это дополнительные уточняющие вопросы, которые дерево задаёт объекту.
- **Листья (leafs)** — конечные вершины дерева. Это вершины, в которых содержится конечный «ответ» — класс объекта.

Максимально возможная длина от корня до самых дальних листьев (не включая корневую) называется **максимальной глубиной дерева (max depth)**.

Во внутренней или корневой вершине признак проверяется на некий логический критерий, по результатам которого мы движемся всё глубже по дереву. Например, «количество кредитов <= 1».

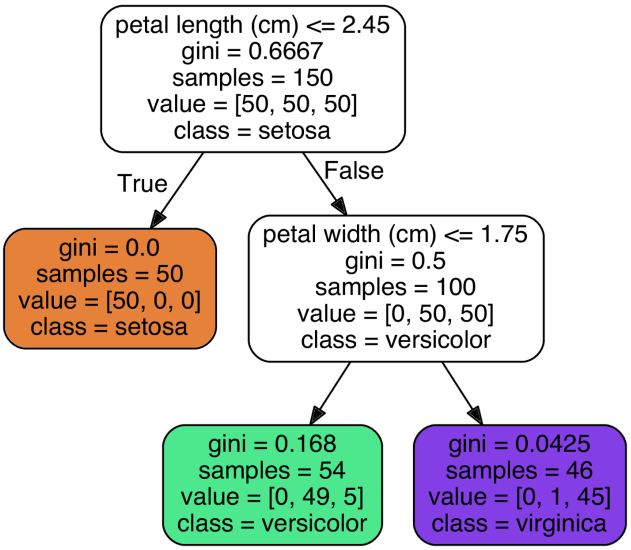
Логический критерий, который находится в каждой вершине, называется **предикатом, или решающим правилом**.

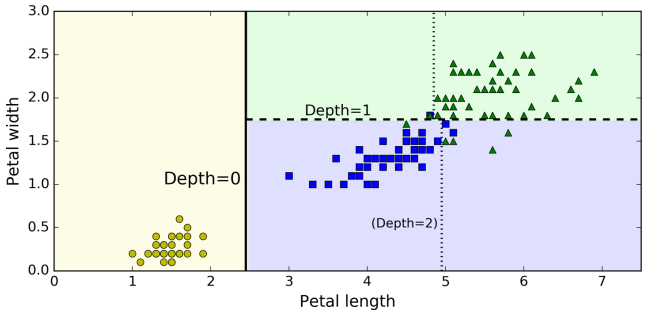
На самом деле все предикаты — это просто взятие порога по значению какого-то признака. Формально это записывается следующим образом:

$$B_v(x,\ t)\ =\ I[x_j \leq t]$$

Каждый новый вопрос дерева решений при его обучении разбивает пространство признаков на две части: в первую отправляются наблюдения, для которых предикат истинен, а во вторую — те, для которых он ложен.

Пример:





Построение дерева решений: алгоритм CART

> **CART (Classification and Regression Tree)** — это алгоритм, который предназначен для построения **бинарных деревьев решений** (деревьев, у которых каждая вершина связана с двумя другими вершинами нижнего уровня). Данный алгоритм, как следует из его названия, предназначен для решения задач классификации и регрессии.

Построение дерева решений можно описать **рекурсией**. Каждая вершина дерева порождает две других вершины, а они в свою очередь порождают новые вершины, и так происходит до тех пор, пока не выполнится некоторый критерий остановки, например в вершине не останутся только наблюдения определённого класса.

Псевдокод рекурсивной функции для построения решающего дерева будет выглядеть следующим образом:

```
def build_decision_tree(X, y):
    node = Node()
    if stopping_criterion(X, y) is True:
        node = create_leaf_with_prediction(y)
            return node
    else:
        X_left, y_left, X_rigth, y_rigth = best_split(X, y)
        node.left = build_decision_tree(X_left, y_left)
        node.right = build_decision_tree(X_rigth, y_rigth)
```

Вершина дерева node задаёт целое поддерево идущих за ним вершин, если такие имеются, а не только саму вершину.

Центральный момент в построении дерева решения по обучающему набору данных — найти такой предикат $B_v(x_j, t) = I[x_j \leq t]$, который обеспечит наилучшее разбиение выборки на классы.

Признак x_j и его пороговое значение t в каждой из вершин и есть **внутренние параметры дерева решений**, которые мы пытаемся отыскать.

Поиск параметров дерева решений

В деревьях мы пытаемся выбрать такие признаки x_j и их пороговые значения t, при которых произойдёт разделение набора на две части наилучшим по какому-то критерию образом. В нашем псевдокоде этот процесс организован в виде функции `best_split()`.

Важно понимать, что дерево решений — это топологический алгоритм, а не аналитический, то есть структуру дерева не получится описать в виде какой-то формулы, как те же линейные модели. Поэтому про стандартные методы оптимизации, такие как градиентный спуск или тем более метод наименьших квадратов можно забыть.

Когда мы работаем с набором данных, у нас ограниченное количество признаков и для них есть ограниченное количество порогов. Тогда мы можем полным перебором найти такую комбинацию j и t, которая обеспечит наилучшее уменьшение неопределённости.

Неопределённость можно измерять различными способами, в деревьях решений для этого используются **энтропия Шеннона** и **критерий Джини**.

Функция ошибки дерева:

$$L(j, t) = \frac{n_v^{left}}{n_v} H(Q_{left}) + \frac{n_v^{right}}{n_v} H(Q_{right})$$

где $H(Q)$ — это функция, которая называется **критерием информативности**. Её значение уменьшается с уменьшением разброса ответов на выборке.

Критерии информативности:

1. **Энтропия Шеннона:**

$$H(Q) = -\sum_{i=1}^{k} P_i log_2 P_i$$

где k — количество классов. p_i — вероятность принадлежности объекта к k-ому классу. log_2 — логарифм по основанию 2.

2. **Критерий Джини:**

$$H(Q) = -\sum_{i=1}^{k} P_i(1 - P_i)$$

где k — количество классов. P_i — вероятность принадлежности объекта к k-ому классу.

Деревья решений в sklearn

Модель дерева решений для решения задачи классификации реализована в классе **DecisionTreeClassifier**. Данный класс реализует обучение по алгоритму CART. Расположение — модуль tree в библиотеке sklearn.

```
from sklearn import tree
```

Основные параметры DecisionTreeClassifier:

- `criterion` — критерий информативности. Может быть равен `'gini'` — критерий Джини и `'entropy'` — энтропия Шеннона.
- `max_depth` — максимальная глубина дерева. По умолчанию — `None`, глубина дерева не ограничена.
- `max_features` — максимальное число признаков, по которым ищется лучшее разбиение в дереве. По умолчанию — `None`, то есть обучение производится на всех признаках.
  Это нужно потому, что при большом количестве признаков будет «дорого» искать лучшее (по критерию типа прироста информации) разбиение среди всех признаков.
- `min_samples_leaf` — минимальное число объектов в листе. По умолчанию — 1.
  У этого параметра есть понятная интерпретация: если он равен 5, то дерево будет порождать только те решающие правила, которые верны как минимум для 5 объектов.

- `random_state` — параметр, отвечающий за генерацию случайных чисел.

Обучение (поиск параметров) — метод fit():

```
#Создаём объект класса DecisionTreeClassifier
dt_clf = tree.DecisionTreeClassifier(
    criterion='entropy',
    max_depth=3,
    random_state=42
)
#Обучаем дерево решений по алгоритму CART
dt_clf.fit(X, y)
```

Предсказание класса — метод predict():

```
y_pred = dt_clf.predict(X)
```

Предсказание вероятностей принадлежности — метод predict_proba():

```
y_pred_proba = dt_clf.predict_proba(X)
```

Обученное дерево можно визуализировать в виде графа, чтобы посмотреть, как дерево делает предсказание. Для этого есть функции `plot_tree()` из модуля tree.

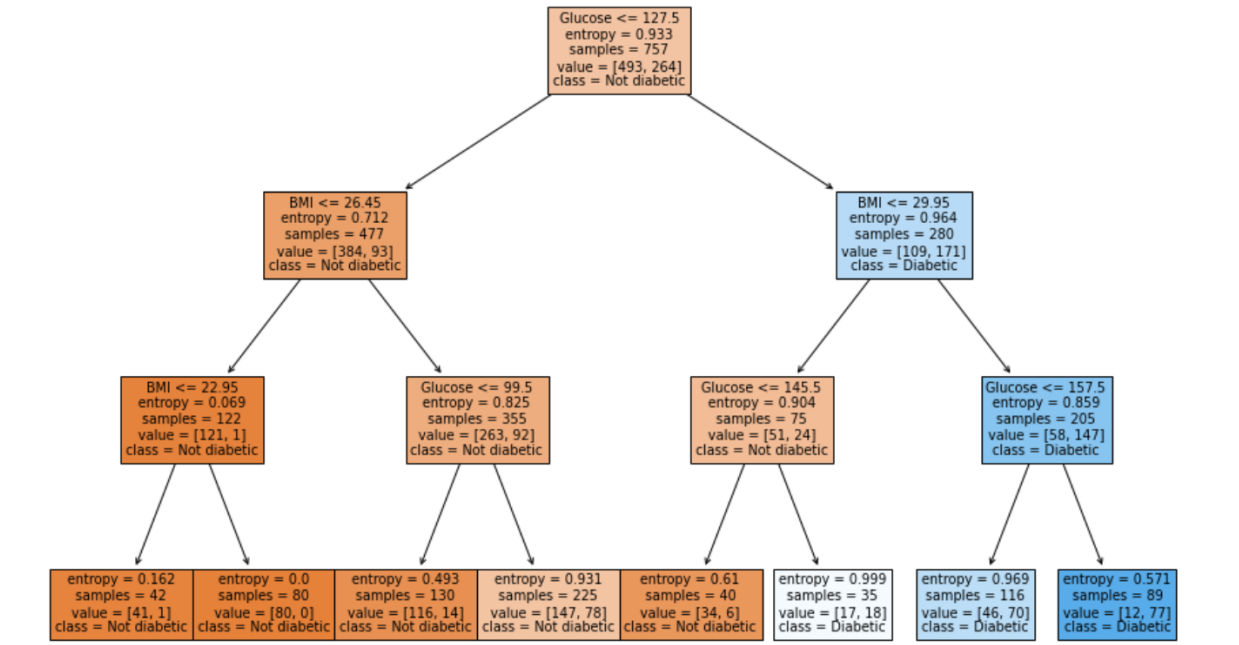
Основные параметры функции:

- `decision_tree` — объект обученного дерева решений,
- `feature_names` — наименования факторов,
- `class_names` — названия классов,
- `filled` — добавлять ли расцветку вершин графа.

```
tree.plot_tree(
    dt_clf,
    feature_names=X.columns,
    class_names=["0 - Not diabetic", "1 - Diabetic"],
    filled=True,
```

```
);
```

Пример:



В каждой из вершин записана следующая информация:

- предикат, по которому происходит разбиение;
- entropy — значение энтропии в текущей выборке;
- samples — количество объектов;
- values — количество объектов каждого из классов;
- class — преобладающий класс, на основе которого будет сделано предсказание.

Коэффициенты важности признаков — атрибут feature_importances_():

```
print(dt_clf.feature_importances_)
```

Достоинства и недостатки деревьев решений

Достоинства	Недостатки
➔ Дерево решений не требует нормализации/стандартизации данных. ➔ Наличие пропусков не оказывает существенного влияния на построение дерева. ➔ За счёт своей простоты модель деревьев решений интуитивно понятна и легко объяснима даже людям, не разбирающимся в методе. ➔ Приятный побочный эффект построения дерева решений — получение значимости признаков. Однако коэффициенты значимости целиком и полностью зависят от сложности дерева.	➔ В силу дискретной топологической структуры дерево не дифференцируется по параметрам — стандартные алгоритмы поиска параметров, такие как градиентный спуск, не работают. Приходится использовать полный перебор. ➔ Метод является жадным — долго обучается из-за полного перебора. Требует больших затрат вычислительных мощностей (по сравнению с другими алгоритмами). Особенно это ощутимо при большом количестве признаков на глубоких деревьях. ➔ Огромная склонность к переобучению. Необходим подбор внешних параметров — max_depth, min_sample_leaf и другие. ➔ Небольшое изменение в данных может заметно повлиять на структуру дерева. ➔ При работе с непрерывными числовыми признаками дерево делит их на категории и теряет информацию. Лучше всего дерево работает, если перевести числовые признаки в категориальные.

Ансамблевые алгоритмы

Ансамблевые модели, или просто **ансамбли (ensembles)** — это метод машинного обучения, где несколько простых моделей (часто называемых

«слабыми учениками») обучаются для решения одной и той же проблемы и объединяются для получения лучших результатов.

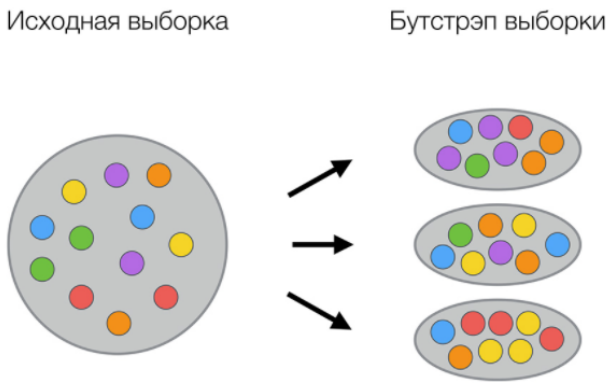
Существует **три проверенных способа построения ансамблей:**

- **Бэггинг** — параллельно обучаем множество **одинаковых** моделей, а для предсказания берём среднее по предсказаниям каждой из моделей.
- **Бустинг** — последовательно обучаем множество **одинаковых** моделей, где каждая новая модель концентрируется на тех примерах, где предыдущая допустила ошибку.
- **Стекинг** — параллельно обучаем множество **разных** моделей, отправляем их результаты в финальную модель, и уже она принимает решение.

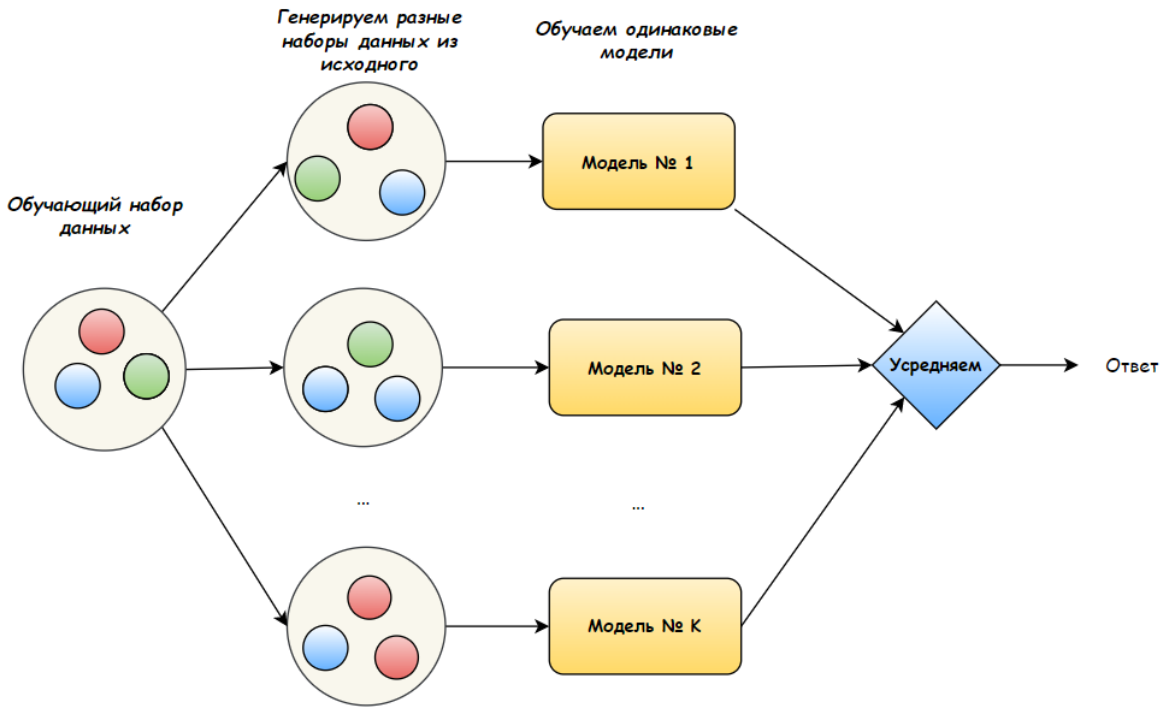
Бэггинг

Бэггинг (bagging) — это алгоритм построения ансамбля путём параллельного обучения множества независимых друг от друга моделей.

В основе алгоритма лежит статистический метод, который называется **бутстрэпом (bootstrap)**. Идея бутстрэпа заключается в генерации n выборок размера b (бутстрэп-выборок) из исходного набора данных размера m путём случайного выбора элементов с повторениями в каждом из наблюдений.



Обучим n одинаковых моделей на каждой из сгенерированных выборок, сделаем предсказания, а затем усредним их. Так мы и получим бэггинг.



В бэггинге в голосовании принимает участие модель одного вида. Эта модель называется **базовой моделью (base model)**.

Утверждения:

- ➔ Смещение (bias) бэггинг-ансамбля не больше (≤) смещения одного алгоритма из этого ансамбля.
- ➔ Однако разброс (variance) бэггинг-ансамбля в k раз меньше, чем разброс одного алгоритма из ансамбля, где k — количество алгоритмов в ансамбле.

Случайный лес

Случайный лес (Random Forest) — это самая распространённая реализация бэггинга, основанная на использовании в качестве базовой модели дерева решений.

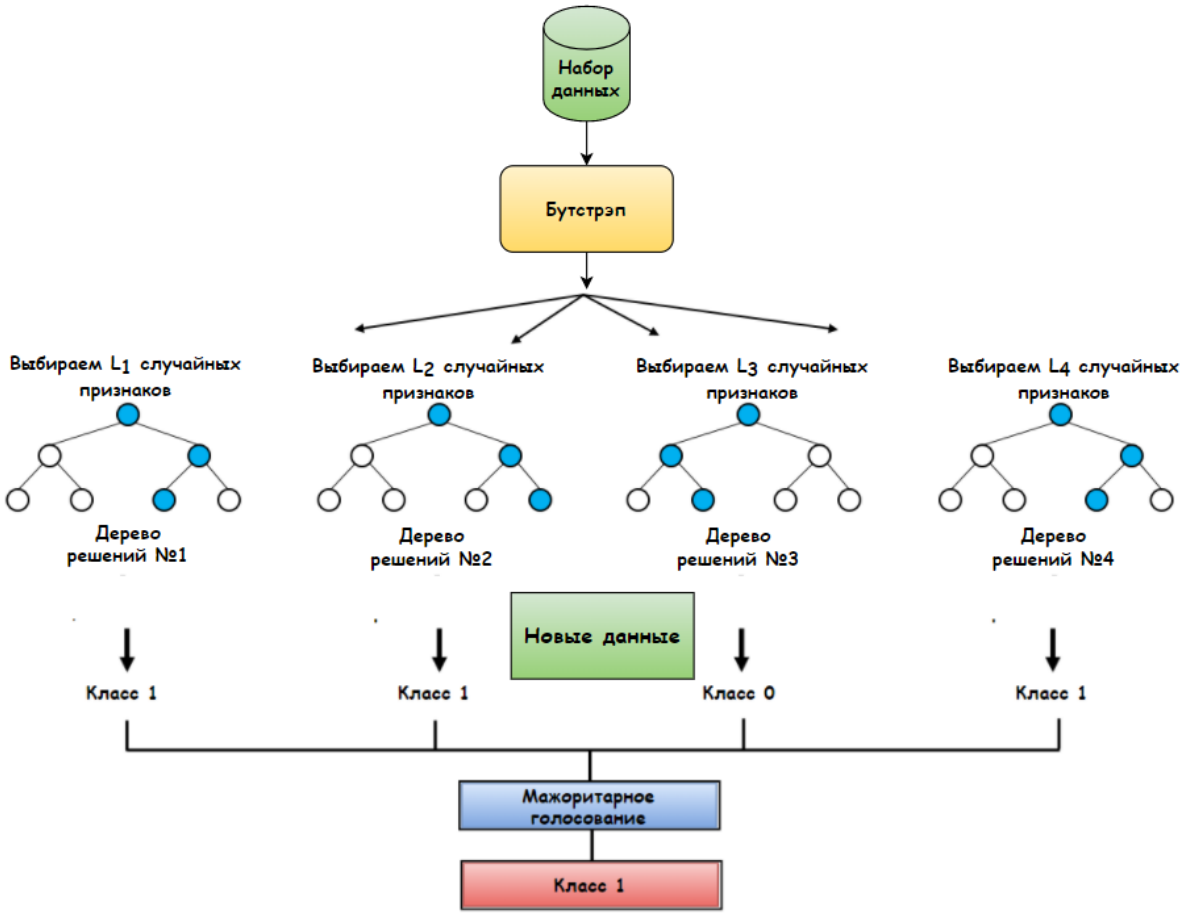
Помимо бутстрэпа, случайный лес использует метод случайных подпространств. Метод заключается в том, что каждая модель обучается не на всех признаках, а только на части из них. Такой подход позволяет уменьшить коррелированность между ответами деревьев и сделать их независимыми друг от друга.

Алгоритм построения случайного леса для задачи классификации

Пусть количество объектов в наборе данных равно N, а количество признаков — M. То есть размер набора данных $(N,\ M)$. Количество деревьев в лесу равно K. Тогда для обучения случайного леса необходимо выполнить следующие шаги:

1. С помощью бутстрэпа создать K наборов данных размера $(N,\ M)$.
2. Для каждого сгенерированного набора данных применить метод случайных подпространств: выбрать $L\ <\ M$ случайных признаков и получить K новых наборов данных размером $(N,\ L)$.
3. На каждом наборе данных обучить K деревьев решений.

Предсказание на новых данных осуществляется путём объединения результатов отдельных деревьев мажоритарным голосованием или путём комбинирования вероятностей.



Случайный лес в sklearn

В sklearn все ансамблевые методы реализованы в модуле ensemble.

```
from sklearn import ensemble
```

В библиотеке sklearn модель случайного леса для решения задачи классификации реализована в классе **RandomForestClassifier**.

Основные параметры RandomForestClassifier:

- `n_estimators` — количество деревьев в лесу (число K из бэггинга). По умолчанию — 100.
- `criterion` — критерий информативности разбиения для каждого из деревьев. Может быть равен `'gini'` — критерий Джини и `'entropy'` — энтропия Шеннона. По умолчанию — `'gini'`.
- `max_depth` — максимальная глубина одного дерева. По умолчанию — `None`, то есть глубина дерева не ограничена.
- `max_features` — максимальное число признаков, которые будут использоваться каждым из деревьев (число L из метода случайных подпространств). По умолчанию `'sqrt'` — для обучения каждого из деревьев используется $\sqrt{m}$ признаков, где m — число признаков в начальном наборе данных.
- `min_samples_leaf` — минимальное число объектов в листе. По умолчанию — 1.
- `random_state` — параметр, отвечающий за генерацию случайных чисел.

Обучение (поиск параметров) — метод fit():

```
rf = ensemble.RandomForestClassifier(
    n_estimators=300,
    criterion='entropy',
    max_depth=6,
    max_features='sqrt',
    random_state=42
)
#Обучаем модель
rf.fit(X, y)
```

Предсказание класса — метод predict():

```
y_pred = rf.predict(X)
```

Предсказание вероятностей принадлежности — метод predict_proba():

```
y_pred_proba = rf.predict_proba(X)
```

Коэффициенты важности признаков — атрибут feature_importances_():

```
print(rf.feature_importances_)
```