Машинное обучение, DS-поток

Домашнее задание 3A

Правила:

- Дедлайн **09 марта 23:59**. После дедлайна работы не принимаются кроме случаев наличия уважительной причины.
- Выполненную работу нужно отправить на почту `mipt.stats@yandex.ru` , указав тему письма `"[ml] Фамилия Имя - задание 3A"` . Квадратные скобки обязательны. Если письмо дошло, придет ответ от автоответчика.
- Прислать нужно ноутбук и его pdf-версию (без архивов). Названия файлов должны быть такими: `3A.N.ipynb` и `3A.N.pdf` , где `N` - ваш номер из таблицы с оценками.
- Решения, размещенные на каких-либо интернет-ресурсах не принимаются. Кроме того, публикация решения в открытом доступе может быть приравнена к предоставлении возможности списать.
- Для выполнения задания используйте этот ноутбук в качестве основы, ничего не удаляя из него.
- Никакой код из данного задания при проверке запускаться не будет.

Задание стоит 15 баллов.

In [1]:

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

warnings.filterwarnings('ignore')
sns.set(font_scale=1.6)
plt.rcParams['axes.facecolor'] = 'lightgrey'
```

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: <...>

Реализация критериев информативности.

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

Ответ: <...>

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

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

In [ ]:

```
# Код функций, реализующих критерии разбиения.

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

    return <...>


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

    return <...>


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

    return <...>


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

    return <...>


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

    return <...>
```

Протестируйте реализованные функции.

Тесты для распределения вероятностей на классах.

In [ ]:

```
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]))
```

Тесты для критериев разбиения.

In [ ]:

```
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)
```

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

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

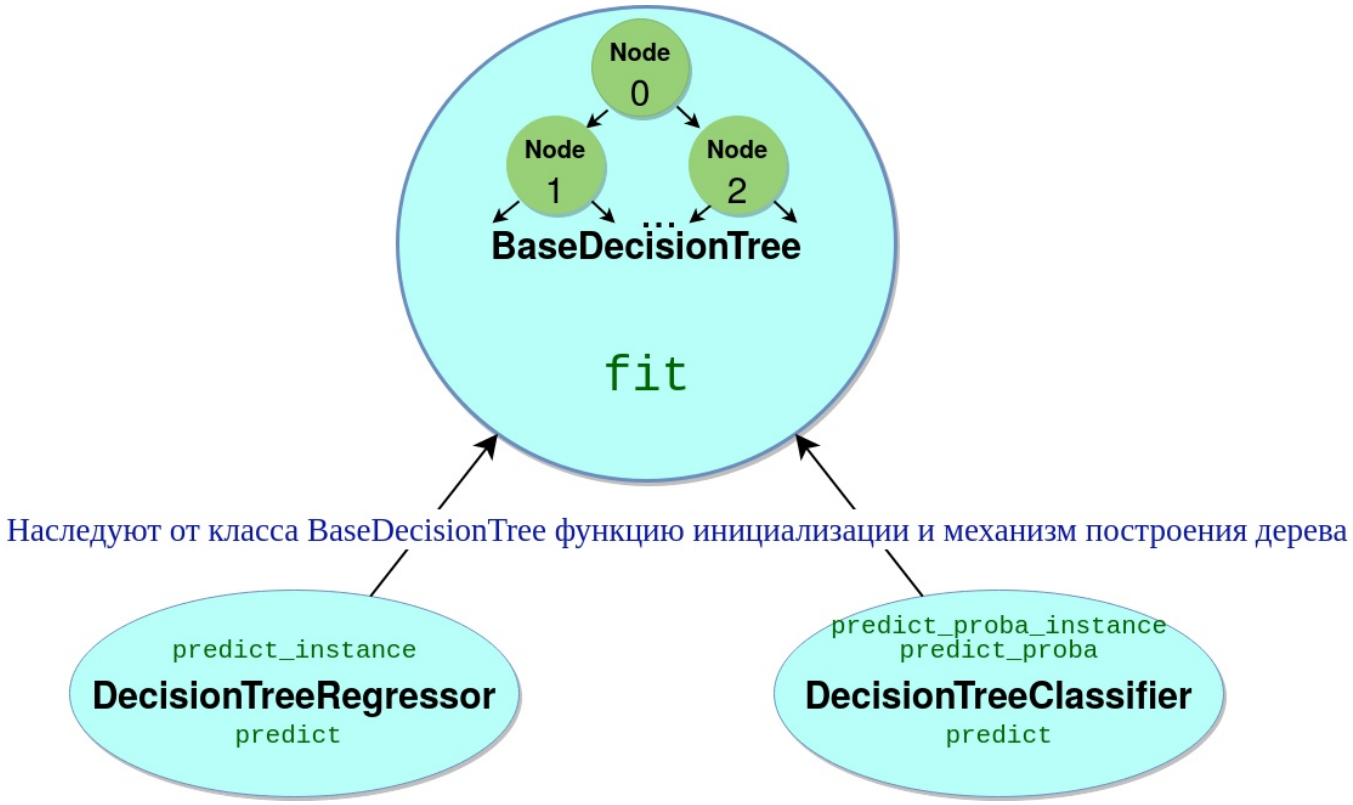
2) `predict_proba` - предсказание распределения вероятностей по классам для всех элементов выборки,

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: <...>

Структура решающего дерева



In [ ]:

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

    return arr.copy()[arr != None]


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 поддерживать
        # 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
        # Список всех вершин дерева. В самом начале
        # работы алгоритма есть только одна
        # вершина - корень.
```

```
60          self.nodes = [Node()]
61          # Количество классов. Актуально только
62          # при решении задачи классификации.
63          self.class_count = 1
64          # Сюда нужно будет записать все значения
65          # для каждого из признаков датасета.
66          self.feature_values = None
67
68      def build_(self, v, X, y, depth=0):
69          '''
70          Рекурсивная функция построения решающего дерева.
71
72          Параметры.
73          1) v - номер рассматриваемой вершины
74          2) X, y - обучающая выборка, попавшая в текущую вершину
75          3) depth - глубина вершины с номером v
76          '''
77
78          if depth == self.max_depth or len(y) < self.min_samples_split:
79              # Если строим дерево для классификации, то
80              # сохраняем метки классов всех элементов выборки,
81              # попавших в вершину.
82              if callable(getattr(self, "set_class_count", None)):
83                  self.nodes[v].y_values = y.copy()
84              # Для регрессии сразу вычислим среднее всех
85              # элементов вершины.
86              else:
87                  self.nodes[v].y_values = np.mean(y)
88              return
89
90          best_criterion_value = np.inf
91          best_feature, best_threshold = 0, 0
92          sample_size, feature_count = X.shape
93
94          # переберём все возможные признаки и значения порогов,
95          # найдём оптимальный признак и значение порога
96          # и запишем их в best_feature, best_threshold
97          for feature_id in range(feature_count):
98              for threshold in self.feature_values[feature_id]:
99                  <...>
100
101         # сохраним найденные параметры в класс текущей вершины
102         self.nodes[v].split_feature = <...>
103         self.nodes[v].split_threshold = <...>
104         # разделим выборку на 2 части по порогу
105         <..>
106
107         # создаём левую и правую дочерние вершины,
108         # и кладём их в массив self.nodes
109         self.nodes.append(Node())
110         self.nodes.append(Node())
111         # сохраняем индексы созданных вершин в качестве
112         # левого и правого сына вершины v
113         self.nodes[v].left_son, self.nodes[v].right_son =\
114             len(self.nodes)-2, len(self.nodes)-1
115         # рекурсивно строим дерево для дочерних вершин
116         self.build_(self.nodes[v].left_son, X_l, y_l, depth+1)
117         self.build_(self.nodes[v].right_son, X_r, y_r, depth+1)
118
119     def fit(self, X, y):
120         '''
```

```
        Функция, из которой запускается построение
        решающего дерева по обучающей выборке.

        Параметры.
        X, y - обучающая выборка
        '''

        # сохраним заранее все пороги для каждого из
        # признаков обучающей выборки
        X, y = np.array(X), np.array(y)
        self.feature_values = []
        for feature_id in range(X.shape[1]):
            self.feature_values.append(
                np.unique(get_not_nans(X[:, feature_id]))
            )

        set_class_count = getattr(self, "set_class_count", None)
        # если строится дерево для классификации,
        # то нужно посчитать количество классов
        if callable(set_class_count):
            set_class_count(y)
        self.build_(0, X, y)
```

Теперь, когда общий код решающего дерева написан, нужно сделать обёртки над `BaseDecisionTree` - классы `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 [ ]:

```python
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
        '''

        # если вершина - лист, возвращаем в качестве предсказания
        # среднее всех элементов, содержащихся в ней
        if self.nodes[v].left_son == None:
            <...>

        # 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 <...>
        elif x[self.nodes[v].split_feature] >= self.nodes[v].split_threshold:
            return <...>
        # 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 <...>

    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]
```

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 [ ]:

```
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 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 <...>
        elif x[self.nodes[v].split_feature] >= self.nodes[v].split_threshold:
            return <...>
        # а если у объекта отсутствует значение
        # разделяющего признака, то будем спускаться
        # в оба поддерева
        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 <...>

    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)
```

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 писать 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 [ ]:

```
from sklearn.datasets import load_wine

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

Далее для критерия Джини и энтропийного критерия найдем оптимальные параметры обучения дерева - `max_depth` и `min_samples_split`.

In [ ]:

```
classification_criteria = <...>
criterion_names = <...>
```

С начала надо разбить выборку на train и test.

In [ ]:

```
X_train, X_test, y_train, y_test = <...>
```

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

In [ ]:

```
for criterion, criterion_name in zip(classification_criteria, criterion_names):
    accuracy = <...>
    print('accuracy:', accuracy)
    assert(accuracy >= 0.85, "Something is wrong with your classifier")
```

Построение графиков.

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

In [ ]:

```
# код построения графиков
<...>
```

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

Вывод.

<...>

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 [ ]:

```
from sklearn.datasets import load_boston

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

In [ ]:

```
regression_criteria = <...>
criterion_names = <...>
```

In [ ]:

```
<...>
```

Разобьём выборку на обучение и тест.

In [ ]:

```
X_train, X_test, y_train, y_test = <...>
```

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

In [ ]:

```
<...>
```

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.

<...>

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 [2]:

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

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 10000 строк данных.

In [ ]:

```
adult_df = pd.read_csv('adult.data', header=None)[:10000]
adult_df.columns = column_names
target = adult_df['target'] == ' >50K'
adult_df = adult_df.drop(['target'], axis=1)
adult_df.head()
```

Предобработаем датасет, заменив категориальные признаки one-hot векторами.

In [ ]:

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

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 [ ]:

```
all_indices = np.arange(adult_df.shape[0])

for feature in column_names:
    if f'{feature}_ ?' in adult_df.columns:
        none_indices = all_indices[adult_df[f'{feature}_ ?'] == 1]

        for dummy_feature in adult_df.columns:
            if not dummy_feature.startswith(f'{feature}_ '):
                continue
            if dummy_feature != f'{feature}_ ?':
                adult_df[dummy_feature][none_indices] = None
        adult_df = adult_df.drop(f'{feature}_ ?', axis=1)
```

Посмотрим на распределение пропущенных значений по признакам.

In [ ]:

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

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

In [ ]:

```
X_adult_train, X_adult_test, y_adult_train, y_adult_test = train_test_split(
    adult_df, target, random_state=777
)
```

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

In [ ]:

```
<...>
```

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

In [ ]:

```
<...>
```

Вывод.

<...>