Métodos Quantitativos

Marcos Rodrigues, Matheus Alcântara, Pedro Ruas

Maio 2017

1 Introdução

Este documento apresenta uma relação de códigos e procedimentos implementados, utilizando as linguagens Python e R, tal como a ferramenta *Minitab Express*, para explorar as técnicas dentro do contexto de Métodos Quantitativos em Ciência da Computação.

Para a linguagem Python, foram utilizadas bibliotecas que implementam os cálculos e operações necessárias, sendo elas a biblioteca NumPy (incluindo o pacote Linalg) e SKLearn. A linguagem R contempla em suas bibliotecas as operações necessárias para o desenvolvimento dos códigos aqui descritos.

2 Revisão de matrizes

Neste capítulo demonstramos exemplos de implementações que efetuam operações em matrizes. Para exemplo, foram geradas matrizes aleatórias de tamanho $5x5$, composta de inteiros variando de -9 a 10. Para a geração das matrizes de números aleatórios, utilizamos os códigos que seguem. Considere $m1$, $m2$, A e B como as matrizes utilizadas nos exemplos.

Python: Revisão de matrizes

```python
# Importa a biblioteca NumPy para uso
import numpy as np

# Gerar matrizes aleatórias
m1 = np.random.randint(-9, 10, size=(5, 5))
m2 = np.random.randint(-9, 10, size=(5, 5))

# A seguir, uma série de operações que podem ser realizadas utilizando matrizes,
↪  em códigos usando Python e R

#Soma algébrica
soma = np.add(m1, m2) # Entre duas matrizes
soma = np.add(4, m1)  # Entre um escalar e uma matriz

# Multiplicação
mult =  np.matmul(m1, m2)   # m1 x m2
mult =  np.matmul(m2, m1)   # m2 x m1
mult =  np.matmul(m1,  4)   # m1 x 4, um escalar

#Triangular superior e inferior
superior = np.triu(m1)
inferior = np.tril(m1)

#Diagonal
diag = np.diag(m1)

#Identidade
identidade=np.identity(5)

#Transposta e simétrica
```

```
transposta = np.matrix.transpose(m1)
isSimetrica = np.array_equal(m1,transposta)  # Verifica se a matriz é simétrica

#Inversa
inversa = np.linalg.inv(m1)

#Determinante
determinante = np.linalg.det(m1)

#Adjunta (adjunta = determinante * inversa)
adjunta = np.linalg.det(m1) * np.linalg.inv(m1)

#Menor complementar
def menorComp(matriz, i, j):
        # Função: retira a linha e coluna (i,j) e retorna o determinante
        temp = np.delete(np.delete(matriz, j, 1), i, 0)
        return np.linalg.det(temp)
# Menor complementar da linha 1 e coluna 2
minor = menorComp(m1, 1, 2))

#Matriz de cofatores
def matrizDeCofatores(matriz):
        # Para cada elemento, calcular o menor complementar (determinante
        ↪  removendo a linha e coluna)
        # Em cada valor obtido, multiplicar por ((-1)**(i+j)))
        temp = np.full_like(matriz, 0)
        for index in np.ndindex(matriz.shape):
                i,j = (index)
                temp[index] = ((-1)**(i+j)) * menorComp(matriz, i, j)
        return temp
cofatores = matrizDeCofatores(m1)

#Posto ou Rank
rank = np.linalg.matrix_rank(m1)

#Autovalor e autovetor
autovalor, autovetor = np.linalg.eig(m1)

#Equações lineares
# Exemplo:
#   x { y { z + w = 10
#   2x + 3y + 5z { 2w = 21
#   4x { 2y { z + w = 16
coeficientes = np.array([[ 1,-1,-1, 1],
                                                [ 2, 3, 5,-2],
                                                [ 4,-2,-1, 1]])
valores = np.array([10,21,16])

# Este método requer uma matrix quadrática
resultado = np.linalg.solve(coeficientes, valores)

# Método 'least-squares-fitting', que minimiza o erro da solução aproximada
resultado = np.linalg.lstsq(coeficientes, valores)
```

R: Revisão de matrizes

```
# Gerar matrizes aleatórias
A <- matrix(sample(-9:10,15), nrow=5, ncol=5)
B <- matrix(sample(-9:10,15), nrow=5, ncol=5)
```

```r
#Soma algébrica
cat("Soma (A+B):\n") ; A+B

# Multiplicação
A ; 3*A              # Multiplicação por um escalar
A ; B ; A*B          # Multiplicação dos elementos das matrizes
A ; B ; A %*% B    # Multiplicação das matrizes

# Triangular superior e inferior
SUPERIOR <- upper.tri(A, diag=TRUE)
INFERIOR <- lower.tri(A, diag=TRUE)

# Diagonal
D <- diag(A)

# Identidade
I <- diag(1,nrow=dim(A)[1]) # Matriz identidade

# Transposta e simétrica
TRANSPOSTA <- t(A)
isSimetrica - isSymmetric.matrix(A)

# Inversa
INV <- solve(A)

# Determinante
DET <- det(A)
DET <- det(t(A)) # Determinante da transposta de A

# Adjunta (adjunta = determinante * inversa)
ADJ <- det(A) * solve(A)

# Matriz de cofatores
A <- matrix(data=c(1, 2, 3, 2, 5, 9, 5, 7, 8), nrow=3, ncol=3)
minor <- function(A, i, j) det( A[-i,-j] )
cofactor <- function(A, i, j) (-1)^(i+j) * minor(A,i,j)
adjoint1 <- function(A) { #Loop
  n <- nrow(A)
  B <- matrix(NA, n, n)
  for( i in 1:n )
    for( j in 1:n )
      B[j,i] <- cofactor(A, i, j)
  B
}

adjoint2 <- function(A) { #Outer
  n <- nrow(A)
  t(outer(1:n, 1:n, Vectorize(
    function(i,j) cofactor(A,i,j)
  )))
}
cat("Cofator da matriz A:\n") ; A %*% adjoint1(A)
cat("Cofator da matriz A:\n") ; A %*% adjoint2(A)

# Posto ou Rank
POSTO <- matrix(rank(A), nrow=length(A[,1]), ncol=length(A[1,]))
```

```r
# Autovalor e autovetor
A <- matrix(c(3,-1,1,-1,5,-1,1,-1,3), nrow=3, ncol=3)
y <- eigen(A)
cat("Autovalores da matriz A:\n") ; y$values
cat("Autovetores da matriz A:\n") ; round(y$vectors)

# Equações lineares
A <- matrix(data=c(1, 2, 3, 2, 5, 9, 5, 7, 8), nrow=3, ncol=3, byrow=TRUE)
x <- matrix(data=c(20, 100, 200), nrow=3, ncol=1, byrow=FALSE)
b <- round(solve(A, x), 3) ; rownames(b) <- c("x","y","z")
cat("Sistema de equacoes [A][x]=[y]\n")
cat("Matriz A:\n") ; A
cat("Matriz x:\n") ; x
cat("Matriz solucao b:\n") ; b
```

3 Análise de Componentes Principais

Neste capítulo demonstramos exemplos de implementações envolvidas na Análise de Componentes Principais, ou em inglês *Principal Component Analysis (PCA)*. Para tanto, fizemos o uso da biblioteca SKLEARN em Python, com o módulo PCA. Em R, as funções já estão nativamente implementadas.

Python: Análise de Componentes Principais

```python
# Importa a biblioteca NumPy e as funções de PCA da biblioteca SKLEARN
import numpy as np
from sklearn.decomposition import PCA

# Gera uma matriz aleatória
X = np.random.randint(0, 10, size=(7, 7))

# Inicia o PCA, sobre a matriz X
pca = PCA(n_components=5)
pca.fit(X)

# Exibe as principais componentes, representando as direções de variância máxima
↪  nos dados. As componentes são ordenadas pela variância explicada
print (pca.components_)

# Exibe a variância explicada de cada componente selecionado
print(pca.explained_variance_)

# Exibe o percentual de variância explicada por componente
print(pca.explained_variance_ratio_)

# Exibe a variância explicada acumulada
print(pca.explained_variance_ratio_.sum())

# Realiza a redução de dimensionalidade na matriz X
print(pca.transform(X))
```

R: Análise de Componentes Principais

```r
data(decathlon2)

# Extrai os indivíduos ativos e as variáveis para a PCA
decathlon2.active <- decathlon2[1:23, 1:10]
head(decathlon2.active[, 1:6])
res.pca <- prcomp(decathlon2.active, scale=TRUE)
```

```r
# res.pca, extraídos da função prcomp, são os valores, e podem ser exibidos com
↪  este comando
names(res.pca)
print(res.pca)

# Os desvios padrões dos componentes principais (raiz quadrada dos autovalores)
head(res.pca$sdev)

# A matriz com os pesos das variáveis (as colunas são os autovetores)
head(unclass(res.pca$rotation)[, 1:4])

# Autovalores
eig <- (res.pca$sdev)^2
# Variâncias, em %
variance <- eig*100/sum(eig)
# Variâncias acumuladas
cumvar <- cumsum(variance)
eig.decathlon2.active <- data.frame(eig=eig, variance=variance,
↪  cumvariance=cumvar)
head(eig.decathlon2.active)

summary(res.pca)
fviz_screeplot(res.pca, ncp=10)
head(res.pca)
fviz_pca_var(res.pca)
fviz_pca_var(res.pca, col.var="cos2") +
  scale_color_gradient2(low="white", mid="blue",
                        high="red", midpoint=0.5) + theme_minimal()

# Contribuição das variáveis na primeira componente
fviz_pca_contrib(res.pca, choice = "var", axes = 1)
fviz_pca_contrib(res.pca, choice = "var", axes = 1, top = 7)

# Contribuição das variáveis na segunda componente
fviz_pca_contrib(res.pca, choice = "var", axes = 2)

# Contribuição na primeira e segunda componentes
fviz_pca_contrib(res.pca, choice = "var", axes = 1:2)
```

4 Decomposição em Valores Singulares

Neste capítulo demonstramos exemplos de implementações envolvidas na Decomposição em Valores Singulares, ou em inglês *Singular Value Decomposition (SVD)*. Também fizemos o uso da bilbioteca SKLEARN em Python. Da mesma forma que em PCA, as funções já estão nativamente implementadas em na linguagem R.

Python: Decomposição em Valores Singulares

```python
# Importa a biblioteca NumPy e as funções de SVD da biblioteca SKLEARN para uso
import numpy as np
from sklearn.decomposition import TruncatedSVD

# Gera uma matriz aleatória
X = np.random.randint(0, 10, size=(7, 7))

# Inicia a operação de SVD sobre a matriz X e exibe as componentes
svd = TruncatedSVD(n_components=5, n_iter=7, random_state=20)
svd.fit(X)
```

```
print (svd.components_)

# Exibe a variância dos dados de treinamento, transofrmados por uma projeção de
↪  cada componente
print(svd.explained_variance_)

# Exibe o percentual de variância explicada por componente
print(svd.explained_variance_ratio_)

# Exibe a variação explicada acumulada
print(svd.explained_variance_ratio_.sum())

# Realiza a redução de dimensionalidade na matriz X
print(svd.transform(X))
```

R: Decomposição em Valores Singulares

```
# A linha abaixo importa um conjunto de dados da URL fornecida
cnut <- read.dta("http://statistics.ats.ucla.edu/stat/data/cerealnut.dta")

# Centralizando as variáveis
mds.data <- as.matrix(sweep(cnut[, -1], 2, colMeans(cnut[, -1])))
dismat <- dist(mds.data)
mds.m1 <- cmdscale(dismat, k = 8, eig = TRUE)
mds.m1$eig

mds.m1 <- cmdscale(dismat, k = 2, eig = TRUE)
x <- mds.m1$points[, 1]
y <- mds.m1$points[, 2]
plot(x, y)
text(x + 20, y, label=cnut$brand)

# Autovalores
xx <- svd(mds.data %*% t(mds.data))
xx$d

# Coordenadas
xxd <- xx$v %*% sqrt(diag(xx$d))
x1 <- xxd[, 1]
y1 <- xxd[, 2]

# Para plotar os valores, mostrando o gráfico
plot(x1, y1)
text(x1 + 20, y1, label=cnut$brand)
```

5 Análise Fatorial

Neste capítulo demonstramos exemplos de implementações envolvidas na Decomposição em Valores Singulares, ou em inglês *Singular Value Decomposition (SVD)*. Também fizemos o uso da bilbioteca SKLEARN em Python. Da mesma forma que em PCA, as funções já estão nativamente implementadas em na linguagem R.

Python: Análise Fatorial

```
# Importa a biblioteca NumPy e as funções de Análise Fatorial da biblioteca
↪  SKLEARN para uso
import numpy as np
from sklearn.decomposition import FactorAnalysis
```

```python
# Gera uma matriz aleatória
X = np.random.randint(0, 10, size=(7, 7))

# Inicia a operação de Análise Fatorial sobre a matriz X e exibe as componentes
fa = FactorAnalysis(n_components=5)
fa.fit(X)
print (fa.components_)

# Calcula a covariância dos dados com o modelo de Análise Fatorial
print (fa.get_covariance())

# Exibe probabilidade logarítmica em cada iteração da Análise Fatorial
print(fa.loglike_)

# Exibe variância do ruído estimado para cada feature
print(fa.noise_variance_)

# Calcula a probabilidade logarítmica média das amostras (scores)
print(fa.score(X))

# Realiza a redução de dimensionalidade na matriz X
print(fa.transform(X))
```

R: Análise Fatorial

```r
# Importa bibliotecas necessárias e bases de teste disponíveis no R
install.packages("psych")
library(psych)
require(foreign)
library( GPArotation )

# Inicia os dados de teste
data(bfi)
head(bfi)
describe(bfi)

# Faz a análise fatorial nos dados
df <- bfi[1:25]
scree(df)
fa.parallel(bfi)

# Exibe os resultados e plota o diagrama para consulta
out <- fa(df,6,fm='minres',rotate='oblimin')
print(out$loadings,cut=.2)
fa.diagram( out )
out <- fa(df,5,fm='minres','oblimin')
print(out$loadings,cut=.2)
fa.diagram( out )
```

6 Regressão Linear Simples e Multipla

Neste capítulo demonstramos exemplos de implementações relacionadas a Regressão Linear, simples ou multipla, e também a Função Logística. Novamente, fizemos o uso das bibliotecas NumPy e SKLEARN em Python, enquanto na linguagem R há implementação nativa. Adicionalmente, em Python, usamos a biblioteca MatPlot para plotar gráficos para melhor visualização dos resultados da regressão linear.

Python: Regressão Linear Simples

```python
# Importa as bibliotecas usadas.
# Note que a SKLEARN disponibiliza um módulo datasets, com bases de dados de
↪  exemplos
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model

# Carrega a base de testes relacionada à diabetes
diabetes = datasets.load_diabetes()

# Usar somente uma feature da base
diabetes_X = diabetes.data[:, np.newaxis, 2]

# Divide os dados em 2 conjuntos distintos: treinamento e testes
diabetes_X_train = diabetes_X[:-20]
diabetes_X_test = diabetes_X[-20:]

# Divide os alvos em 2 conjuntos distintos: treinamento e testes
diabetes_y_train = diabetes.target[:-20]
diabetes_y_test = diabetes.target[-20:]

# Inicia um objeto para fazer a regressão linear
regr = linear_model.LinearRegression()

# Treina o modelo de regressão usando a base de treinamento
regr.fit(diabetes_X_train, diabetes_y_train)

# Exibe os coeficientes calculados
print('Coeficientes: \n', regr.coef_)

# Exibe o coeficiente R2 da predição
print('R2: \n', diabetes_X_test,diabetes_y_test)

# Exibe o erro quadrático médio
print("Erro médior: %.2f"
      % np.mean((regr.predict(diabetes_X_test) - diabetes_y_test) ** 2))

# Exibe o score da variância explicada: 1 significa predição perfeita
print('Score: %.2f' % regr.score(diabetes_X_test, diabetes_y_test))

# Exibe a predição usando o modelo linear
print('Predição: \n', regr.predict(diabetes_X))

# Exibe o gráfico para visualização
plt.scatter(diabetes_X_test, diabetes_y_test,  color='black')
plt.plot(diabetes_X_test, regr.predict(diabetes_X_test), color='blue',
         linewidth=3)
plt.xticks(())
plt.yticks(())
plt.show()
```

R: Regressão Linear Simples

```r
# Utiliza a base de dados Galapagos
galap <-
↪  read.table("https://sites.google.com/site/vllandeiror/dados/galapagos.txt",
↪  sep=" ", header=TRUE, quote="\"", colClasses="character",
↪  fileEncoding="UTF-8")
```

```r
riqueza <- scan(text=galap[,3], dec=".")
area <- scan(text=galap[,2], dec=".")
plot(area,riqueza)

# Note que a relação não é linear e que um ponto assemelha-se a um outlier. Para
↪  linearizar os dados e reduzir o efeito do outlier tranformamos os dados em
↪  log.
riqueza.log <-log10(strtoi(riqueza,base=10))
area.log <- log10(as.double(area))
plot(area.log, riqueza.log)

# Fazendo a regressão e mostrando os coeficientes a (intercepto) e b (inclinação)
↪  da regressão
resultado<-lm(riqueza.log ~ area.log)
resultado

# Detalhamento da regressão
summary(resultado)

# Estatísticas: valores de t, probabilidades, os graus de liberdade e R2
summary.aov(resultado)

# Gráficos para visualização, com a linha de tendência da regressão
plot(area.log,riqueza.log)
abline(resultado)
```

Python: Regressão Linear Múltipla

```python
# Imports
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model

# Função Logística
# O conjunto de dados de teste é uma linha, com alguns ruídos gaussianos
xmin, xmax = -5, 5
n_samples = 100
np.random.seed(0)
X = np.random.normal(size=n_samples)
y = (X > 0).astype(np.float)
X[X > 0] *= 4
X += .3 * np.random.normal(size=n_samples)
X = X[:, np.newaxis]

# Classificar
clf = linear_model.LogisticRegression(C=1e5)
clf.fit(X, y)

# Gerando alguns gráficos para visualização
plt.figure(1, figsize=(7, 6))
plt.clf()
plt.scatter(X.ravel(), y, color='black', zorder=20)
X_test = np.linspace(-5, 10, 300)

# Mais gráficos relacionados à Regressão com Função Logística
def model(x):
    return 1 / (1 + np.exp(-x))
loss = model(X_test * clf.coef_ + clf.intercept_).ravel()
plt.plot(X_test, loss, color='red', linewidth=3)
```

```
ols = linear_model.LinearRegression()
ols.fit(X, y)
plt.plot(X_test, ols.coef_ * X_test + ols.intercept_, linewidth=1)
plt.axhline(.5, color='.5')

plt.ylabel('y')
plt.xlabel('X')
plt.xticks(range(-5, 10))
plt.yticks([0, 0.5, 1])
plt.ylim(-.25, 1.25)
plt.xlim(-4, 10)
plt.legend(('Logistic Regression Model', 'Linear Regression Model'),
           loc="lower right", fontsize='small')
plt.show()
```

R: Regressão Linear Múltipla

```
# Importa a biblioteca necessária
library(car)

# Utiliza a base de dados de formigas
formigas <-
 ↪  read.table("https://sites.google.com/site/vllandeiror/dados/formigas.txt",
 ↪  sep=" ", header=TRUE, quote="\"", colClasses="character",
 ↪  fileEncoding="UTF-8")

# Latitude e altitude (dados da base)
lat <- scan(text=formigas[,1], dec=".")
alt <- scan(text=formigas[,2], dec=".")

# Densidade
dens.formig <- scan(text=formigas[,3], dec=".")
lm(log10(dens.formig) ~ lat+alt)

# Fazendo a regressão múltipla
reg.mult <- lm(log10(dens.formig) ~ lat+alt)
reg.mult
summary(reg.mult)
summary.aov(reg.mult)

# Exibe o gráfico dos parciais desta regressão múltipla
avPlots(reg.mult)
```

7 Estatística descritiva

O objetivo da estatística descritiva consiste em aplicar técnicas para descrever um conjunto de dados, resumindo suas características e comportamento. Neste capítulo demonstramos exemplos dessas técnicas, tendo com suporte a ferramenta *Minitab Express*.

Uma visão geral do sofware *Minitab Express* é exibida na Figura 1. Bases de exemplo da própria ferramenta estão disponíveis, e foram utilizadas. À direita da imagem, temos uma base de teste relativa ao consumo energético apurado na residência de 25 famílias, na qual aplicaremos alguns conceitos de estatística descritiva.

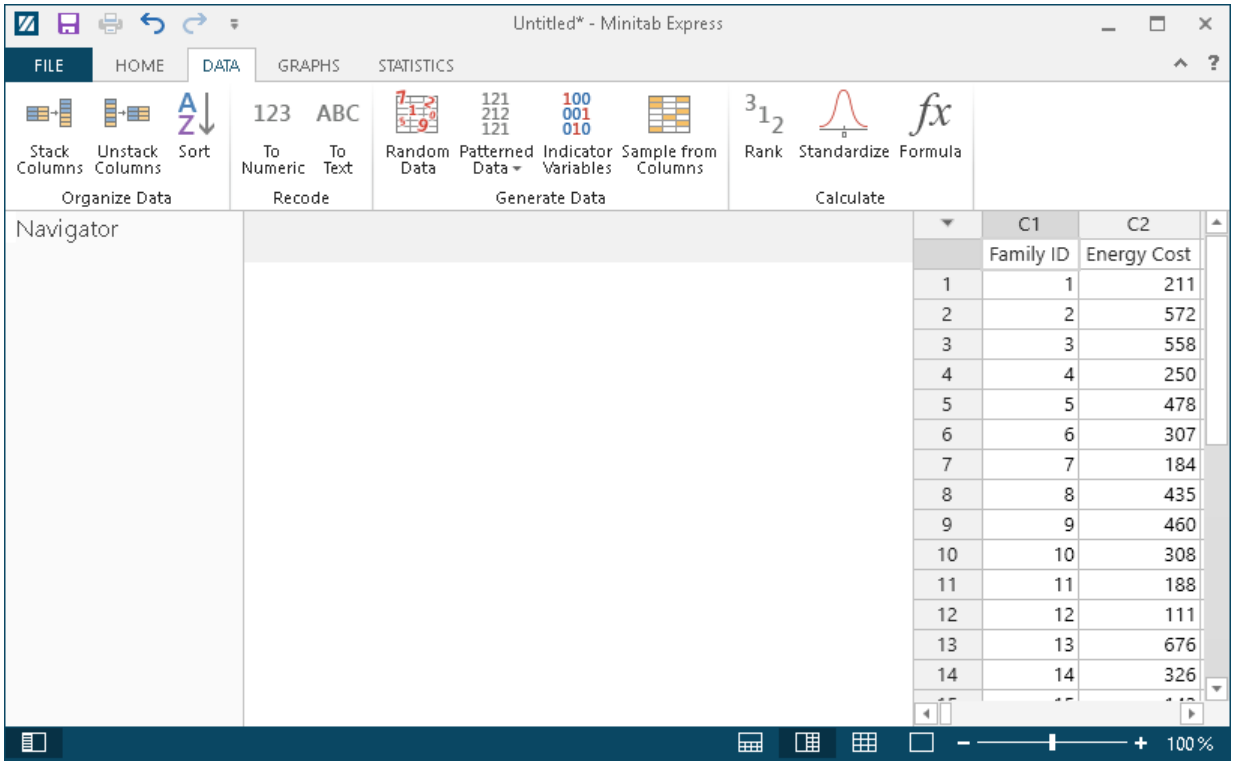

Figura 1: Minitab - Visão geral

É possível ainda gerar um conjunto de números aleatórios, com base em alguns parâmetros pré-definidos, através da opção *Random Data* como exibido na Figura 2. Dentre os parâmetros, a média, o desvio padrão e a distribuição (Normal, Poisson, Weibull, dentre outras).

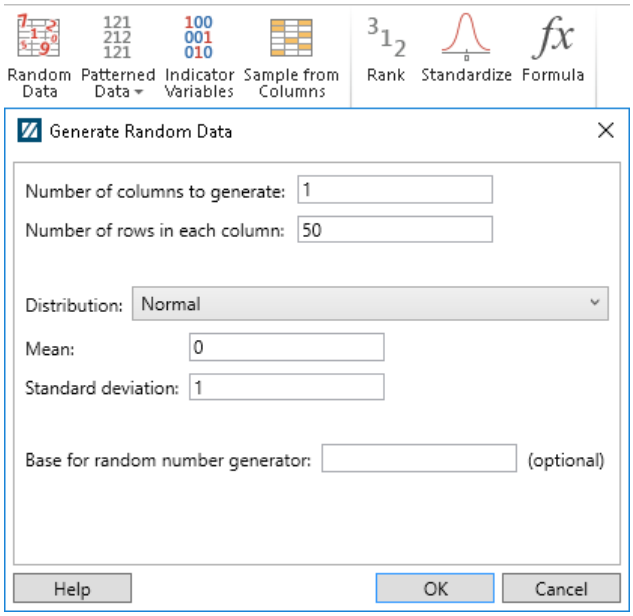

Figura 2: Geração de números aleatórios

A partir da base de dados, é possível, através da opção *Describe* > *Descriptive Statistics*, ver um resumo estatístico dessa base. Na Figura 3 é exibido a opção no software, bem como os resultados da base utilizada de consumo energético.

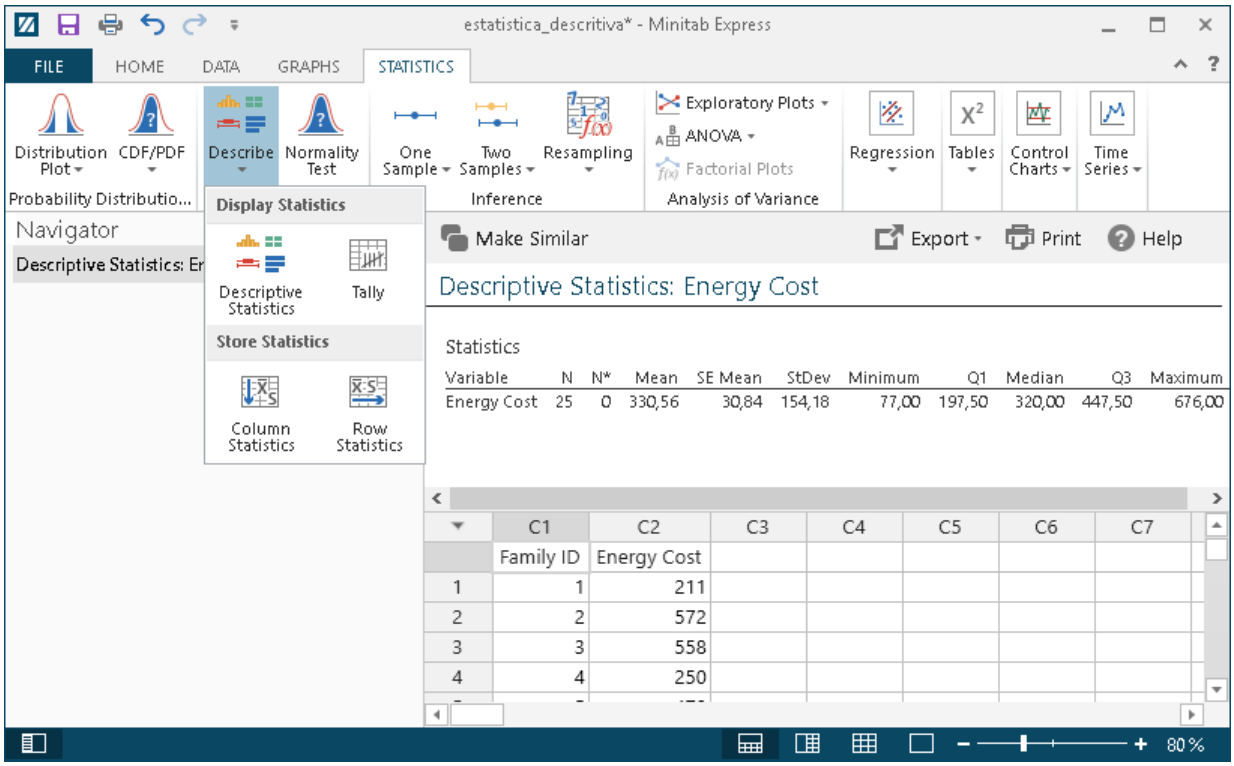

Figura 3: Medidas descritivas de variáveis quantitativas

Também é possível gerar uma série de gráficos distintos no âmbito de estatística descritiva, para análises. Como exemplo, o Histograma, gerado através da opção *Histogram* > *Simple*. Na Figura **??** é exibido a opção no software, bem como o histograma gerado.

Da mesma forma, o gráfico BoxPlot é exibido na Figura 5. Esse gráfico pode ser gerado através da opção *Boxplot* > *Simple*.

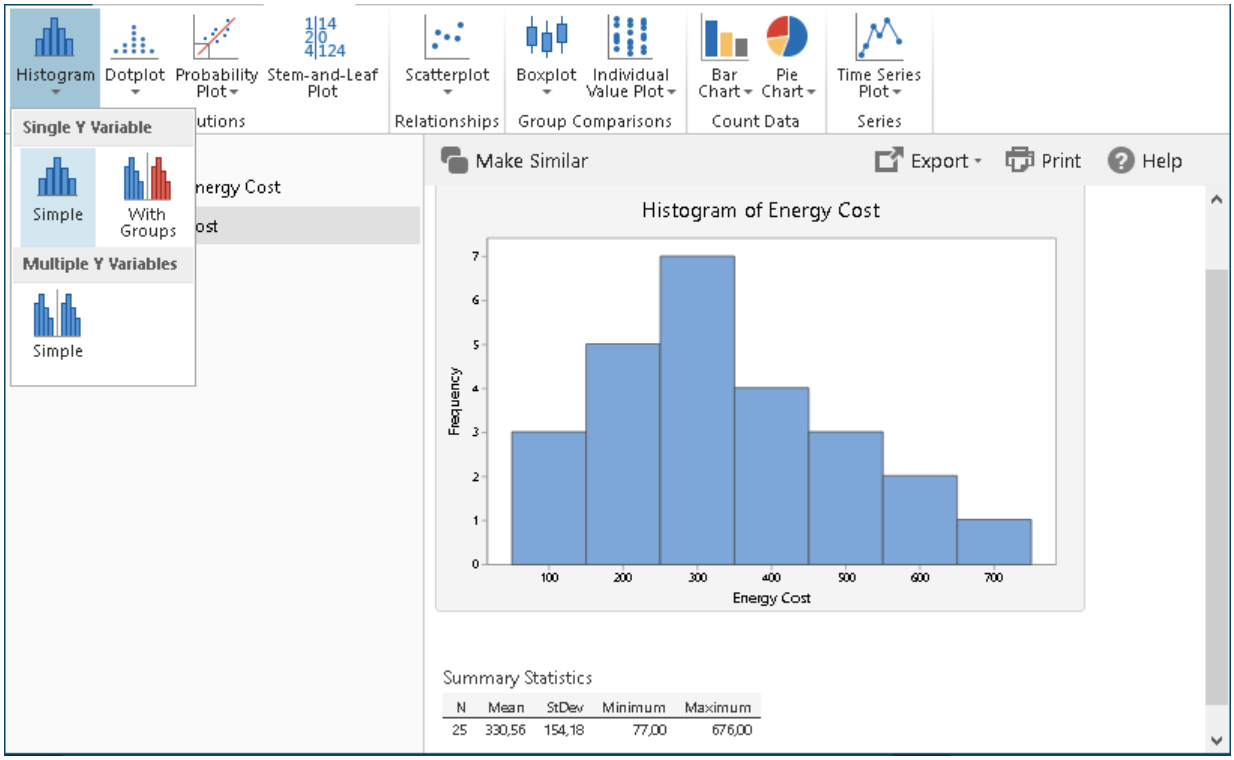

Figura 4: Histograma

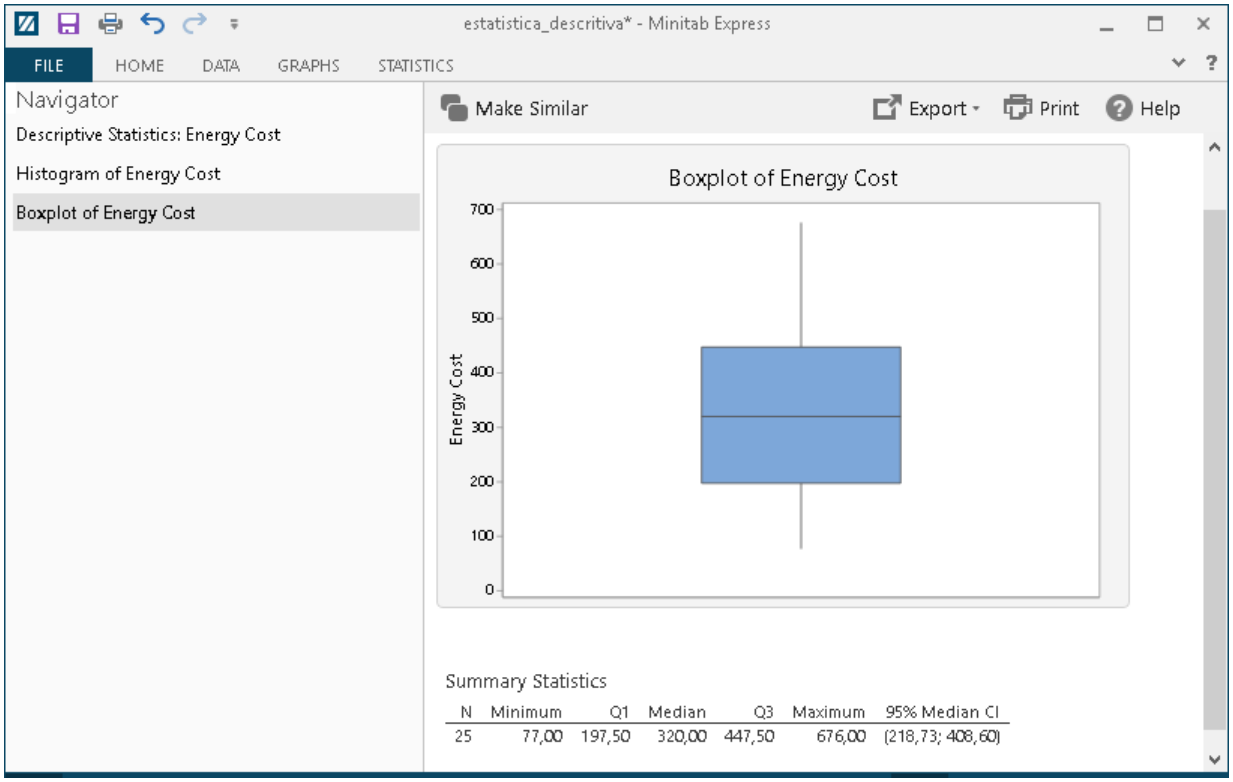

Figura 5: BoxPlot

Por fim, é possível fazer a análise da distribuição de probabilidades, de maneira a identificar se o conjunto de dados segue uma distribuição Normal. A Figura 6 demonstra essa análise no *Minitab*, acionada através da opção *Normality Test*.

Nesta análise, são exibidos os resultados do teste de Anderson Darling, com as variáveis *AD-value* e

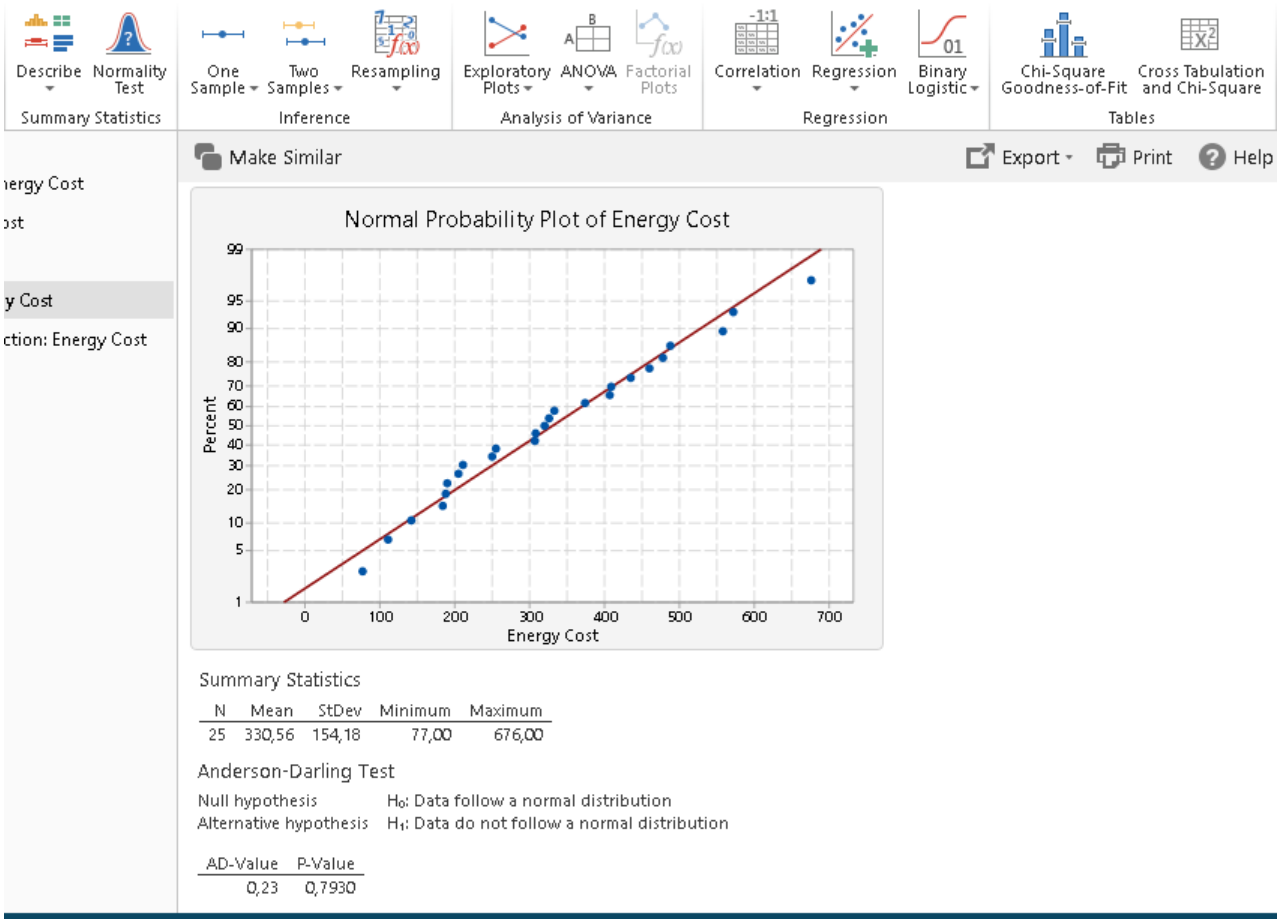

Figura 6: Teste de normalidade

P-value. Para análises mais completas, considerando outras distribuições, é necessário utilizar a versão completa do *Minitab*, disponível para compra.

8 Considerações sobre este documento

Os códigos aqui apresentados utilizaram base de dados de teste, mas podem ser facilmente adaptados para contemplar dados reais (basta mudar as matrizes ou conjuntos inicializados nos códigos).

A seguir, uma lista de arquivos que acompanham este documento, que podem ser usados em futuras implementações:

- 1_matrizes.py
- 1_Matrizes.R
- 2_Espacos_Subespacos.py
- 2_Espacos_Subespacos.R
- 3_PCA_AnáliseFatorial_SVD.py
- 3_PCA_AnáliseFatorial_SVD.R
- 4_RegressaoLinear_Simples_Multivariavel.py
- 4_RegressaoLinear_Simples_Multivariavel.R
- 5_estatistica_descritiva.mpjx
- metodos_quantitativos.tex

Os arquivos com extensão *.R*, são implementações em linguagem R, enquanto aqueles com extensão *.py*, são implementações em Python. Os arquivos com extensão *.mpjx* são arquivos de projeto do *Minitab Express*. Também acompanha o código fonte deste documento.