Решающие деревья

In [1]:

```
import numpy as np
import pandas as pd
import warnings

from sklearn.base import BaseEstimator
from sklearn.metrics import accuracy_score, r2_score
from sklearn.model_selection import GridSearchCV, train_test_split

import matplotlib.pyplot as plt
import seaborn as sns

warnings.filterwarnings('ignore')
sns.set(font_scale=1.6)
```

started 17:38:44 2020-03-14, finished in 808ms

Вспомним, как именно происходит построение решающего дерева. Для построения дерева в каждой нелистовой вершине происходит разбиение подвыборки на две части по некоторому признаку x_j. Этот признак и порог t, по которому будет происходить разбиение, мы хотим брать не произвольно, а основываясь на соображениях оптимальности. Для этого нам необходимо знать некоторый фукционал качества, который будем оптимизировать при построении разбиения.

Обозначим через X_m -- множество объектов, попавших в вершину m, разбиваемую на данном шаге, а через X_l и X_r -- объекты, попадающие в левое и правое поддерево соответственно при заданном правиле $I\{x_j < t\}$. Пусть также H -- используемый критерий информативности (impurity criterion).

Выпишите функционал, который необходимо минимизировать при разбиении вершины:

Ответ:

$$\frac{|X_l|}{|X|}H(X_l) + \frac{|X_r|}{|X|}H(X_r).$$

Реализация критериев информативности.

Вспомните ещё раз, на какой общей идее основаны критерии информативности и какую характеристику выборки они стремятся оптимизировать?

Ответ. Чем меньше разнообразие целевой переменной, тем меньше должно быть значение критерия информативности - и, соответственно, мы будем пытаться минимизировать его значение. Тогда функционал качества $Q(R_m, j, t)$ будет показывать выигрыш при выборе разбиения по предикату {x^j < t}. Его мы пытаемся максимизировать по всем таким предикатам.

Перед тем, как непосредственно работать с решающими деревьями, реализуйте критерии информативности. Использовать готовые реализации критериев или классов для решающих деревьев из `sklearn` и из других библиотек **запрещено.** Также при реализации критериев информативности по причине неэффективности **запрещается использовать циклы**. Воспользуйтесь библиотекой `numpy`.

Каждая функция принимает на вход одномерный `numpy`-массив размерности `(n,)` из значений отклика.

In [2]:

```python
# Код функций, реализующих критерии разбиения.

def mean_square_criterion(y):
    ''' Критерий для квадратичной функции потерь. '''

    return np.mean((y - np.mean(y))**2)


def mean_abs_criterion(y):
    ''' Критерий для абсолютной функции потерь. '''

    return np.mean(np.absolute(y - np.median(y)))


def get_probs_by_y(y):
    ''' Возвращает вектор частот для каждого класса выборки. '''

    _, counts = np.unique(y, return_counts=True)
    return counts / np.sum(counts)


def gini_criterion(y):
    ''' Критерий Джини. '''

    probs = get_probs_by_y(y)
    return np.sum(probs * (1 - probs))


def entropy_criterion(y):
    ''' Энтропийный критерий. '''

    probs = get_probs_by_y(y)
    return -np.sum(probs * np.log(probs))
```

started 17:38:45 2020-03-14, finished in 9ms

Протестируйте реализованные функции.

Тесты для распределения вероятностей на классах.

In [3]:

```python
assert np.allclose(get_probs_by_y([1, 1, 2, 2, 7]),
                   np.array([0.4, 0.4, 0.2]))
assert np.allclose(get_probs_by_y([1]), np.array([1]))
```

started 17:38:45 2020-03-14, finished in 12ms

Тесты для критериев разбиения.

In [4]:

```
assert np.allclose(entropy_criterion([25]), 0)
assert np.allclose(gini_criterion([25]), 0)
assert np.allclose(mean_square_criterion([10, 10, 10]), 0)
assert np.allclose(mean_abs_criterion([10, 10, 10]), 0)
```

started 17:38:45 2020-03-14, finished in 6ms

Реализация класса решающего дерева.

Для того, чтобы лучше понять, как устроены решающие деревья и как именно устроен процесс их построения, вам предлагается реализавать класс `BaseDecisionTree`, реализующий базовые функции решающего дерева. Большая часть кода уже написана.

Используются следующие классы:

Класс 1. `BaseDecisionTree` - класс для решающего дерева, в котором реализовано построение дерева. Все вершины дерева хранятся в списке `self.nodes`, при этом вершина с номером 0 - корень.

1) `__init__` - инициализация дерева. Здесь сохраняются гиперпараметры дерева: `criterion`, `max_depth`, `min_samples_split` и инициализируется список вершин, состоящий только из одной вершины - корневой,

2) `build_` - рекурсивная функция построения дерева. В ней при посещении каждой вершины дерева проверяются условия, стоит ли продолжать разбивать эту вершину. Если да, то перебираются все возможные признаки и пороговые значения и выбирается та пара (признак, значение), которой соответствует наименьшее значение критерия информативности,

3) `fit` - функция обучения дерева, принимающая на вход обучающую выборку. В этой функции происходит предподсчёт всех возможных пороговых значений для каждого из признаков, а затем вызывается функция `build_`.

Класс 2. `Node` - класс вершины дерева. Внутри вершины, помимо раздяляющего признака и порога хранятся `self.left_son`, `self.right_son` - номера дочерних вершин, а также `self.left_prob` и `self.right_prob` - вероятности попадания элемента в каждую из них. При этом в листовых вершиных хранятся также `self.y_values` - значения соответствующих элементов выборки, попавших в вершину.

1) `__init__` - инициализация вершины. Принимает в качестве аргументов разделяющий признак и пороговое значение и сохраняет их.

Класс 3. `DecisionTreeRegressor` - наследник класса `BaseDecisionTree`, в котором реализованы функции для предсказаний при решении задачи регрессии.

1) `predict_instance` - получение предсказания для одного элемента выборки. Выполняется посредством спуска по решающему дереву до листовой вершины,

2) `predict` - получение предсказаний для всех элементов выборки.

Класс 4. `DecisionTreeClassifier` - наследник класса `BaseDecisionTree`, в котором реализованы функции для предсказаний при решении задачи классификации.

1) `predict_proba_instance` - предсказание распределение вероятностей по классам для одного элемента выборки,

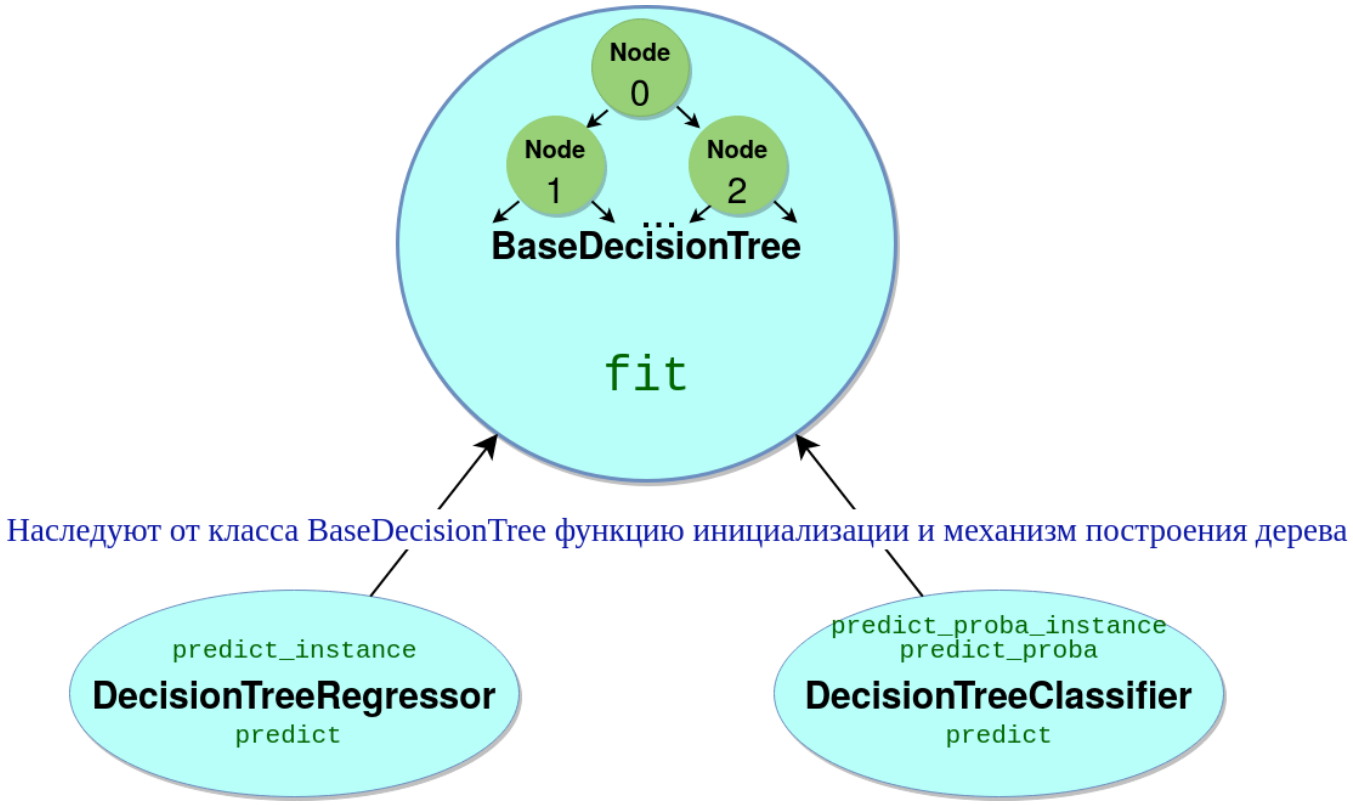
2) `predict_proba` - предсказание распределение вероятностей по классам для всех элементов выборки,

3) `predict_proba` - предсказание меток классов для всех элементов выборки.

Перед написанием кода разбиения дерева, ответьте на вопрос, какие пороговые значения для каждого из признаков вы будете перебирать. Почему рассматривать другие значения в качестве пороговых не имеет смысла?

Ответ: в качестве пороговых значений каждого из признаков нужно перебирать не все возможные действительные числа (тем более, это было бы невозможно, так как тогда пришлось бы перебирать бесконечное число вариантов), а только те значения признаков, которые содержатся в обучающей выборке. Другие значения не имеет смысла рассматривать в качестве пороговых, так как каждое из них будет находится между двумя соседними (в порядке возрастания) значениями признака и это никак не поменяет разбиение выборки на 2 части.

Структура решающего дерева



In [5]:

```
def get_not_nans(arr):
    '''
    Функция, которая создаёт и возвращает новый массив
    из всех элементов переданного массива, не являющихся None.
    '''

    return arr.copy()[arr != np.nan]


class Node(object):
    def __init__(self, split_feature=None, split_threshold=None):
        '''
        Функция инициализации вершины решающего дерева.

        Параметры.
        1) split_feature - номер разделяющего признака
        2) split_threshold - пороговое значение
        '''

        self.split_feature = split_feature
        self.split_threshold = split_threshold
        # По умолчанию считаем, что у вершины нет дочерних вершин.
        self.left_son, self.right_son = None, None
        # Вероятности попадания в каждую из дочерних вершин нужно поддерживат
        # для корректной обработки данных с пропусками
        self.left_prob, self.right_prob = 0, 0
        # Массив значений y. Определён только для листовых вершин дерева
        self.y_values = None


class BaseDecisionTree(BaseEstimator):
    '''
    Здесь содержится реализация всех основных функций для работы
    с решающим деревом.

    Наследование от класса BaseEstimator нужно для того, чтобы
    в дальнейшем данный   класс можно было использовать в
    различных функциях библиотеки sklearn, например, в функциях
    для кросс-валидации.
    '''

    def __init__(self,
                 criterion,
                 max_depth=np.inf,
                 min_samples_split=2):
        '''
        Функция инициализации решающего дерева.

        Параметры.
        1) criterion - критерий информативности,
        2) max_depth - максимальная глубина дерева,
        3) min_samples_split - минимальное количество элементов
        обучающей выборки,  которое должно попасть в вершину,
        чтобы потом происходило разбиение этой вершины.
        '''

        self.criterion = criterion
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
```

```
60          # Список всех вершин дерева. В самом начале
61          # работы алгоритма есть только одна
62          # вершина - корень.
63          self.nodes = [Node()]
64          # Количество классов. Актуально только
65          # при решении задачи классификации.
66          self.class_count = 1
67          # Сюда нужно будет записать все значения
68          # для каждого из признаков датасета.
69          self.feature_values = None
70
71      def build_(self, v, X, y, depth=0):
72          '''
73          Рекурсивная функция построения решающего дерева.
74
75          Параметры.
76          1) v - номер рассматриваемой вершины
77          2) X, y - обучающая выборка, попавшая в текущую вершину
78          3) depth - глубина вершины с номером v
79          '''
80
81          if depth == self.max_depth or len(y) < self.min_samples_split:
82              # Если строим дерево для классификации, то
83              # сохраняем метки классов всех элементов выборки,
84              # попавших в вершину.
85              if callable(getattr(self, "set_class_count", None)):
86                  self.nodes[v].y_values = y.copy()
87              # Для регрессии сразу вычислим среднее всех
88              # элементов вершины.
89              else:
90                  self.nodes[v].y_values = np.mean(y)
91              return
92
93          best_criterion_value = np.inf
94          best_feature, best_threshold = 0, 0
95          sample_size, feature_count = X.shape
96
97          # переберём все возможные признаки и значения порогов,
98          # найдём оптимальный признак и значение порога
99          # и запишем их в best_feature, best_threshold
100         for feature_id in range(feature_count):
101             for threshold in self.feature_values[feature_id]:
102                 # делим вершину по рассматриваемому признаку
103                 # и пороговому значению
104                 y_l = y[np.nan_to_num(X[:, feature_id], threshold) < threshol
105                 y_r = y[np.nan_to_num(X[:, feature_id], threshold - 1) >= thr
106                 if len(y_l) == 0 or len(y_r) == 0:
107                     continue
108
109                 left_fraction = len(y_l) / float(sample_size)
110                 criterion_left = left_fraction * self.criterion(y_l)
111                 criterion_right = (1 - left_fraction) * self.criterion(y_r)
112                 # если для рассматриваемого признака и
113                 # порога значение критерия лучше, чем для
114                 # всех предыдущих пар (признак, порог),
115                 # то обновляем оптимальный признак и
116                 # порог разбиения
117                 if criterion_left + criterion_right < best_criterion_value:
118                     best_criterion_value = criterion_left + criterion_right
119                     best_feature = feature_id
120                     best_threshold = threshold
```

```
121
122         # сохраним найденные параметры в класс текущей вершины
123         self.nodes[v].split_feature = best_feature
124         self.nodes[v].split_threshold = best_threshold
125         # разделим выборку на 2 части по порогу
126         left_indices = np.nan_to_num(X[:, best_feature],
127                                      best_threshold) < best_threshold
128         right_indices = np.nan_to_num(X[:, best_feature],
129                                       best_threshold - 1) >= best_threshold
130         left_indices_or_nans = np.logical_or(left_indices,
131                                              X[:, best_feature] == None)
132         right_indices_or_nans = np.logical_or(right_indices,
133                                               X[:, best_feature] == None)
134
135         X_l, y_l = X[left_indices_or_nans, :], y[left_indices_or_nans]
136         X_r, y_r = X[right_indices_or_nans, :], y[right_indices_or_nans]
137         self.nodes[v].left_prob = np.sum(left_indices) / len(y)
138         self.nodes[v].right_prob = np.sum(right_indices) / len(y)
139
140         # создаём левую и правую дочерние вершины,
141         # и кладём их в массив self.nodes
142         self.nodes.append(Node())
143         self.nodes.append(Node())
144         # сохраняем индексы созданных вершин в
145         # качестве левого и правого сына вершины v
146         self.nodes[v].left_son, self.nodes[v].right_son \
147             = len(self.nodes)-2, len(self.nodes)-1
148         # рекурсивно вызываем алгоритм построения
149         # дерева для дочерних вершин
150         self.build_(self.nodes[v].left_son, X_l, y_l, depth+1)
151         self.build_(self.nodes[v].right_son, X_r, y_r, depth+1)
152
153     def fit(self, X, y):
154         '''
155         Функция, из которой запускается построение
156         решающего дерева по обучающей выборке.
157
158         Параметры.
159         X, y - обучающая выборка
160         '''
161
162         # сохраним заранее все пороги для каждого
163         # из признаков обучающей выборки
164         X, y = np.array(X), np.array(y)
165         self.feature_values = []
166         for feature_id in range(X.shape[1]):
167             self.feature_values.append(
168                 np.unique(get_not_nans(X[:, feature_id]))
169             )
170
171         set_class_count = getattr(self, "set_class_count", None)
172         # если строится дерево для классификации,
173         # то нужно посчитать количество классов
174         if callable(set_class_count):
175             set_class_count(y)
176         self.build_(0, X, y)
```

started 17:38:45 2020-03-14, finished in 26ms

Теперь, когда общий код решающего дерева написан, нужно сделать обёртки над `BaseDecisionTree` - `DecisionTreeRegressor` и `DecisionTreeClassifier` для использования решающего дерева в

задачах регрессии и классификации соответственно.

Допишите функции `predict_instance` и `predict_proba_instance` в классах для регрессии и классификации соответственно. В этих функциях нужно для одного элемента x выборки промоделировать спуск в решающем дереве, а затем по листовой вершине, в которой окажется объект, посчитать для классификации - распределение вероятностей, а для регрессии - число y.

In [6]:

```
class DecisionTreeRegressor(BaseDecisionTree):
    def predict_instance(self, x, v):
        '''
        Рекурсивная функция, предсказывающая значение
        y для одного элемента x из выборки.

        Параметры.
        1) x - элемент выборки, для которого
        требуется предсказать значение y
        2) v - рассматриваемая вершина дерева
        '''

        if self.nodes[v].left_son == None:
            return np.mean(self.nodes[v].y_values)

        # если у объекта x значение признака по
        # которому происходит разделение,  меньше
        # порогового, то спускаемся в левое поддерево,
        # иначе - в правое
        if x[self.nodes[v].split_feature] < self.nodes[v].split_threshold:
            return self.predict_instance(x, self.nodes[v].left_son)
        elif x[self.nodes[v].split_feature] >= self.nodes[v].split_threshold:
            return self.predict_instance(x, self.nodes[v].right_son)
        # а если у элемента отсутствует значение
        # разделяющего признака,  то будем спускаться
        # в оба поддерева
        else:
            left_predict = self.predict_instance(x, self.nodes[v].left_son)
            right_predict = self.predict_instance(x, self.nodes[v].right_son)

            return self.nodes[v].left_prob * left_predict +\
                   self.nodes[v].right_prob * right_predict

    def predict(self, X):
        '''
        Функция, предсказывающая значение
        y для всех элементов выборки X.

        Параметры.
        X - выборка, для которой требуется
        получить вектор предсказаний y
        '''

        return [self.predict_instance(x, 0) for x in X]
```

started 17:38:45 2020-03-14, finished in 9ms

Для удобства реализации функции `predict_proba_instance` класса `DecisionTreeClassifier` будем считать, что все классы имеют целочисленные метки от 0 до $k-1$, где k - количество классов. Если бы это условие не было выполнено, то нужно было бы сначала сделать предобработку меток классов в датасете.

In [7]:

```
class DecisionTreeClassifier(BaseDecisionTree):
    def set_class_count(self, y):
        '''
        Функция, вычисляющая количество классов
        в обучающей выборке.

        Параметры.
        y - значения класса в обучающей выборке
        '''

        self.class_count = np.max(y) + 1

    def predict_proba_instance(self, x, v):
        '''
        Рекурсивная функция, предсказывающая вектор
        вероятностей принадлежности  объекта x
        к классам

        Параметры.
        1) x - элемент выборки, для которого
        требуется предсказать значение y
        2) v - вершина дерева, в которой
        находится алгоритм
        '''

        if self.nodes[v].left_son == None:
            result = np.zeros(self.class_count)
            classes, counts = np.unique(self.nodes[v].y_values,
                                        return_counts=True)
            result[classes.astype(int)] = counts
            return result / np.sum(result)

        # если у x значение признака по которому происходит разделение,
        # меньше порогового, то спускаемся в левое поддерево, иначе - в право
        if x[self.nodes[v].split_feature] < self.nodes[v].split_threshold:
            return self.predict_proba_instance(x, self.nodes[v].left_son)
        elif x[self.nodes[v].split_feature] >= self.nodes[v].split_threshold:
            return self.predict_proba_instance(x, self.nodes[v].right_son)
        # а если у элемента отсутствует значение разделяющего признака,
        # то будем спускаться в оба дерева
        else:
            left_predict \
                = self.predict_proba_instance(x, self.nodes[v].left_son)
            right_predict = \
                self.predict_proba_instance(x, self.nodes[v].right_son)

            return self.nodes[v].left_prob * left_predict +\
                   self.nodes[v].right_prob * right_predict

    def predict_proba(self, X):
        '''
        Функция, предсказывающая вектор вероятностей
        принадлежности объекта x к классам для
        каждого x из X

        Параметры.
        X - выборка, для которой требуется получить вектор предсказаний y
        '''

```

```
        return [self.predict_proba_instance(x, 0) for x in X]

    def predict(self, X):
        '''
        Функция, предсказывающая метку класса для
        всех элементов выборки X.

        Параметры.
        X - выборка, для которой требуется получить
        вектор предсказаний y
        '''

        return np.argmax(self.predict_proba(X), axis=1)
```

started 17:38:45 2020-03-14, finished in 14ms

Подбор параметров.

В этой части задания вам предлагается поработать с написанным решающим деревом, применив его к задачи классификации и регрессии, и в обеих задачах подобрать оптимальные параметры для построения.

Не забывайте делать выводы.

1. Задача классификации.

Теперь - самое время протестировать работу написанного нами решающего дерева. Делать мы это будем на датасете для классификации вина из `sklearn` .

In [8]:

```
from sklearn.datasets import load_wine

X, y = load_wine(return_X_y=True)
```

started 17:38:45 2020-03-14, finished in 25ms

Для критерия Джини и энтропийного критерия найдите оптимальные параметры обучения дерева - `max_depth` и `min_samples_split` . Используйте для этого кросс-валидацию.

In [9]:

```
classification_criteria = [gini_criterion, entropy_criterion]
criterion_names = ['gini', 'entropy']
```

started 17:38:45 2020-03-14, finished in 3ms

Используя кросс-валидацию, найдите оптимальные параметры на обучающей выборке и выведите их вместе с метрикой качества.

С начала надо разбить выборку на train и test.

In [10]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=777)
```

started 17:38:45 2020-03-14, finished in 17ms

Теперь проведите кросс-валидацию для каждого из критериев разбиения вершин.

In [11]:

```
for criterion, criterion_name in zip(classification_criteria,
                                     criterion_names):
    dt = GridSearchCV(
        estimator=DecisionTreeClassifier(criterion),
        param_grid={
            'max_depth': np.arange(2, 11),
            'min_samples_split': np.arange(3, 7)
        },
        scoring='accuracy',
        cv=5,  # разбиение выборки на 5 фолдов
        verbose=3,  # насколько часто печатать сообщения
        n_jobs=2,  # кол-во параллельных процессов
    )

    dt.fit(X_train, y_train)
    predictions = dt.predict(X_test)

    print('Оптимальные параметры для {}:'.format(criterion_name),
          dt.best_params_)
    accuracy = accuracy_score(y_test, predictions)
    print('Точность на тесте:', accuracy)
    assert accuracy >= 0.85, "Something is wrong with your classifier"
```

started 17:38:45 2020-03-14, finished in 7m 5s

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits

[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent work
ers.
[Parallel(n_jobs=2)]: Done  28 tasks      | elapsed:    7.6s
[Parallel(n_jobs=2)]: Done 124 tasks      | elapsed:  1.4min
[Parallel(n_jobs=2)]: Done 180 out of 180 | elapsed:  3.0min finished

Оптимальные параметры для gini: {'max_depth': 3, 'min_samples_split':
3}
Точность на тесте: 0.9333333333333333
Fitting 5 folds for each of 36 candidates, totalling 180 fits

[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent work
ers.
[Parallel(n_jobs=2)]: Done  28 tasks      | elapsed:   12.4s
[Parallel(n_jobs=2)]: Done 124 tasks      | elapsed:  2.3min
[Parallel(n_jobs=2)]: Done 180 out of 180 | elapsed:  4.0min finished

Оптимальные параметры для entropy: {'max_depth': 4, 'min_samples_spli
t': 3}
Точность на тесте: 0.9333333333333333
```

Построение графиков.

Постройте графики зависимости accuracy на обучающей и тестовой выборке от максимальной глубины дерева для каждого критерия на train и на test. В качестве максимальной глубины используйте значения от 3 до 7.

Поскольку при кросс-валидации наилучшее значение `min_samples_split` для обоих критериев равно 3, будем строить графики зависимости `accuracy` от `max_depth` при фиксации `min_samples_split` =3.

In [12]:

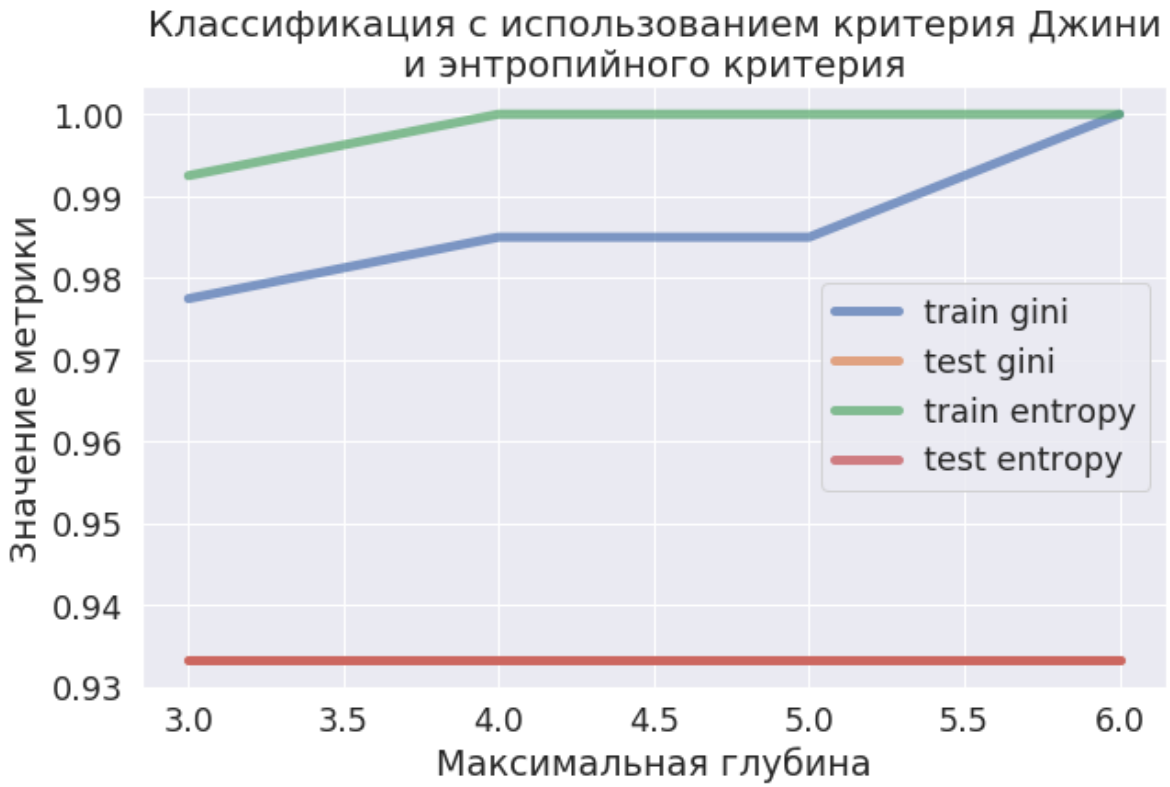
```
def plot_graphs(tree_estimator, criterions, scorer, title,
                max_depth_list=np.arange(3, 7), min_samples_split=3):
    '''
    Функция для рисования графиков.

    Параметры.
    1) tree_estimator - экземпляр класса решающего дерева,
    2) criterions - критерии разбиения дерева,
    3) scorer - метрика качества,
    4) title - заголовок графика,
    5) max_depth_list - список рассматриваемых значений макс.
    глубины дерева,
    6) min_samples_split - значение параметра min_samples_split.
    '''

    plt.figure(figsize=(10, 6))

    for criterion in criterions:
        score_train = []
        score_test = []

        for max_depth in max_depth_list:
            estimator = tree_estimator(criterion[0], max_depth=max_depth,
                                       min_samples_split=min_samples_split)
            estimator.fit(X_train, y_train)
            score_train.append(scorer(estimator.predict(X_train), y_train))
            score_test.append(scorer(estimator.predict(X_test), y_test))

        plt.plot(max_depth_list, score_train,
                 lw=5, alpha=0.7, label='train {}'.format(criterion[1]))
        plt.plot(max_depth_list, score_test,
                 lw=5, alpha=0.7, label='test {}'.format(criterion[1]))

    plt.xlabel('Максимальная глубина')
    plt.ylabel('Значение метрики')
    plt.legend()
    plt.title(title, fontsize=20)
    plt.show()
```

started 17:45:50 2020-03-14, finished in 7ms

In [13]:

```
plot_graphs(DecisionTreeClassifier,
            [(gini_criterion, 'gini'), (entropy_criterion, 'entropy')],
            accuracy_score,
            'Классификация с использованием критерия Джини\n' \
            + 'и энтропийного критерия')
```

started 17:45:50 2020-03-14, finished in 16.6s



Сделайте выводы. Почему графики получились такими? Как соотносятся оптимальные значения параметров на обучающей и на тестовой выборках?

Вывод.

На обоих графиках видно, что с некоторого значения максимальной глубины точность классификации на тестовой выборке перестаёт расти. Это указывает на то, что, возможно, при больших значениях макс. глубины, чем это, происходит переобучение дерева.

Решающее дерево, использующее энтропийный критерий при разбиении, дало более высокий результат как на тестовой, так и на обучающей выборке.

2. Задача регрессии.

Проделайте аналогичные шаги для задачи регрессии. В качестве датасете возьмите `boston` из `sklearn` , а в качестве критерия качества возьмите r2_score. Рассмотрим более широкий диапозон значений для `max_depth` : от 3 до 14.

In [14]:

```
from sklearn.datasets import load_boston

boston_X, boston_y = load_boston(return_X_y=True)
```

started 17:46:07 2020-03-14, finished in 14ms

In [15]:

```
regression_criteria = [mean_square_criterion, mean_abs_criterion]
criterion_names = ['mean_square', 'mean_abs']
```

started 17:46:07 2020-03-14, finished in 7ms

Разобьём выборку на обучение и тест.

In [16]:

```
X_train, X_test, y_train, y_test = train_test_split(boston_X, boston_y,
                                                    random_state=777)
```

started 17:46:07 2020-03-14, finished in 7ms

Проведите эксперименты, аналогичны тем, что были сделаны для задачи классификации.

In [17]:

```
for criterion, criterion_name in zip(regression_criteria, criterion_names):
    dt_regressor = GridSearchCV(
        estimator=DecisionTreeRegressor(criterion),
        param_grid={
            'max_depth': np.arange(2, 11)
        },
        scoring='r2',
        cv=5,  # разбиение выборки на 5 фолдов
        verbose=3,  # насколько часто печатать сообщения
        n_jobs=2,  # кол-во параллельных процессов
    )

    dt_regressor.fit(X_train, y_train)
    predictions = dt_regressor.predict(X_test)

    print('Оптимальные параметры для {}:'.format(criterion_name),
          dt.best_params_)
    print('r2_score на тесте:', r2_score(y_test, predictions))
```

started 17:46:07 2020-03-14, finished in 9m 14s

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits

[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent work
ers.
[Parallel(n_jobs=2)]: Done  28 tasks      | elapsed:  1.2min
[Parallel(n_jobs=2)]: Done  45 out of  45 | elapsed:  4.0min finished

Оптимальные параметры для mean_square: {'max_depth': 4, 'min_samples_s
plit': 3}
r2_score на тесте: 0.6752210376459766
Fitting 5 folds for each of 9 candidates, totalling 45 fits

[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent work
ers.
[Parallel(n_jobs=2)]: Done  28 tasks      | elapsed:  1.6min
[Parallel(n_jobs=2)]: Done  45 out of  45 | elapsed:  5.1min finished

Оптимальные параметры для mean_abs: {'max_depth': 4, 'min_samples_spli
t': 3}
r2_score на тесте: 0.6397486631555982
```

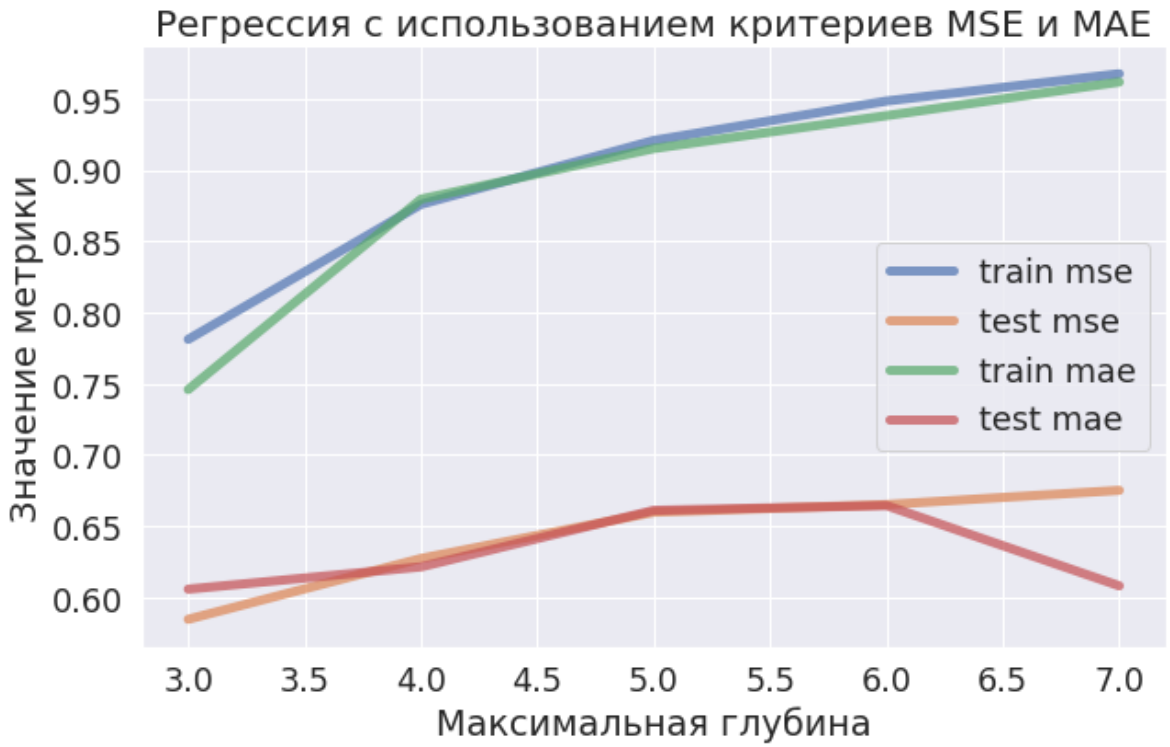
Вывод.

В данном случае оптимальные параметры для обоих критериев разбиения совпадают. Однако, видно, что критерий, основанный на mse, даёт гораздо более высокий результат.

In [18]:

```
plot_graphs(DecisionTreeRegressor,
            [(mean_square_criterion, 'mse'), (mean_abs_criterion, 'mae')],
            r2_score,
            'Регрессия с использованием критериев MSE и MAE',
            max_depth_list=range(3, 8))
```

started 17:55:21 2020-03-14, finished in 1m 4.18s



Сделайте вывод, в котором объясните, почему графики получились такими.

Скорее всего, вы заметили, что дерево в этих экспериментах строится довольно медленно. Как можно ускорить его построение? Можно ли ускорить нахождение оптимального разбиения по некоторому вещественному признаку?

Вывод.

Качество результатов, выдаваемых решающим деревом, построенным с использованием mae критерия, оказалось значительно выше, чем даёт дерево, использующее mse критерий. Но, тем не менее, на втором графике видно, что при `max_depth > 5` решающее дерево начинает переобучаться, так как `score` на обучающей выборке растёт, а на тестовой - падает.

Можно ускорить построение решающего дерева, снизив асимптотику разбиения по признаку с $O(u^2)$ до $O(u)$, где u - количество различных значений признака.

Обработка пропусков с использованием решающих деревьев.

А теперь рассмотрим датасет, в котором часть данных пропущена. В качестве примера возьмём датасет https://archive.ics.uci.edu/ml/datasets/Adult (https://archive.ics.uci.edu/ml/datasets/Adult) для определения категории дохода работников, по таким признакам, как возраст, образование, специальность, класс работы, пол, кол-во отрабатываемых часов в неделю и некоторым другим.

In [19]:

```
column_names = [
    'age', 'workclass', 'fnlwgt', 'education1',
    'education2', 'marital-status', 'occupation',
    'relationship', 'race', 'sex', 'capital-gain',
    'capital-loss', 'hours-per-week', 'native-country', 'target'
]
```

started 17:56:25 2020-03-14, finished in 3ms

Поскольку предсказание в дереве на данных с пропусками часто занимает сильно больше времени, чем в случае отсутствия пропусков (так как часто приходится спускаться разу в 2 поддерева), то для экономии времени сократим датасет, взяв из него только первые 2000 строк данных.

In [20]:

```
adult_df = pd.read_csv('adult.data', header=None)[:2000]
adult_df.columns = column_names
target = adult_df['target'] == ' >50K'
adult_df = adult_df.drop(['target'], axis=1)
adult_df.head()
```

started 17:56:25 2020-03-14, finished in 175ms

Out[20]:

	age	workclass	fnlwgt	education1	education2	marital-status	occupation	relationship	race
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black

In [21]:

```
adult_df.shape
```

started 17:56:25 2020-03-14, finished in 3ms

Out[21]:

```
(2000, 14)
```

Предобработаем датасет, заменив категориальные признаки one-hot векторами.

In [22]:

```
adult_df = pd.get_dummies(adult_df)
adult_df.head()
```

started 17:56:25 2020-03-14, finished in 38ms

Out[22]:

	age	fnlwgt	education2	capital-gain	capital-loss	hours-per-week	workclass_?	workclass_Federal-gov	workclass_Local-gov	wo
0	39	77516	13	2174	0	40	0	0	0	
1	50	83311	13	0	0	13	0	0	0	
2	38	215646	9	0	0	40	0	0	0	
3	53	234721	7	0	0	40	0	0	0	
4	28	338409	13	0	0	40	0	0	0	

5 rows × 102 columns

Поскольку все пропущенные значения относились к категориальным признакам и помечались в датасете знаком `?`, то для каждого категориального признака `feature` исходного датасета надо выполнить следующую процедуру: рассмотреть признак `feature_?` нового датасета и для всех строк, для которых выполнено `feature_?=1`, значениях всех признаков с префиксом `feature` установить в `None`.

In [23]:

```
for feature in column_names:
    if f'{feature}_ ?' in adult_df.columns:
        none_indices = np.arange(adult_df.shape[0])[
            adult_df[f'{feature}_ ?'] == 1
        ]

        for dummy_feature in adult_df.columns:
            if dummy_feature.startswith(f'{feature}_ ') \
                and dummy_feature != f'{feature}_ ?':
                adult_df[dummy_feature][none_indices] = np.nan

        adult_df = adult_df.drop(f'{feature}_ ?', axis=1)
```

started 17:56:25 2020-03-14, finished in 109ms

Посмотрим на распределение пропущенных значений по признакам.

In [24]:

```
1  np.sum(adult_df.isnull(), axis=0)
```

started 17:56:26 2020-03-14, finished in 8ms

Out[24]:

```
age                                  0
fnlwgt                               0
education2                           0
capital-gain                         0
capital-loss                         0
                                   ..
native-country_ Taiwan              39
native-country_ Thailand            39
native-country_ Trinadad&Tobago     39
native-country_ United-States       39
native-country_ Yugoslavia          39
Length: 99, dtype: int64
```

Разобьём данные на обучающую и тестовую выборки в отношении 3:1.

In [25]:

```
1 ▾ X_train, X_test, y_train, y_test = train_test_split(
2       adult_df.values, target.values, random_state=777
3   )
```

started 17:56:26 2020-03-14, finished in 10ms

При помощи кросс-валидации найдём оптимальные гиперпараметры для каждого из критериев разбиения для классификации.

In [26]:

```
for criterion, criterion_name in zip(classification_criteria,
                                     criterion_names):
    dt_classifier = GridSearchCV(
        estimator=DecisionTreeClassifier(criterion),
        param_grid={
            'max_depth': np.arange(2, 10),
            'min_samples_split': np.arange(3, 7)
        },
        scoring='accuracy',
        cv=5,  # разбиение выборки на 5 фолдов
        verbose=3,  # насколько часто печатать сообщения
        n_jobs=2,  # кол-во параллельных процессов
    )

    dt_classifier.fit(X_train, y_train)
    predictions = dt_classifier.predict(X_test)

    print('Оптимальные параметры для {}:'.format(criterion_name),
          dt.best_params_)
    accuracy = accuracy_score(y_test, predictions)
    print('Точность на тесте:', accuracy)
```

started 17:56:26 2020-03-14, finished in 41m 58s

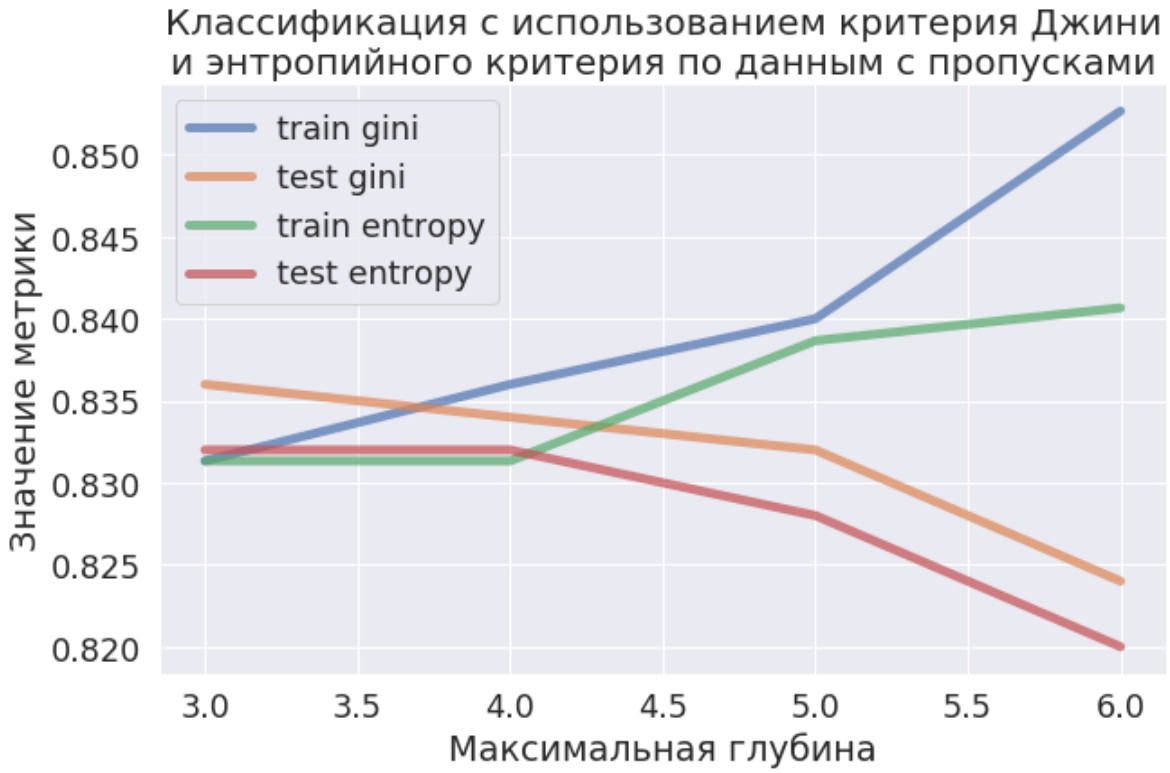
```
Fitting 5 folds for each of 32 candidates, totalling 160 fits

[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent work
ers.
[Parallel(n_jobs=2)]: Done  28 tasks      | elapsed:   28.1s
[Parallel(n_jobs=2)]: Done 124 tasks      | elapsed: 10.3min
[Parallel(n_jobs=2)]: Done 160 out of 160 | elapsed: 19.1min finished

Оптимальные параметры для mean_square: {'max_depth': 4, 'min_samples_s
plit': 3}
Точность на тесте: 0.832
Fitting 5 folds for each of 32 candidates, totalling 160 fits

[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent work
ers.
[Parallel(n_jobs=2)]: Done  28 tasks      | elapsed:   25.8s
[Parallel(n_jobs=2)]: Done 124 tasks      | elapsed:  9.9min
[Parallel(n_jobs=2)]: Done 160 out of 160 | elapsed: 22.6min finished

Оптимальные параметры для mean_abs: {'max_depth': 4, 'min_samples_spli
t': 3}
Точность на тесте: 0.832
```

Проведите эксперименты с построением графиков, аналогичные тем, что были сделаны в предыдущем пункте для задач классификации и регрессии.

In [27]:

```
plot_graphs(
    DecisionTreeClassifier,
    [(gini_criterion, 'gini'), (entropy_criterion, 'entropy')],
    accuracy_score,
    'Классификация с использованием критерия Джини\n'
    'и энтропийного критерия по данным с пропусками'
)
```

started 18:38:23 2020-03-14, finished in 1m 17.5s



Вывод.

По графикам видно, что дерево начинается переобучаться для макс. глубины >= 5. Также можно заметить, что использование критерия Джини даёт лучшие результаты как на обучающей, так и на тестовой выборке.