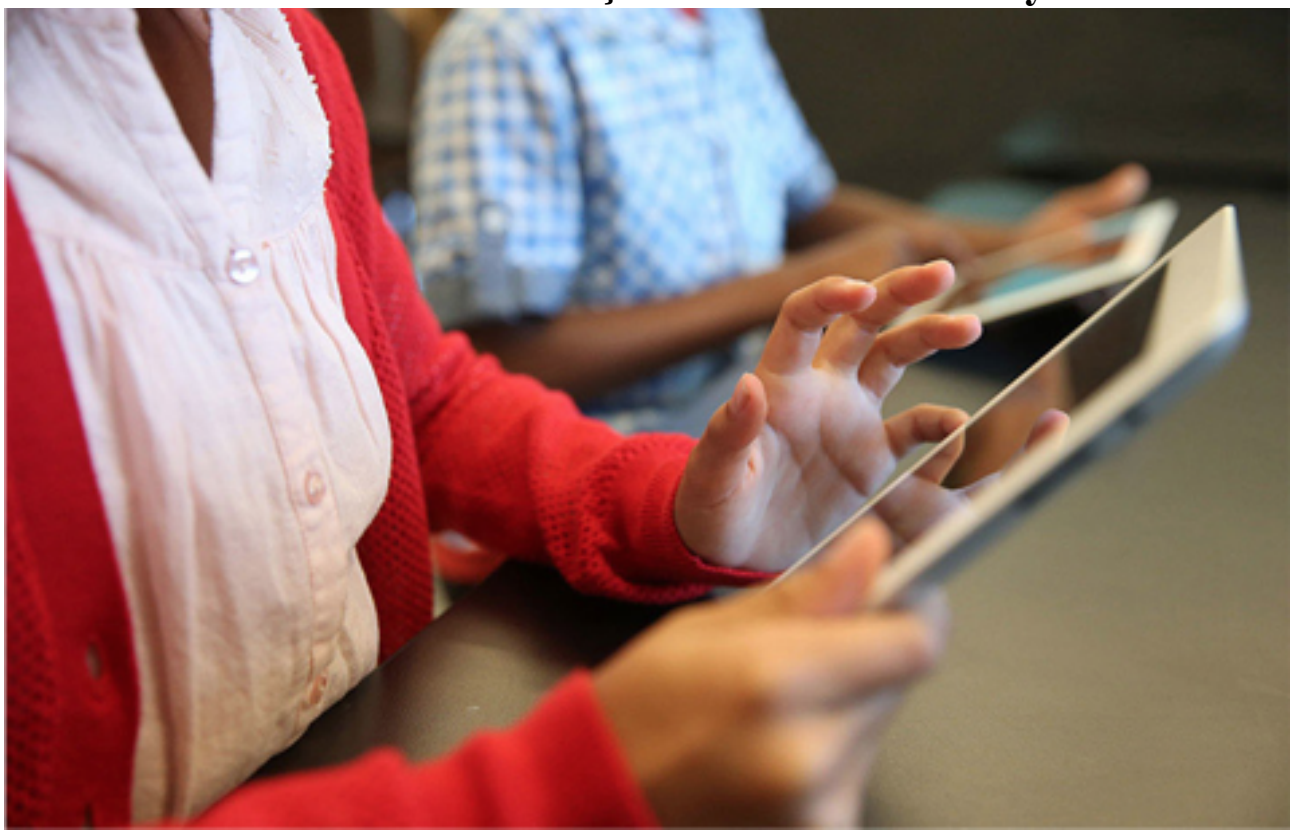
Mineração de dados
Descoberta e Visualização de Padrões com Python



Alan Carlos, Francisco Moura, Marcelo Stefanelli, Robert Carlos

26 de outubro de 2016

Tradução e adaptação do original Data Mining - Discovering and Visualizing Patterns with Python, de Giuseppe Vettigli [7].

Introdução

Mineração de dados ou Data Mining é a extração de informação implícita, previamente desconhecida e potencialmente útil a partir de dados. Ela é aplicada numa grande variedade de domínios e suas técnicas tornaram-se fundamentais para diversas aplicações.

Este guia de referência aborda as ferramentas utilizadas na prática de Mineração de Dados para descobrir e descrever padrões estruturais em dados usando a linguagem de programação Python. Nos últimos anos, Python vendo sendo utilizada cada vez mais para o desenvolvimento de aplicações centrada em dados, graças ao suporte de uma grande comunidade de computação científica e ao aumento do número de bibliotecas disponíveis para análise de dados.

Em particular, será visto:

- Importar e Visualizar de dados
- Classificar e Agrupar dados (Cluster)
- Descobrir relações nos dados utilizando medidas de regressão e correlação
- Reduzir a dimensão dos dados, a fim de comprimir e visualizar as informações que ele traz
- Analisar dados estruturados

Cada tópico será coberto por exemplos de código baseado em quatro das principais bibliotecas Python para análise e manipulação: SciPy (NumPy, Matplotlib) [2], scikit-learn [5] e NetworkX [1].

Ambiente Python

Para executar os códigos-fontes presentes neste guia, é necessário que o ambiente de execução esteja com as versões mais recentes de Python, podendo ser um ambiente configurado com Python 2 ou Python 3.

No momento em que este guia de referência estava sendo escrito, os códigos-fontes foram testados nas seguintes versões de Python:

- **Python 2**: versão 2.7.12
- **Python 3**: versão 3.5.2

Abaixo, são listadas as bibliotecas necessárias, bem como as versões utilizadas neste guia:

Nome da biblioteca	Versão
NumPy	1.11.2
scikit-learn	0.18
scipy	0.18.1
Matplotlib	1.5.3
NetworkX	1.11

Para configurar o ambiente em seu sistema operacional, consulte a documentação no site do Python (www.python.org). No apêndice nós mostramos como instalar as bibliotecas mencionadas neste guia.

Executando o código-fonte

Os exemplos de código poderão ser copiados e executados interativamente no terminal interpretador (console de comandos) do Python, ou executados a partir da invocação de arquivos com extensão **.py** contendo os comandos. O código-fonte também é disponibilizado no apêndice deste guia de referência de modo que poderá ser copiado e salvo em arquivos com extensão .**py**.

Github

Este guia, juntamente com o código-fonte, é disponibilizado no repositório Github - https://github.com/franciscomoura/data-science-and-bigdata. Acesse o item **Mineração de dados com Python**.

Conjunto de dados Íris

Ao longo do guia, utilizaremos o conjunto de dados Íris [3].

O conjunto de dados Íris é um conjunto de dados multivariado, que igualmente divide 150 amostras da flor iris em 3 classes (Setosa, Versicolor e Virgínica). Cada classe possui 50 ocorrências. Cada ocorrência tem quatro características (ou variáveis) sendo, o comprimento e a largura da sépala e pétala, em centímetros e a última coluna contém a descrição da classe. Ao todo, o conjunto de dados tem 5 colunas. As primeiras 4 colunas contêm os valores das características, enquanto que a última coluna representa a classe das amostras (Iris Setosa, Iris Versicolour, Iris Virginica). Uma classe é linearmente separada das outras duas; as duas últimas não são linearmente separadas uma das outras.

Informações sobre os atributos no arquivo:
Coluna 1. sepal length in cm (comprimento da sépala, em centímetros)
Coluna 2. sepal width in cm (largura da sépala, em centímetros)
Coluna 3. petal length in cm (comprimento da pétala, em centímetros)
Coluna 4. petal width in cm (largura da pétala, em centímetros)
Coluna 5. class:
-- Iris Setosa
-- Iris Versicolour
-- Iris Virginica

Iris dataset - https://archive.ics.uci.edu/ml/datasets/Iris

Parte 1 - Importação e Visualização de dados

Normalmente, o primeiro passo de uma análise de dados consiste em obter os dados e realizar a carga destes dados no nosso ambiente de trabalho. Facilmente podemos fazer o download de dados usando a seguinte capacidade do Python:

```
# -*- coding: utf-8 -*-
"""
O arquivo iris.csv será gravado no mesmo diretório do arquivo-fonte
"""
"""
Workarround para executar em ambiente python 2 e python 3
"""
try:
    import urllib.request as url_lib
except ImportError:
    import urllib2 as url_lib

url = 'http://aima.cs.berkeley.edu/data/iris.csv'
remote_file = url_lib.urlopen(url)
with open('iris.csv', 'wb') as local_file:
    local_file.write(remote_file.read())
local_file.close()
```

No código-fonte acima foi utilizada a biblioteca **urllib2** ou a **urllib.request** (dependente do ambiente de execução, se Python 2 ou Python 3) para acessar o arquivo no site da Universidade de Berkley e salvá-lo para o disco usando os métodos do objeto **File** fornecido pela biblioteca padrão do Python.

O arquivo contém o conjunto de dados Íris.

O conjunto de dados é armazenado no formato CSV (valores separados por vírgula). É conveniente analisar o arquivo CSV (fazer o parse) e armazenar as informações que ele contém usando uma estrutura de dados mais apropriada. O CSV pode ser facilmente analisado utilizando a função ***genfromtxt*** da biblioteca **NumPy**:

```
# -*- coding: utf-8 -*-
"""
Executar no mesmo diretório do arquivo iris.csv
"""
import numpy as np
# lê as primeiras 4 colunas
data = np.genfromtxt('iris.csv', delimiter=',', usecols=(0,1,2,3))
# lê a quinta coluna(última)
target_names = np.genfromtxt('iris.csv', delimiter=',', usecols=(4), dtype=str)
```

Neste trecho de código-fonte , foi criada uma matriz com as características das plantas (*data*) e um vector que contém os nomes das classes dessas plantas (*target_names*). Podemos confirmar o tamanho do nosso conjunto de dados olhando a forma das estruturas de dados que nós carregamos, usando o atributo ***shape***:

```
print(data.shape)
# Saída: (150, 4)

print(target_names.shape)
# Saída: (150,)
```

Podemos inspecionar o elemento *target_names* para saber quantas classes ele possui e os seus nomes:

```
# constrói uma coleção contendo elementos únicos
print(set(target_names))

# Saída python 2: set(['setosa', 'versicolor', 'virginica'])
# Saída python 3: {'virginica', 'versicolor', 'setosa'}
```

Uma tarefa importante quando se trabalha com novos dados é tentar entender quais informações estes dados contém e como eles estão estruturados. A visualização ajuda-nos a explorar esta informação graficamente de modo a ganhar a compreensão e visão sobre os dados.

Usando os recursos de plotagem (funções ***plot*** e ***show***) da biblioteca **Pylab** (que é uma interface para **Matplotlib**) podemos construir um gráfico de dispersão bi-dimensional que nos permite analisar duas dimensões do conjunto de dados, traçando os valores de um recurso contra os valores do outro:

```
20 import pylab as pl
21
22 pl.plot(data[target_names == 'setosa', 2], data[target_names == 'setosa', 3], 'bo')
23 pl.plot(data[target_names == 'versicolor', 2], data[target_names == 'versicolor', 3], 'ro')
24 pl.plot(data[target_names == 'virginica', 2], data[target_names == 'virginica', 3], 'go')
25 pl.ylabel('Largura da petala - cm')
26 pl.xlabel('Comprimento da petala - cm')
27 pl.axis([0.5, 7, 0, 3])
28 pl.show()
```

O código-fonte acima usa a terceira e a quarta dimensão do conjunto de dados (comprimento e largura da pétala) e o resultado é mostrado na figura 1.

No gráfico da figura 1 temos 150 pontos e as cores representam as classe das flores; os pontos azuis representam as amostras que pertencem à espécie setosa, os vermelhos representam à espécie versicolor e os verdes representam à espécie virginica.

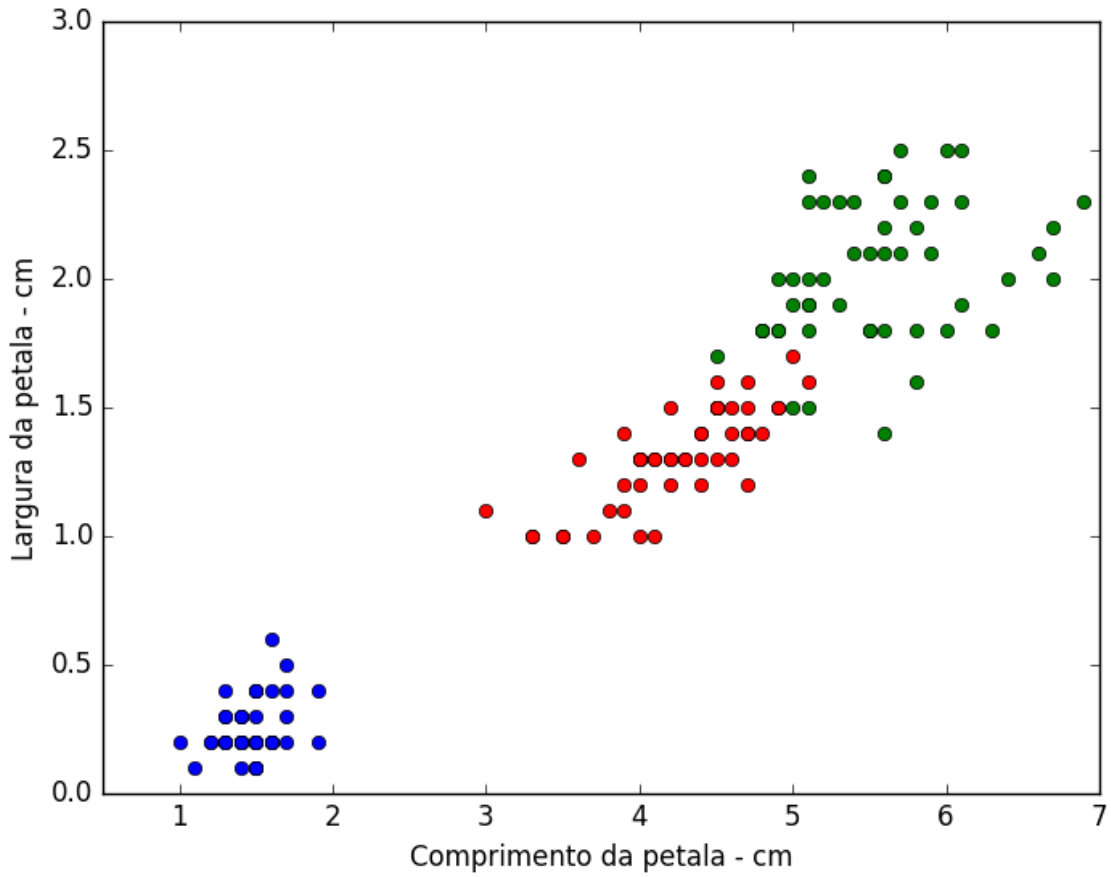


Figura 1: Gráfico de dispersão do comprimento e largura da sépala

Outra forma comum de olharmos para os dados é traçar o histograma das características individuais. Neste caso, uma vez que os dados são divididos em três classes, podemos comparar as distribuições das características que estamos analisando de cada classe. Com o código-fonte abaixo podemos traçar a distribuição da primeira característica de nossos dados (comprimento da sépala) para cada classe:

```
import pylab as pl
xmin = min(data[:, 0])
xmax = max(data[:, 0])
pl.figure()
pl.subplot(411) # distribuição da classe setosa (primeira, no topo)
pl.hist(data[target_names == 'setosa', 0], color='b', alpha=.7)
pl.xlim(xmin, xmax)
pl.xlabel('Comprimento da sepala - cm')
pl.ylabel('Ocorrencias')

pl.subplot(412) # distribuição da classe setosa versicolor class (segunda)
pl.hist(data[target_names == 'versicolor', 0], color='r', alpha=.7)
pl.xlim(xmin, xmax)
pl.xlabel('Comprimento da sepala - cm')
pl.ylabel('Ocorrencias')

pl.subplot(413) # distribuição da classe setosa virginica class (terceira)
pl.hist(data[target_names == 'virginica', 0], color='g', alpha=.7)
pl.xlim(xmin, xmax)
pl.xlabel('Comprimento da sepala - cm')
pl.ylabel('Ocorrencias')

pl.subplot(414) # histograma global (quarto, último)
pl.hist(data[:, 0], color='y', alpha=.7)
pl.xlim(xmin, xmax)
pl.xlabel('Comprimento da sepala - cm')
pl.ylabel('Ocorrencias')

pl.show()
```

E o resultado pode ser visualizado no histograma da figura 2.

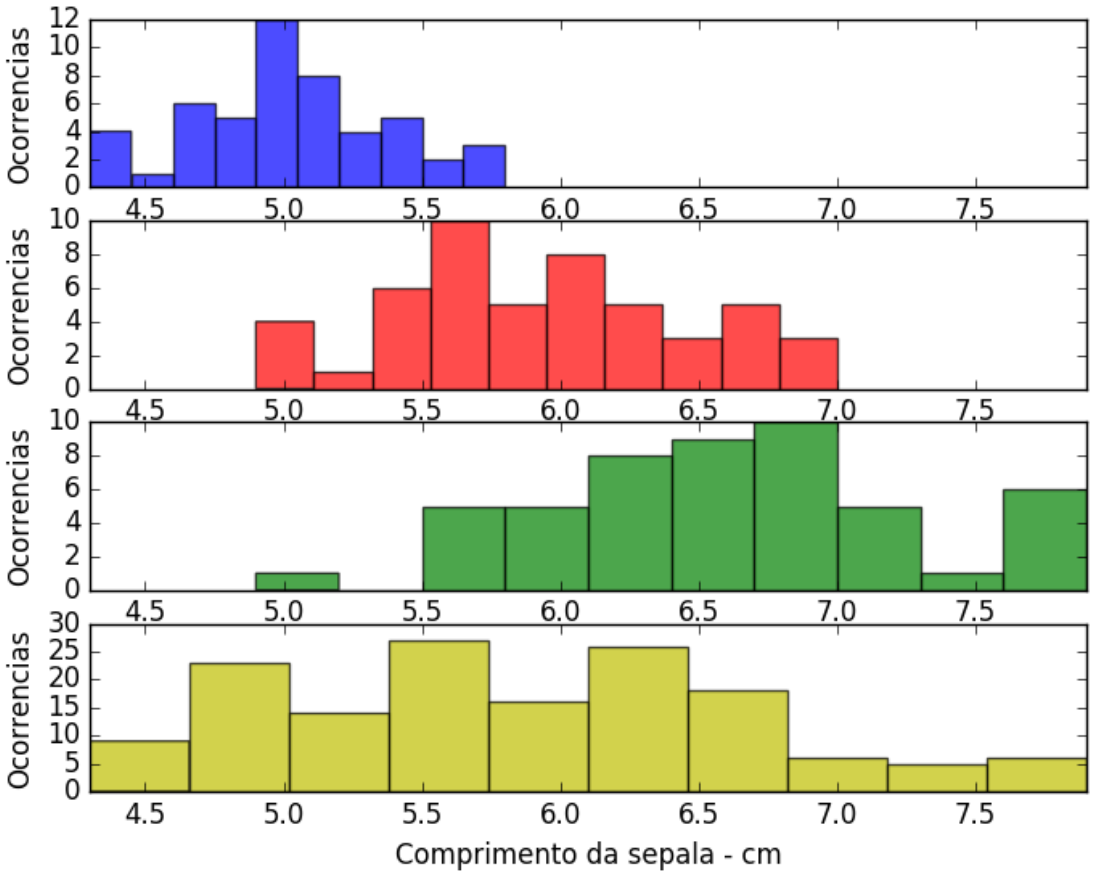


Figura 2: Histograma do Comprimento da sépala - cm x Número de ocorrências

Olhando para os histogramas acima, podemos entender algumas características que poderiam nos ajudar a distinguir os

dados de acordo com as classes que temos. Por exemplo, pode-se observar que, em média, as plantas íris setosa têm um comprimento de sépala menor em comparação com as plantas íris virginica.

Parte 2 - Classificação

Classificação é uma função de mineração de dados que indica a classe que um determinado registro pertence. Os modelos que implementam esta função são chamados de classificadores. Há dois passos básicos para usar um classificador: treinamento e classificação. O treinamento é o processo de obtenção de dados que são conhecidos por pertencer à classes específicas e a criação de um classificador com base nos dados conhecidos. Classificação é o processo de tomar um classificador construído com o conjunto de dados de treinamento e executá-lo em dados desconhecidos para determinar a associação de classe para as amostras desconhecidas.

A biblioteca **scikit-learn** contém a implementação de muitos modelos para a classificação e nesta seção veremos como usar o ***Gaussian Naive Bayes***, a fim de identificar as plantas íris como setosa, versicolor ou virginica, usando o conjunto de dados da primeira parte. Para este fim, vamos converter o vetor de strings que contém a classe em números inteiros:

```
58 # converter o vetor de strings que contêm a classe em números inteiros
59 target = np.zeros(len(target_names), dtype=np.int)
60 target[target_names == 'setosa'] = 0
61 target[target_names == 'versicolor'] = 1
62 target[target_names == 'virginica'] = 2
```

Agora estamos prontos para instanciar e treinar nosso classificador:

```
63 from sklearn.naive_bayes import GaussianNB
64 # instanciar e treinar o classificador
65 classifier = GaussianNB()
66 # treinar no conjunto de dados iris
67 classifier.fit(data, target)
```

A classificação pode ser feita com o método de predição e é fácil testá-lo com uma amostra:

```
68 print(classifier.predict(data[0].reshape(1, -1)))
69 # Saída: [1]
70
71 print(target[0])
72 # Saída: 1
```

Neste caso, a classe prevista é igual a correta (setosa), mas é importante avaliar o classificador em uma gama mais ampla de amostras e testá-lo com dados não usados no processo de treinamento. Para este fim, dividir os dados em conjunto de treinamento e conjunto de teste, colhendo amostras aleatoriamente da base de dados original. Vamos usar o primeiro conjunto para treinar o classificador e o segundo para testar o classificador. A função ***train_test_split*** pode fazer isso por nós:

```
73 # dividir o conjunto de dados em amostras de treinamento e teste
74 from sklearn.model_selection import train_test_split
75 data_train, data_test, target_train, target_test = train_test_split(data, target, test_size=0.4,
   ↪  random_state=0)
```

O conjunto de dados foi dividido e o tamanho do conjunto de ensaio (teste) é de 40% do tamanho do original como especificado com parâmetro da função *test_size*. Com esses dados podemos novamente treinar o classificador e imprimir a sua precisão:

```
76 # treinar o classificador com os conjuntos de treino específicos
77 classifier.fit(data_train, target_train) # treino
78 print(classifier.score(data_test, target_test)) # teste
79 # Saída: 0.93333333333333335
```

Neste caso, temos 93% de precisão. A precisão de um classificador é determinado pelo número de amostras classificadas correctamente dividido pelo número total de amostras classificadas. Em outras palavras, isto significa que é a proporção do número total de previsões que eram correctos.

Outra ferramenta para estimar o desempenho de um classificador é a matriz de confusão. Nesta matriz cada coluna representa as instâncias de uma classe prevista, enquanto cada linha representa as instâncias de uma classe real. Usando as métricas de módulo, é muito fácil de calcular e imprimir a matriz:

```
80 # matriz de confusão
81 from sklearn.metrics import confusion_matrix
82 print(confusion_matrix(classifier.predict(data_test),target_test))
```

Saída:

```
[[16  0  0]
 [ 0 23  4]
 [ 0  0 17]]
```

Nesta matriz de confusão, podemos ver que todas as flores Iris setosa e virginica foram classificadas corretamente, mas, das reais 27 flores Iris versicolor, o sistema previu que 4 eram virginica. Se levarmos em conta que todas as estimativas correctas estão localizadas na diagonal da tabela, que é fácil de inspecionar visualmente a tabela de erros, uma vez que estão representados pelos valores não nulos fora da diagonal.

Uma função que nos dá um relatório completo sobre o desempenho do classificador está também disponível:

```
83 # relatório de classificação
84 from sklearn.metrics import classification_report
85 print(classification_report(classifier.predict(data_test), target_test, target_names=['setosa',
    ↪  'versicolor', 'virginica']))
```

Gerando a seguinte saída:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	16
versicolor	1.00	0.85	0.92	27
virginica	0.81	1.00	0.89	17
av / total	0.95	0.93	0.93	60

Aqui está um resumo das medidas utilizadas pelo relatório:

- ***Precision* (precisão)**: a proporção de casos positivos previstos que estavam corretos (A medida de precisão mensura a quantidade de casos que foram corretamente classificados como positivos dentre todos os casos que foram julgados como positivos [6]).
- ***Recall* (abrangência/cobertura/sensibilidade)** (ou também *true positive rate*): a proporção de casos positivos que foram corretamente identificados (ao contrário da precisão, é a relação entre a quantidade de casos que foram corretamente classificados como positivos dentre todos os casos que são realmente da classe de positivos [6]).
- ***F1-Score*** (**medida F**): a média harmônica entre os valores de Precision e Recall.

O suporte é simplesmente a quantidade de elementos da classe dada utilizada para o teste. No entanto, dividindo os dados, podemos reduzir o número de amostras que podem ser utilizadas para o treinamento, e os resultados da avaliação podem depender de uma escolha aleatória particular para o par (conjunto de treinamento, conjunto de teste).

Para avaliar realmente um classificador e compará-lo com outros, nós temos que usar um modelo de avaliação mais sofisticado como ***Cross Validation*** (Validação Cruzada). A ideia por trás do modelo é simples: o dado é dividido em conjuntos de treinamento e teste várias vezes consecutivas e o valor médio das contagens da predição obtidos com os diferentes conjuntos consiste na avaliação do classificador. A biblioteca **scikit-learn** nos fornece uma função para executar o modelo:

```
86 from sklearn.model_selection import cross_val_score
87 # cross validation com 6 iterações
88 scores = cross_val_score(classifier, data, target, cv=6)
89 print(scores)
90 # Saída: [ 0.92592593  1.     0.91666667  0.91666667  0.95833333  1.   ]
```

Como podemos ver, a saída desta implementação é um vector que contém a precisão obtida com cada iteração do modelo. Podemos facilmente calcular a precisão média da seguinte forma:

```
91 print(np.mean(scores))
92 # Saída: 0.952932098765
```

Parte 3 - Agrupamento (*Clustering*)

Muitas vezes não temos rótulos anexados aos dados que nos dizem a classe das amostras; temos que analisar os dados, a fim de agrupa-los com base em critérios de semelhança/similaridade onde grupos (clusters) são conjuntos de amostras similares. Este tipo de análise é chamado de análise não supervisionada de dados. Uma das mais famosas ferramentas de agrupamento é o algoritmo k-médias (k-means), que pode ser executado da seguinte forma:

```
93  from sklearn.cluster import KMeans
94  # inicialização correta para o cluster mostrar o mesmo resultado a cada execução
95  kmeans = KMeans(n_clusters=3, init="k-means++", random_state=3425)
96  kmeans.fit(data)
```

O trecho de código acima executa o algoritmo e agrupa os dados em 3 *clusters* (conforme especificado pelo parâmetro *n_clusters*). Agora podemos usar o modelo para identificar cada amostra em um dos *clusters* (agrupamentos):

```
97  clusters = kmeans.predict(data)
```

E podemos avaliar os resultados de agrupamento (*clustering*), comparando-os com os rótulos que já temos com a contagem da completude e homogeneidade (*completeness and the homogeneity score*):

```
98  from sklearn.metrics import completeness_score, homogeneity_score
99  print(completeness_score(target, clusters))
100 # Saída: 0.764986151449
101
102 print(homogeneity_score(target, clusters))
103 # Saída: 0.751485402199
```

A contagem da completude se aproxima de 1, quando a maioria dos pontos de dados que são membros de uma determinada classe são elementos do mesmo grupo, enquanto a contagem da homogeneidade se aproxima de 1, quando todos os agrupamentos contêm apenas os dados que são membros de uma única classe.

Definição:
Homogeneidade: cada *cluster* contém apenas membros de uma única classe.
Completude: todos os membros de uma determinada classe são atribuídos ao mesmo grupo.
sckit-learn - http://scikit-learn.org/stable/modules/clustering.html

Também podemos visualizar o resultado do agrupamento e comparar as designações com os rótulos reais visualmente:

```
104 import pylab as pl
105
106 pl.figure()
107
108 pl.subplot(211)  # topo, figura com as classes reais
109 pl.plot(data[target == 0, 2], data[target == 0, 3], 'bo', alpha=.7)  # 0 setosa
110 pl.plot(data[target == 1, 2], data[target == 1, 3], 'ro', alpha=.7)  # 1 versicolor
111 pl.plot(data[target == 2, 2], data[target == 2, 3], 'go', alpha=.7)  # 2 virginica
112 pl.xlabel('Comprimento da petala - cm')
113 pl.ylabel('Largura da petala - cm')
114 pl.axis([0.5, 7, 0, 3])
115
116 pl.subplot(212)  # embaixo, figura com as classes atribuídas automaticamente
117 pl.plot(data[clusters == 0, 2], data[clusters == 0, 3], 'go', alpha=.7)  # clusters 0 verginica
118 pl.plot(data[clusters == 1, 2], data[clusters == 1, 3], 'bo', alpha=.7)  # clusters 1 setosa
119 pl.plot(data[clusters == 2, 2], data[clusters == 2, 3], 'ro', alpha=.7)  # clusters 2 versicolor
120
121 pl.xlabel('Comprimento da petala - cm')
122 pl.ylabel('Largura da petala - cm')
123 pl.axis([0.5, 7, 0, 3])
124
125 pl.show()
```

O gráfico mostra o resultado:

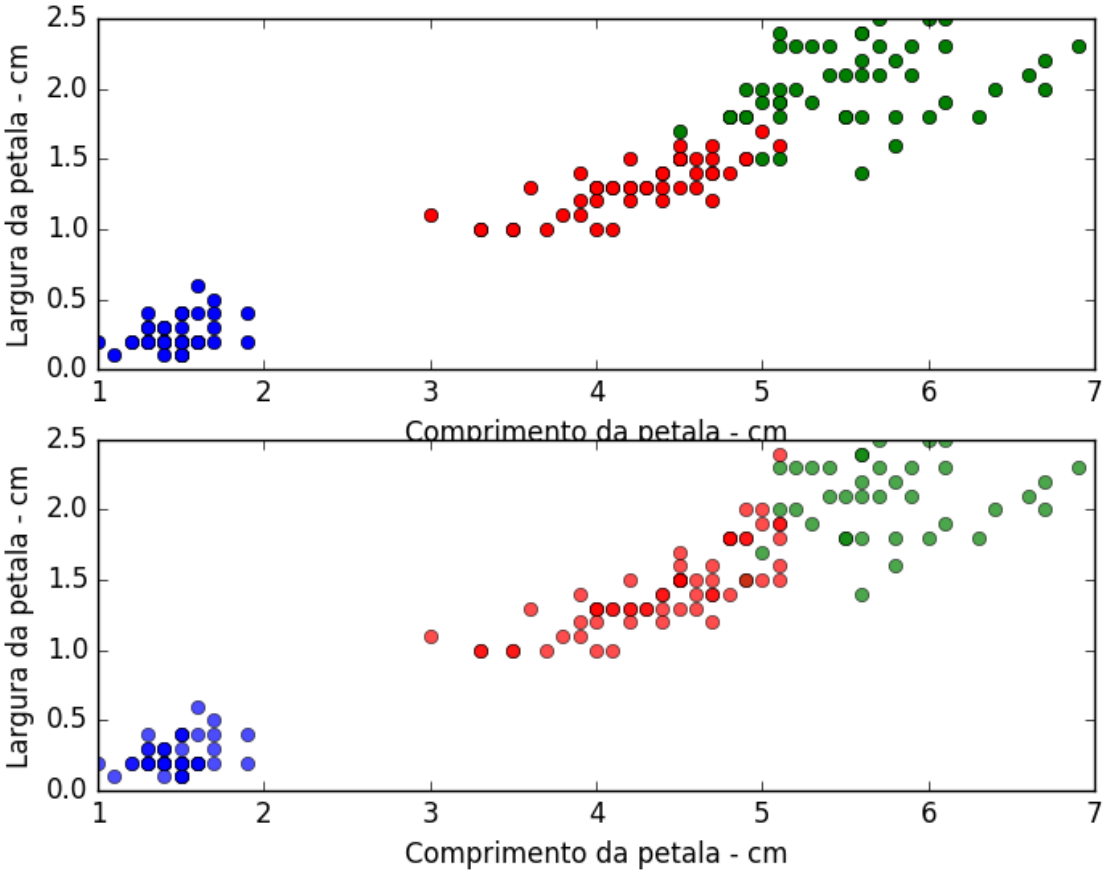


Figura 3: Agrupamento k-means

Observando o gráfico, vemos que o agrupamento da espécie setosa (cor azul) foi completamente identificado por k-means, enquanto os outros dois agrupamentos foram identificados com alguns erros.

Parte 4 - Regressão

A regressão é um método para investigar as relações funcionais entre as variáveis que podem ser usadas para fazer predições. Consideremos o caso em que temos duas variáveis, uma é considerada para ser explicativa, e a outra é considerada para ser um dependente. Queremos descrever a relação entre as variáveis usando um modelo; Quando esta relação é expressa com uma linha temos a regressão linear.

De modo a aplicar a regressão linear, construímos um conjunto de dados sintético composto tal como descrito acima:

```
1 import numpy as np
2 x = np.random.rand(40,1) # variável explicativa
3 y = x*x*x+np.random.rand(40,1)/5 # variável dependente
```

Agora podemos usar o modelo de LinearRegression que encontramos no módulo sklearn.linear_model. Este modelo calcula a linha de melhor ajuste para os dados observados, minimizando a soma dos quadrados dos desvios verticais a partir de cada ponto de dados para a linha. O uso é semelhante aos outros modelos implementados em sckit-learn que foi visto previamente:

```
4 from sklearn.linear_model import LinearRegression
5 linreg = LinearRegression()
6 linreg.fit(x,y)
```

E podemos traçar esta linha ao longo dos pontos de dados reais para avaliar o resultado:

```
7 import pylab as pl
8 xx = np.linspace(0,1,40)
9 pl.plot(x, y, 'o', xx, linreg.predict(np.matrix(xx).T),'--r')
10 pl.show()
```

A plotagem deve ser a seguinte:

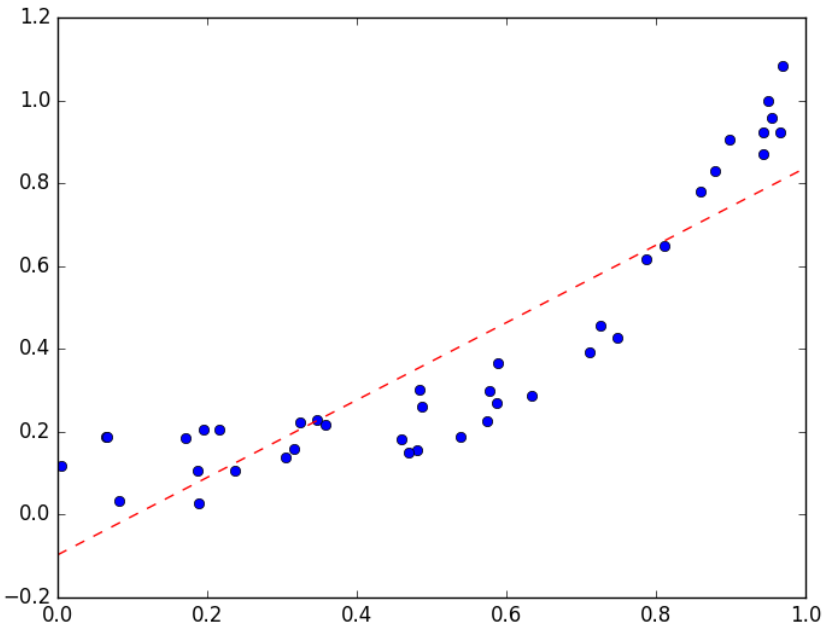


Figura 4: Regressão linear

Neste gráfico podemos observar que a linha atravessa o centro de nossos dados e nos permite identificar a tendência crescente.

Também podemos quantificar como o modelo se ajusta aos dados originais usando o erro médio quadrático:

```
from sklearn.metrics import mean_squared_error
print(mean_squared_error(linreg.predict(x),y))
# Saída: 0.0202788893782
```

Esta métrica mede o quadrado da distância esperada entre a predição e os verdadeiros dados. É 0 (zero) quando a predição é perfeita.

Parte 5 - Correlação

Estudamos a correlação para entender como e se pares de variáveis estão fortemente relacionados. Este tipo de análise nos ajuda a localizar as variáveis criticamente importantes das quais outros dependem. A melhor medida de correlação é o coeficiente de correlação produto-momento de Pearson. É obtido dividindo a covariância das duas variáveis pelo produto dos seus desvios padrão. Podemos calcular este índice entre cada par de variáveis para o conjunto de dados das flores íris da seguinte forma:

```
126 corr = np.corrcoef(data.T) # T. dá a transposição
127 print(corr)
```

Gerando a seguinte saída:

```
[[1.          -0.10936925  0.87175416  0.81795363 ]
[-0.10936925  1.          -0.4205161  -0.35654409]
[0.87175416  -0.4205161   1.          0.9627571  ]
[0.81795363  -0.35654409  0.9627571   1.         ]]
```

A função ***corrcoef*** retorna uma matriz simétrica de coeficientes de correlação calculados a partir de uma matriz de entrada em que as linhas são variáveis e colunas são observações. Cada elemento da matriz representa a correlação entre duas variáveis.

A correlação é positiva quando os valores aumentam em conjunto. É negativo quando um valor diminui à medida que os outros aumentam. Em particular, temos que 1 é uma correlação positiva perfeita, 0 (zero) não há correlação e -1 é uma correlação negativa perfeita.

Quando o número de variáveis cresce podemos visualizar convenientemente a matriz de correlação utilizando uma plotagem pseudocores (pseudocolor):

```
128 import pylab as pl
129 pl.pcolor(corr)
130 pl.colorbar() # adicionar barra de cores
131 # organiza os nomes das variáveis nos eixos cartesianos
132 pl.xticks(np.arange(0.5, 4.5), ['sepal length', 'sepal width', 'petal length', 'petal width'],
      ↪  fontsize=6)
133 pl.yticks(np.arange(0.5, 4.5), ['sepal length', 'sepal width', 'petal length', 'petal width'],
      ↪  fontsize=6)
134 pl.grid(True)
135 pl.title('Coeficiente de correlacao plantas Iris')
136 pl.show()
```

A imagem seguinte mostra o resultado:

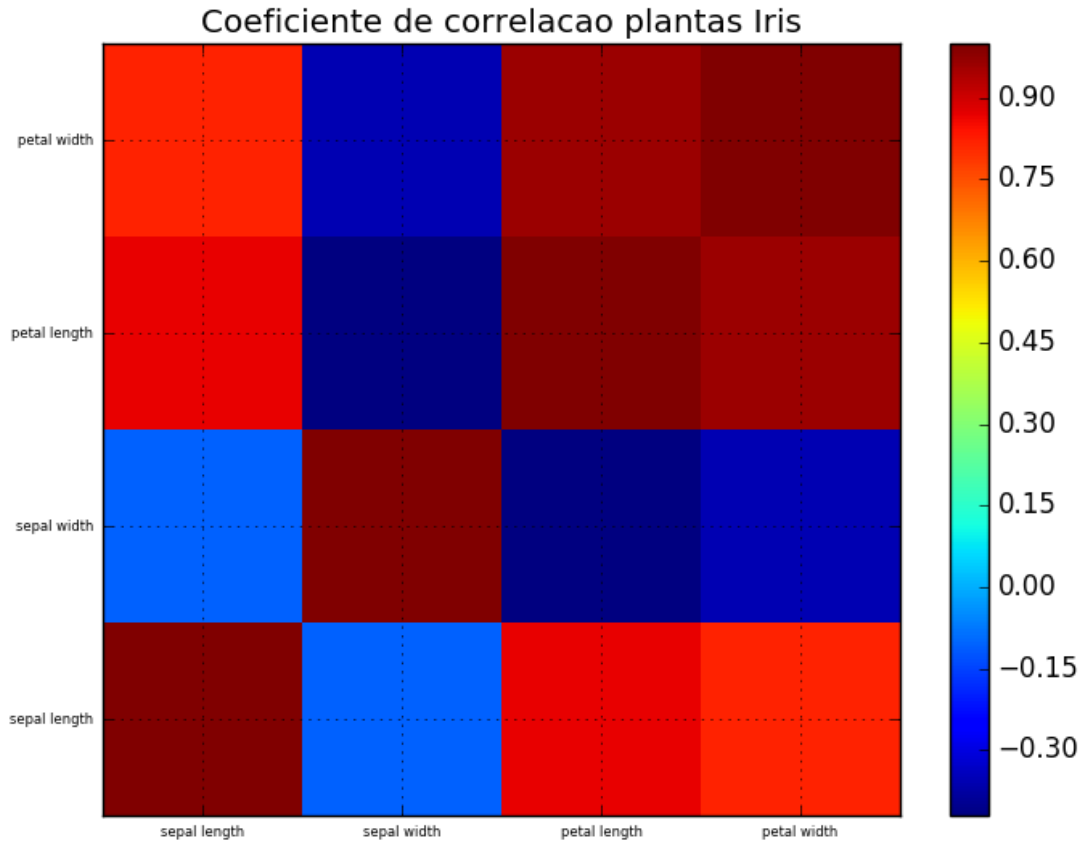


Figura 5: Coeficiente de Correlação

Olhando para a barra de cores à direita da figura, podemos associar a cor na plotagem para um valor numérico. Neste caso, o vermelho é associado com altos valores de correlação positiva e podemos ver que a correlação mais forte em nosso conjunto de dados é entre as variáveis ”largura da pétala (petal width)” e ”comprimento pétala (petal length)”.

Parte 6 - Redução de dimensionalidade

Na primeira parte deste estudo, vimos como visualizar duas dimensões do conjunto de dados das flores íris. Com esse método sozinho, temos uma visão de apenas uma parte do conjunto de dados. Uma vez que o número máximo de dimensões que podemos plotar, ao mesmo tempo, é 3, para ter uma visão global dos dados é necessário incorporar todos os dados em um número de dimensões que podemos visualizar. Este processo de incorporação é chamado de redução de dimensionalidade. Uma das mais famosas técnicas de redução de dimensionalidade é a Análise de Componentes Principais (PCA). Esta técnica transforma as variáveis de nossos dados em um número igual ou menor de variáveis não correlacionadas chamados componentes principais (PCs).

Por sua vez, a biblioteca **sckit-learn** fornece-nos tudo o que precisamos para realizar nossa análise:

```
137 from sklearn.decomposition import PCA
138 pca = PCA(n_components=2)
```

No código acima, instanciamos um objeto PCA que podemos usar para calcular os dois primeiros PCs.

A transformação é calculado como se segue:

```
139 pcad = pca.fit_transform(data)
```

E podemos plotar o resultado, como de costume:

```
140 import pylab as pl
141 pl.plot(pcad[target_names == 'setosa', 0], pcad[target_names == 'setosa', 1], 'bo')
142 pl.plot(pcad[target_names == 'versicolor', 0], pcad[target_names == 'versicolor', 1], 'ro')
143 pl.plot(pcad[target_names == 'virginica', 0], pcad[target_names == 'virginica', 1], 'go')
144 pl.show()
```

O resultado é o seguinte:

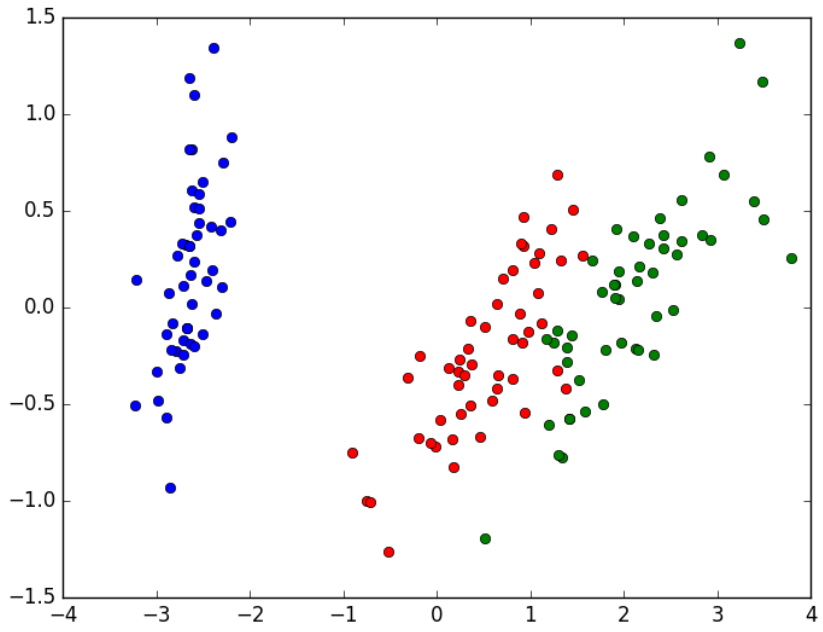


Figura 6: Redução de dimensionalidade

Notamos que a figura acima é semelhante ao proposto na primeira parte, mas desta vez a separação entre a espécie versicolor (em vermelho) e da espécie virginica (em verde) é mais clara.

O PCA projeta os dados em um espaço onde a variância é maximizada e podemos determinar a quantidade de informação armazenada nos PCs olhando para a razão de variância:

```
145 print(pca.explained_variance_ratio_)
146 # Saída: [ 0.92461621  0.05301557]
```

Agora sabemos que o primeiro PC responde por 92% das informações do conjunto de dados original, enquanto o segundo representa os restantes 5%. Nós também podemos imprimir a quantidade de informação que perdemos durante o processo de transformação:

```
147 print(1-sum(pca.explained_variance_ratio_))
148 # Saída: 0.0223682249752
```

Neste caso, perdemos 2% da informação.
Neste ponto, podemos aplicar a transformação inversa para obter os dados originais de volta:

```
149 data_inv = pca.inverse_transform(pcad)
```

Sem dúvida, a transformação inversa não nos dá exatamente os dados originais, devido à perda de informação. Podemos estimar quanto o resultado do inverso é provável para os dados originais da seguinte forma:

```
150 print(abs(sum(sum(data - data_inv))))
151 # Saída: 2.88657986403e-15
```

Temos que a diferença entre os dados originais e à aproximação calculado com a transformação inversa está perto de zero. É interessante notar a quantidade de informação que se pode preservar, variando o número de componentes principais:

```
152 for i in range(1, 5):
153     pca = PCA(n_components=i)
154     pca.fit(data)
155     print("{0:1d} {1:11} = {2:.5f}%".format(i, "componente" if i == 1 else "componentes",
          ↪  sum(pca.explained_variance_ratio_) * 100))
```

A saída deste código é a seguinte:

```
1 componente  = 92.46162%
2 componentes = 97.76318%
3 componentes = 99.48169%
4 componentes = 100.00000%
```

Quanto mais PCs usamos mais a informação é preservada, mas esta análise nos ajuda a entender quantos componentes podemos usar para salvar uma certa quantidade de informações. Por exemplo, a partir da saída do código acima, podemos ver que no conjunto de dados das flores Iris podemos economizar quase 100% da informação usando apenas três PCs.

Parte 7 - Mineração de dados em redes

Muitas vezes, os dados que temos que analisar é estruturado na forma de redes, por exemplo nossos dados poderiam descrever as amizades entre um grupo de usuários do Facebook ou as co-autorias de artigos entre os cientistas. Aqui, os objetos para estudos são descritos por nós e por arestas que descrevem conexões entre eles.

Nesta parte, veremos os passos básicos para a análise desse tipo de dados usando NetworkX, que é uma biblioteca que nos ajuda na criação, a manipulação e o estudo das redes. Em particular, vamos ver como usar uma medida de centralidade, a fim de construir uma visualização significativa dos dados e como encontrar um grupo de nós, onde as conexões são densas.

Precisamos de uma rede para explorar. Poderíamos construir uma adicionando nós e arestas um de cada vez, mas a NetworkX pode ler dados em muitos formatos diferentes e analisá-lo em sua representação gráfico interna. Um formato padrão de dados de rede é o formato de texto Graph Modeling Language (GML). Existem vários repositórios públicos de dados de rede interessantes formatados em GML; de um destes nós utilizaremos um gráfico das relações entre os personagens do romance Les Miserables - https://gephi.org/datasets/lesmiserables.gml.zip.
No entanto, este conjunto de dados está formatado de maneira que será lido somente com a versão 1.9.1 e anteriores da NetworkX. Felizmente, podemos reformatá-lo para que as versões mais novas desta biblioteca possam executar as análises como esperado. No apêndice encontra-se o procedimento para adequação do formato GML.

Usando NetworkX, podemos facilmente importar os dados estruturados em GML e visualizar a rede carregada:

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3 G = nx.read_gml('lesmiserables.gml') # lê o arquivo gml
4 nx.draw_networkx(G, alpha = 0.5, font_size = 10) # desenha a rede
5 plt.show() # plota em tela o gráfico da rede
```

O resultado deve ser o seguinte:

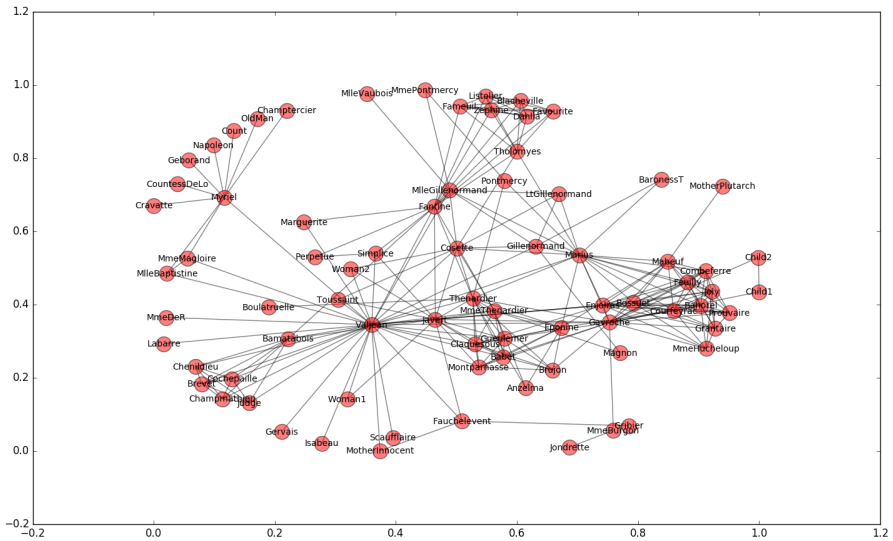


Figura 7: Rede Les Miserables

Nesta rede cada nó representa um personagem do romance e a conexão entre dois personagens representa a aparição no mesmo capítulo. É fácil ver que o gráfico não é realmente útil. A maioria dos detalhes da rede ainda estão ocultas e é impossível compreender quais são os nós mais importantes. A fim de ganhar alguns insights sobre os nossos dados,

podemos estudar o grau dos nós. O grau de um nó é considerado uma das mais simples medidas de centralidade e que consiste do número de ligações que um nó tem. Podemos resumir a distribuição de graus de uma rede verificando os seus valores de máximo, mínimo, mediana, primeiro quartil e terceiro quartil:

```
import numpy as np
deg = nx.degree(G)
values = list(deg.values())
print(np.min(values))
print(np.percentile(values, 25)) # calcula o primeiro quartil
print(np.median(values))
print(np.percentile(values, 75)) # calcula o terceiro quartil
print(np.max(values))
```

Saída:

```
1
2.0
6.0
10.0
36
```

A partir desta análise, podemos decidir observar apenas os nós com um grau maior do que 10. A fim de exibir somente os nós podemos criar um novo gráfico com apenas os nós que queremos visualizar:

```
Gt = G.copy()
dn = nx.degree(Gt)
for n in Gt.nodes():
    if dn[n] <= 10:
        Gt.remove_node(n)
nx.draw_networkx(Gt, alpha = 0.5, font_size = 10) # desenha a rede
plt.show()
```

A imagem abaixo mostra o resultado:

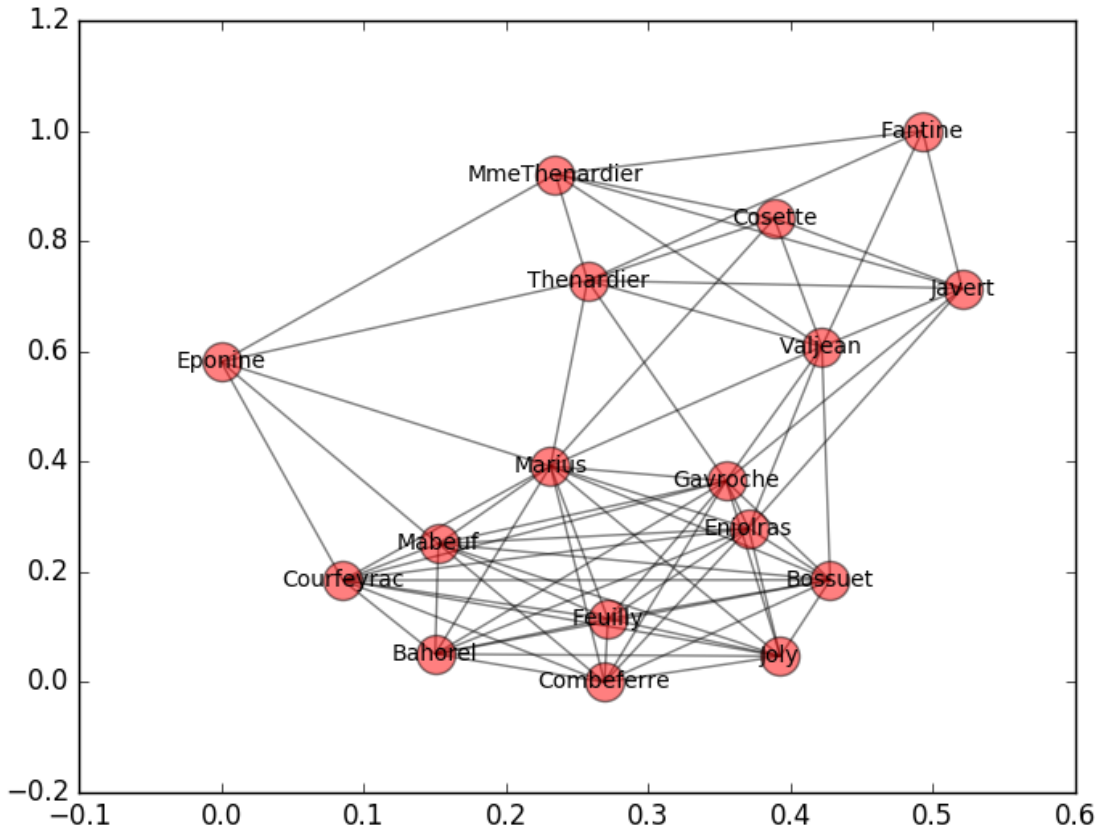


Figura 8: Rede 2

Desta vez, o gráfico é mais legível. Somos capazes de observar os personagens mais relevantes e seus relacionamentos.

Também é interessante estudar a rede através da identificação dos seus cliques. Um clique é um grupo em que um nó é conectado a todos os outros e um clique máximo é um clique que não é um subconjunto de qualquer outra clique na rede. Podemos encontrar todos os cliques máximos da nossa rede da seguinte forma:

```
21 cliques = list(nx.find_cliques(G))
22 print(max(cliques, key=lambda l: len(l)))
```

e podemos visualizar o maior clique na seguinte saída:

[u'Joly', u'Gavroche', u'Bahorel', u'Enjolras', u'Courfeyrac', u'Bossuet', u'Combeferre', u'Feuilly', u'Prouvaire', u'Grantaire']

Podemos ver que a maioria dos nomes na lista são os mesmos do cluster de nós do gráfico anterior.

Parte 8 - Apêndice

Instalando as bibliotecas Python

Este guia assume que já existe um ambiente Python instalado e executando. O comando ***pip*** também está disponível. Se necessário, consulte o site do pip - https://pip.pypa.io/en/stable/installing/.

Consultando versão instalada do pip:

```
$ pip --version
```

ou

```
$ pip3 --version
```

Atualizando o pip:

```
$ pip install -U pip
```

ou

```
$ pip install --upgrade pip
```

Instalando módulos:

```
$ pip install <nome_do_modulo>
```

Procurando módulos do pip:

```
$ pip search <nome>
```

Reformatando arquivos GML [4]

Formato suportado pela NetworkX até a versão 1.9.1:

```
graph
[
  directed 0
  node
  [
    id 0
    label "Beak"
  ]
  .
  .
  .
```

Formato suportado pelas versões mais recentes, a partir da versão 1.9.1:

```
graph [
  directed 0
  node [
    id 0
    label "Beak"
  ]
  .
  .
  .
```

Para formatar o conteúdo do arquivo gml e deixá-lo no formato suportado pelas versões mais recentes da NetworkX, substitua os caracteres fim de linha e abre colchetes (\n [) por um espaço seguido de um abre colchetes ([). Utilizando um editor de textos que permite substituir caracteres invisíveis, como o Notepad++ no Windows, TextWrangler no macOS ou o Vim no Linux, substitua as seguintes sequências de caracteres:

- ***fim de linha***, seguido por ***um abre colchetes***:

```
\n
[
```

- e ***fim de linha***, seguido de *dois espaços* e ***um abre colchetes***:

```
\n
␣␣[
```

por ***um espaço*** seguido de ***um abre colchetes***:

```
␣[
```

Substitua cada sequência uma por vez!

Vídeo sobre mineração de dados no governo federal

Big Data no Planalto: como o governo minera datasets gigantescos para reprimir crimes (https://www.infoq.com/br/presentations/big-data-no-planalto-como-o-governo-minera-datasets)

Referências Bibliográficas

[1] Hagberg, A. A., Schult, D. A., and Swart, P. J. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)* (Pasadena, CA USA, Aug. 2008), pp. 11–15.

[2] Jones, E., Oliphant, T., Peterson, P., et al. SciPy: Open source scientific tools for Python, 2001–. [Online; acessado em 2016-10-26].

[3] Lichman, M. UCI machine learning repository, 2013.

[4] Makan. Unexpected error reading gml graph, oct 2015. [Online; acessado em 2016-10-26].

[5] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[6] Santos, F. L. d. Mineração de opinião em textos opinativos utilizando algoritmos de classificação.

[7] Vettigli, G. Practical Data Mining with Python. [Online; acessado em 2016-10-26].