Tarea1

July 18, 2017

1 Tarea 1: alumno Jorge III Altamirano Astorga

1.1 Introducción

Presento la siguiente tarea para la materia de Propedéutico de la Maestría en Ciencia de Datos del Instituto Tecnológico Autónomo de México. Le agradezco al Maestro Mauricio García Tec por su comprensión y por compartir conocimiento nuevo para mí.

1.1.1 Importación e inicialización de arrays

import numpy as np
x = [1, 2, 3] y = [4, 5, 6] x + y

Matrices como arrays Presentación de matrices

```
Matríz con numpy:
[[1,2,3],[4,5,6]]
```

Corresponde a la matriz

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

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

        X = np.array([[1, 2, 3], [4, 5, 6]])
        X

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

Operaciones con arrays

$$X + 2B = \begin{bmatrix} 3 & 6 & 9 \\ 12 & 15 & 18 \end{bmatrix}$$

```
In [2]: X + 2*X

Out[2]: array([[ 3,  6,  9],
               [12, 15, 18]])
```

Multiplicación de matrices con numpy.matmul()

$$X^t \times X = \begin{bmatrix} 17 & 22 & 27 \\ 22 & 29 & 36 \\ 27 & 36 & 45 \end{bmatrix}$$

```
In [3]: np.matmul(X.transpose(), X) #X^t * X

Out[3]: array([[17, 22, 27],
               [22, 29, 36],
               [27, 36, 45]])
```

Obteniendo datos específicos

```
In [4]: X[1,1]

Out[4]: 5
```

Selección y multi selección de elementos...

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
In [5]: X[1, :] #1. Fila entera

Out[5]: array([4, 5, 6])

In [6]: X[:, 1] #2. Columna entera

Out[6]: array([2, 5])

In [7]: X[0:2, 0:2] #3. Slice de n:m, n,n+1,...,m-1

Out[7]: array([[1, 2],
               [4, 5]])

In [8]: X.shape #Dimensión de arrays

Out[8]: (2, 3)
```

1.1.2 Vectores

Numpy es *vector-ready*

```
In [9]: vec = np.array([1, 2, 3])
        print(vec)

[1 2 3]
```

1.2 Creando clases en Python

Creando clases en Python from scratch

```
In [10]: class Array:
             "Clase mínima para Álgebra Lineal"
             def __init__(self, list_of_rows):
                 "Constructor"
                 self.data = list_of_rows
                 self.shape = (len(list_of_rows), len(list_of_rows[0]))

In [11]: A = Array([[1, 2, 3], [4, 5, 6]])
         A.__dict__ # la propiedad oculta *dict* muestra las propiedades internas de la clase

Out[11]: {'data': [[1, 2, 3], [4, 5, 6]], 'shape': (2, 3)}

In [12]: A.data #accediendo a la propiedad *data* de la clase

Out[12]: [[1, 2, 3], [4, 5, 6]]

In [13]: A.shape #accediendo a la propiedad *shape* de la clase

Out[13]: (2, 3)
```

... implementando métodos que faciliten la utilización de dicha clase para nuestros fines de Álgegra Lineal

Métodos especiales de clase

1. Método de *Pretty Print* He aquí el método para imprimir bonito, primero mostramos como se imprime por default sin existir el método

```
In [14]: Array([[1,2,3], [4,5,6]])

Out[14]: <__main__.Array at 0x7f824734d860>

In [15]: print(Array([[1,2,3], [4,5,6]]))

<__main__.Array object at 0x7f824734db38>


In [16]: # Prueba de clase
         class JorgeClass:
             def __init__(self):
                 pass # para no hacer nada
             def say_hi(self):
                 print("¡Método de impresión simple!")
             def __repr__(self):
                 return "Representación sin imprimir nada"
             def __str__(self):
                 return "Método explícito para *print* como objeto"
```

```
In [17]: x = JorgeClass()
         x.say_hi()
```

```
¡Método de impresión simple!
```

```
In [18]: x
```

```
Out[18]: Representación sin imprimir nada
```

```
In [19]: print(x)
```

```
Método explícito para *print* como objeto
```

1.2.1 EJERCICIO 1

Validador

```
In [20]: import re

         class Array:
             "Clase mínima para Álgebra Lineal"
             data = list()
             def __init__(self, list_of_rows):
                 "Constructor"
                 self.data = list_of_rows
                 nrow = len(list_of_rows)
                 #  ___caso vector: redimensionar correctamente
                 if not isinstance(list_of_rows[0], list):
                     nrow = 1
                     self.data = [[x] for x in list_of_rows]
                 # ahora las columnas deben estar bien aunque sea un vector
                 ncol = len(self.data[0])
                 self.shape = (nrow, ncol)
                 # validar tamano correcto de filas
                 if any([len(r) != ncol for r in self.data]):
                     raise Exception("Ejercicio 1: Las filas deben ser del mismo tamano. ¡Valida
             def say_hi(self):
                 print("Ejercicio 1: say_hi() method")
             def __repr__(self):
                 retval = "["
                 for list in self.data:
                     retval += "["
                     for x in list:
                         retval += str(x) + ", "
                     retval += "], "
                 retval = retval + "]"
                 return retval.replace(", ]", "]")
```

```
    def __str__(self):
        return self.__repr__()
        #return "Ejercicio 1: print"
    def __getitem__(self, index):
        return(self.data[index[0]][index[1]])
    def __setitem__(self, index, newval):
        self.data[index[0]][index[1]] = newval
        return
```

Prueba de validador

```
In [21]: Array([[1,2,3],[4,5]])
```

```
---------------------------------------------------------------------------

Exception                                 Traceback (most recent call last)

<ipython-input-21-b33be5094d7d> in <module>()
----> 1 Array([[1,2,3],[4,5]])


<ipython-input-20-e362da19f62a> in __init__(self, list_of_rows)
     17         # validar tamano correcto de filas
     18         if any([len(r) != ncol for r in self.data]):
---> 19             raise Exception("Ejercicio 1: Las filas deben ser del mismo tamano. ąVal
     20     def say_hi(self):
     21         print("Ejercicio 1: say_hi() method")


Exception: Ejercicio 1: Las filas deben ser del mismo tamano. ąValidador funcionando!
```

Prueba con vectores

```
In [22]: vec = Array([1,2,3])
         vec.data
```

```
Out[22]: [[1], [2], [3]]
```

Index & Item Assignment

```
In [24]: X = Array([[1,2],[3,4]])
         X[1,0]
```

```
Out[24]: 3
```

1.2.2 EJERCICIO 2

Método setter

```
In [25]: X[0,0] = 0
         X[0,0]

Out[25]: 0

In [26]: X

Out[26]: [[0, 2], [3, 4]]
```

Inicialización de una matriz en ceros con numpy

```
In [27]: np.zeros((3,6))

Out[27]: array([[ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

1.2.3 EJERCICIO 3

Función para crear arrays "vacíos" (en ceros)

```
In [28]: import re
         import numpy as np

         class zeros:
             "Clase para llenar una matriz en Ceros"
             data = list()
             shape = None
             def __init__(self, shape):
                 "Constructor"
                 self.data = list()
                 print(shape)
                 for i in range(0, shape[1]):
                     self.data.append(list())
                     for j in range(0, shape[0]):
                         self.data[i].append(list())
                         self.data[i][j] = 0.
                         #self.data[i].append(list())
                     #    self.data[i][j] = 0
                 self.shape = shape
                 #self.data = np.zeros(shape)
             def eye(self):
                 if self.shape[0] != self.shape[1]:
                     raise Exception("Ejercicio 3: las columnas y las filas deben ser de igual n
                 for i in range(0,self.shape[1]):
                     for j in range(0, self.shape[0]):
```

```
                    if(i == j):
                        self.data[i][j] = 1.0
            def __repr__(self):
                retval = "[\n"
                for list in self.data:
                    retval += "["
                    for x in list:
                        retval += str(x) + ", "
                    retval += "], \n"
                retval = retval + "]"
                return retval.replace(", ]", "]")
            def __str__(self):
                return self.__repr__()
                #return "Ejercicio 1: print"
            def __getitem__(self, index):
                return(self.data[index[0]][index[1]])
            def __setitem__(self, index, newval):
                self.data[index[0]][index[1]] = newval
                return
```

```
In [29]: X = zeros([3,4])
         X
```

```
[3, 4]
```

```
Out[29]: [
         [0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0],
         ]
```

```
In [30]: X.eye()
         X
```

```
        ---------------------------------------------------------------------------

        Exception                                 Traceback (most recent call last)

        <ipython-input-30-7a2759e8335c> in <module>()
    ----> 1 X.eye()
          2 X


        <ipython-input-28-003304790700> in eye(self)
         21     def eye(self):
         22         if self.shape[0] != self.shape[1]:
```

```
---> 23             raise Exception("Ejercicio 3: las columnas y las filas deben ser de igua
     24         for i in range(0,self.shape[1]):
     25             for j in range(0, self.shape[0]):


Exception: Ejercicio 3: las columnas y las filas deben ser de igual número n = m
```

```
In [31]: X = zeros((4, 4))
         X.eye()
         X
```

```
(4, 4)
```

```
Out[31]: [
         [1.0, 0.0, 0.0, 0.0],
         [0.0, 1.0, 0.0, 0.0],
         [0.0, 0.0, 1.0, 0.0],
         [0.0, 0.0, 0.0, 1.0],
         ]
```

Transposición

```
In [32]: np.array([[1,2,3],[4,5,6],[7,8,9]]).transpose()
```

```
Out[32]: array([[1, 4, 7],
                [2, 5, 8],
                [3, 6, 9]])
```

1.2.4 EJERCICIO 4

Función para transponer arrays

```
In [33]: import re

         class transpose:
             "Clase para llenar una matriz en Ceros"
             data = list()
             shape = None
             def __init__(self, list_of_rows):
                 "Constructor"
                 # obtener dimensiones
                 self.data = list_of_rows
                 nrow = len(list_of_rows)
                 #  ___caso vector: redimensionar correctamente
                 if not isinstance(list_of_rows[0], list):
                     nrow = 1
                     self.data = [[x] for x in list_of_rows]
```

```
        # ahora las columnas deben estar bien aunque sea un vector
        ncol = len(self.data[0])
        self.shape = (nrow, ncol)
        # validar tamano correcto de filas
        if any([len(r) != ncol for r in self.data]):
            raise Exception("Las filas deben ser del mismo tamano")
    def transpose(self):
        transposed = list()
        for i in range(0, len(self.data[1])):
            transposed.append(list())
            for j in range(0, len(self.data[0])):
                transposed[i].append(list())
                transposed[i][j] = self.data[j][i]
        print(transposed)
    def eye(self):
        if self.shape[0] != self.shape[1]:
            raise Exception("Ejercicio 3: las columnas y las filas deben ser de igual n
        for i in range(0,self.shape[1]):
            for j in range(0, self.shape[0]):
                if(i == j):
                    self.data[i][j] = 1.0
    def __repr__(self):
        retval = "[\n"
        for list in self.data:
            retval += "["
            for x in list:
                retval += str(x) + ", "
            retval += "], \n"
        retval = retval + "]"
        return retval.replace(", ]", "]")
    def __str__(self):
        return self.__repr__()
        #return "Ejercicio 1: print"
    def __getitem__(self, index):
        return(self.data[index[0]][index[1]])
    def __setitem__(self, index, newval):
        self.data[index[0]][index[1]] = newval
        return
    def __add__(self, other):
        "Hora de sumar"
        if isinstance(other, transpose):
            if self.shape != other.shape:
                raise Exception("Las dimensiones son distintas!")
            rows, cols = self.shape
            suma = transpose([[0. for c in range(cols)] for r in range(rows)])
            for r in range(rows):
                for c in range(cols):
                    suma.data[r][c] = self.data[r][c] + other.data[r][c]
```

```
                return suma
            elif isinstance(2, (int, float, complex)): # en caso de que el lado derecho sea
                rows, cols = self.shape
                newArray = Array([[0. for c in range(cols)] for r in range(rows)])
                for r in range(rows):
                    for c in range(cols):
                        newArray.data[r][c] = self.data[r][c] + other
                return newArray
            else:
                return NotImplemented # es un tipo de error particular usado en estos metod
        def __mul__(self, other):
            "Método de multiplicación"
            if isinstance(other, transpose):
                if self.shape[1] != other.shape[0]:
                    raise Exception("Las dimensiones son distintas!")
                rows, cols = self.shape
                suma = transpose([[0. for c in range(cols)] for r in range(rows)])
                for r in range(rows):
                    for c in range(cols):
                        suma.data[r][c] = self.data[r][c] + other.data[r][c]
                return suma
            elif isinstance(2, (int, float, complex)): # en caso de que el lado derecho sea
                rows, cols = self.shape
                newArray = Array([[0. for c in range(cols)] for r in range(rows)])
                for r in range(rows):
                    for c in range(cols):
                        newArray.data[r][c] = self.data[r][c] + other
                return newArray
            else:
                return NotImplemented # es un tipo de error particular usado en estos metod
```

```
In [34]: X = transpose([[1,2,3],[4,5,6],[7,8,9]])
         X.transpose()
```

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

Suma

```
In [35]: np.array([[1,2,3],[4,5,6],[7,8,9]]) + np.array([[10,11,12],[13,14,15],[16,17,18]])
```

```
Out[35]: array([[11, 13, 15],
                [17, 19, 21],
                [23, 25, 27]])
```

```
In [36]: Y = transpose([[10,11,12],[13,14,15],[16,17,18]])
         X + Y
```

```
Out[36]: [
         [11, 13, 15],
         [17, 19, 21],
         [23, 25, 27],
         ]

In [37]: Z = X + 10
         Z

Out[37]: [[11, 12, 13], [14, 15, 16], [17, 18, 19]]

In [38]: Z.data

Out[38]: [[11, 12, 13], [14, 15, 16], [17, 18, 19]]

In [39]: transpose([[1,1],[1,1]]) + transpose([[2,2],[2,2]])

Out[39]: [
         [3, 3],
         [3, 3],
         ]

In [40]: import re

         class transpose:
             "Clase para llenar una matriz en Ceros"
             data = list()
             shape = None
             def __init__(self, list_of_rows):
                 "Constructor"
                 # obtener dimensiones
                 self.data = list_of_rows
                 nrow = len(list_of_rows)
                 #  ___caso vector: redimensionar correctamente
                 if not isinstance(list_of_rows[0], list):
                     nrow = 1
                     self.data = [[x] for x in list_of_rows]
                 # ahora las columnas deben estar bien aunque sea un vector
                 ncol = len(self.data[0])
                 self.shape = (nrow, ncol)
                 # validar tamano correcto de filas
                 if any([len(r) != ncol for r in self.data]):
                     raise Exception("Las filas deben ser del mismo tamano")
             def transpose(self):
                 transposed = list()
                 for i in range(0, len(self.data[1])):
                     transposed.append(list())
                     for j in range(0, len(self.data[0])):
                         transposed[i].append(list())
```

```
            transposed[i][j] = self.data[j][i]
    print(transposed)
def eye(self):
    if self.shape[0] != self.shape[1]:
        raise Exception("Ejercicio 3: las columnas y las filas deben ser de igual n
    for i in range(0,self.shape[1]):
        for j in range(0, self.shape[0]):
            if(i == j):
                self.data[i][j] = 1.0
def __repr__(self):
    retval = "[\n"
    for list in self.data:
        retval += "["
        for x in list:
            retval += str(x) + ", "
        retval += "], \n"
    retval = retval + "]"
    return retval.replace(", ]", "]")
def __str__(self):
    return self.__repr__()
    #return "Ejercicio 1: print"
def __getitem__(self, index):
    return(self.data[index[0]][index[1]])
def __setitem__(self, index, newval):
    self.data[index[0]][index[1]] = newval
    return
def __add__(self, other):
    "Hora de sumar"
    if isinstance(other, transpose):
        if self.shape != other.shape:
            raise Exception("Las dimensiones son distintas!")
        rows, cols = self.shape
        suma = transpose([[0. for c in range(cols)] for r in range(rows)])
        for r in range(rows):
            for c in range(cols):
                suma.data[r][c] = self.data[r][c] + other.data[r][c]
        return suma
    elif isinstance(2, (int, float, complex)): # en caso de que el lado derecho sea
        rows, cols = self.shape
        newArray = Array([[0. for c in range(cols)] for r in range(rows)])
        for r in range(rows):
            for c in range(cols):
                newArray.data[r][c] = self.data[r][c] + other
        return newArray
    else:
        return NotImplemented # es un tipo de error particular usado en estos metod
def __rmul__(self, other):
    "Multiplicación escalar"
```

```
                if isinstance(2, (int, float, complex)): # en caso de que el lado derecho sea s
                    rows, cols = self.shape
                    newArray = transpose([[0. for c in range(cols)] for r in range(rows)])
                    for r in range(rows):
                        for c in range(cols):
                            newArray.data[r][c] = self.data[r][c] * other
                    return newArray
                else:
                    return NotImplemented # es un tipo de error particular usado en estos metod
            def __mul__(self, other):
                "Multiplicación vectorial"
                if isinstance(other, transpose):
                    if self.shape[1] != other.shape[0] or self.shape[0] != other.shape[1]:
                        raise Exception("Las dimensiones son distintas!")
                    rows, cols = self.shape[0], other.shape[1]
                    rowsL, colsL = other.shape[0], self.shape[1]
                    retval = transpose([[0. for c in range(cols)] for r in range(rows)])
                    for r in range(rows):
                        for c in range(cols):
                            print("R(" + str(r) + ", " + str(c) + ") = ", end=" ")
                            for i in range(colsL):
                                #for j in range(rowsL):
                                print("A("+str(c) + "," + str(i) + ") * B(" + str(r) + "," + st
                            print("")
                                    #retval.data[r][c] += self.data[i][j] * other.data[i][j]
                            #print(str(r) + ", " + str(c) + " = ")
                    return retval
                else:
                    return NotImplemented # es un tipo de error particular usado en estos metod

        transpose([[1,1,1],[1,1,1]])  * transpose([[2,2],[2,2],[2,2]])
```

```
R(0, 0) =  A(0,0) * B(0,0) + A(0,1) * B(0,0) + A(0,2) * B(0,0) +
R(0, 1) =  A(1,0) * B(0,1) + A(1,1) * B(0,1) + A(1,2) * B(0,1) +
R(1, 0) =  A(0,0) * B(1,0) + A(0,1) * B(1,0) + A(0,2) * B(1,0) +
R(1, 1) =  A(1,0) * B(1,1) + A(1,1) * B(1,1) + A(1,2) * B(1,1) +
```

```
Out[40]: [
         [0.0, 0.0],
         [0.0, 0.0],
         ]
```

```
In [41]: np.matmul(np.array([[1,1,1],[1,1,1]]), np.array([[2,2,2],[2,2,2],[2,2,2]]))
```

```
Out[41]: array([[6, 6, 6],
                [6, 6, 6]])
```

1.2.5 EJERCICIO 5: Vectores

```
In [42]: class Vector(transpose):
             "clase de Vectores Array"
             def __init__(self, list_of_numbers):
                 self.vdata = list_of_numbers
                 list_of_rows = [[x] for x in list_of_numbers]
                 return transpose.__init__(self, list_of_rows)
             def __repr__(self):
                 return "Vector(" + str(self.vdata) + ")"
             def __str__(self):
                 return str(self.vdata)
             def __add__(self, other):
                 if type(other) is Vector:
                     new_arr = transpose.__add__(self, other)
                 else:
                     "Suma escalar"
                     rows, cols = self.shape
                     new_arr = transpose([0 for r in range(rows)])
                     for r in range(rows):
                         new_arr.data[r][0] = self.data[r][0] + 10
                     print(new_arr.__dict__)
                 return Vector([x[0] for x in new_arr.data])
```

```
In [43]: Vector([1,2,3]).__dict__
```

```
Out[43]: {'data': [[1], [2], [3]], 'shape': (3, 1), 'vdata': [1, 2, 3]}
```

```
In [44]: Vector([1,2,3])
```

```
Out[44]: Vector([1, 2, 3])
```

```
In [45]: Vector([1,2,3]) + Vector([4,5,6])
```

```
Out[45]: Vector([5, 7, 9])
```

1.2.6 EJERCICIO 6

```
In [46]: Vector([1,2,3]) + 10
```

```
{'data': [[11], [12], [13]], 'shape': (1, 1)}
```

```
Out[46]: Vector([11, 12, 13])
```

1.2.7 EJERCICIO 7

1.2.8 EJERCICIO 8

1.2.9 EJERCICIO 9

1.2.10 EJERCICIO 10