Centro Universitário FEI

**MÉTODO DOS MÍNIMOS QUADRADOS**

Disciplina:

Tópicos Especiais em Aprendizagem (Prof. Reinaldo A. C. Bianchi)

Aluno:

Claudio Aparecido Borges Junior (RA 120122-7)

São Bernardo do Campo

2o semestre / 2020

**EXERCÍCIO:**

*Implemente o Método dos Míninos quadrados em C / C++ / Java / (Python): Básico – Linear, Quadrático, e Robusto (com peso). Teste o funcionamento usando: “Alps Water”, “Books x Grades” e “US Census Dataset”.*

A implantação foi realizada utilizando a linguagem Python versão 3.6. Foi criada uma classe chamada *Matrix* que implementa as principais operações matriciais como transposição, inversão, multiplicação, cálculo de determinante, entre outros. Essa classe não está otimizada em desempenho, porém a mesma não se limita a uma matriz de tamanho fixo.

A função *main* avalia os 3 conjuntos de dados em 3 métodos de mínimos quadrados: linear, quadrático e robusto com peso utilizando o inverso da variância. Os resultados estão apresentados abaixo:

*“Alps Water”:*

*“Books x Grades”*

*“US Census Dataset”*

Utilizando o “Alps Water”, pode-se calcular o valor da pressão para uma temperatura de 200F como:

**ANEXO I – CÓDIGO FONTE**

import copy  
import csv  
from typing import Tuple  
  
  
class Matrix:  
 *""" Represents a matrix and its operations.  
  
 In mathematics, a matrix is a rectangular array or table of numbers,  
 symbols, or expressions, arranged in rows and columns.  
 """* @staticmethod  
 def build(path: str) -> 'Matrix':  
 *""" Builds a matrix from a file path.  
  
 It's expected that each column is separated by tab (\t), the first  
 column is the header, the remaining columns are the data, and all rows  
 have the same number of columns.* ***:param*** *path: the TSV file path.* ***:return****: a Matrix.  
 """* with open(path) as file:  
 rd = csv.reader(file, delimiter='\t', quotechar='"')  
 raw = [data for data in rd]  
 data\_str = raw[1:]  
 data\_int = [[float(elm) for elm in row] for row in data\_str]  
 return Matrix(data\_int)  
  
 def \_\_init\_\_(self, data: list):  
 *""" Initializes a Matrix.* ***:param*** *data: A list of list, being the outer list is the rows and the  
 inner list is the columns.  
 """* self.data = copy.deepcopy(data)  
  
 def \_assert\_squareness(self):  
 *""" Assert that the Matrix is square.  
  
 A square matrix is a matrix with the same number of rows and columns.  
 """* assert len(self.data) == len(self.data[0]), 'Matrix is not square'  
  
 def \_assert\_invertible(self):  
 *""" Assert that the Matrix is invertible.  
  
 An n-by-n square matrix A is called invertible (also nonsingular or  
 nondegenerate), if there exists an n-by-n square matrix B such thar  
 $AB = BA = I\_n$, where $I\_n$ denotes the n-by-n identity matrix.  
 Source: https://en.wikipedia.org/wiki/Invertible\_matrix  
 """* assert self.determinant() != 0, 'Not invertible, determinant is 0'  
  
 def get\_n\_columns(self) -> int:  
 *""" Get the matrix number of columns.* ***:return****: the number of columns.  
 """* return len(self.data[0])  
  
 def get\_n\_rows(self) -> int:  
 *""" Get the matrix number of rows.* ***:return****: the number of rows.  
 """* return len(self.data)  
  
 def get\_element(self, row, column):  
 *""" Get an matrix's element giving a row and column indices.* ***:param*** *row: row index starting from 0.* ***:param*** *column: row index starting from 0.* ***:return****: matrix element.  
 """* return self.data[row][column]  
  
 def get\_column(self, col\_idx: int) -> 'Matrix':  
 *""" Get a column as a Matrix.* ***:param*** *col\_idx: column index starting from 0.* ***:return****: a Matrix with only 1 column.  
 """* col = [[self.data[i][col\_idx]] for i in range(len(self.data))]  
 return Matrix(col)  
  
 def get\_row(self, row\_idx: int) -> 'Matrix':  
 *""" Get a row as a Matrix.* ***:param*** *row\_idx: row index starting from 0.* ***:return****: a Matrix with only 1 row.  
 """* data = [self.data[row\_idx]]  
 return Matrix(data)  
  
 def del\_column(self, col\_idx: int) -> 'Matrix':  
 *""" Deletes a column and return a new Matrix.* ***:param*** *col\_idx: column index to be deleted starting from 0.* ***:return****: a Matrix with 1 column less.  
 """* m = self.data  
 n\_rows = len(m)  
 n\_columns = len(m[0])  
 # Builds the matrix without a column  
 refactored = [[m[i][j] for j in range(n\_columns)  
 if j != col\_idx]  
 for i in range(n\_rows)]  
 return Matrix(refactored)  
  
 def prepend\_val\_in\_rows(self, val: float) -> 'Matrix':  
 *""" Prepend a constant value as the fist column.* ***:param*** *val: value to be prepended.* ***:return****: a Matrix with a prepended column.  
 """* prepended = [[val] + row for row in self.data]  
 return Matrix(prepended)  
  
 def transpose(self) -> 'Matrix':  
 *""" Transpose the matrix.  
  
 In linear algebra, the transpose of a matrix is an operator which flips  
 a matrix over its diagonal; that is, it switches the row and column  
 indices of the matrix A by producing another matrix, often denoted by  
 $A^T$ (among other notations).  
 Source: https://en.wikipedia.org/wiki/Transpose* ***:return****: the transposed Matrix.  
 """* m = self.data  
 n\_row = len(m)  
 n\_col = len(m[0])  
 # Invert rows per columns  
 transposed = [[m[row][col] for row in range(n\_row)]  
 for col in range(n\_col)]  
 # Return a new Matrix's instance  
 return Matrix(transposed)  
  
 def matrix\_minor(self, row\_idx: int, col\_idx: int) -> float:  
 *""" Return a matrix minor a row and column.  
  
 In linear algebra, a minor of a matrix A is the determinant of some  
 smaller square matrix, cut down from A by removing one or more of its  
 rows and columns. Minors obtained by removing just one row and one  
 column from square matrices (first minors) are required for calculating  
 matrix cofactors, which in turn are useful for computing both the  
 determinant and inverse of square matrices.  
 Source: https://en.wikipedia.org/wiki/Minor\_(linear\_algebra)* ***:param*** *row\_idx: row index to be removed starting from 0.* ***:param*** *col\_idx: column index to be removed starting from 0.* ***:return****: the matrix minor.  
 """* # Remove a row and column of a matrix and return a copy of it. This  
 # operation is specially useful for calculating the cofactor matrix.  
  
 # i is the row index and j is the column index.  
 minor = [[self.data[i][j] for j in range(len(self.data[i]))  
 if j != col\_idx]  
 for i in range(len(self.data))  
 if i != row\_idx]  
 return Matrix(minor).determinant()  
  
 def cofactor(self) -> 'Matrix':  
 *""" Return the cofactor matrix.  
  
 The (i, j) cofactor is obtained by multiplying the minor by  
 $(− 1)^{i+j}$.* ***:return****: the cofactor Matrix.  
 """* # Calculate the cofactor matrix  
 self.\_assert\_squareness()  
 m = self.data  
 n = len(m)  
  
 coff = []  
 # 1. For each row  
 for i in range(n):  
 row = []  
 # 2. And for each column  
 for j in range(n):  
 # 3. Calculate the matrix minor  
 det = self.matrix\_minor(i, j)  
 # 4. Adjust the determinant by (-1)^(i+j)  
 def\_adj = det \* (-1) \*\* (i + j + 2)  
 # 5. Build the row  
 row.append(def\_adj)  
 # 6. Build the columns  
 coff.append(row)  
  
 return Matrix(coff)  
  
 def adjugate(self) -> 'Matrix':  
 *""" Return the adjugate matrix.  
  
 In linear algebra, the adjugate or classical adjoint of a square matrix  
 is the transpose of its cofactor matrix.  
 Source: https://en.wikipedia.org/wiki/Adjugate\_matrix* ***:return****: the adjugate Matrix.  
 """* # 1. Calculate the cofactor matrix  
 coff = self.cofactor()  
 # 2. Transpose it  
 adjugate = coff.transpose()  
 return adjugate  
  
 def inverse(self) -> 'Matrix':  
 *""" Invert the matrix.  
  
 An n-by-n square matrix A is called invertible (also nonsingular or  
 nondegenerate), if there exists an n-by-n square matrix B such thar  
 $AB = BA = I\_n$, where $I\_n$ denotes the n-by-n identity matrix.  
 https://en.wikipedia.org/wiki/Invertible\_matrix* ***:return****: the inverted Matrix.  
 """* # 1. Verify if the matrix is invertible  
 self.\_assert\_invertible()  
 # 2. Calculate the matrix determinant  
 det = self.determinant()  
 # 3. Calculate the adjugate matrix  
 adjugate = self.adjugate()  
 # 4. Calculate the 1/det(M) \* Adjugate  
 inverse = adjugate.mult\_scalar(1 / det)  
 return inverse  
  
 def determinant(self) -> float:  
 *""" Calculate the matrix determinant.  
  
 In linear algebra, the determinant is a scalar value that can be  
 computed from the elements of a square matrix and encodes certain  
 properties of the linear transformation described by the matrix.* ***:return****: the matrix determinant value.  
 """* self.\_assert\_squareness()  
 # M is the short form of matrix  
 m = self.data  
 # N is the matrix order (n x n)  
 n = len(m)  
  
 if n == 1:  
 # Determinant of 1x1  
 return m[0][0]  
 elif n == 2:  
 # Determinant of 2x2 matrix is straightforward  
 return (m[0][0] \* m[1][1]) - (m[0][1] \* m[1][0])  
  
 # Build a new matrix to use rule of Sarrus  
 m\_sarrus = [row + row[0:n - 1] for row in m]  
  
 # Apply rule of sarrus  
 partial = 0  
 for col in range(n):  
 moving\_down = 1  
 moving\_up = 1  
 for i in range(n):  
 moving\_down \*= m\_sarrus[i][col + i]  
 moving\_up \*= m\_sarrus[n - i - 1][col + i]  
 partial += moving\_down - moving\_up  
 return partial  
  
 def mult\_scalar(self, scalar: float) -> 'Matrix':  
 *""" Multiply a matrix to a scalar.  
  
 Multiply each element of the matrix by the scalar.* ***:param*** *scalar: value to multiply the matrix.* ***:return****: a matrix multiplied by a scalar.  
 """* scaled = [[scalar \* elm for elm in row] for row in self.data]  
 return Matrix(scaled)  
  
 def mult\_vector(self, vector: list) -> 'Matrix':  
 *""" Multiply a matrix by an vector.  
  
 Multiply each row by a the vector. Both matrix row and vector must  
 have the same dimension.* ***:param*** *vector: the list of elements to multiply each row.* ***:return****: the matrix multiplied by the vector.  
 """* # Multiply a matrix to a vector  
 m = self.data  
  
 assert len(m[0]) == len(vector), 'Impossible to multiply'  
  
 multiplied = [[m[i][j] \* vector[j] for j in range(len(m[0]))]  
 for i in range(len(m))]  
 return Matrix(multiplied)  
  
 def mult\_matrix(self, matrix2: 'Matrix') -> 'Matrix':  
 *""" Multiply a matrix by another matrix.  
  
 In mathematics, particularly in linear algebra, matrix multiplication is  
 a binary operation that produces a matrix from two matrices. For matrix  
 multiplication, the number of columns in the first matrix must be equal  
 to the number of rows in the second matrix. The resulting matrix, known  
 as the matrix product, has the number of rows of the first and the  
 number of columns of the second matrix.  
 Source: https://en.wikipedia.org/wiki/Matrix\_multiplication* ***:param*** *matrix2: the second matrix to multiply this matrix.* ***:return****: the result of the matrixes multiplication.  
 """* # Register the matrices m1 and m2  
 m1 = self.data  
 m2 = matrix2.data  
  
 # Register the number of rows and columns of both matrices m1 and m2  
 n\_rows\_m1 = len(m1)  
 n\_cols\_m1 = len(m1[0])  
 n\_rows\_m2 = len(m2)  
 n\_cols\_m2 = len(m2[0])  
  
 # Multiplication property: the number of columns of the first matrix  
 # must be the same as the number of rows of the second matrix  
 assert n\_cols\_m1 == n\_rows\_m2, 'Matrices can not be multiplied'  
  
 # For each row of m1, multiply by each column of m2 to find a single  
 # element  
 result = []  
 for row\_m1 in range(n\_rows\_m1):  
 row = []  
 for col\_m2 in range(n\_cols\_m2):  
 row.append(sum([m1[row\_m1][i] \* m2[i][col\_m2]  
 for i in range(n\_cols\_m1)]))  
 result.append(row)  
  
 # Return a new Matrix's instance  
 return Matrix(result)  
  
 def power(self, power) -> 'Matrix':  
 *""" Power each matrix's element (element wise operation).* ***:param*** *power: the power value.* ***:return****: return the matrix powered (element wise).  
 """* powered = [[val \*\* power for val in row] for row in self.data]  
 return Matrix(powered)  
  
 def merge(self, matrix: 'Matrix') -> 'Matrix':  
 *""" Merge matrixes with the same number of rows.* ***:param*** *matrix: matrix to be merged to this matrix.* ***:return****: the result of the merge.  
 """* # Merge a matrix with the same number of rows  
 m1 = self.data  
 m2 = matrix.data  
  
 assert len(m1) == len(m2), 'Impossible to merge'  
  
 merged = [row\_1 + row\_2 for row\_1, row\_2 in zip(m1, m2)]  
 return Matrix(merged)  
  
 def \_\_str\_\_(self) -> str:  
 return 'data=' + str(self.data)  
  
  
def least\_squares(y: Matrix, x: Matrix) -> Matrix:  
 *""" Calculate the least squares.  
  
 The method of least squares is a standard approach in regression analysis  
 to approximate the solution of overdetermined systems (sets of equations  
 in which there are more equations than unknowns) by minimizing the sum of  
 the squares of the residuals made in the results of every single equation.  
 Source: https://en.wikipedia.org/wiki/Least\_squares* ***:param*** *y: a single column matrix representing the Y axis, the dependent  
 variable.* ***:param*** *x: a matrix representing the x axis, the features or independent  
 variables.* ***:return****: the regression matrix.  
 """* # 1. Calculate (x^T \* x)^-1  
 partial\_1 = x.transpose().mult\_matrix(x).inverse()  
 # 2. Calculate x^T \* y  
 partial\_2 = x.transpose().mult\_matrix(y)  
 # 3. Calculate the result, it is the multiplication of steps 4 and 5  
 result = partial\_1.mult\_matrix(partial\_2)  
 return result  
  
  
def least\_squares\_weighted(y: Matrix, x: Matrix, w: list) -> Matrix:  
 *""" Calculate the least squares with weights.  
  
 The method of least squares is a standard approach in regression analysis  
 to approximate the solution of overdetermined systems (sets of equations  
 in which there are more equations than unknowns) by minimizing the sum of  
 the squares of the residuals made in the results of every single equation.  
 Source: https://en.wikipedia.org/wiki/Least\_squares* ***:param*** *y: a single column matrix representing the Y axis, the dependent  
 variable.* ***:param*** *x: a matrix representing the x axis, the features or independent  
 variables.* ***:param*** *w: a list of weights.* ***:return****: the regression matrix.  
 """* # 1. Calculate (x^T \* w \* x)^-1  
 partial\_1 = x.transpose().mult\_vector(w).mult\_matrix(x).inverse()  
 # 2. Calculate x^T \* w \* y  
 partial\_2 = x.transpose().mult\_vector(w).mult\_matrix(y)  
 # 3. Calculate the result, it is the multiplication of steps 4 and 5  
 result = partial\_1.mult\_matrix(partial\_2)  
 return result  
  
  
def split\_matrix\_into\_x\_and\_y(matrix: Matrix) -> Tuple[Matrix, Matrix]:  
 *""" Return the Y and X value of a Matrix.  
  
 Matrix is expected to have both Xs and Y. The Y is expected to be  
 in the last column of the matrix.* ***:param*** *matrix: the matrix to be split.* ***:return****: a tuple of y and x matrixes.  
 """* # 1. Find the last column  
 last\_column = matrix.get\_n\_columns()  
 # 2. Retrieve it as y  
 y = matrix.get\_column(last\_column - 1)  
 # 3. Retrieve the x axis, it is the remaining columns  
 x = matrix.del\_column(last\_column - 1)  
 return y, x  
  
  
def apply\_least\_squares\_linear(matrix: Matrix) -> Matrix:  
 *""" Apply linear least squares* ***:param*** *matrix: a matrix containing the Y and Xs values.* ***:return****: the regression model.  
 """* # 1. Retrive Y axis and X axises  
 y, x\_pre = split\_matrix\_into\_x\_and\_y(matrix)  
 # 2. Prepend 1 to include Beta 0 in the Beta matrix  
 x = x\_pre.prepend\_val\_in\_rows(1)  
 # 3. Run the calculation  
 return least\_squares(y, x)  
  
  
def apply\_least\_squares\_quadratic(matrix: Matrix) -> Matrix:  
 *""" Apply quadratic least squares* ***:param*** *matrix: a matrix containing the Y and Xs values.* ***:return****: the regression model.  
 """* # 1. Retrieve Y axis and X axises  
 y, x\_pre = split\_matrix\_into\_x\_and\_y(matrix)  
 # 2. Calculate the power of 2 for each element of X  
 x\_pre\_squared = x\_pre.power(2)  
 # 3. Build a single matrix for X with the number 1 prepended to include  
 # Beta 0 in the Beta matrix  
 x = x\_pre.merge(x\_pre\_squared).prepend\_val\_in\_rows(1)  
 # 4 Run the calculation  
 return least\_squares(y, x)  
  
  
def apply\_least\_squares\_weighted\_robust(matrix: Matrix) -> Matrix:  
 *""" Apply robust (weighted) least squares.  
  
 The weighted value is computed by the inverse of variance.* ***:param*** *matrix: a matrix containing the Y and Xs values.* ***:return****: the regression model.  
 """* # 1. Calculate the linear least squares  
 linear = apply\_least\_squares\_linear(matrix)  
 # 2. Retrieve Y axis and X axises  
 y, x\_pre = split\_matrix\_into\_x\_and\_y(matrix)  
 # 3. Prepend 1 to include Beta 0 in the Beta matrix  
 x = x\_pre.prepend\_val\_in\_rows(1)  
 # 4. Calculate the weights using the inverse of variance  
 w = []  
 n\_rows = x.get\_n\_rows()  
 for i in range(n\_rows):  
 estimation = x.get\_row(i).mult\_matrix(linear).get\_element(0, 0)  
 variance = y.get\_element(i, 0) - estimation  
 w\_i = abs(1/variance)  
 w.append(w\_i)  
 # 5. Calculate the least squares using the weights  
 return least\_squares\_weighted(y, x, w)  
  
  
def evaluate(path: str):  
 *""" Evaluate a dataset.  
  
 The evaluation run 3 different least squares methods: linear, quadratic,  
 and robust (weighted).* ***:param*** *path: dataset path.  
 """* matrix = Matrix.build(path)  
 print('-' \* 80)  
 print('Evaluation: ' + path)  
 print('Linear: ' + str(apply\_least\_squares\_linear(matrix)))  
 print('Quadratic: ' + str(apply\_least\_squares\_quadratic(matrix)))  
 print('Linear Weighted: ' + str(apply\_least\_squares\_weighted\_robust(matrix)))  
  
  
def main():  
 # Evaluate the 3 datasets  
 evaluate('d1\_alpswater.tsv')  
 evaluate('d2\_books\_attend\_grade.tsv')  
 evaluate('d3\_us\_census.tsv')  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

**ANEXO II – SAÍDA DO PROGRAMA**

--------------------------------------------------------------------------------  
Evaluation: d1\_alpswater.tsv  
Linear: data=[[-81.06372712864686], [0.522892400784599]]  
Quadratic: data=[[38.82928171008825], [-0.6547704091062769], [0.002889711086254465]]  
Linear Weighted: data=[[-81.36733077487588], [0.5243294760032597]]  
--------------------------------------------------------------------------------  
Evaluation: d2\_books\_attend\_grade.tsv  
Linear: data=[[37.379185204571286], [4.0368926110093355], [1.283477274709945]]  
Quadratic: data=[[0.06230835531275236], [-0.3493624778801523], [1.2794887604338177], [-0.08526151736744608], [0.0010906510793434031]]  
Linear Weighted: data=[[33.89660258350932], [4.049644310350338], [1.5394228560559924]]  
--------------------------------------------------------------------------------  
Evaluation: d3\_us\_census.tsv  
Linear: data=[[-3783.9455909089884], [2.025302727272731]]  
Quadratic: data=[[32294.198410987854], [-34.98766851751134], [0.009490508241924545]]  
Linear Weighted: data=[[-3778.840082823066], [2.022789272835894]]

**ANEXO III – CONJUNTO DE DADOS “Alps Water”**

temperature pressure  
194.5 20.79  
194.3 20.79  
197.9 22.40  
198.4 22.67  
199.4 23.15  
199.9 23.35  
200.9 23.89  
201.1 23.99  
201.4 24.02  
201.3 24.01  
203.6 25.14  
204.6 26.57  
209.5 28.49  
208.6 27.76  
210.7 29.04  
211.9 29.88  
212.2 30.06

**ANEXO IV – CONJUNTO DE DADOS “Books x Grades”**

books attend grade  
0 9 45  
1 15 57  
0 10 45  
2 16 51  
4 10 65  
4 20 88  
1 11 44  
4 20 87  
3 15 89  
0 15 59  
2 8 66  
1 13 65  
4 18 56  
1 10 47  
0 8 66  
1 10 41  
3 16 56  
0 11 37  
1 19 45  
4 12 58  
4 11 47  
0 19 64  
2 15 97  
3 15 55  
1 20 51  
0 6 61  
3 15 69  
3 19 79  
2 14 71  
2 13 62  
3 17 87  
2 20 54  
2 11 43  
3 20 92  
4 20 83  
4 20 94  
3 9 60  
1 8 56  
2 16 88  
0 10 62

**ANEXO V – CONJUNTO DE DADOS “US Census Dataset”**

year population  
1900 75.9950  
1910 91.9720  
1920 105.7110  
1930 123.2030  
1940 131.6690  
1950 150.6970  
1960 179.3230  
1970 203.2120  
1980 226.5050  
1990 249.6330  
2000 281.4220