Машинное обучение практика

ФИО преподавателя: Оцоков Шамиль Алиевич

e-mail: shamil24@mail.ru

Библиотека NumPy

Библиотека NumPy (сокращение от Numerical Python — «числовой Python») обеспечивает эффективный интерфейс для хранения и работы с плотными буферами данных. Массивы библиотеки NumPy похожи на встроенный тип данных языка Python list, но обеспечивают гораздо более эффективное хранение и операции с данными при росте размера массивов.

```
import numpy
```

Библиотека NumPy

```
In[12]: # Создаем массив целых чисел длины 10, заполненный нулями
        np.zeros(10, dtype=int)

Out[12]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In[13]: # Создаем массив размером 3 x 5 значений с плавающей точкой,
        # заполненный единицами
        np.ones((3, 5), dtype=float)

Out[13]: array([[ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.]])

In[14]: # Создаем массив размером 3 x 5, заполненный значением 3.14
        np.full((3, 5), 3.14)

Out[14]: array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```

Библиотека NumPy

```
In[15]: # Создаем массив, заполненный линейной последовательностью,
        # начинающейся с 0 и заканчивающейся 20, с шагом 2
        # (аналогично встроенной функции range())
        np.arange(0, 20, 2)

Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In[16]: # Создаем массив из пяти значений,
        # равномерно располагающихся между 0 и 1
        np.linspace(0, 1, 5)

Out[16]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])

In[17]: # Создаем массив размером 3 x 3 равномерно распределенных
        # случайных значения от 0 до 1
        np.random.random((3, 3))

Out[17]: array([[ 0.99844933,  0.52183819,  0.22421193],
                [ 0.08007488,  0.45429293,  0.20941444],
                [ 0.14360941,  0.96910973,  0.946117  ]])
```

Библиотека NumPy

```
In[19]: # Создаем массив размером 3 x 3 случайных целых числа
        # в промежутке [0, 10)
        np.random.randint(0, 10, (3, 3))

Out[19]: array([[2, 3, 4],
                [5, 7, 8],
                [0, 5, 0]])

In[20]: # Создаем единичную матрицу размером 3 x 3
        np.eye(3)

Out[20]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])

In[21]: # Создаем неинициализированный массив из трех целочисленных
        # значений. Значениями будут произвольные, случайно оказавшиеся
        # в соответствующих ячейках памяти данные
        np.empty(3)

Out[21]: array([ 1.,  1.,  1.])
```

Библиотека NumPy

Обсудим некоторые атрибуты массивов. Начнем с описания трех массивов случайных чисел: одномерного, двумерного и трехмерного. Воспользуемся генератором случайных чисел библиотеки NumPy, задав для него начальное значение, чтобы гарантировать генерацию одних и тех же массивов при каждом выполнении кода:

```
In[1]: import numpy as np
       np.random.seed(0)  # начальное значение для целей воспроизводимости

       x1 = np.random.randint(10, size=6)           # одномерный массив
       x2 = np.random.randint(10, size=(3, 4))      # двумерный массив
       x3 = np.random.randint(10, size=(3, 4, 5))   # трехмерный массив
```

У каждого из массивов есть атрибуты `ndim` (размерность), `shape` (размер каждого измерения) и `size` (общий размер массива):

```
In[2]: print("x3 ndim: ", x3.ndim)
       print("x3 shape:", x3.shape)
       print("x3 size: ", x3.size)

x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Библиотека NumPy

Еще один полезный атрибут — `dtype`, тип данных массива (который мы уже ранее обсуждали в разделе «Работа с типами данных в языке Python» этой главы):

```
In[3]: print("dtype:", x3.dtype)

dtype: int64
```

Другие атрибуты включают `itemsize`, выводящий размер (в байтах) каждого элемента массива, и `nbytes`, выводящий полный размер массива (в байтах):

```
In[4]: print("itemsize:", x3.itemsize, "bytes")
```

```
x2 = np.random.randint(10, size=(3, 4))
# двумерный массив
x = np.arange(10)
x[:5] # первые пять элементов
x[начало:конец:шаг]
```

Библиотека NumPy

```
In[38]: grid = np.arange(1, 10).reshape((3, 3))
        print(grid)

[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

np.array(myList)
# Преобразование в вектор-строку с помощью reshape
x.reshape((1, 3))

Слияние массивов

Слияние, или объединение, двух массивов в библиотеке NumPy выполняется в основном с помощью методов `np.concatenate`, `np.vstack` и `np.hstack`. Метод `np.concatenate` принимает на входе кортеж или список массивов в качестве первого аргумента:

```
In[43]: x = np.array([1, 2, 3])
        y = np.array([3, 2, 1])
        np.concatenate([x, y])

Out[43]: array([1, 2, 3, 3, 2, 1])
```

Библиотека NumPy

Для объединения двумерных массивов можно также использовать np.concatenate:

```
In[45]: grid = np.array([[1, 2, 3],
                         [4, 5, 6]])

In[46]: # слияние по первой оси координат
        np.concatenate([grid, grid])

Out[46]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])

In[47]: # слияние по второй оси координат (с индексом 0)
        np.concatenate([grid, grid], axis=1)

Out[47]: array([[1, 2, 3, 1, 2, 3],
                [4, 5, 6, 4, 5, 6]])
```

Библиотека NumPy

```
In[48]: x = np.array([1, 2, 3])
        grid = np.array([[9, 8, 7],
                         [6, 5, 4]])

        # Объединяет массивы по вертикали
        np.vstack([x, grid])

Out[48]: array([[1, 2, 3],
                [9, 8, 7],
                [6, 5, 4]])

In[49]: # Объединяет массивы по горизонтали
        y = np.array([[99],
                      [99]])
        np.hstack([grid, y])

Out[49]: array([[ 9,  8,  7, 99],
                [ 6,  5,  4, 99]])
```

Библиотека NumPy

Универсальные функции библиотеки NumPy очень просты в использовании, поскольку применяют нативные арифметические операторы языка Python. Можно выполнять обычные сложение, вычитание, умножение и деление:

```
In[7]: x = np.arange(4)
       print("x     =", x)
       print("x + 5 =", x + 5)
       print("x - 5 =", x - 5)
       print("x * 2 =", x * 2)
       print("x / 2 =", x / 2)
       print("x // 2 =", x // 2)  # деление с округлением в меньшую сторону
```

```
x     = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [ 0.   0.5  1.   1.5]
x // 2 = [0 0 1 1]
```

```
In[10]: np.add(x, 2)
```

Библиотека NumPy

Оператор	Эквивалентная универсальная функция	Описание
+	np.add	Сложение (например, 1 + 1 = 2)
–	np.subtract	Вычитание (например, 3 – 2 = 1)
–	np.negative	Унарная операция изменения знака (например, –2)
*	np.multiply	Умножение (например, 2 * 3 = 6)
/	np.divide	Деление (например, 3 / 2 = 1.5)
//	np.floor_divide	Деление с округлением в меньшую сторону (например, 3 // 2 = 1)
**	np.power	Возведение в степень (например, 2 ** 3 = 8)
%	np.mod	Модуль/остаток (например, 9 % 4 = 1)

Библиотека NumPy

В библиотеке NumPy также реализованы операторы сравнения, такие как < («меньше») и > («больше») в виде поэлементных универсальных функций. Результат этих операторов сравнения всегда представляет собой массив с булевым типом данных. Доступны для использования все шесть
стандартных операторов сравнения:

```
In[4]: x = np.array([1, 2, 3, 4, 5])

In[5]: x < 3  # меньше

Out[5]: array([ True,  True, False, False, False], dtype=bool)

In[6]: x > 3  # больше

In[11]: (2 * x) == (x ** 2)

Out[11]: array([False,  True, False, False, False], dtype=bool)
```

Библиотека NumPy

Срезы также поддерживаются:
arr[0:5] = 100

```
In [90]: arr
Out[90]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

In [91]: slice_of_arr = arr[0:6]

In [92]: slice_of_arr
Out[92]: array([0, 1, 2, 3, 4, 5])

In [94]: slice_of_arr[:] = 99

In [95]: slice_of_arr
Out[95]: array([99, 99, 99, 99, 99, 99])

In [96]: arr
Out[96]: array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
```

```
arr_copy = arr.copy()
```

Библиотека NumPy

```
arr_2d
```

```
array([[ 5, 10, 15],
       [20, 25, 30],
       [35, 40, 45]])
```

```
arr_2d[:2,1:]
```

```
array([[10, 15],
       [25, 30]])
```

Задание. Вырезать из матрицы 6 на 6 подматрицу 4 на 4 посередине

Библиотека NumPy

```
arr2 = np.arange(1,11)
```

```
arr2
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
arr2>5
```

```
array([False, False, False, False, False,  True,  True,  True,  True,
        True])
```

```
arr2[arr2>5]
```

```
array([ 6,  7,  8,  9, 10])
```

Pandas

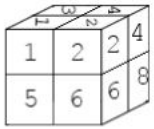
На самом примитивном уровне объекты библиотеки Pandas можно считать расширенной версией структурированных массивов библиотеки NumPy, в которых строки и столбцы идентифицируются метками, а не простыми числовыми индексами. Библиотека Pandas предоставляет множество полезных утилит, методов и функциональности в дополнение к базовым структурам данных, но все последующее изложение потребует понимания этих базовых структур. Позвольте познакомить вас с тремя фундаментальными структурами данных библиотеки Pandas: классами `Series`, `DataFrame` и `Index`.

Начнем наш сеанс программирования с обычных импортов библиотек NumPy и Pandas:

```
In[1]: import numpy as np
       import pandas as pd
```

NumPy

Скаляр	Вектор	Матрица	Тензор
1	$\begin{bmatrix}1\\2\end{bmatrix}$	$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$	$\begin{bmatrix}\begin{bmatrix}1 & 2\end{bmatrix} & \begin{bmatrix}3 & 4\end{bmatrix}\\\begin{bmatrix}5 & 6\end{bmatrix} & \begin{bmatrix}7 & 8\end{bmatrix}\end{bmatrix}$




```
tensor = np.random.randint(-5, 5, (3, 2, 2))    # Во вторых скобках заданы размеры тензора в высоту, ширину, глубину (оси X,Y,Z)
print(tensor)                                   # Вывод трехмерного тензора
```

```
[[[-2 -5]
  [ 1 -2]]

 [[-3  2]
  [ 4  0]]

 [[-1  4]
  [-3  0]]]
```

print('Количество осей: ',tensor.ndim)
print('Форма массива: ',tensor.shape)
print('Количество значений: ',tensor.size)
Количество осей: 3
Форма массива: (3, 2, 2)
Количество значений: 12

NumPy

```
array_1D = np.random.randint(-5, 5, 4) # Задается одномерный массив
print(array_1D)
[-1 -4 -4 -4]

array_2D = random_matrix = np.random.randint(-, 5, (5, 2))   # Задается двумерный масс
ив
print(array_2D)

[[-4 -2]
[ 0 -2]
[-4 3]
[ 3 -2]
 [-3 -4]]
```

NumPy

1. Создайте три вектора, из пяти элементов каждый. Один вектор должен быть заполнен нулями, второй - единицами, а третий - цифрами 3.

2. Создайте список (не массив), состоящий из 10 имен и назовите его **list_names**
Создайте массив из созданного ранее списка с помощью метода np.array() и назовите его **array_names** Отобразите тип переменных **list_names** и **array_names**

3. Создайте случайный массив из десяти элементов (можно воспользоваться любым из доступных методов из модуля np.random).
Выведите на экран тип элементов созданного массива с помощью dtype

4. Создайте массив из ста целочисленных значений (можно воспользоваться любым способом).
Выведите на экран размерность массива.
Выведите на экран построчно 1-й, 32-й и предпоследний элементы массива.

5. Создайте массив из 15 единиц.
Замените каждый третий элемент массива на 2.
Выведите финальный результат на экран.

NumPy

6. Создайте три вектора, из пяти элементов каждый. Один вектор должен быть заполнен нулями, второй - единицами, а третий - цифрами 3.

7. Создайте двумерный массив 6 на 6 (любым способом).
Создайте переменную my_mean и запишите в нее среднее значение созданного массива.

8. Создайте массив 7 на 7, состоящий из нулей.
Любым способом добавьте в этот массив крест из едениц.
то есть необходимо получить следующий массив:

```
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
1 1 1 1 1 1 1
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
```

Постарайтесь найти оптимальное решение)

Pandas

- Series
- DataFrames
- Missing Data
- GroupBy
- Merging,Joining,and Concatenating
- Operations
- Data Input and Output

Pandas

Объект Series библиотеки Pandas

Объект `Series` библиотеки Pandas — одномерный массив индексированных данных. Его можно создать из списка или массива следующим образом:

```
In[2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
       data

Out[2]: 0    0.25
        1    0.50
        2    0.75
        3    1.00
        dtype: float64
```

Как мы видели из предыдущего результата, объект `Series` служит адаптером как для последовательности значений, так и последовательности индексов, к которым можно получить доступ посредством атрибутов `values` и `index`. Атрибут `values` представляет собой уже знакомый нам массив NumPy:

Pandas

```
In[3]: data.values

Out[3]: array([ 0.25,  0.5 ,  0.75,  1.  ])
```

index — массивоподобный объект типа pd.Index, который мы рассмотрим подробнее далее:

```
In[4]: data.index

Out[4]: RangeIndex(start=0, stop=4, step=1)[1]
```

Pandas

Явное описание индекса расширяет возможности объекта `Series`. Такой индекс не должен быть целым числом, а может состоять из значений любого нужного типа. Например, при желании мы можем использовать в качестве индекса строковые значения:

```
In[7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=['a', 'b', 'c', 'd'])
       data

Out[7]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        dtype: float64
```

Спасибо за внимание!