Дисциплина «Искусственный интеллект»

Рабочая тетрадь № 5

Теоретический материал – Деревья принятия решений

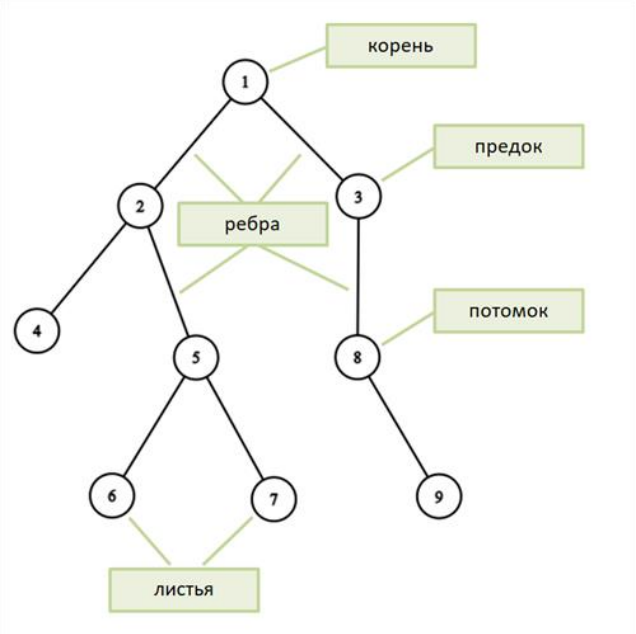
Деревья решений являются одним из наиболее эффективных инструментов интеллектуального анализа данных и предсказательной аналитики, которые позволяют решать задачи классификации и регрессии.

Перед тем как непосредственно перейти к решению задач с использование данного инструмента рассмотрим общее понятие "дерево" в информатике и способы задания деревьев в языке Python.

Деревья принадлежат к числу основных структур данных, используемых в программировании. Древовидная структура является одним из способов представления иерархической структуры в графическом виде. Такое название она получила потому, что граф выглядит как перевернутое дерево. Корень дерева (корневой узел) находится на самом верху, а листья (потомки) — внизу.

Деревья широко применяются в компьютерных технологиях. Примером является файловая система, представляющая собой иерархическую структуру из файлов и каталогов.

Схематично дерево и его основные элементы приведены на рисунке ниже.



На рисунке изображены родительские отношения (ребра, ветви дерева) между узлами (вершинами) дерева. На верхнем уровне каждый «родитель» указывает на своих «потомков». То есть в этой иерархической структуре вершина всегда «знает» своих потомков.

Для того чтобы более точно оперировать структурой Дерево, нужно дать определение некоторым ключевым понятиям:

– корневой узел — самый верхний узел дерева, он не имеет предков;

– лист, листовой или терминальный узел — конечный узел, то есть не имеющий потомков;

– внутренний узел — любой узел дерева, имеющий потомков, то есть не лист.

С корневого узла начинается выполнение большинства операций над деревом. Чтобы получить доступ к любому элементу структуры, необходимо, переходя по ветвям, перебирать элементы, начиная с головы — корневого узла. Корневой узел — это своеобразный вход в дерево. Большинство алгоритмов работы с деревом строятся на том, что каждый узел дерева рассматриваются как корневой узел поддерева, «растущего» из этого узла. Такой подход дает возможность зацикливать выполнение операций при прохождении по элементам дерева. Но в связи с тем, что при прохождении по дереву (в отличие от массива) неизвестно сколько шагов будет в этом цикле, используется другой инструмент — рекурсивный вызов.

Двоичное (бинарное) дерево — это древовидная структура данных, где каждый узел имеет не более двух детей. Этих детей называют левым (Л) и правым (П) потомком или «сыном». На рисунке выше дерево является двоичным.

1.1. Теоретический материал – Основы объектно-ориентированного программирования в Python

В предыдущих разделах мы рассматривали в основном традиционное программирование на Python, когда вся программа разбивается (или не разбивается) на отдельные модули, содержащие функции. Такое программирование соответствует парадигме структурного программирования. Само структурное программирование оказалось колоссальным шагом в построении программ. Однако еще большим шагом является парадигма объектно-ориентированного программирования. В этом подходе программа состоит из отдельных классов, которые объединяют в себе как переменные, называемые полями класса, так и функции, называемые методами класса.

На самом деле мы уже сталкивались с классами, когда создавали объекты для решения задач классификации и регрессии в Scikit-learn. В данном разделе подробнее познакомимся с основами объектно-ориентированного программирования (ООП).

Объектно-ориентированное программирование состоит из трех китов:

– инкапсуляция;
– наследование;
– полиморфизм.

Рассмотрим на примерах эти понятия. Первое - инкапсуляция - это объединение в одном объекте данных и программного кода таким образом, что для внешней работы внутренняя часть объекта может быть скрыта от пользователя. Инкапсуляция может быть реализована не только с помощью классов, но и с помощью модулей, но классы позволяют сделать

инкапсуляцию естественным путем. Создадим класс в Python. Для этого необходимо определить класс (новый тип данных) и создать объект, называемый экземпляром класса. Мы рекомендуем имена классов начинать с заглавной буквы "T", подчеркивая тем самым, что речь идет о типе данных. Делается это так:

```
class TAnimal:
  name = ""
  def __init__(self, name):
     self.name = name
  def say(self):
     print(self.name)
```

Теперь создадим экземпляр этого класса. Экземпляр класса представляет собой переменную, с которой можно работать обычным образом.

```
Animal = TAnimal("Обезьяна")
Animal.say()
```

Рассмотрим синтаксис Python при создании классов. Все начинается с ключевого слова class. Далее в блоке из отступов мы определяем переменные, которые будем называть полями и функции, которые называются методами. Методы определяются, как обычные функции и могут возвращать значения. Единственное отличие состоит в том, что у всех методов есть обязательный первый параметр, который по традиции всегда называем self в котором передается ссылка на экземпляр класса. Поэтому когда внутри класса метод хочет обратиться к своему полю, то необходимо использовать конструкцию self.name. Заметим, что при вызове методов мы первый параметр не задаем.

Далее, у каждого класса есть метод, с именем __init__, который называется конструктором класса. Этот метод вызывается в момент создания экземпляра Animal = TAnimal("Обезьяна"). Конструктор может иметь любое количество параметров. Предположим, что теперь нам нужно сделать класс для описания конкретного животного - кошки. Для это мы используем наследование классов, когда можно определять новые классы, как наследники существующих. При этом новый класс будет иметь все поля и методы наследуемого класса. Вот как это делается:

```
class TAnimal:
   name = ""
   def __init__(self, name):
      self.name = name
   def say(self):
      print(self.name)


class TCat(TAnimal):
```

```
    def may(self):
        print("Мяу!")

Cat = TCat("Кошка")
Cat.say()
Cat.may()
```

Мы видим, что у наследованного класса сохранился конструктор и метод say. В последнем примере мы выдели, что наследный класс, также как и исходный имеет конструктор, который принимает в качестве параметра - название животного тогда, что в данном случае излишне. Для решения этой проблемы мы воспользуемся объектно-ориентированным механизмом - полиморфизмом. Полиморфизм - это возможность замены методов при наследовании. Сделаем так, чтобы не нужно было передавать в конструкторе название "Кошка".

```
class TCat(TAnimal):
    def __init__(self):
        super().__init__("Кошка")
    def may(self):
        print("Мяу!")

Cat = TCat()
Cat.say()
Cat.may()
```

Результат выполнения этой программы будет аналогичный, но теперь при использовании этого класса нам не нужно передавать в конструкторе никаких параметров. Полиморфное перекрытие методов делается простым объявлением метода (в данном случае конструктора). При этом нельзя можно менять входные параметры. Если в результате написания кода метода возникает необходимость вызвать перекрытый метод, то для этого необходимо использовать функцию super(), которая по сути просто возвращает ссылку на родительский класс. Самое удивительное в полиморфизме, что изменяя метод, он меняется даже когда на него есть ссылки родительского класса. Рассмотрим еще один пример. Пусть у нас есть класс:

```
class TDo:
    def Operation(self, x, y):
        return x + y
    def Run(self):
        x = int(input("Enter x > "))
        y = int(input("Enter y > "))
        z = self.Operation(x, y)
        print("Result = " + z.__str__())
```

```
Do = TDo()
Do.Run()
```

С помощью полиморфизма заменим функцию Operation на другую в наследном классе:

```
class TDo2(TDo):
   def Operation(self, x, y):
      return x * y
```

1.2.1 Пример

Задача:

Необходимо разработать виртуальную модель процесса обучения. В программе должны быть объекты-ученики, учитель, кладезь знаний.

Потребуется три класса – "учитель", "ученик", "данные". Учитель и ученик во многом похожи, оба – люди. Значит, их классы могут принадлежать одному надклассу "человек". Однако в контексте данной задачи у учителя и ученика вряд ли найдутся общие атрибуты. Определим, что должны уметь объекты для решения задачи "увеличить знания":

- Ученик должен уметь брать информацию и превращать ее в свои знания.
- Учитель должен уметь учить группу учеников.
- Данные могут представлять собой список знаний. Элементы будут извлекаться по индексу.

Решение:

```
class Data:
    def __init__(self, *info):
        self.info = list(info)
    def __getitem__(self, i):
        return self.info[i]

class Teacher:
    def teach(self, info, *pupil):
        for i in pupil:
            i.take(info)

class Pupil:
    def __init__(self):
        self.knowledge = []
    def take(self, info):
        self.knowledge.append(info)

lesson = Data('class', 'object', 'inheritance', 'polymorphism', 'encapsulation')
marIvanna = Teacher()
vasy = Pupil()
pety = Pupil()
marIvanna.teach(lesson[2], vasy, pety)
marIvanna.teach(lesson[0], pety)
print(vasy.knowledge)
print(pety.knowledge)
```

Ответ:

```
['inheritance']
['inheritance', 'class']
```

1.2.2 Пример

Задача:

Напишите программу по следующему описанию. Есть класс "Воин". От него создаются два экземпляра-юнита. Каждому устанавливается здоровье в 100 очков. В случайном порядке они бьют друг друга. Тот, кто бьет, здоровья не теряет. У того, кого бьют, оно уменьшается на 20 очков от одного удара. После каждого удара надо выводить сообщение, какой юнит атаковал, и сколько у противника осталось здоровья. Как только у кого-то заканчивается ресурс здоровья, программа завершается сообщением о том, кто одержал победу.

Решение:

```
import random
class Warrior:
    def __init__(self,health):
        self.health = health

    def hit(self,target,target1):
        if target.health > 0:
            target.health -= 20
        if target1 == warrior1:
            target1 = "Warrior1"
        if target1 == warrior2:
            target1 = "Warrior2"
        print(target1, " has attacked")
        print(target.health, " left")
        if target.health == 0:
           print(target1, " has won")

warrior1 = Warrior(100)
warrior2 = Warrior(100)
q = int(input("Enter 1 to attack. Enter 2 to stop program:"))

while q != 2:
    if q == 1:
        j = random.randint(1,3)
        if j % 2 == 0:
            warrior1.hit(warrior2,warrior1)
            q = int(input("Enter 1 to let some warrior attack:"))
        else:
            warrior2.hit(warrior1,warrior2)
            q = int(input("Enter 1 to let some warrior attack:"))
    else:
        print("Wrong input.")
        break
```

Ответ:

```
Enter 1 to attack. Enter 2 to stop program:1
Warrior1  has attacked
80  left
Enter 1 to let some warrior attack:1
Warrior2  has attacked
80  left
Enter 1 to let some warrior attack:1
Warrior1  has attacked
60  left
Enter 1 to let some warrior attack:1
Warrior2  has attacked
60  left
Enter 1 to let some warrior attack:1
Warrior2  has attacked
40  left
Enter 1 to let some warrior attack:1
Warrior2  has attacked
20  left
Enter 1 to let some warrior attack:1
Warrior2  has attacked
0  left
Warrior2  has won
Enter 1 to let some warrior attack:2
```

1.2.3 Пример

Задача:

Создайте класс по работе с дробями. В классе должна быть реализована следующая функциональность:

- сложение дробей;
- вычитание дробей;
- умножение дробей;
- деление дробей.

Решение:

```python
class Rational:

    @staticmethod
    def gcd(a,b):
        while (b != 0):
            (a,b) = (b,a%b)
        return a

    @staticmethod
    def sgn(x):
        if x>0:
            return 1
        elif x<0:
            return -1
        else:
            return 0

    def __init__(self,n,d):
        if n==0:
            self.num=0
            self.den=1
        else:
            z=self.sgn(n)*self.sgn(d)
            n=abs(n)
            d=abs(d)
            k=self.gcd(n,d)
            self.num=z*n//k
            self.den=d//k

    def __str__(self):
        if self.num==0:
            return "0"
        else:
            return str(self.num)+"/"+str(self.den)

    def __add__(self,o):
        n1=self.num
        d1=self.den
        if type(o)==int:
            n2=o
            d2=1
        else:
            n2=o.num
            d2=o.den
        n=n1*d2+n2*d1
        d=d1*d2
        return Rational(n,d)
```

```
    def __radd__(self,o):
        n1=self.num
        d1=self.den
        if type(o)==int:
            n2=o
            d2=1
        else:
            n2=o.num
            d2=o.den
        n=n1*d2+n2*d1
        d=d1*d2
        return Rational(n,d)

    def __sub__(self,o):
        n1=self.num
        d1=self.den
        n2=o.num
        d2=o.den
        n=n1*d2-n2*d1
        d=d1*d2
        return Rational(n,d)

    def __mul__(self,o):
        n1=self.num
        d1=self.den
        n2=o.num
        d2=o.den
        n=n1*n2
        d=d1*d2
        return Rational(n,d)

    def __floordiv__(self,o):
        n1=self.num
        d1=self.den
        n2=o.num
        d2=o.den
        n=n1*d2
        d=d1*n2
        return Rational(n,d)


d1=Rational(1,2)
d2=Rational(1,3)
d3=d1+d2
print(d3)
d4=d1-d2
print(d4)
d5=d1*d2
print(d5)
d6=d1*d2
print(d6)
d7=d1//d2
print(d7)
d8=6+d1
print(d8)
```

Ответ:

```
5/6
1/6
1/6
1/6
3/2
13/2
```

Задание:

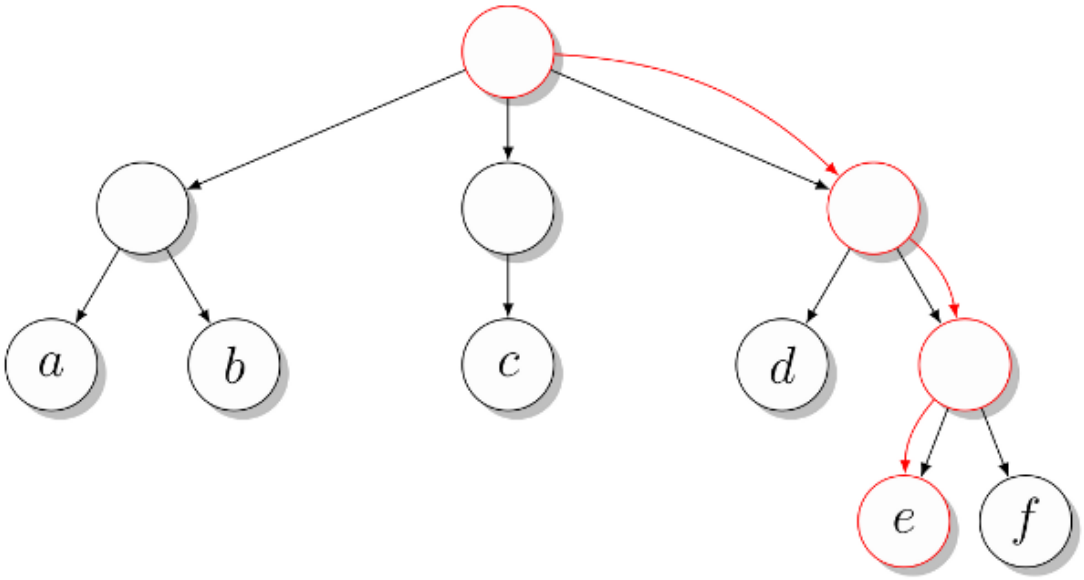
Создайте класс по работе с тригонометрическими функциями. В классе должны быть реализованы функции вычисления:

- косинуса;
- синуса;
- тангенса;
- арксинуса;
- арккосинуса;
- арктангенса;
- перевода из градусов в радианы.

1.2. Теоретический материал – Реализация деревьев в Python

Любое представление графов, естественно, можно использовать для представления деревьев, потому что деревья — это особый вид графов. Однако, деревья играют свою большую роль в алгоритмах, и для них разработано много соответствующих структур и методов. Большинство алгоритмов на деревьях (например, поиск по деревьям) можно рассматривать в терминах теории графов, но специальные структуры данных делают их проще в реализации.

Проще всего описать представление дерева с корнем, в котором ребра спускаются вниз от корня. Такие деревья часто отображают иерархическое ветвление данных, где корень отображает все объекты (которые, возможно, хранятся в листьях), а каждый внутренний узел показывает объекты, содержащиеся в дереве, корень которого — этот узел. Это описание можно использовать, представив каждое поддерево списком, содержащим все его поддеревья-потомки. Рассмотрим простое дерево, показанное на рисунке ниже.



Мы можем представить это дерево как список списков:

T = [["a", "b"], ["c"], ["d", ["e", "f"]]]

print(T[0][1])

print(T[2][1][0])

Каждый список в сущности является списком потомков каждого из внутренних узлов. Во втором примере мы обращаемся к третьему потомку корня, затем ко второму его потомку и в конце концов — к первому потомку предыдущего узла (этот путь отмечен на рисунке). В ряде случаев возможно заранее определить максимальное число потомков каждого узла. (Например, каждый узел бинарного дерева может иметь до двух потомков). Поэтому можно использовать другие представления, скажем, объекты с отдельным атрибутом для каждого из потомков как в листинге ниже.

1.2.1 Пример

Задача:

Определите класс бинарного дерева и задайте его объекты с отдельным атрибутом для каждого из потомков.

Решение:

```
class Tree:
    def __init__(self, left, right):
        self.left = left
        self.right = right

t = Tree(Tree("a", "b"), Tree("c", "d"))
t.right.left
```

Ответ:

```
'c'
```

1.2.2 Пример

Для обозначения отсутствующих потомков можно использовать None (в случае если у узла только один потомок). Само собой, можно комбинировать разные методы (например, использовать списки или множества потомков для каждого узла).

Распространенный способ реализации деревьев, особенно на языках, не имеющих встроенной поддержки списков, это так называемое представление «первый потомок, следующий брат». В нем каждый узел имеет два «указателя» или атрибута, указывающих на другие узлы, как в бинарном дереве. Однако, первый из этих атрибутов ссылается на первого потомка узла, а второй — на его следующего брата (т.е. узел, имеющий того же родителя, но находящийся правее, — прим. перев). Иными словами, каждый узел дерева имеет указатель на связанный список его потомков, а каждый из этих потомков ссылается на свой собственный аналогичный список. Таким образом, небольшая модификация бинарного дерева даст нам многопутевое дерево, показанное в листинге ниже.

Решение:

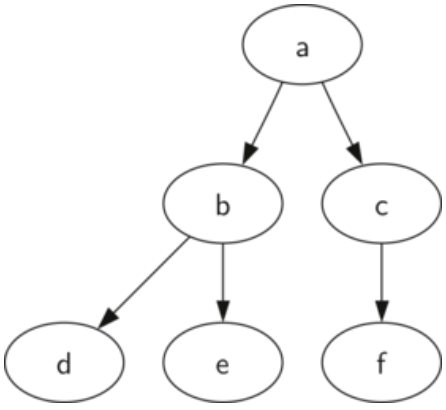
```
class Tree:
    def __init__(self, kids, next=None):
        self.kids = self.val = kids
        self.next = next

t = Tree(Tree("a", Tree("b", Tree("c", Tree("d")))))
t.kids.next.next.val
```

Ответ:

```
'c'
```

Задание

Представьте дерево показанное на рисунке с использованием списка из списков. Выведите на печать корень дерева, а также его левое и правое поддеревья.



Задание:

Дан класс, описывающий бинарное дерево.

```
class Tree:
  def __init__(self, data):
    self.left = None
    self.right = None
    self.data = data
  def PrintTree(self):
    print(self.data)
```

Реализуйте в классе функцию для вставки нового элемента в дерево по следующим правилам:

- Левое поддерево узла содержит только узлы со значениями меньше, чем значение в узле.
- Правое поддерево узла содержит только узлы со значениями меньше, чем значение в узле.
- Каждое из левого и правого поддеревьев также должно быть бинарным деревом поиска.
- Не должно быть повторяющихся узлов.

Метод вставки сравнивает значение узла с родительским узлом и решает куда доваить элемент (в левое или правое поддерево). Перепишите, метод PrintTree для печати полной версии дерева.

1.3. Теоретический материал – Деревья решений

Дерево решений – это один из наиболее часто и широко используемых алгоритмов контролируемого машинного обучения, который может выполнять как регрессионные, так и классификационные задачи.

Использование деревьев решений для прогнозного анализа имеет ряд преимуществ:

1. Деревья решений могут быть использованы для прогнозирования как непрерывных, так и дискретных значений, т. е. они хорошо работают как для задач регрессии, так и для задач классификации.
2. Они требуют относительно меньших усилий для обучения алгоритма.
3. Они могут быть использованы для классификации нелинейно разделимых данных.
4. Они очень быстры и эффективны по сравнению с KNN и другими алгоритмами классификации.

Решим модельные примеры классификации и регрессии, разобранные в предыдущих раочих тетрадях, но с использованием деревьев принятия решений.

1.3.1 Пример

Задача:

Построим дерево решений для задачи классификации, для этого, построим границу решения для каждого класса. В качестве данных будем использовать уже знакомый нам и встроенный в библиотеку sklearn набор данных ирисов Фишера. Импортируем библиотеки, набор данных и посмотрим его характеристики.

Решение:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

dataset = sns.load_dataset('iris')
dataset
```

```
dataset.shape
```

```
dataset.head()
```

Далее, разделим наши данные на атрибуты и метки, а затем выделим в общей совокупности полученных данных обучающие и тестовые наборы. Таким образом, мы можем обучить наш алгоритм на одном наборе данных, а затем протестировать его на совершенно на другом наборе, который алгоритм еще не видел. Это дает вам более точное представление о том, как на самом деле будет работать ваш обученный алгоритм.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    # поскольку iris это pandas-таблица, для нее нужно указывать iloc
    dataset.iloc[:, :-1],  # берем все колонки кроме последней в признаки
    dataset.iloc[:, -1],  # последнюю в целевую переменную (класс)
    test_size = 0.20 # размер тестовой выборки 20%
)

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
X_train.head()
```

```
y_train.head()
```

После того, как данные были разделены на обучающие и тестовые наборы, последний шаг состоит в том, чтобы обучить алгоритм дерева решений на этих данных и сделать прогнозы. Scikit-Learn содержит библиотеку tree , которая содержит встроенные классы/методы для различных алгоритмов дерева решений. Поскольку мы собираемся выполнить здесь задачу классификации, мы будем использовать класс DecisionTreeClassifier для этого примера. Метод fit этого класса вызывается для обучения алгоритма на обучающих данных, которые передаются в качестве параметра методу fit . Выполним следующий сценарий для обучения алгоритма.

```
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier()
classifier.fit(X_train, y_train)
```

```
#построим делево решений
from sklearn import tree
tree.plot_tree(classifier)
```

Теперь, когда наш классификатор обучен, давайте сделаем прогнозы по тестовым данным. Для составления прогнозов используется метод predict класса Decision Tree Classifier. Взгляните на следующий код для использования.

```
y_pred = classifier.predict(X_test)
y_pred
```

На данный момент мы обучили наш алгоритм и сделали некоторые прогнозы. Теперь посмотрим, насколько точен наш алгоритм. Для задач классификации обычно используются такие метрики, как матрица путаницы, точность. Библиотека Scikit-Learn metrics содержит методы classification_report и confusion_matrix, которые могут быть использованы для расчета этих метрик.

```
from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Из матрицы оценок алгоритма вы можете видеть, что из 30 тестовых экземпляров наш алгоритм неправильно классифицировал только 3. Это приблизительно 91 % точности.

Ответ:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

150 rows × 5 columns

```
(150, 5)
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

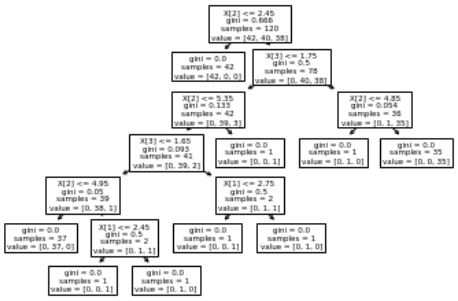
```
((120, 4), (30, 4), (120,), (30,))
```

	sepal_length	sepal_width	petal_length	petal_width
112	6.8	3.0	5.5	2.1
25	5.0	3.0	1.6	0.2
24	4.8	3.4	1.9	0.2
79	5.7	2.6	3.5	1.0
69	5.6	2.5	3.9	1.1

```
112     virginica
25         setosa
24         setosa
79     versicolor
69     versicolor
Name: species, dtype: object
```

```
DecisionTreeClassifier()
```

```
[Text(182.61818181818182, 201.90857142857143, 'X[2] <= 2.45\ngini = 0.666\nsamples = 120\nvalue = [42, 40, 38]'),
 Text(152.1818181818182, 170.84571428571428, 'gini = 0.0\nsamples = 42\nvalue = [42, 0, 0]'),
 Text(213.05454545454546, 170.84571428571428, 'X[3] <= 1.75\ngini = 0.5\nsamples = 78\nvalue = [0, 40, 38]'),
 Text(152.1818181818182, 139.78285714285715, 'X[2] <= 5.35\ngini = 0.133\nsamples = 42\nvalue = [0, 39, 3]'),
 Text(121.74545454545455, 108.72, 'X[3] <= 1.65\ngini = 0.093\nsamples = 41\nvalue = [0, 39, 2]'),
 Text(60.872727272727275, 77.65714285714284, 'X[2] <= 4.95\ngini = 0.05\nsamples = 39\nvalue = [0, 38, 1]'),
 Text(30.436363636363637, 46.59428571428572, 'gini = 0.0\nsamples = 37\nvalue = [0, 37, 0]'),
 Text(91.30909090909091, 46.59428571428572, 'X[1] <= 2.45\ngini = 0.5\nsamples = 2\nvalue = [0, 1, 1]'),
 Text(60.872727272727275, 15.531428571428563, 'gini = 0.0\nsamples = 1\nvalue = [0, 0, 1]'),
 Text(121.74545454545455, 15.531428571428563, 'gini = 0.0\nsamples = 1\nvalue = [0, 1, 0]'),
 Text(182.61818181818182, 77.65714285714284, 'X[1] <= 2.75\ngini = 0.5\nsamples = 2\nvalue = [0, 1, 1]'),
 Text(152.1818181818182, 46.59428571428572, 'gini = 0.0\nsamples = 1\nvalue = [0, 0, 1]'),
 Text(213.05454545454546, 46.59428571428572, 'gini = 0.0\nsamples = 1\nvalue = [0, 1, 0]'),
 Text(182.61818181818182, 108.72, 'gini = 0.0\nsamples = 1\nvalue = [0, 0, 1]'),
 Text(273.92727272727274, 139.78285714285715, 'X[2] <= 4.85\ngini = 0.054\nsamples = 36\nvalue = [0, 1, 35]'),
 Text(243.4909090909091, 108.72, 'gini = 0.0\nsamples = 1\nvalue = [0, 1, 0]'),
 Text(304.3636363636364, 108.72, 'gini = 0.0\nsamples = 35\nvalue = [0, 0, 35]')]
```



```
array(['virginica', 'versicolor', 'virginica', 'versicolor', 'virginica',
       'virginica', 'versicolor', 'setosa', 'versicolor', 'versicolor',
       'setosa', 'virginica', 'versicolor', 'versicolor', 'versicolor',
       'setosa', 'setosa', 'virginica', 'versicolor', 'versicolor',
       'setosa', 'virginica', 'versicolor', 'virginica', 'versicolor',
       'virginica', 'versicolor', 'virginica', 'versicolor', 'virginica'],
      dtype=object)
```

```
[[ 5  0  0]
 [ 0 12  1]
 [ 0  2 10]]
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00         5
  versicolor       0.86      0.92      0.89        13
   virginica       0.91      0.83      0.87        12

    accuracy                           0.90        30
   macro avg       0.92      0.92      0.92        30
weighted avg       0.90      0.90      0.90        30
```

Задание
Задача:
Постройте классификатор на основе дерева принятия решений следующего датасета:

```
# данные
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
target =  [0, 0, 0, 1, 1, 1]
```

1.4. Теоретический материал – Дерево решений для регрессии

Дерево решений для регрессии

Процесс решения регрессионной задачи с деревом решений с помощью Scikit Learn очень похож на процесс классификации. Однако для регрессии мы используем класс DecisionTreeRegressor древовидной библиотеки. Кроме того, оценочные показатели регрессии отличаются от показателей классификации. В остальном процесс почти такой же.

Построим регрессию с использованием дерева решений в Python и библиотеки scikit-learn. В качестве исходного набора данных будем использовать зависимость заработной платы от опыта работы из предыдущей тетради:

https://raw.githubusercontent.com/AnnaShestova/salary-years-simple-linear-regression/master/Salary_Data.csv

1.4.1 Пример

Задача:

Постойте регрессию с использованием дерева решений, реализованного в Python.

Решение:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

url = r'https://raw.githubusercontent.com/AnnaShestova/salary-years-simple-linear-regression/master/Salary_Data.csv'
dataset = pd.read_csv(url)
dataset.head()

#Исследуем набор данных
print(dataset.shape)
dataset.describe()
```

```
# Нарисуем точечную диаграмму
plt.scatter (dataset['YearsExperience'], dataset['Salary'], color = 'b', label = "Заработная плата")
plt.xlabel("Опыт(лет)")
plt.ylabel("Заработная плата")
plt.show()
```

```
from sklearn.tree import DecisionTreeRegressor
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 1].values
print(X)
print(y)

# Теперь, когда у нас есть атрибуты и метки, необходимо разделить их на а обучающий и тестовый наборы.
# Приведенный фрагмент разделяет 80% данных на обучающий набор, а 20% данных – на набор тестов
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

# далее можно обучить алгоритм линейной регрессии
# необходимо импортировать класс LinearRegression, создать его экземпляр и вызвать метод fit()
regressor = DecisionTreeRegressor()
regressor.fit(X_train, y_train)
```

```
from sklearn import tree
tree.plot_tree(regressor)
```

Построим прогноз:

```
y_pred = regressor.predict(X_test)
y_pred
```

Теперь сравним некоторые из наших прогнозируемых значений с фактическими значениями:

```
df=pd.DataFrame({'Actual':y_test, 'Predicted':y_pred})
df
```

Расчитаем среднюю абсолютную и среднеквадратичную ошибку регрессии:

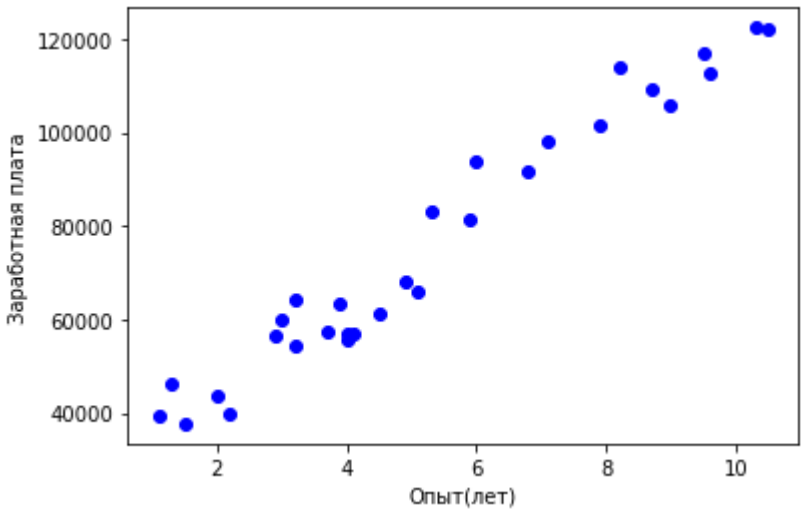
```
from sklearn import metrics
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
```

```
metrics.mean_absolute_error(y_test, y_pred) / np.average(y) * 100
```

Ответ:

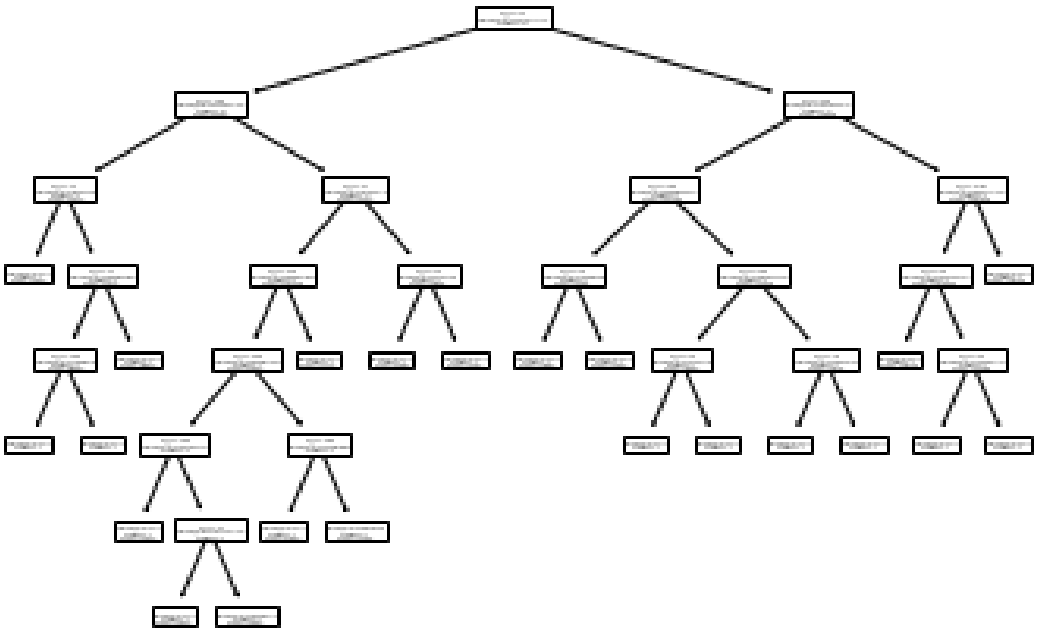
```
(30, 2)
```

	YearsExperience	Salary
count	30.000000	30.000000
mean	5.313333	76003.000000
std	2.837888	27414.429785
min	1.100000	37731.000000
25%	3.200000	56720.750000
50%	4.700000	65237.000000
75%	7.700000	100544.750000
max	10.500000	122391.000000




```
[[ 1.1]
 [ 1.3]
 [ 1.5]
 [ 2. ]
 [ 2.2]
 [ 2.9]
 [ 3. ]
 [ 3.2]
 [ 3.2]
 [ 3.7]
 [ 3.9]
 [ 4. ]
 [ 4. ]
 [ 4.1]
 [ 4.5]
 [ 4.9]
 [ 5.1]
 [ 5.3]
 [ 5.9]
 [ 6. ]
 [ 6.8]
 [ 7.1]
 [ 7.9]
 [ 8.2]
 [ 8.7]
 [ 9. ]
 [ 9.5]
 [ 9.6]
 [10.3]
 [10.5]]
[ 39343.  46205.  37731.  43525.  39891.  56642.  60150.  54445.  64445.
  57189.  63218.  55794.  56957.  57081.  61111.  67938.  66029.  83088.
  81363.  93940.  91738.  98273. 101302. 113812. 109431. 105582. 116969.
 112635. 122391. 121872.]

DecisionTreeRegressor()
```

```
[Text(165.95689655172413, 203.85, 'X[0] <= 5.2\nsquared_error = 614737637.832\nsamples = 24\nvalue = 73886.208'),
 Text(69.26896551724138, 176.67000000000002, 'X[0] <= 2.55\nsquared_error = 81200345.857\nsamples = 14\nvalue = 54976.0'),
 Text(23.089655172413792, 149.49, 'X[0] <= 1.2\nsquared_error = 7820714.0\nsamples = 4\nvalue = 42241.0'),
 Text(11.544827586206896, 122.31, 'squared_error = 0.0\nsamples = 1\nvalue = 39343.0'),
 Text(34.63448275862069, 122.31, 'X[0] <= 2.1\nsquared_error = 6694994.667\nsamples = 3\nvalue = 43207.0'),
 Text(23.089655172413792, 95.13, 'X[0] <= 1.65\nsquared_error = 1795600.0\nsamples = 2\nvalue = 44865.0'),
 Text(11.544827586206896, 67.94999999999999, 'squared_error = 0.0\nsamples = 1\nvalue = 46205.0'),
 Text(34.63448275862069, 67.94999999999999, 'squared_error = 0.0\nsamples = 1\nvalue = 43525.0'),
 Text(46.179310344827584, 95.13, 'squared_error = 0.0\nsamples = 1\nvalue = 39891.0'),
 Text(115.44827586206895, 149.49, 'X[0] <= 4.7\nsquared_error = 19731272.6\nsamples = 10\nvalue = 60070.0'),
 Text(92.35862068965517, 122.31, 'X[0] <= 4.25\nsquared_error = 9499922.484\nsamples = 8\nvalue = 58341.625'),
 Text(80.81379310344828, 95.13, 'X[0] <= 3.45\nsquared_error = 9604901.143\nsamples = 7\nvalue = 57946.0'),
 Text(57.72413793103448, 67.94999999999999, 'X[0] <= 2.95\nsquared_error = 14313358.25\nsamples = 4\nvalue = 58920.5'),
 Text(46.179310344827584, 40.77000000000001, 'squared_error = 0.0\nsamples = 1\nvalue = 56642.0'),
 Text(69.26896551724138, 40.77000000000001, 'X[0] <= 3.1\nsquared_error = 16777116.667\nsamples = 3\nvalue = 59680.0'),
 Text(57.72413793103448, 13.590000000000003, 'squared_error = 0.0\nsamples = 1\nvalue = 60150.0'),
 Text(80.81379310344828, 13.590000000000003, 'squared_error = 25000000.0\nsamples = 2\nvalue = 59445.0'),
 Text(103.90344827586206, 67.94999999999999, 'X[0] <= 3.85\nsquared_error = 372490.889\nsamples = 3\nvalue = 56646.667'),
 Text(92.35862068965517, 40.77000000000001, 'squared_error = 0.0\nsamples = 1\nvalue = 57189.0'),
 Text(115.44827586206895, 40.77000000000001, 'squared_error = 338142.25\nsamples = 2\nvalue = 56375.5'),
 Text(103.90344827586206, 95.13, 'squared_error = 0.0\nsamples = 1\nvalue = 61111.0'),
 Text(138.53793103448277, 122.31, 'X[0] <= 5.0\nsquared_error = 911070.25\nsamples = 2\nvalue = 66983.5'),
 Text(126.99310344827586, 95.13, 'squared_error = 0.0\nsamples = 1\nvalue = 67938.0'),
 Text(150.08275862068965, 95.13, 'squared_error = 0.0\nsamples = 1\nvalue = 66029.0'),
 Text(262.6448275862069, 176.67000000000002, 'X[0] <= 8.05\nsquared_error = 160167356.45\nsamples = 10\nvalue = 100360.5'),
 Text(213.57931034482758, 149.49, 'X[0] <= 5.95\nsquared_error = 53566814.556\nsamples = 6\nvalue = 91617.333'),
 Text(184.71724137931034, 122.31, 'X[0] <= 5.6\nsquared_error = 743906.25\nsamples = 2\nvalue = 82225.5'),
 Text(173.17241379310343, 95.13, 'squared_error = 0.0\nsamples = 1\nvalue = 83088.0'),
 Text(196.26206896551724, 95.13, 'squared_error = 0.0\nsamples = 1\nvalue = 81363.0'),
 Text(242.44137931034481, 122.31, 'X[0] <= 6.95\nsquared_error = 13823368.688\nsamples = 4\nvalue = 96313.25'),
 Text(219.35172413793103, 95.13, 'X[0] <= 6.4\nsquared_error = 1212201.0\nsamples = 2\nvalue = 92839.0'),
 Text(207.80689655172412, 67.94999999999999, 'squared_error = 0.0\nsamples = 1\nvalue = 93940.0'),
 Text(230.8965517241379, 67.94999999999999, 'squared_error = 0.0\nsamples = 1\nvalue = 91738.0'),
 Text(265.5310344827586, 95.13, 'X[0] <= 7.5\nsquared_error = 2293710.25\nsamples = 2\nvalue = 99787.5'),
 Text(253.98620689655172, 67.94999999999999, 'squared_error = 0.0\nsamples = 1\nvalue = 98273.0'),
 Text(277.07586206896553, 67.94999999999999, 'squared_error = 0.0\nsamples = 1\nvalue = 101302.0'),
 Text(311.7103448275862, 149.49, 'X[0] <= 10.05\nsquared_error = 33407056.688\nsamples = 4\nvalue = 113475.25'),
 Text(300.1655172413793, 122.31, 'X[0] <= 8.6\nsquared_error = 13207004.222\nsamples = 3\nvalue = 110676.333'),
 Text(288.6206896551724, 95.13, 'squared_error = 0.0\nsamples = 1\nvalue = 113812.0'),
 Text(311.7103448275862, 95.13, 'X[0] <= 9.3\nsquared_error = 12436202.25\nsamples = 2\nvalue = 109108.5'),
 Text(300.1655172413793, 67.94999999999999, 'squared_error = 0.0\nsamples = 1\nvalue = 105582.0'),
 Text(323.2551724137931, 67.94999999999999, 'squared_error = 0.0\nsamples = 1\nvalue = 112635.0'),
 Text(323.2551724137931, 122.31, 'squared_error = 0.0\nsamples = 1\nvalue = 121872.0')]
```



```
array([ 46205. , 121872. ,  56375.5,  56375.5, 112635. , 105582. ])
```

	Actual	Predicted
0	37731.0	46205.0
1	122391.0	121872.0
2	57081.0	56375.5
3	63218.0	56375.5
4	116969.0	112635.0
5	109431.0	105582.0

```
from sklearn import metrics
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
```

```
Mean Squared Error: 25498988.416666668
Mean Absolute Error: 4120.666666666667
```

```
5.421715809463662
```

Средняя абсолютная ошибка для нашего алгоритма составляет 4120.66, что составляет менее 6 процентов от среднего значения всех значений в столбце.

Задание

Задача:

Задание. Постройте модель регрессии для данных из предыдущей рабочей тетради.Для примера можно взять потребления газа (в миллионах галлонов) в 48 штатах США или набор данных о качестве красного вина:
https://raw.githubusercontent.com/likarajo/petrol_consumption/master/data/petrol_consumption.csv
https://raw.githubusercontent.com/aniruddhachoudhury/Red-Wine-Quality/master/winequality-red.csv
Постройте прогноз. Оцените точность модели.