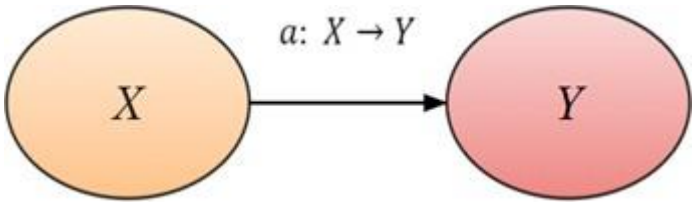
Постановка задачи

Пусть задано обучающее множество пар объектов и ответов к ним $Q = \{(x, y)\}$, где $x \in X$ – векторы из $\mathbb{R}^M$, описываемые M координатами (признаками), $y \in Y$ – целевая метка объекта, $|Q| = N$.

Множество Y может быть непрерывным (задача регрессии) с мощностью $|Y| = N$ или дискретным (задача классификации) с мощностью $|Y| = K$, где K — количество классов.

Наша цель – составить функцию $f(x)$ – модель, которая наилучшим образом определяет зависимость между векторами x из множества X и целевой переменной y из множества Y - $f: X \to Y$.



Такую функцию мы будем искать не в аналитическом виде, как мы делали, например, в случае линейной регрессии, а в виде алгоритма, то есть в виде последовательности действий. Обычно в математике алгоритм обозначается как $a(x)$ или $a: X \to Y$.

Алгоритм $a(x)$ мы будем искать в семействе деревьев решений.

Определение решающего дерева

> **Дерево решений** представляет собой последовательность условий вида $x_j \leq t$, которые называются **предикатами** (или **решающими правилами**).

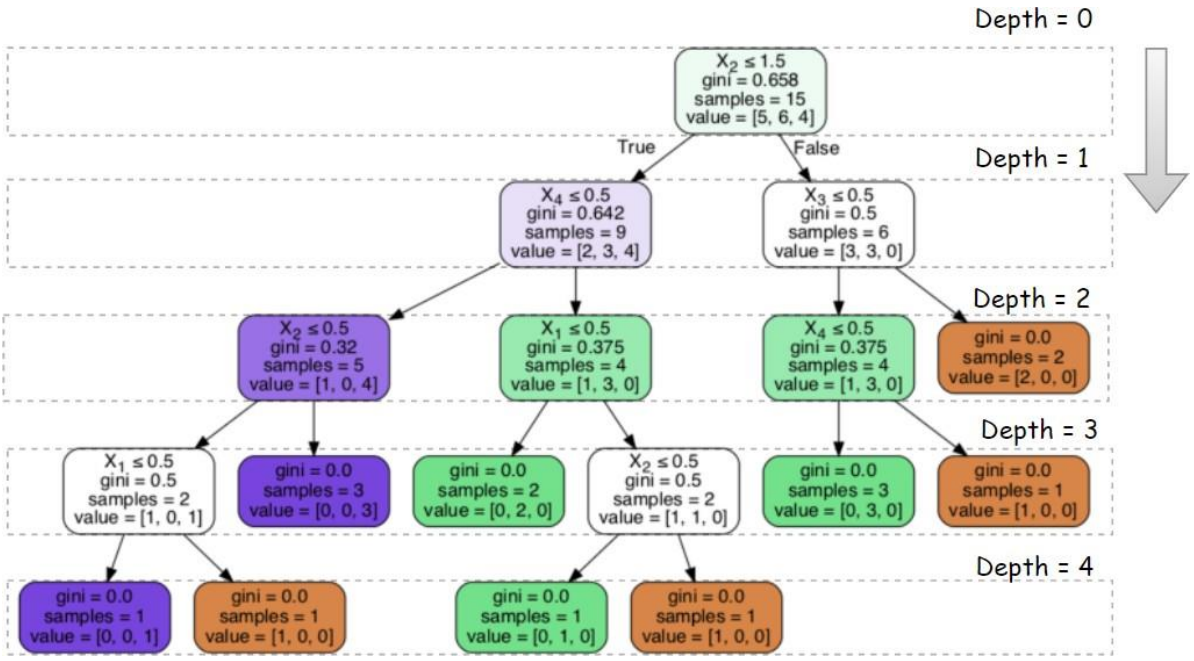
Формально сами предикаты в вершинах мы ранее обозначали как:

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

где v — номер вершины графа, а I с квадратными скобками [] — обозначение индикаторной функции. Последняя равна 1 (True), если условие внутри скобок выполняется, и 0 (False) — в противном случае.

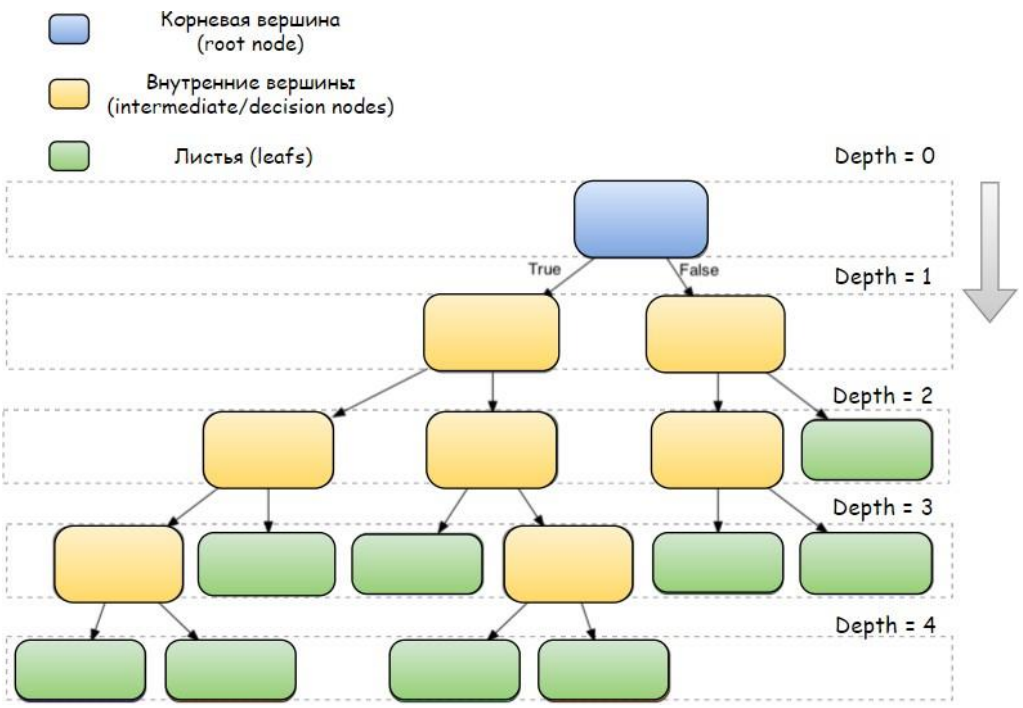
Каждый предикат разделяет множество объектов X на два подмножества: левое – множество тех объектов, для которых заданное в предикате условие выполняется ($X^{left} = \{x \mid x_j \leq t\}$), и правое – множество тех объектов, для которых условие ложно ($X^{right} = X \setminus X^{left} = \{x \mid x_j > t\}$).

Последовательность предикатов лучше всего представлять в виде **ациклического связного графа**. Пример:



В построенном графе выделяют **три типа вершин:**

- Корневая вершина (*root node*) – откуда всё начинается.
- Внутренние вершины (*intermediate nodes*).
- Листья (*leafs*) – конечные вершины дерева. Это вершины, в которых определяется конечный «ответ» — прогноз дерева решений.



Вершины графа группируются в уровни, которые называются **глубиной дерева (depth)**. Отсчёт уровней ведётся с 0 (то есть корневая вершина не считается при подсчёте глубины дерева).

Корневая и внутренние вершины содержат предикаты $B(x_j, t)$, на основе которых организуется движение по графу. В ходе предсказания осуществляется проход по дереву к некоторому листу. Для каждого объекта из выборки движение начинается из корня.

В вершине v проход осуществляется вправо, если для объекта x условие, записанное в предикате, выполняется, то есть $B_v(x_j, t) = 1$, и влево, если условие является для объекта x ложным, то есть $B_v(x_j, t) = 0$. Проход продолжается до момента, пока будет достигнут некоторый лист. Ответом алгоритма для объекта x считается прогноз $\hat{y} = a(x)$, приписанный этому листу.

В задаче регрессии прогноз целевой переменной $\hat{y}$ будет определяться принципом усреднения, то есть объекту x в качестве прогноза

В задаче классификации прогноз целевой переменной $\hat{y}$ будет определяться принципом голосования большинства, то есть

присваивается среднее (или медианное) значение целевой переменной по объектам, попавшим в лист на этапе обучения дерева.	объекту x присваивается самый популярный класс объектов, попавших в лист на этапе обучения дерева.

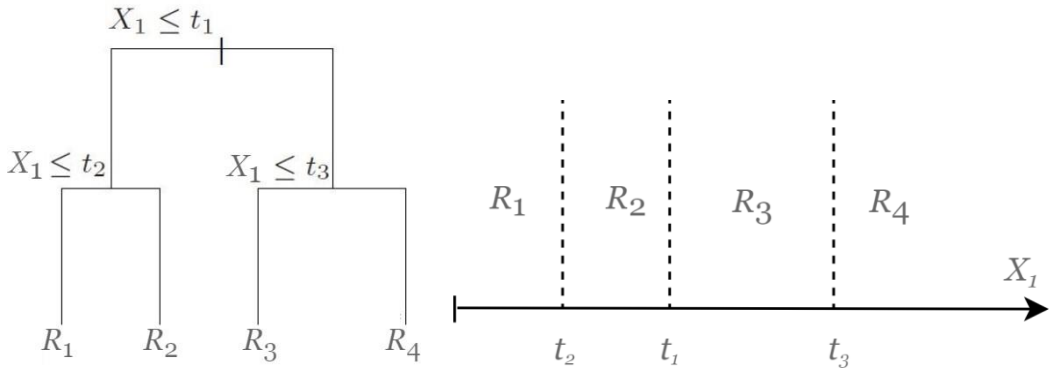
Геометрическая интерпретация дерева решений

Геометрически каждый предикат $B(x_j, t) = [x_j \leq t]$ задаёт в пространстве факторов разделяющую плоскость. Эта плоскость всегда будет перпендикулярна оси фактора x_j и будет делить пространство факторов на две части: ту, для которой условие $x_j \leq t$ выполняется, и ту, для которой это условие ложно.

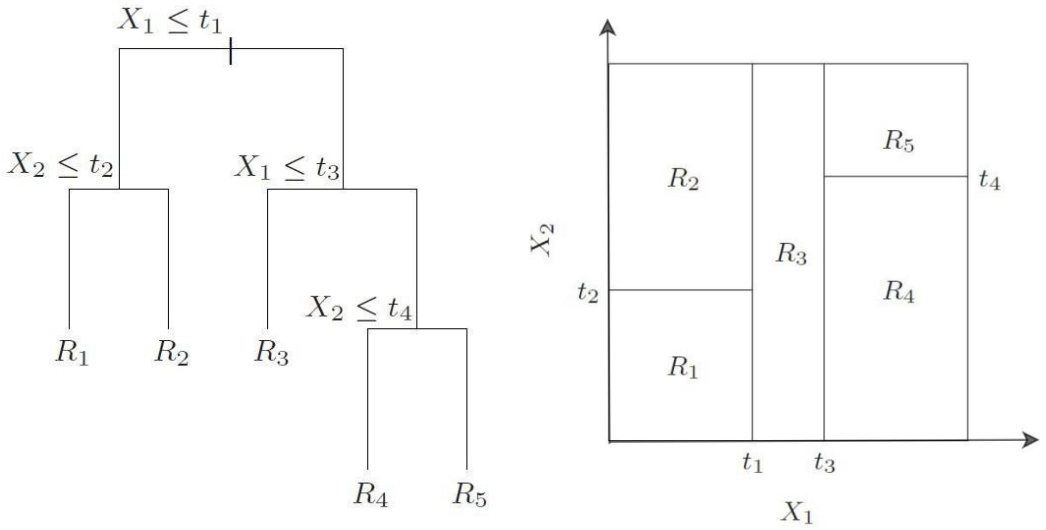
Следующие предикаты будут делить каждую из образованных при прошлом разделении областей на две другие части, и так далее, пока не будут образованы листовые вершины.

В результате построения дерева решений мы получаем пространство факторов, разделённое на l областей, каждая из которых соответствует своей листовой вершине. Обозначим эти области как R_p, $p = 1, 2, \ldots, l$.

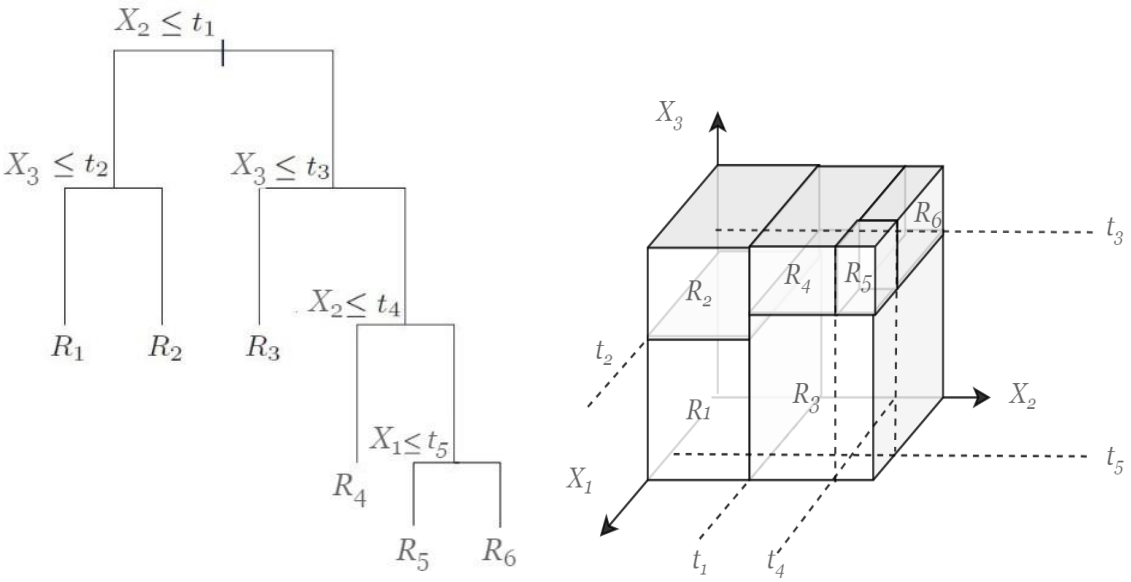
Случай для $M = 1$:



Случай для $M = 2$:



Случай для $M = 3$:



В общем случае, когда факторов m штук, то есть объект $x \in X$ является вектором из пространства $\mathbb{R}^m$, мы получаем m-мерное пространство, разбитое на области, представляющие собой m-мерные параллелепипеды или, как их называют в математике, **гиперпараллелепипеды**.

В случае задачи регрессии (Decision Tree Regressor) ответ модели (значение целевого признака) будет определяться как среднее значение

В случае задачи классификации (Decision Tree Classifier) ответ модели (класс объекта) будет определяться голосованием большинства внутри

целевой переменной y внутри области R_p, соответствующей листовой вершине:

$$\hat{y} = a(x) = \frac{1}{|R_p|} \sum_{y \in R_p} y$$

области R_p. То есть дерево будет возвращать тот класс, который наиболее популярен в области R_p, соответствующей листовой вершине:

$$P_k = \frac{1}{|R_p|} \sum_{y \in R_p} [y = k],$$

$$\hat{y} = a(x) = argmax_{k \in K}(P_k)$$

Свойства алгоритма дерева решений

Из структуры дерева решений следует несколько интересных свойств:

- Полученная функция $a(x)$ является кусочно-постоянной, а у таких функций производная равна нулю во всех точках, где задана функция. Следовательно, при поиске оптимального решения о градиентных методах, таких как SGD, можно забыть.
- В случае задачи регрессии дерево решений (в отличие от, например, линейной регрессии) не может предсказывать значения целевой переменной за границами области значений обучающей выборки (на самом деле может, но для этого нужны специальные манипуляции над данными, которые не входят в рамки данного модуля).
- Дерево решений способно идеально приблизить обучающую выборку, но при этом ничего не выучить: для этого достаточно построить такое дерево, в каждый лист которого будет попадать только один объект.

  Такая модель будет обладать идеальным качеством на обучающей выборке, однако при использовании на реальных данных качество будет плохим. Иначе говоря, модель будет переобученной.

Методы построения дерева решений. Алгоритм CART

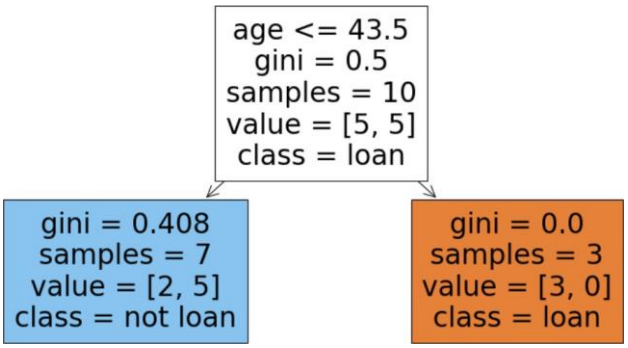
CART (Classification and Regression Tree) – алгоритм, предназначенный для построения **бинарных деревьев решений** (деревьев, у которых каждая вершина связана с двумя другими вершинами нижнего уровня). Алгоритм очень похож на C4.5, однако, в отличие от последнего, предназначен как для

задач классификации, так и для задач регрессии. CART важен для нас, поскольку именно он используется для построения моделей решающих деревьев в sklearn.

Пусть у нас есть множество наблюдений $x \in X$ и правильных ответов к ним $y \in Y$. Необходимо построить алгоритм $a(x)$ — дерево решений, которое для объектов из множества X выдаёт ответ из множества Y. Алгоритм $a(x)$ зависит от некоторых внутренних параметров w.

Под внутренними параметрами w понимается структура дерева, а именно последовательность предикатов $B_v(x_j, t) = [x_j \leq t]$ и сами параметры: j — номер признака, по которому строится условие в предикате, и t — пороговое значение для условия. То есть $w = (j, t)$.

Чтобы понять, как происходит построение дерева произвольной глубины по алгоритму CART, нам сначала необходимо научиться строить **решающие пни** — деревья глубины 1. Пример такого решающего пня вы можете видеть ниже:



Корневая вершина в данном случае является **родительской**, а две другие – её **потомками**.

Чтобы построить решающий пень, нам нужно лишь определить параметры предиката в родительской вершине — j и t.

Кандидаты в параметры разбиения

Будем рассматривать простые предикаты вида:

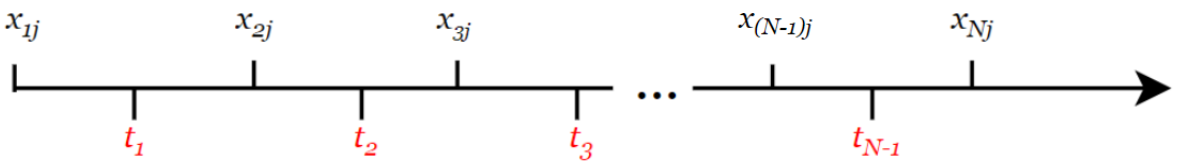
$$B(x_j, t) = [x_j \leq t]$$

Введём критерий оптимальности – некоторую меру **неоднородности (impurity)** $G(X,\ y,\ w)$, которая будет показывать, насколько большой разброс целевой переменной y для объектов из множества X наблюдается в дочерних вершинах при параметрах сплита w.

Будем строить решающий пень так, чтобы минимизировать $G(X, y,\ w)$:

$$G(X,\ y, w) \to \min_{w}$$

Параметры $w = (j, t)$ можно найти простым перебором. В качестве кандидатов на пороговое значение t можно рассматривать среднее значение между двумя соседними уникальными значениями отсортированного фактора x_j.



Для каждого из возможных предикатов $B_v(x_j,\ t)$ нам необходимо подсчитать значение неоднородности $G(X, y, w)$ на всей выборке и определить такую комбинацию $w_{opt} = (j_{opt}, t)$, которая даёт минимум:

$$w_{opt} = \operatorname{argmin}_{w} G(X,\ y, w)$$

Псевдокод такого алгоритма будет выглядеть так (запускать его не нужно):

```
min_loss = inf
optimal_split_params = None
for j in range(M):
    thresholds = find_candidates_for_thresholds(X[:, j], y)
    for t in thresholds:
        split_params = (j, t)
        loss = calculate_loss(X, y, split_params)
        if loss < min_loss:
            min_loss = loss
            optimal_split_params = split_params
```

Оптимизированный алгоритм поиска пороговых значений для фактора x_j:

1. Отсортировать числовой фактор x_j по возрастанию:
$$\{x_{ij} \mid x_{ij} < x_{(i+1)j}\}$$
2. Вычислить среднее между двумя соседними уникальными значениями фактора:
$$\{x_{ij}^{mean} \mid x_{ij}^{mean} = \frac{x_{ij} + x_{(i+1)j}}{2}\}$$
3. В качестве кандидатов на пороговые значений t выбираются только те значения, при которых целевой признак меняет своё значение.
$$\{x_{ij}^{mean} \mid y_i - y_{(i+1)} \neq 0\}$$

Ветвление. Неоднородность

Пары объектов и ответов y, соответствующих им, обозначим буквой $Q = \{(x, y)\}$. Пусть это множество Q содержит $N = |Q|$ объектов из обучающей выборки.

? Как сильно ответы в выборке Q различаются между собой? Иначе говоря: какой разброс целевой переменной в выборке Q?

Чтобы это измерить, ввёдем некоторую функцию $H(Q)$, которую назовём **критерием неоднородности (impurity criterion)**, или **критерием информативности** (более распространённое название).

Предположим, что мы выбрали конкретные параметры w. Тогда множество Q разбивается на две части:

$$Q^{left} = \{(x, y) \mid x_j \leq t\}$$
$$Q^{right} = Q \setminus Q^{left} = \{(x, y) \mid x_j > t\}$$

Каждая из полученных выборок будет иметь свои размеры — мощности множеств, назовём их $N^{left} = |Q^{left}|$ и $N^{right} = |Q^{right}|$.

? Насколько уменьшился разброс целевой переменной после сплита с такими параметрами?

Для ответа на этот вопрос необходимо вычислить **взвешенную неоднородность** $G(Q, w)$ в полученных левой и правой частях:

$$G(Q, w) = \frac{N^{left}}{N} H(Q^{left}) + \frac{N^{right}}{N} H(Q^{right})$$

? Какие параметры w нам подойдут?

Нам нужно выбрать такие параметры w для предиката $B(x_j, t)$, при которых неоднородность после ветвления будет наименьшей.

$$G(Q, w) \to \min_w$$

$$w_{opt} = (j_{opt}, t_{opt}) = \operatorname{argmin}_w G(Q, w)$$

От вида функции $H(Q)$, зависит то, как будет выглядеть итоговое выражение для неоднородности. Её значение должно уменьшаться с уменьшением разброса ответов на выборке.

Дополнительно введём такое понятие как **прирост информации (information gain)**:

$$IG(Q, w) = H(Q) - G(Q, w) = H(Q) - \frac{N^{left}}{N} H(Q^{left}) - \frac{N^{right}}{N} H(Q^{right})$$

Интерпретация $IG(Q_v, w)$: сколько новой информации о целевой переменной y удалось получить при использовании предиката $B(x_j, t)$.

Можно рассматривать задачу поиска параметров в контексте прироста информации. Тогда нашей целью будет найти максимум прироста информации:

$$IG(Q, w) \to \max_w$$

Оптимальными будут те параметры w, которые дают наибольший прирост информации в результате разбиения:

$$w_{opt} = (j_{opt}, t_{opt}) = \operatorname{argmax}_w IG(Q, w)$$

Критерии информативности в задаче регрессии

- **Квадратичная ошибка (Squared Error):**

$$H(Q) = \frac{1}{N} \sum_{y \in Q} (y - \bar{y})^2,$$

$$\bar{y} = \frac{1}{N} \sum_{y \in Q} y$$

Данный критерий информативности измеряет дисперсию целевой переменной для объектов в вершине.

- **Абсолютная ошибка (Absolute Error)**:

$$H(Q) = \frac{1}{N} \sum_{y \in Q} |y - median(y)|,$$

Данный критерий информативности измеряет отклонение целевой переменной от медианы для объектов в вершине.

- **Пуассоновская ошибка (Poisson Error)**:

Данный критерий используется редко, но он будет хорошим выбором в задачах, где целевая переменная дискретная и подчинена распределению Пуассона (например, число отказов оборудования).

Формула выводится из распределения Пуассона с помощью метода максимального правдоподобия. Мы опустим сам вывод формулы и приведём только конечный вид критерия информативности:

$$H(Q) = \frac{1}{N} \sum_{y \in Q} (y \log(\frac{y}{\bar{y}}) - y + \bar{y})$$

Критерии информативности в задаче классификации

Все критерии для задачи классификации, которые мы будем рассматривать, основаны на **вероятности принадлежности к классу**. Пусть у нас есть K классов. Тогда оценка вероятности принадлежности к классу под номером k определяется как:

$$P_k = \frac{1}{N} \sum_{y \in Q} [y = k]$$

- **Энтропия Шеннона (entropy):**

$$H(Q) = -\sum_{k=1}^{K} P_k log(P_k)$$

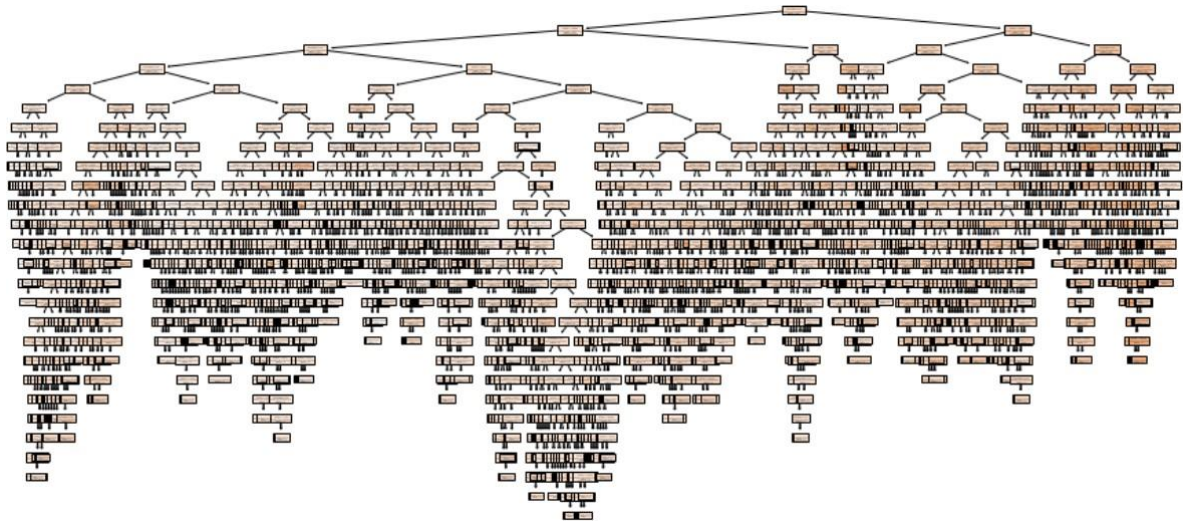
Энтропия численно отражает меру хаоса и активно используется во многих прикладных задачах — от шифрования до машинного обучения.

- **Критерий Джини (gini)**:

$$H(Q) = \sum_{k=1}^{K} P_k (1 - P_k)$$

Деревья произвольной глубины

Легко убедиться, что для любой выборки можно построить решающее дерево, не допускающее на ней ни одной ошибки. Даже с простыми предикатами $[x_j \leq t]$ можно сформировать дерево, в каждом листе которого находится ровно по одному объекту выборки:

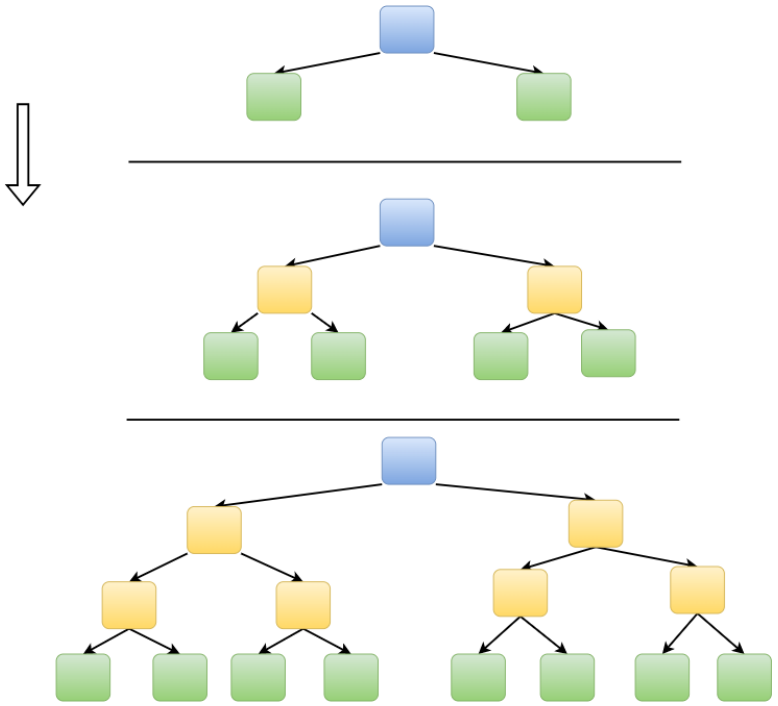


Очевидно, что такие деревья нам не подойдут.

Тогда можно было бы поставить задачу поиска дерева, которое является минимальным (с точки зрения количества листьев) среди всех деревьев, не

допускающих ошибок на обучении — в этом случае можно было бы надеяться на наличие у дерева обобщающей способности. К сожалению, было доказано, что эта задача является слишком сложной (если быть точнее, NP-полной), а время построения такого дерева неприемлемо велико.

Придётся отказаться от идеи поиска оптимальной структуры дерева. Будем искать не всю структуру дерева, а строить его последовательно, уровень за уровнем.



Процедура построения дерева является **рекурсией**. Мы рекурсивно используем алгоритм построения решающего пня для каждой из дочерних вершин, тем самым создавая всё новые и новые вершины графа.

Пусть $Q_v = \{(x, y)\}$ — множество объектов и ответов к ним, попавших в вершину с номером v, $N_v = |Q_v|$. Пусть задан критерий информативности $H(Q)$, а также некоторый критерий остановки рекурсии $stop\_criterion(Q_v)$.

Наш **алгоритм построения дерева решений CART** будет выглядеть следующим образом:

1. Создаём вершину под номером v.
2. Проверяем критерий остановки $stop\_criterion()$.

По умолчанию ветвление дерева решений прекращается, когда достигается **однородность**, то есть $H(Q_v) = 0$.

2.1. Если условие остановки выполнено.

Формируем листовую вершину v и ставим в соответствие этому листу ответ, который будет выдаваться для новых объектов, которые дойдут до этого листа. Назовём эту часть $create\_leaf()$.

- В случае **задачи регрессии**:

$$\hat{y}_v = \frac{1}{N_v} \sum_{y \in Q_v} y$$

или

$$\hat{y}_v = median_{y \in Q_v}(y)$$

- В случае **задачи классификации**:

$$P_{vk} = \frac{1}{N_v} \sum_{y \in Y_v} [y = k],$$

$$\hat{y}_v = argmax_{k \in K}(P_{vk})$$

2.2. Если условие остановки не выполнено.

Строим решающий пень. Формируем решающее правило $B(x_j, t)$ для вершины v. Из возможных комбинаций $w = (j, t)$ находим такую, которая определяет наилучшее разбиение текущего множества объектов и ответов к ним Q_v на две части: $Q_v^{left} = \{(x, y) | x_j \leq t\}$ и $Q_v^{right} = \{(x, y) | x_j > t\}$:

$$G(Q, w) = \frac{N_v^{left}}{N_v} H(Q_v^{left}) + \frac{N_v^{right}}{N_v} H(Q_v^{right})$$

$$w_{opt} = (j_{opt}, t_{opt}) = argmin_w G(Q, w)$$

$$B_v(x_j, t) = [x_{j_{opt}} \leq t_{opt}]$$

Эту часть алгоритма мы обозначали ранее как $best\_split()$.

Для выборок Q_v^{left} и Q_v^{right} процедура будет повторяться рекурсивно, пока не выполнится критерий остановки.

3. Возвращаем созданную вершину.

Критерии остановки

Мы должны иметь такое условие, а лучше целый набор условий, при которых ветвление дерева прекратится. В нашем алгоритме мы назвали данный шаг $stop\_critetion(Q_v)$.

- достижение однородности $H(Q) = 0$;
- ограничение максимальной глубины дерева (параметр `max_depth`);
- ограничение максимального количества листьев в дереве (параметр `max_leaf_node`);
- ограничение минимального количества объектов, при которых допускается ветвление дерева (параметр `min_samples_split`);
- ограничение минимального количества объектов, необходимых для создания листа (параметр `min_samples_leaf`).

Значимость признаков

Вспомним, что такое **прирост информации (information gain)**. Это разница между неоднородностью в вершине v до её деления и взвешенной неоднородностью после деления по предикату $B_v(x_j, t) = [x_j \leq t]$:

$$IG(Q_v, w) = H(Q_v) - G(Q_v, w) = H(Q_v) - \frac{N_v^{left}}{N_v} H(Q_v^{left}) - \frac{N_v^{right}}{N_v} H(Q_v^{right})$$

Умножим всё выражение на N_v:

$$IG(Q_v, w) = N_v \cdot H(Q_v) - N_v^{left} \cdot H(Q_v^{left}) - N_v^{right} \cdot H(Q_v^{right})$$

Рассчитаем суммарный прирост информации, получаемый деревом от фактора x_j, на один объект из обучающей выборки. Эту величину как раз и назовём **(абсолютной) значимостью признака** x_j:

$$F(x_j) = \frac{1}{n} \sum_{v \in V_j} IG(Q_v, w)$$

Здесь запись $\sum_{v \in V_j}$ означает, что мы суммируем прирост информации по тем вершинам , которые содержат признак x_j внутри условия предиката.

Последний шаг — нормировка. Она производится, чтобы сумма всех значимостей была равна 1. Результатом будет **относительная значимость** признаков, которая позволяет воспринимать значение значимости каждого из признаков как выраженный в долях вклад фактора x_j в построение дерева решений:

$$F(x_j) = \frac{F(x_j)}{\sum_{j=1}^{M} F(x_j)}$$

Реализация собственного дерева решений на Python

```
def find_candidates_for_thresholds(x, y):
    x = x.sort_values().drop_duplicates()
    x_roll_mean = x.rolling(2).mean().dropna()
    y = y[x_roll_mean.index]
    y_roll_mean = y.diff()
    candidates = x_roll_mean[y_roll_mean != 0]
    return candidates.values

def squared_error(y):
    y_pred = y.mean()
    return ((y - y_pred) ** 2).mean()

def entropy(y):
    p = y.value_counts(normalize=True)
    entropy = -np.sum(p * np.log2(p))
    return entropy

def split(X, y, split_params):
```

```
    j, t =  split_params
    predicat = X.iloc[:, j] <= t
    X_left, y_left = X[predicat], y[predicat]
    X_right, y_right = X[~predicat], y[~predicat]
    return X_left, y_left, X_right, y_right

def calculate_weighted_impurity(X, y, split_params, criterion):
    X_left, y_left, X_right, y_right = split(X, y, split_params)
    N, N_left, N_right  = y.size, y_left.size, y_right.size
    score = N_left / N * criterion(y_left) + N_right / N *
criterion(y_right)
    return score

def best_split(X, y, criterion):
    M = X.shape[1]
    min_weighted_impurity = np.inf
    optimal_split_params = None
    for j in range(M):
        thresholds = find_candidates_for_thresholds(X.iloc[:, j], y)
        for t in thresholds:
            split_params = (j, t)
            weighted_impurity = calculate_weighted_impurity(X, y,
split_params, criterion)
            if weighted_impurity < min_weighted_impurity:
                min_weighted_impurity = weighted_impurity
                optimal_split_params = split_params
    return optimal_split_params

class Node:
    def __init__(self, left=None,
                 right=None, value=None,
                 split_params=None, impurity=None,
                 samples=None, is_leaf=False):
        self.left = left
        self.right = right
        self.split_params = split_params
        self.value =  value
        self.impurity = impurity
        self.samples = samples
        self.is_leaf = is_leaf

def create_leaf_prediction(y):
    value = y.mode()[0]
```

```
    return value

def stopping_criterion(X, y, criterion):
    return criterion(y) == 0

def build_decision_tree(X, y, criterion):
    if stopping_criterion(X, y, criterion):
        value = create_leaf_prediction(y)
        node = Node(
            value=value,
            impurity=criterion(y),
            samples=y.size,
            is_leaf=True
        )
    else:
        split_params = best_split(X, y, criterion=entropy)
        X_left, y_left, X_rigth, y_right = split(X, y, split_params)
        left = build_decision_tree(X_left, y_left, criterion)
        right = build_decision_tree(X_rigth, y_right, criterion)
        node = Node(
            left=left, right=right,
            split_params=split_params,
            impurity=criterion(y),
            samples=y.size
        )
    return node

def print_decision_tree(node, depth=0):
    depth += 1
    if node.is_leaf:
        print('    ' * depth, 'class: {}'.format(node.value))
    else:
        print('    ' * depth, 'feature_{} <=
{:.3f}:'.format(*node.split_params))
        print_decision_tree(node.left, depth=depth)
        print('    ' * depth, 'feature_{} >
{:.3f}:'.format(*node.split_params))
        print_decision_tree(node.right, depth=depth)

def predict_sample(node, x):
    if node.is_leaf:
        return  node.value
    j, t = node.split_params
```

```
    if x[j] <= t:
        return predict_sample(node.left, x)
    else:
        return predict_sample(node.right, x)

def predict(decision_tree, X):
    predictions = [predict_sample(decision_tree, x) for x in X.values]
    return np.array(predictions)
def stopping_criterion(X, y, criterion, max_depth=None, depth=0):
    if max_depth is None:
        return (criterion(y) == 0)
    else:
        return (criterion(y) == 0) or (depth > max_depth)

def build_decision_tree(X, y, criterion, max_depth=None, depth=0):
    depth += 1
    if stopping_criterion(X, y, criterion, max_depth, depth):
        value = create_leaf_prediction(y)
        node = Node(
            value=value,
            impurity=criterion(y),
            samples=y.size,
            is_leaf=True
        )
    else:
        split_params = best_split(X, y, criterion=entropy)
        X_left, y_left, X_rigth, y_right = split(X, y, split_params)
        left = build_decision_tree(X_left, y_left, criterion, max_depth,
depth)
        right = build_decision_tree(X_rigth, y_right, criterion,
max_depth, depth)
        node = Node(
            left=left, right=right,
            split_params=split_params,
            impurity=criterion(y),
            samples=y.size
        )
    return node

def calculate_feature_importances(node, feature_importance=None):
    if feature_importance is None:
        feature_importance = np.zeros(X.shape[1])
    if node.value is None:
```

```
        j = node.split_params[0]
        feature_importance[j] += node.impurity * node.samples - \
                                 node.left.impurity * node.left.samples
- \
                                 node.right.impurity *
node.right.samples
        calculate_feature_importances(node.left, feature_importance)
        calculate_feature_importances(node.right, feature_importance)
    feature_importance /= node.samples
    feature_importance /= feature_importance.sum()
    return feature_importance
```