ریحانه شاهرخیان ۹۹۵۲۱۳۶۱

تمرین اول درس هوش محاسباتی

دکتر مزینی

سوال ۱:

از آنجایی که صورت سوال از ما خواسته مسئله با کتابخانه numpy حل شود، باید از ۴ توابع آماده‌ی آن برای این کار استفاده کرد که به صورت مقابل است:

```
greater_result = np.greater(array1, array2)
greater_equal_result = np.greater_equal(array1, array2)
less_result = np.less(array1, array2)
less_equal_result = np.less_equal(array1, array2)
```

همانطور که از اسم توابع پیداست، تابع اول مقایسه‌ی عنصر به عنصر بزرگتری، تابع دوم بزرگتر مساوی، تابع سوم کوچکتری و تابع آخر کوچکتر مساوی بودن را بررسی می‌کنند.

خروجی این توابع آرایه‌ای به سایز آرایه‌های ورودی است که در صورت برقراری رابطه‌ی مدنظر بین دو عنصر آرایه‌های ورودی، عنصر متناظر True و در غیر این صورت False است.

```
array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[1, 2], [2, 3]])

greater, greater_equal, less, less_equal = element_wise_comparison(array1, array2)

print("Greater than:")
print(greater)
print("\nGreater than or equal to:")
print(greater_equal)
print("\nLess than:")
print(less)
print("\nLess than or equal to:")
print(less_equal)
```

```
Greater than:
[[False False]
 [ True  True]]

Greater than or equal to:
[[ True  True]
 [ True  True]]

Less than:
[[False False]
 [False False]]

Less than or equal to:
[[ True  True]
 [False False]]
```

سوال ۲:

از آنجایی که صورت سوال از ما خواسته مسئله با کتابخانه numpy حل شود، باید از ۲ تابع multiply و dot استفاده کرد که به صورت مقابل است:

```
if method == "element-wise":
    result = np.multiply(array1, array2)
elif method == "matrix-multiply":
    result = np.dot(array1, array2)
```

تابع multiply در واقع تابع آماده‌ی numpy برای ضرب عنصر به عنصر و تابع dot برای ضرب ماتریسی است.

```
array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[2, 0], [1, 2]])

# Perform element-wise multiplication
element_wise_result = array_multiply(array1, array2, method="element-wise")
print("Element-wise multiplication:")
print(element_wise_result)

# Perform matrix multiplication
matrix_multiply_result = array_multiply(array1, array2, method="matrix-multiply")
print("\nMatrix multiplication:")
print(matrix_multiply_result)
```

```
Element-wise multiplication:
[[2 0]
 [3 8]]

Matrix multiplication:
[[ 4  4]
 [10  8]]
```

سوال ۳:

وقتی بخواهیم دو آرایه با سایزهای n*n و n*1 را باهم جمع کنیم، می‌توانیم از broadcasting استفاده کنیم که آرایه‌ی دوم یا به صورت افقی و یا عمودی به آرایه‌ی اول اضافه می‌شود. برای column-wise addition باید بعد اول هر دو آرایه یکسان باشند و برای row-wise addition باید بعد اول آرایه‌ی دوم با بعد دوم آرایه‌ی اول یکسان باشند، در غیر این صورت انجام پذیر نیستند. نحوه‌ی پیاده سازی آن به صورت مقابل است:

```
if method == "column-wise":
    if p.shape[0] != q.shape[0]:
        raise ValueError("Shapes are incompatible for the column-wise method")
    result = p + q.reshape(-1, 1)

elif method == "row-wise":
    if p.shape[1] != q.shape[0]:
        raise ValueError("Shapes are incompatible for the row-wise method")
    result = p + q

else:
    raise ValueError("Invalid method is provided")

return result
```

نکته‌ی که وجود دارد برای column-wise باید ابتدا آرایه‌ی دوم را به نوعی rotate کنیم تا با انجام عملیات + بین دو آرایه به صورت درستی انجام شود(در واقع تمامی عناصر ردیف اول آرایه‌ی اول با عنصر اول آرایه‌ی دوم جمع می‌شود و ...) ولی برای row-wise با انجام عملیات + بین دو آرایه جمع انجام می‌شود(تمامی عناصر ستون اول آرایه‌ی اول با عنصر اول آرایه‌ی دوم جمع می‌شود و ...)

واضح است که اگر شرایط ابعاد برقرار نباشد یا متد درستی(row-wise یا column-wise) صدا زده نشده باشد، ارور متناسب raise می‌شود در غیر این صورت جمع انجام می‌شود و آرایه‌ی خروجی، آرایه‌ای هم‌سایز با آرایه‌ی اول است.

```
# Example usage with different-shaped arrays
p = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
q = np.array([10, 20, 30])

# Add q row-wise to p
row_wise_result = broadcast_add(p, q, method="row-wise")
print("Row-wise addition:")
print(row_wise_result)

# Add q column-wise to p
column_wise_result = broadcast_add(p, q, method="column-wise")
print("Column-wise addition:")
print(column_wise_result)
```

```
Row-wise addition:
[[11 22 33]
 [14 25 36]
 [17 28 39]]
Column-wise addition:
[[11 12 13]
 [24 25 26]
 [37 38 39]]
```

سوال ۴:

همانطور که سوال از ما خواسته ابتدا یک ماتریس تصادفی به ابعاد 4x4 با مقادیر تصادفی از بازه 1 تا 10 با استفاده از تابع np.random.uniform ایجاد می‌کنیم. سپس برای نرمال‌سازی ابتدا مقادیر حداقل و حداکثر در ماتریس ایجاد شده را محاسبه می‌کنیم و تمام عناصر ماتریس از مقدار حداقل کم می‌کنیم و در آخر آن‌ها تقسیم بر اختلاف مقدار حداکثر و مقدار حداقل می‌کنیم تا تمام مقادیر ماتریس در بازه 0 تا 1 قرار گیرند. کد این مراحل به صورت مقابل است:

```python
# Initialize the random matrix
x = np.random.uniform(1, 10, size=(4, 4))

print("Original Array:")
print(x)

# Do the normalization
#x = (x - 1) / 9
min = x.min()
max = x.max()
x = (x - min) / (max - min)

np.set_printoptions(precision=5)

print("After normalization:")
print(x)
```

```
Original Array:
[[4.60959637 1.54897217 2.10200603 4.57556142]
 [6.19212486 1.78458978 2.78370932 6.36303299]
 [4.29950084 8.46440721 3.04701076 9.27062372]
 [9.20392132 7.58621746 2.81012799 3.29294848]]
After normalization:
[[0.39637 0.      0.07162 0.39196]
 [0.60132 0.03051 0.15991 0.62345]
 [0.35621 0.89559 0.194   1.     ]
 [0.99136 0.78186 0.16333 0.22586]]
```

سوال ۵:

۵-۱: در این بخش با توجه به تساوی `(x - y) / y = (x / y) - 1` می‌توانیم قیمت پایانی روز را بر قیمت روز قبلی تقسیم کنیم و در انتها یک واحد از جواب کم کنیم. برای دسترسی به قیمت روز قبل از تابع shift استفاده می‌کنیم. از آنجایی که داده‌ی روز اول، روز قبلی ندارد و جواب تقسیم تعریف نشده می‌شود، عنصر اول daily_output را حذف می‌کنیم.

```
daily_output = (data['Closing Price'] / data['Closing Price'].shift(1)) - 1
del daily_output[0]
```

```
daily outputs:
1     -0.002145
2      0.013911
3      0.010884
4     -0.024109
5      0.013518
         ...
359    0.006720
360   -0.015886
361    0.008170
362    0.006454
363   -0.011010
Name: Closing Price, Length: 363, dtype: float64
```

۲-۵: با استفاده از تابع mean کتابخانه numpy میانگین بازده روزانه که در بخش قبل محاسبه کردیم را به دست می‌آوریم.

```
average_daily_output = np.mean(daily_output)
```

```
average daily outputs:
0.000555
```

۳-۵: برای محاسبه‌ی انحراف معیار از تابع std کتابخانه‌ی numpy استفاده می‌کنیم.

```
std_deviation_daily_output = np.std(daily_output)
```

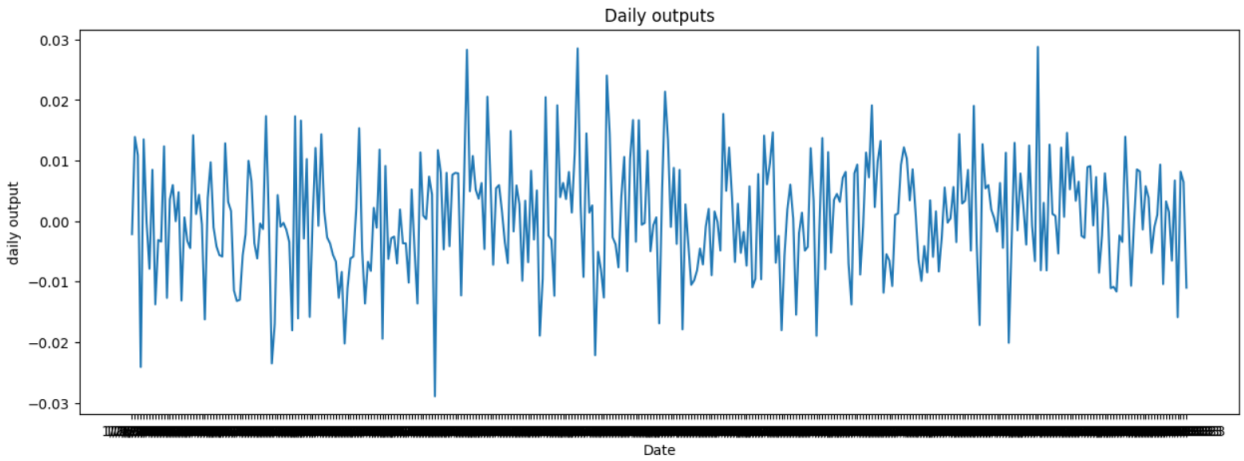
```
std deviation of daily outputs:
0.009443
```

۴-۵: با استفاده از matplotlib قیمت‌های پایانی هر روز را بر روی نمودار نشان می‌دهیم.

```
plt.figure(figsize=(15, 5))
plt.plot(data['Date'], data['Closing Price'])
plt.xlabel('Date')
plt.ylabel('Closing Price')
plt.title('Daily Closing Prices')
plt.show()
```

۵-۵: برای نشان دادن میزان بازده روزانه روی نمودار مانند بخش قبل عمل می‌کنیم و از آنجایی که برای روز اول بازده محاسبه نشده است، ایندکس تاریخ هم از ۱ شروع می‌کنیم.

```
plt.figure(figsize=(15, 5))
plt.plot(data['Date'][1:], daily_output)
plt.xlabel('Date')
plt.ylabel('daily output')
plt.title('Daily outputs')
plt.show()
```



۶-۵: ابتدا مقدار بیشترین و کمترین بازده روزانه را با استفاده از max و min کتابخانه numpy محاسبه می‌کنیم. سپس ایندکس مربوط به آن را با استفاده از argmax و argmin پیدا می‌کنیم و از آنجایی که روز اول از daily_output حذف شده، یک واحد به ایندکس به دست آمده اضافه می‌کنیم و در انتها تاریخ مدنظر را بدست می‌آوریم.

```
max_output = np.max(daily_output)
min_output = np.min(daily_output)
max_output_date = data['Date'][np.argmax(daily_output) + 1]
min_output_date = data['Date'][np.argmin(daily_output) + 1]
```

```
max output: Date = 11/9/2023, output = 0.028786
min output: Date = 4/16/2023, output = -0.028964
```

۷-۵: بیشترین و کمترین قیمت پایانی روز را با استفاده از max و min محاسبه می‌کنیم. برای بدست آوردن تاریخ آن‌ها هم ابتدا از idmax و idmin استفاده می‌کنیم تا ایندکس آن‌ها را بدست آوریم و سپس با توجه به ایندکس تاریخ را خروجی می‌دهیم.

```
max_price = data['Closing Price'].max()
min_price = data['Closing Price'].min()
max_price_date = data['Date'][data['Closing Price'].idxmax()]
min_price_date = data['Date'][data['Closing Price'].idxmin()]
```

```
max price: Date = 11/29/2023, price = 124.618011
min price: Date = 4/16/2023, price = 82.968210
```

سوال ۶:

در روش اول (پیاده سازی با for) باید با دو for تو در تو عملیات ضرب داخلی ماتریسی را انجام دهیم و در روش دوم تنها با استفاده از تابع dot کتابخانه numpy این کار انجام می‌شود.

```
outputs = np.zeros((X.shape[0], 1))

for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        outputs[i] += X[i, j] * w[j]
return outputs
```

```
outputs = np.dot(X, w)
return outputs
```

```
Time spent on calculating the outputs using for loops:
2.824991226196289
Time spent on calculating the outputs using vectorization:
0.011034011840820312
```

همانطور که مشخص است مدت زمان اجرا با for تقریبا 256 برابر روش دوم است. بنابراین سرعت vectorization به طرز چشمگیری بیشتر از for است.

سوال ۷:

از آنجایی که صورت سوال از ما خواسته مسئله با کتابخانه numpy حل شود، می‌توان با استفاده از تابع where شرط بزرگتر بودن عناصر آرایه از threshold را قرار دهیم که در صورت برقراری مقدار آرایه‌ی modified_arr برابر 1 و در غیر این صورت 0 شود.

```
modified_arr = np.where(array > threshold, 1, 0)
```

واضح است که سایز آرایه‌ی خروجی هم اندازه‌ی آرایه‌های ورودی است.

```
input_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
threshold_value = 5
result_array = replace_elements_above_threshold(input_array, threshold_value)
print(result_array)
```

```
[[0 0 0]
 [0 0 1]
 [1 1 1]]
```

سوال ۸:

ابتدا در تابع init که constructor کلاس Matrix است، مقدار دهی اولیه را انجام می‌دهیم.

```
self.matrix = matrix
```

۸-۱: در تابع is_equal تنها با استفاده از == تساوی دو آرایه را چک می‌کنیم و واضح است که خروجی == boolean است.

```
return self.matrix == second_matrix.matrix
```

```
matrix1 = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

matrix2 = Matrix([[0, 0, 0], [4, 5, 6], [7, 8, 9]])

# test equality of matrices here and show the result #
print(matrix1.is_equal(matrix2))
```

```
False
```

۸-۲: با توجه به اینکه صورت سوال استفاده از numpy را منع کرده است، برای مقایسه عنصر به عنصر این دو ماتریس از دو for تو در تو استفاده می‌کنیم. برای ذخیره کردن جواب، نتیجه‌ی هر ردیف را در لیستی append می‌کنیم و سپس هر ردیف ذخیره شده را به لیست result اضافه می‌کنیم.

```
result = []
for i in range(len(self.matrix)):
    row = []
    for j in range(len(self.matrix[i])):
        row.append(self.matrix[i][j] > second_matrix.matrix[i][j])
    result.append(row)

return Matrix(result)
```

```
matrix3 = Matrix([[0, 0, 0], [10, 20, 30], [-1, 8, 10]])

# test proportion of matrices here and show the result #
print(matrix3.is_higher_elementwise(matrix2).matrix)
```

```
[[False, False, False], [True, True, True], [False, False, True]]
```

۸-۳: برای بررسی زیر مجموعه بودن ماتریس، باید تمامی عناصر ماتریس اول دقیقا به همان ترتیب و ابعاد در ماتریس دوم باشند. برای این کار در واقع ابتدا با استفاده از دو for تو در تو، روی تمامی subset های مارتیس دوم که هم سایز با ماتریس اول هستند iterate می‌کنیم. سپس چک می‌کنیم اگر تمامی عناصر ماتریس اول با همان ترتیب منطبق بر یکی از آن subset ها شد خروجی true را return می‌کنیم.

```
if len(self.matrix) > len(second_matrix.matrix):
    return False

for i in range(len(second_matrix.matrix) - len(self.matrix) + 1):
    for j in range(len(second_matrix.matrix[0]) - len(self.matrix[0]) + 1):
        if all(self.matrix[row][col] == second_matrix.matrix[row + i][col + j] for row in range(len(self.matrix)) for col in range(len(self.matrix[0]))):
            return True

return False
```

```
matrix4 = Matrix([[5, 6], [8, 9]])
matrix5 = Matrix([[1, 2], [4, 5]])
matrix6 = Matrix([[1, 2], [3, 4]])
matrixx = Matrix([[1, 2], [5, 6]])

# test subset of matrices here and show the result #
print(matrix5.is_subset(matrix1))
print(matrix4.is_subset(matrix2))
print(matrix6.is_subset(matrix1))
print(matrixx.is_subset(matrix1))
```

```
True
True
False
False
```

۸-۴: برای حساب کردن ضرب داخلی دو ماتریس از سه for تو در تو استفاده می‌کنیم. در واقع برای محاسبه‌ی هر عنصر ماتریس result نیاز به دو for تو در تو داریم تا ضرب را انجام دهد و مقادیر ضرب شده را باهم جمع کند. برای ذخیره کردن جواب، نتیجه‌ی عناصر هر ردیف را در لیستی append می‌کنیم و سپس هر ردیف ذخیره شده را به لیست result اضافه می‌کنیم.

```
result = []
for i in range(len(self.matrix)):
    row = []
    for j in range(len(second_matrix.matrix[0])):
        sum_product = 0
        for k in range(len(self.matrix[0])):
            sum_product += self.matrix[i][k] * second_matrix.matrix[k][j]
        row.append(sum_product)
    result.append(row)
return Matrix(result)
```

```
1 matrix7 = Matrix([[3, 1], [2, 4], [-1, 5]])
2 matrix8 = Matrix([[3, 1], [2, 4]])
3
4 # test product of matrices here and show the result #
5 print(matrix7.dot_product(matrix8).matrix)
```

```
[[11, 7], [14, 18], [7, 19]]
```