

Machine Learning na Prática

Modelos em Python

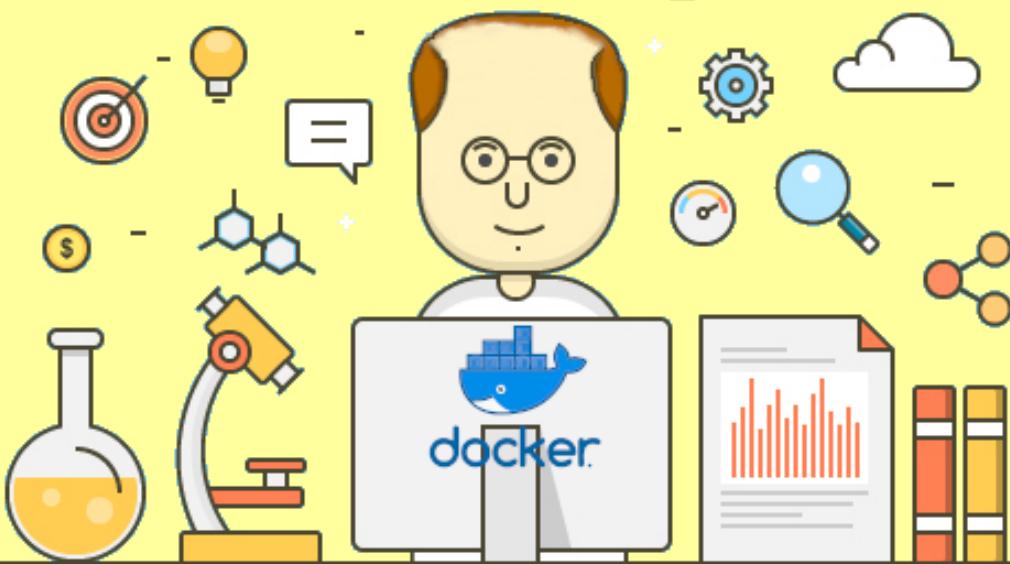
Fernando Anselmo

Copyright © 2020 Fernando Anselmo - v1.0

PUBLICAÇÃO INDEPENDENTE

<http://fernandoanselmo.orgfree.com>

É permitido a total distribuição, cópia e compartilhamento deste arquivo, desde que se preserve os seguintes direitos, conforme a licença da *Creative Commons 3.0*. Logos, ícones e outros itens inseridos nesta obra, são de responsabilidade de seus proprietários. Não possuo a menor intenção em me apropriar da autoria de nenhum artigo de terceiros. Caso não tenha citado a fonte correta de algum texto que coloquei em qualquer seção, basta me enviar um e-mail que farei as devidas retratações, algumas partes podem ter sido cópias (ou baseadas na ideia) de artigos que li na Internet e que me ajudaram a esclarecer muitas dúvidas, considere este como um documento de pesquisa que resolvi compartilhar para ajudar os outros usuários e não é minha intenção tomar crédito de terceiros.



Sumário

1 Entendimento Geral

1.1	Do que trata esse livro?	7
1.2	O que é Mineração de Dados?	7
1.3	O que é Machine Learning?	8
1.4	Formas de Aprendizado	9
1.5	Montagem do Ambiente	11

2 Conceitos Introdutórios

2.1	Termos da Estatística	17
2.2	Termos que devemos saber	18
2.3	Roteiro	20
2.4	Bibliotecas Utilizadas	21
2.5	Distribuição Gaussiana	22
2.6	Distribuição de Poisson	23
2.7	Distribuição Binomial	26
2.8	Feature Selection	28
2.9	K Fold Cross Validation	31

2.10	Matriz de Confusão	33
2.11	Curva ROC e Valor AUC	38
2.12	Terminamos?	42

3 EDA

3.1	Passos da EDA	43
3.2	Passo 1 - Entender os Dados	43
3.3	Passo 2 - Limpar os Dados	48
3.4	Passo 3 - Relacionamento entre os Atributos	51
3.5	Conclusão.	54

4 Modelos Iniciais

4.1	K-Means	55
4.2	Aplicação da Técnica	56
4.3	Plotagem do Resultado do Modelo	58
4.4	K-Nearest Neighbors	59
4.5	Análise de Cluster	62
4.6	Clusterização Hierárquica	65
4.7	Regressão Linear	69
4.8	Árvore de Decisão.	75
4.9	Floresta Aleatória.	79

5 Modelos Preditivos

5.1	Naïve Bayes	83
5.2	Apriori	85
5.3	SVM	87

5.4	Regressão Logística.....	92
------------	---------------------------------	-----------

6 Modelos Complementares

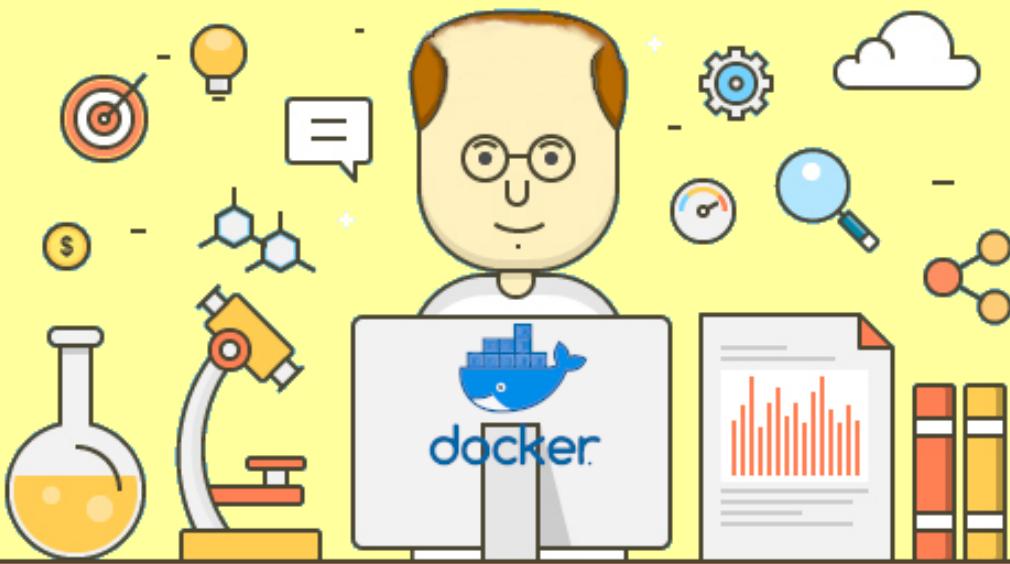
6.1	DBScan	97
------------	---------------------	-----------

6.2	Explicação do noise	98
------------	----------------------------------	-----------

6.3	Atrás dos outliers da Íris	99
------------	---	-----------

A Considerações Finais

A.1	Sobre o Autor	102
------------	----------------------------	------------



1. Entendimento Geral

F Machine Learning é a habilidade de localizar padrões em Dados. (Gil Weinberg - Founding Director of Georgia Tech Center for Music Technology)

1.1 Do que trata esse livro?

Cada vez que leio um livro de **Machine Learning** tenho a sensação que o autor quer mostrar para seus leitores o quanto ele é um bom Matemático, coloca um monte de fórmulas com demonstrações (muitas delas parecem escritas em grego e usam inclusive letras gregas) é isso me faz pensar: *Será que quando indicar um livro para meus alunos vou querer que eles aprendam fórmulas ou que tenham uma base em que possam praticar?*

E graças a isso publiquei uma série de artigos sobre os mais variados modelos de *Machine Learning* na rede **Linkedin** e esses artigos foram a base para esse livro, não espere encontrar aqui muitos conceitos, apesar de ser obrigado a abordá-los ou muita coisa se perderia, mas a ideia aqui é ser totalmente prático.

Já ouvimos isso de prático tantas vezes, existe duas séries de livros especialistas denominadas: "*Hands On*" e "*Cookbook*". Aprecio e tenho muitos desses livros, porém sempre que desejo algo no primeiro caso é muito difícil encontrar bem separado e exposto da forma como queria e no segundo está fragmentado demais. O prático aqui será: Um modelo em linguagem Python, ler bases de dados, com o uso de suas possibilidades, seu treino e melhor forma de obter resultados. Sendo assim não espere encontrar nesse aulas básicas de Python.

1.2 O que é Mineração de Dados?

Antes de surgir o termo "Ciência de Dados", um artigo de 1996 chamado "*Data Mining to knowledge discovery in database*" se referia a um processo geral de descoberta com a utilização das informações contidas nos dados. Em 2001, *William S. Cleveland* levou a Mineração de Dados combinou esta com Ciência da Computação. Realizou estatísticas para expandir as possibilidades da Mineração de Dados.

Durante este período, surgiu a **Web 2.0** no qual sites não eram apenas um "Panfleto Digital", mas um meio para compartilhar, postar e comentar experiências entre milhões de usuários. Sites como *Facebook* e *Linkedin* (2003), *Flickr* (2004), *YouTube* e *reddit* (2005), *Twitter* e *Spotify* (2006) ou *Pinterest* (2010) que

trouxeram muitos dados e se tornou difícil lidar com essa quantidade produzida. Assim criamos um novo termo denominado **Big Data**. Abriu um mundo de possibilidades para encontrarmos valores associados aos dados. Porém precisamos de infraestrutura para lidar com tantos dados, tecnologia de computação paralela como *MapReduce*, *Hadoop* e *Spark*. Em 2010 ocorreu o surgimento da **Ciência de Dados** para dar suporte as necessidades de negócio.

1.3 O que é Machine Learning?

De forma bastante genérica, algoritmos de *Machine Learning* (Aprendizado de Máquina e doravante apenas **ML**) são uma mudança de paradigma da “programação tradicional” onde precisamos passar toda a heurística explicitamente para um novo conceito, onde ao invés de escrever cada ação que o algoritmo deve realizar, apenas passamos diversos exemplos e deixamos que o computador resolva (ou aprenda) quais são as “melhores” (menor custo) decisões. É também chamada de *Statistical Learning* (Autores Hastie, Tibshirani & Friedman 2009) utilizada para extrair um modelo a partir de um sistema de observações ou medidas. Sendo um campo relativamente novo da ciência composto de uma variedade de métodos (algoritmos) computacionais e estatísticos que competem entre si.

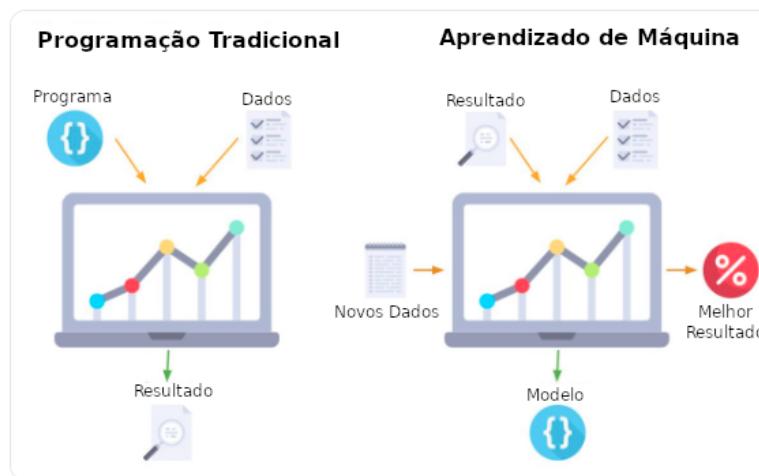


Figura 1.1: Programação Tradicional X Machine Learning

Ou seja, treinamos e melhoramos as respostas até que possam receber **novas observações**, transformamos isso em resultados sem que sejam explicitamente programáveis, isso é chamado "Modelo de ML". Interpretar esses modelos é entender como podemos transformar dados em informação útil. Em geral são classificados em:

- **Clusterização:** Encontrar uma estrutura de dados, summarização.
- **Rregressão e Estimação:** Predição de valores contínuos.
- **Classificação:** Predição de um item de um caso de classe/categoria.
- **Associação:** frequentemente ocorre entre itens e eventos.

É muito importante conhecer vários deles e assim podemos decidir qual se ajusta melhor as observações que temos para treiná-los. Devemos ter em mente que ML pode ser bem diferente de **Estatística**. Aqui não estamos preocupados com inferência, causalidade e exogeneidade. ML está mais preocupada, quase que

exclusivamente, em melhorar previsões ou rapidamente localizar em um mar de informações a resposta de um problema.

Assim podemos pegar a mesma função, por exemplo **Regressão Logística** e analisar do ponto de vista da estatística, que interpretaria se os "betas" são significativos, se os "resíduos" têm uma distribuição normal ou analisar isso do ponto de vista de ML e descobrir como está a relação entre **Precisão** e **Recall**, ou qual a ROC ou AUC do modelo¹.

1.4 Formas de Aprendizado

Quanto as formas de aprendizado se dividem em:

- **Não Supervisionado**, corresponde a um vetor de observações que é utilizado para observar padrões, tendências, verificar estruturas e descobrir relações.
- **Supervisionado**, além do vetor de observações, existe também uma resposta associada a cada questão.
- **Aprendizagem por Reforço**, uma ação ocorre e as consequências são observadas, assim a próxima ação considera os resultados da primeira ação. É um algoritmo dinâmico que parte do princípio "tentativa e erro".

Entender a diferença entre os dois tipos é bem simples, enviamos ao computador uma série de imagens sobre pratos de comida e não informamos absolutamente nada sobre elas, o máximo que acontecerá é a separação dessas em grupos similares. Imaginemos que em seguida mostramos a seguinte imagem:



Figura 1.2: Nova informação

Qual será o prato de comida que o computador associará? Exatamente "Ovo frito e batatas fritas"². No segundo tipo de aprendizado além das imagens dizemos o que cada uma vem a ser, porém ao mostrar o mesmo tipo de imagem não pense que o computador consegue classificá-la corretamente, provavelmente pode se confundir mais uma vez (assim como nos confundimos com a imagem mostrada).

¹As curvas ROC e AUC estão entre as métricas mais utilizadas para avaliação em um modelo de ML.

²Apesar que se olhar mais de perto vemos que isso é iogurte, meio pêssego e tiras de maçã

A resolução com problemas de imagem é bem complicada³, pois conseguimos identificar a diferença devido a nossa experiência não apenas visual mas com informações sobre textura, forma e outras que o computador ainda está engatinhando. Por isso o estudo desses algoritmos é algo tão rico e como vimos anteriormente. É muito importante conhecer boa parte dos modelos e decidir qual trabalha melhor com as observações que temos para treiná-lo.

1.4.1 Algoritmo Não Supervisionado

Não envolve um controle direto, aqui o ponto principal do requisito são desconhecidos e ainda precisam ser definidos. Normalmente são usados para: explorar uma estrutura de informação; extrair informações desconhecidas sobre as observações e detectar padrões.

k-means: é o mais popular, usado para segmentar em categorias exclusivas com base em uma variedade de recursos, por exemplo clientes como seus hábitos de consumo ou casas com seu preço, localidade e área.

t-Distributed Stochastic Neighbor Embedding: t-SNE é utilizado para redução de dimensionalidade que é particularmente adequada para a visualização de conjuntos que possuem dados com alta dimensão.

Principal Component Analysis: PCA é usado para enfatizar a variação e destacar padrões fortes em um conjunto de dados, utilizado para facilitar a exploração e visualização dos dados.

Associate Rule Mining: ARM para encontrar padrões frequentes, associações, relações e correlação, normalmente utilizado em proporcionar análises para cesta de compras. Em termos gerais, é aplicado em várias situações como encontrar associação ou determinar um padrão frequente nos conjuntos de observações.

1.4.2 Algoritmos Supervisionados

Neste tipo temos os atributos alvo rotulados e do ponto de vista da máquina, esse processo é uma rotina de "conectar os pontos" ou achar similaridades entre as observações e o que ela representa. Para "alimentar" o algoritmo determinamos que tipo de resultado é desejado ("sim/não", "verdadeiro/falso", a projeção do valor das vendas, a perda líquida de crédito ou o preço da habitação).

Régressão linear: muito utilizado para prever resultados numéricos contínuos, como preços de casas ou ações, umidade ou temperatura de um local, crescimento populacional.

Régressão logística: É um classificador popular utilizado especialmente no setor de crédito para prever inadimplências de empréstimos.

k-Nearest Neighbors: KNN é um algoritmo usado para classificar as observações em duas ou mais categorias (*cluster*) amplamente usado na separação como preços de casas em grupos, por exemplo com base em preço, área, quartos e toda uma gama de outros preditores.

Support Vector Machines: SVM é um classificador popular utilizado na detecção de imagens e faces, além de aplicativos como reconhecimento de manuscrito.

³ Assim como análise da linguagem, chamada de NLP - *Natural Language Processing*

Tree-Based Algorithms: Algoritmos baseados em árvores, como *Random Forest* (florestas aleatórias) ou *Decision Tree* (árvores de decisão), são usados para resolver problemas de classificação e regressão.

Naive Bayes: Utiliza um modelo matemático de probabilidade para resolver problemas de classificação.

1.4.3 Algoritmos de Aprendizagem por Reforço

Tratam de situações onde a máquina começa a aprender por tentativa e erro ao atuar sobre um ambiente dinâmico. Desta maneira, não é necessário novos exemplos ou um modelo a respeito da tarefa a ser executada: a única fonte de aprendizado é a própria experiência do agente, cujo objetivo formal é adquirir uma política de ações que maximize seu desempenho geral.

Q-Learning: é um algoritmo de aprendizado baseado em valores. Esses tipos atualizam uma função com base em uma equação (particularmente neste caso de *Bellman*) matemática, ou então, com base em políticas, estima a função de valor para uma política gananciosa obtida a partir do último aprimoramento. Q-learning é um algoritmo de políticas. Significa que aprende o valor ideal, independentemente das ações do agente. Por outro lado, um aprendiz de política aprende o valor que está sendo executada pelo agente, incluindo as etapas de exploração e encontrará uma política ideal, leva em consideração a exploração inerente dessa.

Temporal Difference: TD é um agente que aprende por meio de episódios sem conhecimento prévio desse ambiente. Isso significa que a diferença temporal adota uma abordagem de aprendizado sem modelo ou supervisão. Podemos considerar isso como uma tentativa e erro.

Monte-Carlo Tree Search: MCTS é um método geralmente usado nos jogos para prever o caminho (movimentos) que a política deve seguir para alcançar a solução final vencedora. Jogos como Cubo de Rubik, Sudoku, Xadrez, Go, ou um simples Jogo da Velha têm muitas propriedades em comuns que levam ao aumento exponencial do número de possíveis ações que podem ser executadas. Esses passos aumentam exponencialmente à medida que o jogo avança. Idealmente, podemos prever todos os movimentos possíveis e seus resultados que podem ocorrer no futuro e assim aumentarmos a chance de ganhar.

Asynchronous Advantage Actor-Critic: A3C é um dos algoritmos mais recentes a serem desenvolvidos no campo de Reforço Profundo. Foi desenvolvido pelo *DeepMind* do Google e implementa um treinamento no qual vários *workers*, em ambiente paralelo, atualizam independentemente uma função de valor global - portanto "assíncrona". Um dos principais benefícios de ter atores assíncronos é a exploração eficaz e eficiente do espaço de estados.

1.5 Montagem do Ambiente

Vemos atualmente uma grande revolução em torno de *Data Science* (falamos em inglês para parecer algo chique) principalmente em torno as ferramentas que tem se atualizado a uma velocidade assustadora. Porém, essas atualizações constantes muitas vezes carregam problemas que podem afetar o seu Sistema Operacional. A pergunta é: Como ficar atualizado e seguro ao mesmo tempo? A única resposta coerente que consegui encontrar foi: Usar a conteinerização para resolver o problema.

Então o ideal é partir atrás de imagens prontas, visitar sites como HubDocker (<https://hub.docker.com>) e rezar para encontrar a imagem que nos atenda ou fazer algo melhor e personalizar a imagem.

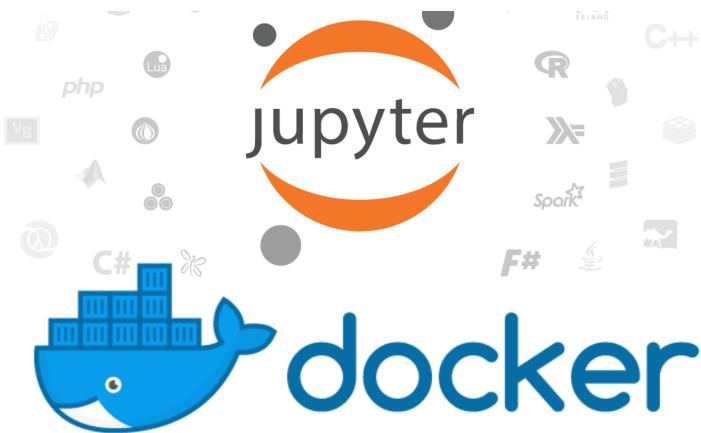


Figura 1.3: Docker e Jupyter para Data Science

Essa segunda alternativa é bem mais próxima a realidade de qualquer um que deseje trabalhar de modo efetivo com Ciência de Dados. A personalização de uma imagem no Docker não é um bicho de sete cabeças (no qual a partir do momento que cortamos uma das cabeças nasce mais duas, então sempre imaginei que seriam mais do que sete) mas algo que pode ser facilmente aprendido por qualquer um que "ao menos" saiba usar o terminal de comando do Linux.

Dica 1.1: Não sabe nem por onde começar com o Docker? Não se desespere, baixe um paper sobre o Docker gratuitamente na minha página no Academia.edu (<https://iesbpreve.academia.edu/FernandoAnselmo>).

Minha primeira personalização de Imagem⁴ surgiu quando descobri que as versões **Jupyter** não me atendiam por completo e **Anaconda** era grande demais. Iremos então trabalhar em uma imagem que possa conter um Jupyter mais adequado e uma Anaconda mais controlada. Com o seguinte resultado do arquivo **Dockerfile** (que obrigatoriamente deve ter esse nome):

```
# Base da Imagem
FROM ubuntu:19.10

# Adiciona o metadata para a imagem com o par: chave,valor
LABEL maintainer="Fernando Anselmo <fernando.anselmo74@gmail.com>"
LABEL version="1.2"

# Variaveis de Ambiente
ENV LANG=C.UTF-8 LC_ALL=C.UTF-8 PATH=/opt/conda/bin:$PATH

# Execucoes iniciais:
# Cria a pasta de ligacao
RUN mkdir ~/GitProjects && \
# instala os pacotes necessarios
apt-get update && apt-get install --no-install-recommends --yes python3 && \
apt-get install -y wget ca-certificates git-core pkg-config tree freetds-dev
apt-utils && \
# Limpeza basica
```

⁴O objetivo não é termos uma imagem pequena, mas uma PERSONALIZÁVEL, se não deseja isso recomendo que acesse o tutorial disponível em <https://jchrist.github.io/conda-docker-tips.html>.

```
apt-get autoclean -y && \
rm -rf /var/lib/apt/lists/* && \
# Configuracao do Jupyter
mkdir ~/.ssh && touch ~/.ssh/known_hosts && \
ssh-keygen -F github.com || ssh-keyscan github.com >> ~/.ssh/known_hosts && \
git clone https://github.com/bobbywlindsey/dotfiles.git && \
mkdir ~/.jupyter && \
mkdir -p ~/.jupyter/custom && \
mkdir -p ~/.jupyter/nbconfig && \
cp /dotfiles/jupyter/jupyter_notebook_config.py ~/.jupyter/ && \
cp /dotfiles/jupyter/custom/custom.js ~/.jupyter/custom/ && \
cp /dotfiles/jupyter/nbconfig/notebook.json ~/.jupyter/nbconfig/ && \
rm -rf /dotfiles && \
# Instalar o Anaconda
echo 'export PATH=/opt/conda/bin:$PATH' > /etc/profile.d/conda.sh && \
wget --quiet https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-x86_64.sh -O \
~/anaconda.sh && \
/bin/bash ~/anaconda.sh -b -p /opt/conda && \
rm ~/anaconda.sh && \
conda uninstall anaconda-navigator && \
conda update conda && \
conda update anaconda && \
conda install nodejs && \
conda update --all && \
# Limpeza basica no Anaconda
find /opt/conda/ -follow -type f -name '*.a' -delete && \
find /opt/conda/ -follow -type f -name '*.pyc' -delete && \
find /opt/conda/ -follow -type f -name '*.js.map' -delete && \
find /opt/conda/lib/python*/site-packages/bokeh/server/static -follow -type f -name \
'*.*js' ! -name '*.min.js' -delete && \
# Instalar os temas para o Jupyter
pip install msgpack jupyterthemes && \
jt -t grade3 && \
# Instalar os pacotes
conda install scipy && \
conda install pymssql mkl=2018 && \
pip install SQLAlchemy missingno json_tricks \
gensim elasticsearch psycopg2-binary \
jupyter_contrib_nbextensions mysql-connector-python \
jupyter_nbextensions_configurator pymc3 apyori && \
# Habilitar as extensoes do Jupyter Notebook
jupyter contrib nbextension install --user && \
jupyter nbextensions_configurator enable --user && \
jupyter nbextension enable codefolding/main && \
jupyter nbextension enable collapsible_headings/main && \
# Adicionar a extensao do vim-binding
mkdir -p $(jupyter --data-dir)/nbextensions && \
git clone https://github.com/lambdalisue/jupyter-vim-binding $(jupyter \
--data-dir)/nbextensions/vim_binding && \
cd $(jupyter --data-dir)/nbextensions \
chmod -R go-w vim_binding && \
# Remover o que nao eh necessario
conda clean -afy && \
apt-get remove -y wget git-core pkg-config && \
apt-get autoremove -y && apt-get autoclean -y && \
# Adicionar o Git ao Jupyter Lab
```

```

pip install --upgrade jupyterlab-git && \
jupyter lab build

# Configurar o acesso ao Jupyter
WORKDIR /root/GitProjects
EXPOSE 8888
CMD jupyter lab --no-browser --ip=0.0.0.0 --allow-root
--NotebookApp.token='data-science'

```

Tento manter sempre o script o mais documentado possível deste modo posso remover ou adicionar propriedades sem me incomodar muito. Para criarmos a imagem é muito simples, supondo que a localização do script esteja em uma pasta chamada docker-data-science, então na pasta anterior digitar o comando:

```
$ docker build -t fernandoanselmo/docker-data-science docker-data-science
```

Obviamente que "fernandoanselmo/docker-data-science" pode ser alterado para o nome que desejar, porém na essência isso cria uma imagem que contém além do sistema operacional uma versão completa do Jupyter (com a inclusão de várias funcionalidades) para a realização do nosso trabalho como Cientista de Dados.

Dica 1.2: Socorro. Mas isso é muito complicado e não entendo nada disso! Sem problemas, basta saltar toda essa parte de criação e construção para os próximos tópicos. Esta imagem foi publicada no **DockerHub** e pode ser baixado sem problemas.

Agora o comando:

```
$ docker run -d -t -i -v --privileged /dev/ttyACM0:/dev/ttyACM0 -v
~/Aplicativos/ipynb:/root/GitProjects --network=host
--name meu-jupyter fernandoanselmo/docker-data-science
```

Realiza a criação de um contêiner. Vejamos os detalhes: após a opção -v aparece a expressão "~/Aplicativos/ipynb", essa se refere a uma determinada pasta no sistema operacional onde serão armazenados os Notebooks produzidos. Abrir o navegador no endereço <http://localhost:8888> e obtemos o seguinte resultado:

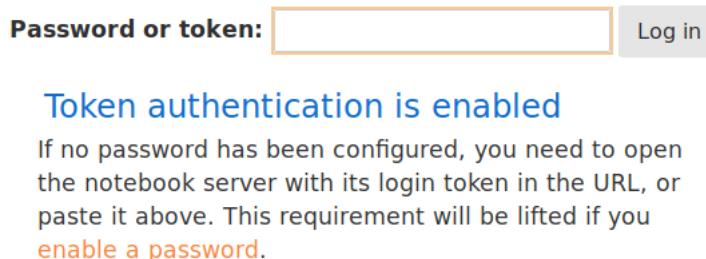


Figura 1.4: Jupyter solicitando o Token

A senha do token está definida na última linha do Script como "data-science", após informá-la o jupyter está pronto para trabalharmos:

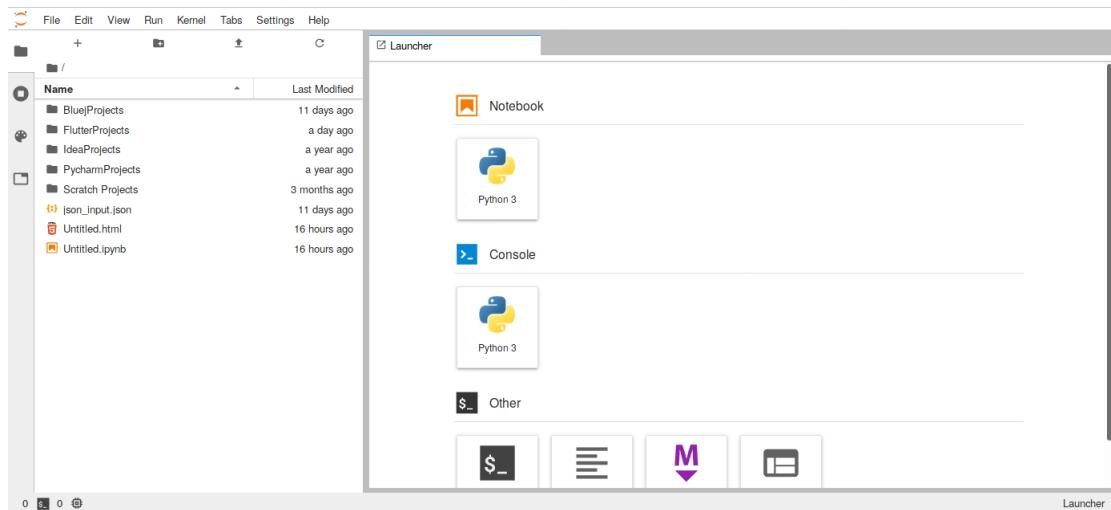


Figura 1.5: Jupyter Lab pronto

A maior vantagem que agora possuímos um ambiente com o **Jupyter Lab** completamente controlado incluindo temas e sincronização com o **Github**. Os próximos comandos são bem mais simples, tais como:

```
$ docker stop meu-jupyter
$ docker start meu-jupyter
```

Respectivamente para encerrar e iniciar o contêiner. Oh não! Agora estou preso, não posso mais fazer atualizações. Devemos entender que os contêineres são **dinâmicos**. Precisamos instalar o Keras/TensorFlow, acessar o contêiner (com ele já iniciado):

```
$ docker exec -it meu-jupyter /bin/bash
```

E instalamos normalmente, como se estivesse em uma máquina com o Ubuntu (com os poderes de superusuário):

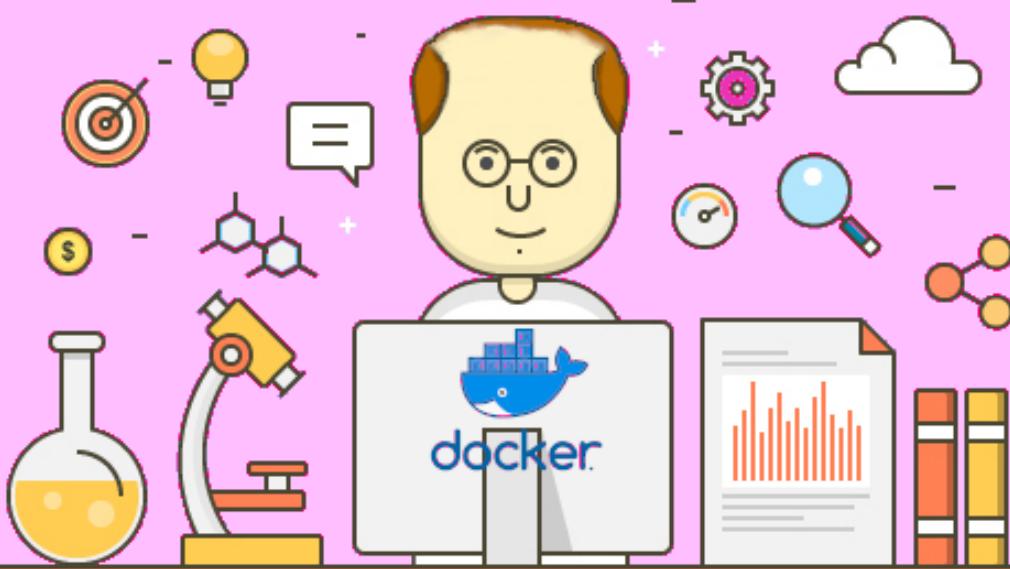
```
# conda install -c conda-forge keras
```

Pronto uma vez testado podemos optar por manter só nesse contêiner, ou então modificar o Script para ao criarmos novos contêineres e estes serão criados com essa funcionalidade embutida.

Verificar qual a versão do Python que está a nossa disposição, em uma célula do Notebook digite:

```
!python --version
```

Ao pressionarmos Ctrl+Enter será mostrada a versão 3.7.7. Agora podemos testar e executar qualquer ferramental sem nos preocuparmos em corromper a máquina. Talvez, no máximo, perdemos um contêiner.



2. Conceitos Introdutórios

F A melhor forma de prever o futuro é cria-lo. (Peter Drucker - Escritor e Pai da Administração Moderna)

2.1 Termos da Estatística

Devemos sempre possuir um vocabulário comum, e mesmo que não se entenda absolutamente nada de estatística (e tenha um Estatístico a sua disposição), o Cientista de Dados deve conhecer os seguintes termos:

- **População:** o conjunto constituído por todos os indivíduos que representam pelo menos uma característica comum, cujo comportamento interessa analisar (inferir). Por exemplo, se em uma empresa o diretor gostaria de saber se os funcionários estão satisfeitos com os benefícios oferecidos, a população de estudo são todos os funcionários dessa empresa. O conceito de população depende do objetivo de estudo.
- **Amostra:** um subconjunto, uma parte selecionada da totalidade de observações abrangidas pela população, através da qual se faz inferência sobre as características da população. Por exemplo, uma rádio tem o interesse de saber como está sua audiência com os ouvintes no trânsito. Não é possível perguntar a todos os motoristas que ouvem rádio qual é aquela que eles preferem. Então buscamos uma parte representativa dessa população, isto significa, perguntar somente a alguns motoristas qual rádio preferem escutar enquanto dirigem. Uma amostra tem que ser representativa, sua coleta bem como seu manuseio requer cuidados especiais para que os resultados não sejam distorcidos.
- **Elemento ou Variável de Interesse:** característica a ser observada em cada indivíduo da amostra. Componentes sobre o qual serão observadas ou medidas as características. Onde cada característica corresponde a um tipo de variável. Por exemplo, se queremos estudar o índice de massa corporal (IMC) de alunos do ensino médio de uma cidade, tomaremos uma amostra dessa população, e mediremos a altura e o peso de cada aluno, já que o IMC é calculado como uma razão entre peso e o quadrado da altura do indivíduo. Nesse caso: peso e altura são as variáveis de interesse.

Tendo em vista as dificuldades de várias naturezas para se observar todos os elementos da população, tomaremos alguns deles para formar um grupo a ser estudado. Este subconjunto da população, em geral com dimensão menor, é denominado **amostra**.

Outros termos de interessante são as "Medidas Resumo", são elas:

- **Média:** É a soma das observações dividida pelo total. Este é o mais comum indicador de uma tendência central de uma variável.
- **Mediana:** Em uma lista ordenada de dados (rol), a mediana é o termo central.
- **Moda:** Refere-se ao termo que mais aparece em uma coleção. Sendo que: Amodal - rol que não tem nenhum valor que se repete; Bimodal - existem 2 valores que se repetem na mesma frequência, N-modal - existem n valores que se repetem na mesma frequência.
- **Variância:** Medida de dispersão dos dados para uma média. Quanto maior for mais distantes da média estão, e quanto menor for mais próximos estão da média.
- **Desvio Padrão (std):** É o resultado positivo da raiz quadrada da variância. Indica como fechado estão os dados em torno da média.
- **Amplitude:** Medida de dispersão da amostra, sendo uma simples diferença entre o menor e maior valor.
- **Coeficiente de Variação** usado para analisar a dispersão em termos relativos a seu valor médio quando duas ou mais séries de valores apresentam unidades de medida diferentes. Dessa forma, podemos dizer que o coeficiente de variação é uma forma de expressar a variabilidade dos dados excluindo a influência da ordem de grandeza da variável.

Probabilidade é uma medida que nos mostra qual o grau de um evento ocorrer novamente. Muita para da ciência de dados é baseada na tentativa de medir a probabilidade de eventos, desde as chances de um anúncio ser clicado até a falha de uma determinada peça em uma linha de montagem.

Dica 2.1: Para saber mais. Quer entender mais sobre Estatística, recomendo este interessante livro online e em constante evolução <http://onlinestatbook.com/2/index.html>.

2.2 Termos que devemos saber

Como toda ciência, existe um vocabulário comum que os cientistas falam e que devemos conhecer, palavras como: treinar um modelo, MSE, *Overfitting* ou bisbilhotagem fazem parte desse vocabulário. Então mesmo sendo este um livro prático, precisamos saber do que se tratam.

feature (atributo) pode ser considerado como **preditor** (ou explicativo) e **alvo** (ou dependente). Atributos preditores são valores de entrada para um algoritmo enquanto que os dependentes de saída (resultado). Um problema que pode ser trabalhado com técnicas de ML é quando existe um padrão e não é fácil defini-lo. Fazendo uma analogia com estatística, é preciso usar um conjunto de observações (amostra) para descobrir um processo subjacente (probabilidade). Os dados são selecionados e aplicados ao modelo que possui um grau de aprendizado (acurácia). Descobrir um padrão não é memorizar *Overfitting*, pois ao injetar novos dados o modelo deve ser capaz de prever o resultado.

Training (treinar) um modelo significa ensinar o modelo a determinar bons valores para todos os pesos e o viés de exemplos rotulados. *Loss* (perda) é a penalidade por uma má previsão em um único exemplo, que pode ser quantificada por métricas como *Mean Square Error* (MSE). Esse é um cálculo conhecido como

loss function (função de perda) sendo que representa uma medida de quão bom um modelo de previsão faz em termos de ser capaz prever o resultado correto.

Existem três princípios em ML:

- **Navalha de Occam:** o modelo mais simples que se ajusta aos dados também é o mais plausível.
- **Viés de amostragem:** se os dados são amostrados de forma tendenciosa, então a aprendizagem também produz resultados tendenciosos.
- **Bisbilhotagem de dados:** se um conjunto de dados afeta qualquer etapa do processo de aprendizado, então a capacidade do mesmo conjunto de dados avaliar o resultado foi comprometida (se usar propriedades adequadas do conjunto de dados, pode assumir relações antes de começar a escolha de modelos).

Teoria Vapnik–Chervonenkis (VC) tenta explicar o processo de aprendizagem do ponto de vista estatístico. A dimensão VC é uma medida da complexidade a um espaço de funções que pode ser aprendido por um algoritmo de classificação estatística, é definido como a cardinalidade do maior conjunto de pontos que o algoritmo pode quebrar. Se tiver poucos dados, modelo mais simples funcionam melhor e complexo é um desastre.

Desigualdade de Hoeffding fornece um limite superior na probabilidade que a soma das variáveis independentes não se desvie do valor esperado acima de uma certa quantia. Quando generalizada, modelos muito sofisticados (com grande número de hipóteses) perdem a ligação entre uma amostra e o total, isso implica em memorizar ao invés de aprendizado.

Na aprendizagem supervisionada, um algoritmo é construído através de muitos exemplos e tenta encontrar um modo de minimizar a perda das ligações. Uma parte dos dados deve ser separada para treinamento e outra para teste. Essa separação ocorre para estimar o erro de predição do modelo e ter a certeza de que não está superajustado. Geralmente, é adotada uma razão de 80% dos dados para treinamento e 20% teste.

Gradient Descent (SGD) calcula o gradiente da curva de perda (inclinação) e indica qual o caminho para minimizar o erro (utilizar um gráfico da perda em função do peso), redefine o peso - quando há vários pesos, o gradiente é um vetor de derivadas parciais com relação aos pesos. Esse vetor gradiente (com direção e magnitude) é multiplicado por um escalar conhecido como taxa de aprendizado (*learning rate* ou *step size*) para determinar o próximo ponto. A heurística de buscar mínimo local começa de diferentes valores para então encontrar o melhor dentre todos. A aleatoriedade permite localizar outros mínimos locais, e assim podemos determinar o melhor, que pode ser o mínimo global.

Step (etapa) é o número total de iterações do treinamento. Uma etapa calcula a perda de um lote e usa esse valor para modificar os pesos do modelo uma vez. *Batch size* (tamanho do lote) é o número total de exemplos (escolhidos aleatoriamente) que foram usados para calcular o gradiente em uma única etapa (conjunto de dados inteiro).

Epoch (época) é um ciclo completo de treinamento e um gráfico de erro/perda indica como é a evolução do modelo. Cada *epoch* pode conter resultados piores, mas o melhor é guardado e exibido sempre *pocket algorithm*, a não ser que surja um melhor. Esse gráfico também pode ser construído destinado a um conjunto de dados para teste. Se a curva de perda em função das *epoch* cair muito para o treinamento e não acontecer o mesmo com os dados de teste, revela que o modelo está treinado em excesso para essas amostras de treino e não consegue prever um novo conjunto de amostras.

Overfitting acontece quando o ruído é ajustado também, só que ruído não tem padrão a ser descoberto.

O ruído pode ser aleatório ou determinístico, relacionado a informação que existe, mas os dados, ou o conjunto de hipóteses, não conseguem promover o aprendizado dessa informação. *Weight decay* (decaimento de peso) é uma constante regularizadora, que restringe os pesos na direção da função alvo e melhora o ajuste e reduz o *overfitting*.

Normalization (normalizar) significa converter valores de recursos numéricos (como, 100 a 900) em um intervalo padrão (como, 0 a 1 ou de -1 a +1). Um exemplo desse tipo de cálculo é: $valor - min \div max - min$. Se o conjunto de dados é composto de múltiplos atributos, a normalização gera benefícios, como ajudar a descida gradiente a convergir mais rapidamente, evitar que um número torne-se NaN por exceder seu limite de precisão e ajudar o modelo a aprender os pesos apropriados para cada atributo. A normalização deve ocorrer depois de separar os dados de treinamento e teste.

Standard (padronização) é um cálculo importante para comparar medidas com diferentes unidades. Um exemplo desse tipo de cálculo é o "Z score": $(valor - media) \div std$. A padronização deve ser feita antes da normalização.

2.3 Roteiro

Devemos saber que um roteiro para ML passa pelas seguintes fases:

- Entender a questão de negócio (falar a mesma língua).
- Identificar a causa raiz.
- Coletar as observações.
- Realizar uma estatística descritiva e compreender suas características.
- Aplicar uma limpeza: entender os atributos e possíveis valores faltantes.
- Ler os dados, se necessário renomear ou corrigir os atributos.
- Procurar por incoerências.
- Criar novas variáveis para modelar melhor o fenômeno, se necessário.
- Levantar Hipóteses sobre o Comportamento do Negócio.
- Definir o tipo do problema: Regressão, Classificação ou Clusterização.
- Realizar uma Análise Exploratória de Dados.
- Quais hipóteses são falsas e quais são verdadeiras?
- Quais as correlações entre os atributos preditores e alvo?
- Verificar se as variáveis possuem o mesmo peso, em termos de importância, para o modelo.
- Aplicar diferentes algoritmos de ML.
- Comparar os modelos, sob a mesma métrica de performance, o mais apropriado para análise.
- Garantir que o modelo não possui *Overfit*, ou seja, memorização ao invés de aprendizado.
- Escrever os valores de previsão e seu intervalo de confiança do arquivo de teste.
- Descrever uma breve explicação do raciocínio da solução.
- Anotar as respostas encontradas.
- Detalhar as possíveis soluções para o problema.

São vários passos, devemos segui-los para garantir o sucesso das nossas análises. Algumas partes são bem preocupantes, entre elas, as que envolvem os conceitos mais básicos, a tendência é sempre saltá-los sem dar muita importância. Porém são cruciais.

2.4 Bibliotecas Utilizadas

Já temos o *Jupyter* e devemos conhecer o ferramental que iremos trabalhar. Abrir uma célula e digitar os seguintes comandos:

```
import numpy as np
import pandas as pd
import matplotlib
import pylab
import matplotlib.pyplot as plt
from scipy import stats
from scipy.stats import norm
from numpy.random import seed
from numpy.random import randn
import matplotlib.colors
import scipy
import sklearn
import seaborn as sns
import statsmodels.api as sm
%matplotlib inline
```

Na próxima célula digitar:

```
print('numpy: {}'.format(np.__version__))
print('pandas: {}'.format(pd.__version__))
print('scipy: {}'.format(scipy.__version__))
print('matplotlib: {}'.format(matplotlib.__version__))
print('sklearn: {}'.format(sklearn.__version__))
print('seaborn: {}'.format(sns.__version__))
print('statsmodel: {}'.format(sm.__version__))
print('\nMais informações:\n\n', sklearn.show_versions())
```

E obtemos como resposta as versões das principais bibliotecas que utilizaremos:

- **NumPy Numerical Python.** Seu recurso mais poderoso é a matriz n-dimensional. Também contém funções básicas de álgebra linear, transformações de *Fourier*, recursos avançados de números aleatórios e ferramentas para integração com outras linguagens de baixo nível, como Fortran, C e C++.
- **Pandas Python and Data Analysis** para operações e manipulações de dados estruturados. Amplamente utilizada para coleta e preparação de dados.
- **SciPy Scientific Python.** Tem por base a NumPy. É uma das bibliotecas mais úteis para diversos módulos de ciência e engenharia de alto nível, como matrizes discretas, álgebra linear, otimização e dispersão.
- **Matplotlib Math Plotting Library.** para gerar uma grande variedade de gráficos, tais como histogramas, gráfico de linhas e mapa de calor.
- **SkLearn ou Scikit Learn SciPy Toolkit Learn.** Tem por base a NumPy, SciPy e Matplotlib. Contém muitas ferramentas para aprendizado de máquina e modelagem estatística, incluindo classificação, regressão, clusterização e redução de dimensionalidade.

- **Seaborn Statistical Data Visualization.** Geração de gráficos atraentes e informativos em Python. Tem por base a Matplotlib. Visa tornar a visualização uma parte central da exploração e compreensão dos dados.
- **Statsmodels Statistical Models.** Permite explorar dados, estimar modelos estatísticos e executar testes. Uma lista extensa de estatísticas descritivas, funções de plotagem e de resultados estão disponíveis para diferentes tipos de dados e para cada estimador.

Caso exista a necessidade de uma atualização, por exemplo da biblioteca **Scikit-learn**, abrir uma nova célula e digitar o comando:

```
!pip install --upgrade scikit-learn
```

Dica 2.2: Bibliotecas Utilizadas. Obtenha uma boa referência sobre essas pois não serão tratadas em nível básico neste livro. Obviamente suas funções serão comentadas mas partiremos do pressuposto que sabemos lidar com estas.

2.5 Distribuição Gaussiana

A Distribuição Gaussiana (também conhecida como Curva Normal) é uma curva em forma de sino e supõe-se que, durante qualquer valor de medição, siga uma distribuição normal de um número igual de medidas acima e abaixo do valor médio. Para entendermos a distribuição normal, é importante conhecer as definições de média, mediana e moda. Se uma distribuição for normal, seus valores são os mesmos. No entanto, o valor da média, mediana e moda podem ser diferentes resultando em uma distribuição inclinada (não gaussiana).

Abrir uma nova célula e iremos gerar um gráfico com uma Distribuição Gaussiana ideal:

```
xAxis = np.arange(-3, 3, 0.001) \\
yAxis = norm.pdf(xAxis, 0, 1) \\
plt.plot(xAxis, yAxis) \\
plt.show()
```

Obtemos uma Distribuição Normal perfeita. Normalmente será muito difícil na realidade a curva ser tão perfeita assim, então podemos nos aproximar mais da realidade e gerar a partir de uma amostra com dados aleatórios:

```
data = 5 * randn(10000) + 50 \\
plt.hist(data, bins=100) \\
plt.show()
```

Obtemos a seguinte Distribuição Normal:

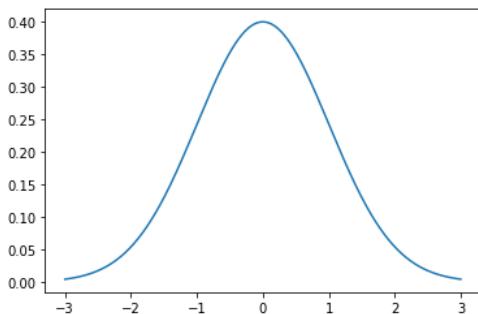


Figura 2.1: Curva da Distribuição Normal

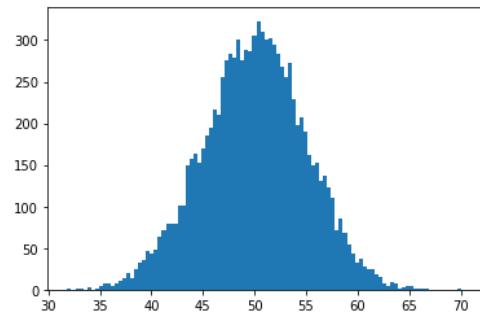


Figura 2.2: A partir de dados randômicos

Também podemos a partir dessa segunda, calcular seus valores principais:

```
print('Média: %.3f' % np.mean(data))
print('Mediana: %.3f' % np.median(data))
print('Moda:', stats.mode(data))
```

2.6 Distribuição de Poisson

Pronuncia-se *Poassom*, é uma distribuição de Probabilidade Discreta para um atributo X (qualquer) que satisfaça as seguintes condições:

- O experimento consistem em calcular quantas vezes (k) que um evento ocorre em um dado intervalo.
- A probabilidade de ocorrer é a mesma em cada intervalo.
- O experimento resulta em resultados que podem ser classificados como **sucessos** ou **falhas**.
- O número médio de sucessos (μ) que ocorre em uma região especificada é conhecido.
- A probabilidade de um sucesso é proporcional ao tamanho da região.
- A probabilidade de um sucesso em uma região extremamente pequena é praticamente zero.
- O número de ocorrências em um intervalo é independente.

Na prática é uma função de ponto percentual, *Poisson* não existe na forma fechada simples é computado numericamente. Essa é uma distribuição discreta, definida apenas para valores inteiros de X, a função de ponto percentual não é suave da mesma forma que a função de ponto percentual normalmente é para uma distribuição contínua. Começamos com a importação das bibliotecas necessárias:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

A NumPy já nos provê a função necessária para vermos como é o gráfico:

```
dp = np.random.poisson(lam=3, size=(1000))
plt.hist(dp)
plt.show()
```

Geramos 1.000 números aleatórios com uma ocorrência 3 e obtemos como resultado:

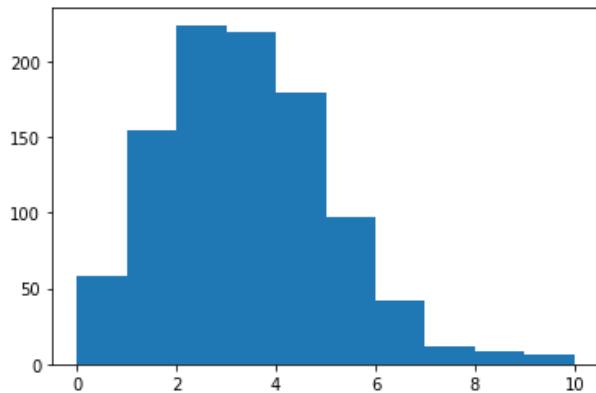


Figura 2.3: Distribuição de Poisson na MatPlotLib

É chamado "Variável Aleatória de Poisson" o número de sucessos resultantes em um experimento de Poisson. Porém esse gráfico fica bem mais apresentável com o uso da Seaborn:

```
sns.distplot(dp)
plt.show()
```

E obtemos:

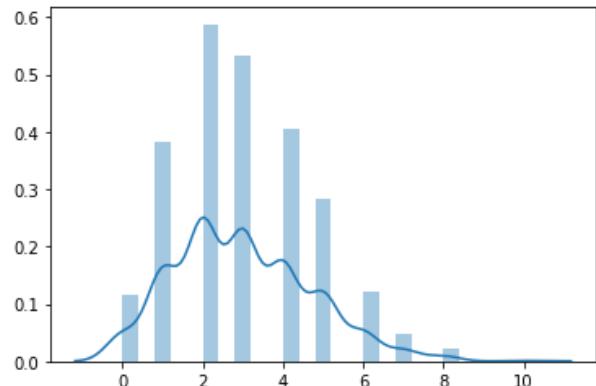


Figura 2.4: Distribuição de Poisson na Seaborn

Dica 2.3: Historicamente falando. 1946, o estatístico britânico RD Clarke publicou "*Uma Aplicação da Distribuição de Poisson*", com sua análise dos acertos de bombas voadoras (mísseis V-1 e V-2) em Londres durante a II Guerra Mundial. Algumas áreas foram atingidas com mais frequência do que outras. Os militares britânicos desejavam saber se os alemães estavam atacando esses distritos (os acertos indicavam grande precisão técnica) ou se a distribuição era por acaso. Se os mísseis fossem de fato apenas alvos aleatórios (dentro de uma área mais geral), os britânicos poderiam simplesmente dispersar instalações importantes para diminuir a probabilidade de serem atingidos.

Se acertarmos a escala, surpreendentemente, ao colocar uma Distribuição Normal e uma de Poisson sobrepostas:

```

sns.distplot(np.random.normal(loc=50, scale=7, size=1000), hist=False,
             label='normal')
sns.distplot(np.random.poisson(lam=50, size=1000), hist=False, label='poisson')
plt.show()

```

Obtemos como resultado:

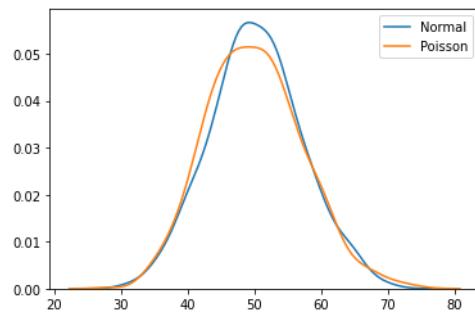


Figura 2.5: Distribuição Normal e de Poisson

E claramente percebemos a similaridade entre ambas. Um caso curioso que ocorre com a Distribuição de Poisson é a "cauda longa", sabemos que em qualquer comércio existe os produtos que mais possuem saída e aqueles outros que estão ali para "compor estoque". Por exemplo:

```

ax = sns.distplot(np.random.poisson(lam=3, size=10000), bins=27, kde=False)
ax.set(xlabel='Mercadorias', ylabel='Unidades Vendidas')
plt.show()

```

Obtemos como resultado:

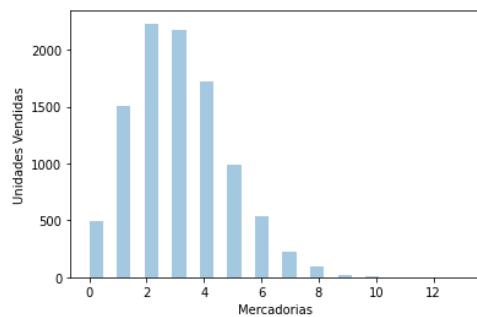


Figura 2.6: A Cauda Longa

A partir do número 6 temos um decrescimento constante, o que demonstra exatamente o que acontece com algumas mercadorias.

2.7 Distribuição Binomial

Neste tipo de distribuição apenas dois resultados são possíveis: sucesso ou fracasso, ganho ou perda, vitória ou perda e a probabilidade é exatamente a mesma para todas as tentativas. No entanto, os resultados não precisam ser igualmente prováveis, e cada estudo é independente um do outro. Podemos ver sua curva com 1.000 lançamentos de 10 possibilidades

```
sns.distplot(np.random.binomial(n=10, p=0.5, size=1000), hist=True)
plt.show()
```

Obtemos como resultado:

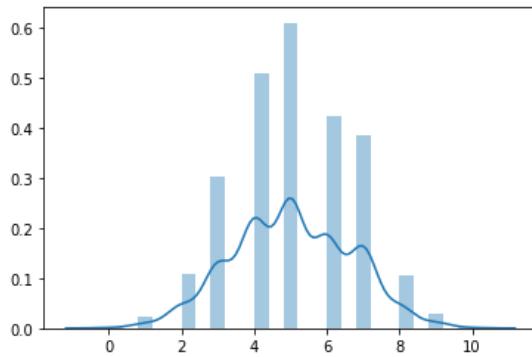


Figura 2.7: Distribuição Binomial

A principal diferença para a normal é que essa é contínua, enquanto que a binomial é discreta, mas se houver pontos de dados suficientes, serão bastante semelhantes (inclusive com a Poisson).

```
sns.distplot(np.random.normal(loc=50, scale=7, size=1000), hist=False,
            label='Normal')
sns.distplot(np.random.poisson(lam=50, size=1000), hist=False, label='Poisson')
sns.distplot(np.random.binomial(n=100, p=0.5, size=200), hist=False,
            label='Binomial')
plt.show()
```

Obtemos como resultado:

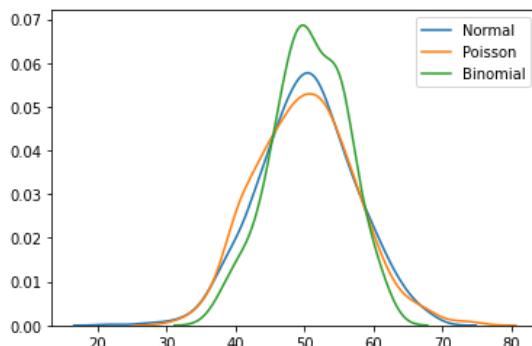


Figura 2.8: Mesmo gráfico 3 Distribuições

Mas e na prática? Imaginemos que 90% dos passageiros reservados chegam de fato para sua viagem. Suponhamos um transporte que pode conter 45 assentos. Muitas vezes as companhias praticam "excesso de reservas"(conhecido por *Overbooking*) isso significa que vende mais passagens do que os assentos disponíveis. Isso se deve ao fato de que às vezes os passageiros não aparecem um assento vazio representa perda. No entanto, ao reservar em excesso corre o risco de ter mais passageiros do que assentos.

Com esses riscos em mente, uma companhia decide vender mais de 45 bilhetes. Supondo que desejam manter a probabilidade de ter mais de 45 passageiros para embarcar no voo abaixo de 0,4 quantas passagens podem vender?

Para resolvermos isso precisamos da SciPy, e procedemos a seguinte codificação:

```
from scipy.stats import binom_test
for i in range(45, 51):
    print(i, "-", binom_test(x=45, n=i, p=0.9, alternative='greater'))
```

Obtemos como resultado:

```
45 - 0.008727963568087723
46 - 0.04800379962448249
47 - 0.13833822255419037
48 - 0.27986215181073293
49 - 0.449690866918584
50 - 0.6161230077242769
```

O parâmetro *alternative* que indica a hipótese, podemos usar:

- *greater* maior que
- *less* menor que
- *two-sided* maior e menor que (valor padrão)

E podemos vender **48 passageiros**. E agora sabemos porque todas as companhias de transporte praticam *overbooking*. Para vermos isso graficamente:

```
sns.distplot(np.random.binomial(n=48, p=0.9, size=400), label='Binomial')
```

Obtemos como resultado:

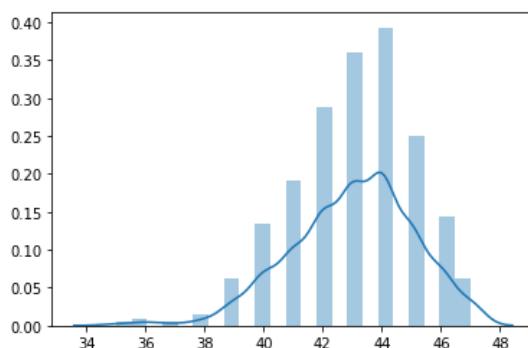


Figura 2.9: Gráfico do Overbooking

2.8 Feature Selection

Seleção de Atributos, Variáveis, Recursos ou Subconjuntos de Variáveis, são os vários nomes do processo que realiza a seleção de atributos relevantes para a construção de modelos.

O objetivo da *Feature Selection* é o de selecionar os atributos que funcionam melhor como **preditores**. Essa etapa ajuda a reduzir o *overfitting*, aumenta a acurácia do modelo e reduz o tempo de treinamento.

São os seguintes métodos que podemos utilizar:

- **Filter Methods**: usam medidas estatísticas para atribuir uma pontuação para cada atributo. Estas são classificadas de modo a serem mantidas ou removidas do modelo. Normalmente usamos testes univariados que consideram a independência do atributo explicativo com o dependente. Exemplo: *chi squared*, pontuações com Coeficiente de Correlação.
- **Wrapper Methods**: selecionam um conjunto de atributos, onde diferentes combinações são preparadas, avaliadas e comparadas. Um modelo preditivo é usado para avaliar a combinação de atributos e dar uma nota a partir da acurácia do modelo. Exemplo: RFE.
- **Embedded Methods**: aprendem quais atributos contribuem melhor para a acurácia do modelo no momento de sua construção. Exemplo: Métodos de Penalização, Algoritmos Lasso, *Elastic NET* e *Ridge Regression*.

Dica 2.4: Bases de Dados. Para TODOS os exemplos neste livro, utilizamos bases de dados que estão disponibilizadas em <https://github.com/fernandoans/machinelearning/tree/master/bases>. Salvo qualquer observação em contrário.

Importação das bibliotecas utilizadas:

```
import pandas as pd
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif, mutual_info_classif
from sklearn.feature_selection import chi2
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier
%matplotlib inline
```

Os classificadores estão contidos na Scikit-Learn e utilizaremos como nossa fonte de dados o arquivo chamado **pima-indians-diabetes.csv**. Criar um DataFrame para esta:

```
colnames = ['gest', 'glic', 'sang', 'skin', 'insul', 'mass', 'familia', 'idade',
           'conf']
df = pd.read_csv('pima-indians-diabetes.csv', names=colnames)
df.head()
```

Esta base são dados do **Instituto Nacional de Diabetes, Doenças Digestivas e Renais** sendo de pacientes do sexo feminino. Consiste de vários atributos que são considerados como preditivos (explicativos) e um atributo alvo (dependente). Os explicativos incluem o número de gestações que a paciente teve, seu IMC, nível de insulina, idade e outras informações como possível causa da Diabetes em Pacientes. Essa base possui 798 linhas sem quaisquer presença de nulos, verificar com: `df.info()`

O atributo *conf* é alvo, se o paciente em questão teve ou não diabetes. Precisamos isolá-lo:

```
X = df.drop(['conf'], axis=1)
y = df['conf']
```

Para os outros desejamos conhecer: qual (ou quais) atributos se comportam melhor para o nosso modelo?

2.8.1 Coeficientes de Coorelação

Como primeira forma aplicamos os testes estatísticos:

```
f_classif1 = SelectKBest(score_func=f_classif, k=4)
fit1 = f_classif1.fit(X,y)
```

Os tipos para o **SelectKBest**, neste caso, são:

- **f_classif**: mais adequado quando os atributos são numéricos e o dependente é categórico.
- **mutual_info_classif**: mais adequando quando não existe uma dependência linear entre os preditores e o alvo.
- **f_regression**: para resolver problemas de regressão.

Visualizar os atributos selecionados:

```
cols = fit1.get_support(indices=True)
df.iloc[:,cols]
```

Indica que número de gestações (*gest*), concentração de glicose no plasma (*glic*), índice de massa corporal (*mass*) e *idade* seriam os melhores candidatos.

2.8.2 Chi Squared

Essa é uma outra forma de medir a dependência, "elimina"os atributos com a maior probabilidade de serem independentes da classe e, portanto, irrelevantes para a classificação:

```
test2 = SelectKBest(chi2, k=4)
fit2 = test2.fit(X, y)
```

Visualizar os atributos selecionados:

```
cols = fit2.get_support(indices=True)
df.iloc[:,cols]
```

Percebemos que aconteceu uma mudança nos atributos: *glic*, *mass* e *idade* continuam, porém ao invés de *gest* aparece a quantidade administrada de insulina (*insul*).

2.8.3 RFE

Recursive Feature Elimination é um método que remove o(s) recurso(s) mais fraco(s) até que o número especificado de recursos seja atingido. Sendo assim é necessário informar ao RFE o número de atributos caso contrário reduz pela metade esse valor de acordo com o número de atributos das observações:

```
model = LogisticRegression(max_iter=2000, solver='lbfgs')
rfe = RFE(model, n_features_to_select=4)
fit3 = rfe.fit(X, y)
```

Visualizar as variáveis selecionadas:

```
cols = fit3.get_support(indices=True)
df.iloc[:,cols]
```

Como resultado obtemos quase a mesma combinação anterior e aparece mais a variável *familia* que indica se existem casos de diabetes na família.

2.8.4 Ensembles Methods

Métodos de agrupamento¹, como o algoritmo *Random Forest*, podem ser usados para estimar a importância de cada atributo, e retorna um valor para cada um, quanto mais alto esse, maior sua importância.

Aplicar o algoritmo:

```
model = RandomForestClassifier(n_estimators=10)
model.fit(X, y)
RandomForestClassifier(bootstrap=True, class_weight=None,
criterion='gini', max_depth=None, max_features='auto',
max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, n_estimators=10,
n_jobs=None, oob_score=False, random_state=None,
verbose=0, warm_start=False)
```

E podemos gerar uma visualização:

```
feature_importancia = pd.DataFrame(model.feature_importances_,
index = X.columns, columns=['importancia']).sort_values('importancia',
ascending=False)
feature_importancia
```

Porém é preferível vê-las através de um gráfico:

```
feature_importancia.plot(kind='bar')
```

Obtemos como resultado:

¹São métodos que utilizam vários algoritmos de aprendizado para obter um melhor desempenho preditivo.

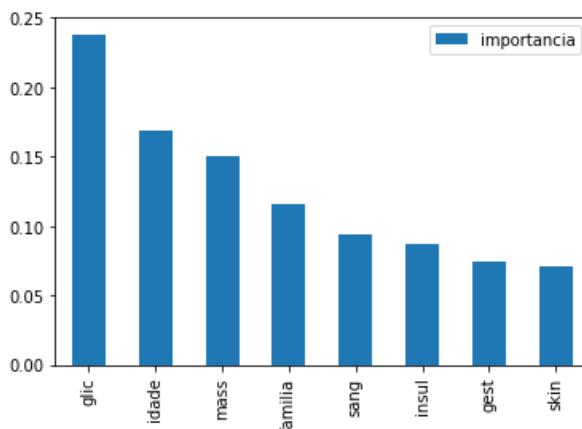


Figura 2.10: Grau de Importância dos Atributos

E afinal de contas com tudo o que vimos. Qual Método utilizar?

- Usar *RFE* caso tenha recursos computacionais para isso.
- Ao trabalhar com Classificação e os atributos forem numéricos, utilizar *f_classif* ou *mutual_info_classif*.
- Ao trabalhar com Regressão e os atributos forem numéricos, utilizar *f_regression* ou *mutual_info_regression*.
- Ao trabalhar com atributos categóricos utilizar *Chi Squared*.

2.9 K Fold Cross Validation

Vimos como encontrar os melhores atributos preditores para trabalhar, porém verificamos um pequeno problema: qual a forma em se escolher o modelo ideal² para nossa base de dados? O que fazemos é testar um modelo várias vezes obtendo pedaços diferentes de treino e teste a cada vez.

Usaremos a mesma base com Casos de Diabetes. Realizar a importação das bibliotecas:

```
from pandas import read_csv
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
%matplotlib inline
```

Em seguida colocar as observações em um *DataFrame*:

```
colnames = ['gest', 'glic', 'sang', 'skin', 'insul', 'mass', 'familia', 'idade',
'conf']
```

²"Ideal"é apenas um conceito para designar qual terá uma melhor acurácia (não usemos como sinônimo para perfeito ou o melhor)

```
df = read_csv('pima-indians-diabetes.data.csv', names=colnames)
df.head()
```

Para não gerar códigos repetitivos, criar uma lista com todos os modelos que executaremos:

```
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('DT', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
```

Trabalharemos com Regressão Logística (LR), Análise Descriminante (LDA), *K-Nearest Neighbors* (KNN), Árvore de Decisão (DT), *Gaussian* tipo NB e SVM. Avaliamos a acurácia de cada um dos modelos:

```
acuracia = []
for sigla, modelo in models:
    kfold = KFold(n_splits=10, random_state=7, shuffle=True)
    resultado = cross_val_score(modelo, X, Y, cv=kfold, scoring='accuracy', n_jobs=-1)
    acuracia.append(resultado.mean())
    print("%s: %f (%f)" % (sigla, resultado.mean(), resultado.std()))
```

O objeto **KFold** criado executa 10 vezes (definido em *n_splits*) cada um dos modelos mantendo um estado de aleatoriedade de 7 registros. O método *cross_val_score* realiza todo o trabalho que tivemos para executar um modelo, dividir a base de dados em treino e teste obtendo o score de cada execução e obtemos o mesmo resultado final, a cada resposta podemos definir o número de vezes a executar (definido em *n_jobs*) neste caso será executado somente 1 vez (o valor é -1).

Obtemos como resultado a lista de 10 valores para cada um dos modelos, para facilitar, trazemos a média (que também será adicionada a lista *acuracia*) e o desvio padrão:

```
LR: 0.778640 (0.047350)
LDA: 0.766969 (0.047966)
KNN: 0.710988 (0.050792)
DT: 0.696770 (0.046741)
NB: 0.759142 (0.038960)
SVM: 0.760458 (0.034712)
```

Também podemos ter isso de forma gráfica:

```
fig = plt.figure(figsize = (8,4))
axes = fig.add_axes([0.0, 0.0, 1.0, 1.0])
axes.set_title('Comparação dos Modelos');
axes.bar([item[0] for item in models], acuracia)
plt.show()
```

E obtemos como resultado:

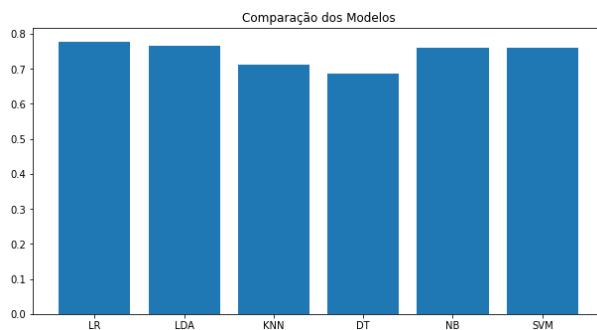


Figura 2.11: Melhor Acurácia dos Modelos

2.10 Matriz de Confusão

A Matriz de Confusão nos auxilia a medir a precisão do nosso modelo. A acurácia é uma boa medida porém um tanto falha, suponhamos que estejamos para realizar um trabalho com dados de pacientes que desejam saber a possibilidade de desenvolver uma determinada doença ou não. São quatro respostas possíveis que o nosso modelo pode prover:

- **Verdadeiro Positivo (TP - true positive)**: no conjunto da classe real, o resultado foi correto. O paciente desenvolveu a doença e o modelo previu que iria desenvolver. (T & T)
- **Falso Positivo (FP - false positive)**: no conjunto da classe real, o resultado foi incorreto. O paciente desenvolveu a doença e o modelo previu que não iria desenvolver. (T & F - Erro tipo 2)
- **Falso Negativo (FN - false negative)**: no conjunto da classe real, o resultado foi incorreto. O paciente não desenvolveu a doença e o modelo previu que iria desenvolver. (F & T - Erro tipo 1)
- **Verdadeiro Negativo (TN - true negative)**: no conjunto da classe real, o resultado foi correto. O paciente não desenvolveu a doença e o modelo previu que não iria desenvolver. (F & F)

Os casos 2 e 3 são erros no modelo, sendo que o 2 é considerado um pior tipo. Para começar com a nossa prática importar as bibliotecas necessárias:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sn
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.linear_model import LogisticRegression
%matplotlib inline
```

A biblioteca `sklearn.metrics` na versão 0.22 ganhou o método **confusion_matrix**. Somente para entendermos como funciona a Matriz de Confusão, suponhamos que foi realizado um teste com base em características de pacientes, existe uma possibilidade de desenvolver (D) ou não (S) uma determinada doença:

```
y_true = pd.Series(['D', 'D', 'D', 'D', 'D', 'D', 'S', 'S', 'S', 'S', 'S', 'S'])
y_pred = pd.Series(['D', 'D', 'S', 'D', 'D', 'D', 'S', 'D', 'S', 'S', 'D', 'S'])
```

A série contendo `y_true` é o resultado real (ou verdadeiro) e `y_pred` foi o resultado que o modelo disse que iria ocorrer. Se fossemos somente pela acurácia obtemos 13 respostas sendo 4 delas erradas, porém como saber se o modelo está realmente agindo bem e o mais importante onde está se confundindo?

```
conf = confusion_matrix(y_true, y_pred)
print(conf)
```

Ao usarmos o método `confusion_matrix` obtemos a seguinte Matriz que nos auxilia a responder essas questões:

```
[[5 1][3 4]]
```

O eixo **X** da Matriz representa o que foi predito e **y** o que realmente aconteceu pelo modelo. Nas diagonais obtemos as corretas, cinco que estão Doentes e quatro que estão sadios e o modelo acertou. Porém, três não estão doentes mas foi previsto que estariam (se pensarmos a notícia não é tão ruim - do ponto de vista para o paciente, por isso esse é o erro tipo 1) e um está doente porém prevemos que não estaria (péssima notícia, por isso esse é o erro tipo 2). Visualizar resultados assim pode ser bem complexo, então tentemos uma adaptação do Mapa de Calor da biblioteca **Seaborn**:

```
data = {
    'Ocorreu': y_true,
    'Predito': y_pred
}
df = pd.DataFrame(data, columns=['Ocorreu', 'Predito'])
conf = pd.crosstab(df['Ocorreu'], df['Predito'], rownames=['Ocorreu'],
                    colnames=['Predito'])
res = sn.heatmap(conf, annot=True, fmt='0f', annot_kws={"size":12},
                  cmap=plt.cm.Blues)
plt.show()
```

E obtemos como resultado:

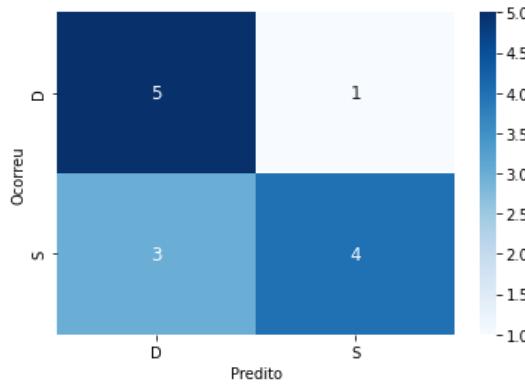


Figura 2.12: Mapa de Calor mostrando a Matriz de Confusão

Quanto mais escuro, maior a quantidade de elementos, assim podemos rapidamente avaliar como nosso modelo se comporta (o ideal é que as diagonais fiquem bem escuras enquanto que as extremidades claras).

2.10.1 Em valor ou percentual?

Um detalhe muito comum de acontecer é: devemos mostrar o valor em decimal ou percentual? Ou seja as quantidades reais das amostras ou um percentual do todo? Usar o seguinte código para gerar uma matriz de um teste realizado com imagens:

```
conf_arr = np.array([[88,14,4],[12,85,11],[5,15,91]])
sum = conf_arr.sum()
df_cm = pd.DataFrame(conf_arr,
    index = [ 'Cão', 'Gato', 'Coelho'],
    columns = [ 'Cão', 'Gato', 'Coelho'])
res = sn.heatmap(df_cm, annot=True, vmin=0.0, vmax=100.0, cmap=plt.cm.Blues)
plt.yticks([0.5,1.5,2.5], [ 'Cão', 'Gato', 'Coelho'], va='center')
plt.title('Matriz de Confusão')
plt.show()
```

E obtemos como resultado:

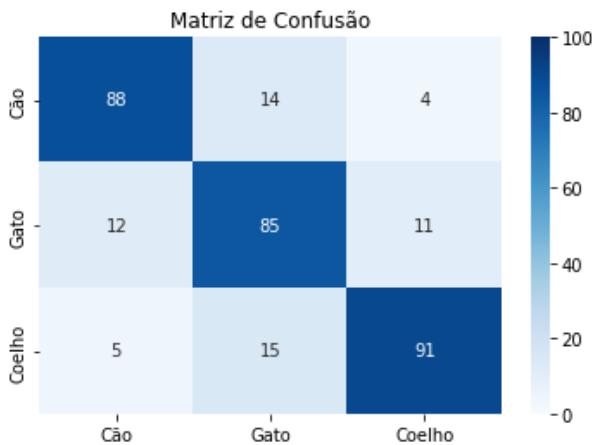


Figura 2.13: Comparativo numérico na Matriz de Confusão

E ao realizarmos um ajuste:

```
conf_arr = conf_arr * 100.0 / ( 1.0 * sum )
conf_arr /= 100
df_cm = pd.DataFrame(conf_arr,
    index = [ 'Cão', 'Gato', 'Coelho'],
    columns = [ 'Cão', 'Gato', 'Coelho'])
res = sn.heatmap(df_cm, annot=True, vmin=0.0, vmax=0.3, fmt='%.2%', cmap=plt.cm.Blues)
plt.title('Matriz de Confusão (em %)')
plt.show()
```

Obtemos a seguinte matriz agora com o resultado em percentual:

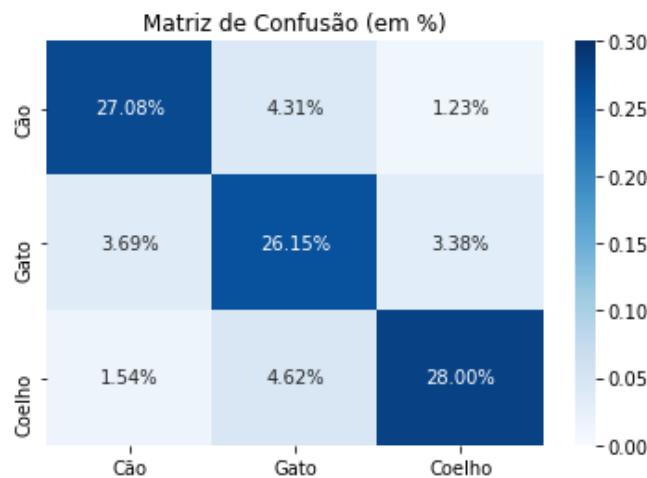


Figura 2.14: Comparativo percentual na Matriz de Confusão

Qual dos dois é melhor? A resposta seria: que fica mais fácil para que o Cientista de Dados consiga explicar o resultado para seu público com a maior clareza, não existe uma regra definida para isso.

2.10.2 Na prática

Voltamos ao nossos dados sobre Diabetes, ler os dados:

```
colnames = ['gest', 'glic', 'sang', 'skin', 'insul', 'mass', 'familia', 'idade',
           'conf']
df = pd.read_csv('pima-indians-diabetes.data.csv', names=colnames)
df.head()
```

Deixar somente os atributos que nos interessam:

```
df = df.drop(columns=['sang'], axis=1)
df = df.drop(columns=['insul'], axis=1)
df = df.drop(columns=['gest'], axis=1)
df = df.drop(columns=['skin'], axis=1)
df.head()
```

Dica 2.5: 100% de Precisão. Sempre que ocorrer 100% podemos ligar todas as antenas pois existe um erro, nenhum modelo possui essa previsão tão perfeita. Considere também que abaixo de 70% não é preditivo.

Agora acontece um erro clássico, a base contém *conf* o atributo alvo. Se seguimos em frente para separar as bases em teste e treino o modelo provavelmente acusa uma acurácia (errada) com 100% de precisão, é como se ele estivesse colando as respostas. Então devemos isolar esse atributo:

```
target = df['conf']
df = df.drop(columns=['conf'], axis=1)
```

E assim podemos separar as observações em treino e teste:

```
X_train, X_test, y_train, y_test = train_test_split(df, target, test_size = .25)
```

Usamos uma medida de 25%, ou seja 75% para treino e o restante para testar nosso modelo. Existem 768 registros não nulos no total, temos como resultado final: 576 para o modelo treinar e 192 para testar. Normalmente deixamos 25% para teste e verificação da acurácia do modelo esse número pode ser aumentado ou diminuído conforme seus dados, não existe uma regra definida.

Conforme nossa verificação do melhor modelo, vimos que devemos trabalhar com **Regressão Logística**, então executar o modelo e verificar a acurácia:

```
clf = LogisticRegression(max_iter=10000)
clf.fit(X_train, y_train)
print('Acurácia:', clf.score(X_test, y_test))
```

Esse resultado pode variar mas está em torno dos 76%, o problema é, onde esse modelo está errando? Executar a matriz de confusão para descobrir:

```
y_pred = clf.predict(X_test)
conf = confusion_matrix(y_test, y_pred)
data = {
    'Ocorreu': y_test,
    'Predito': y_pred
}
df2 = pd.DataFrame(data, columns=['Ocorreu', 'Predito'])
conf = pd.crosstab(df2['Ocorreu'], df2['Predito'], rownames=['Ocorreu'],
                    colnames=['Predito'])
res = sn.heatmap(conf, annot=True, fmt='%.0f', annot_kws={"size":12},
                  cmap=plt.cm.Blues)
plt.show()
```

E obtemos como resultado:

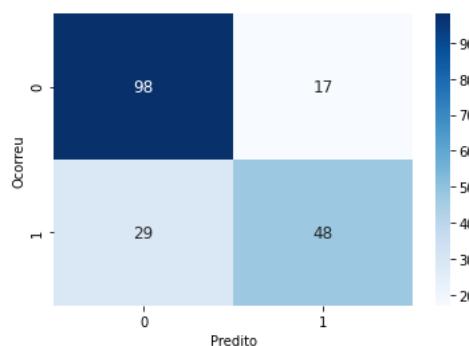


Figura 2.15: Resultado da Matriz de Confusão

Vemos que o modelo se comporta bem nos casos que o paciente não tem diabetes, acerta 85,2% e erra 14,8% das vezes. Já quando tem a doença acerta 62,3% e erra 37,7% das vezes. Ou seja, para melhorarmos a acurácia precisamos de mais dados com pacientes que desenvolveram a diabetes.

2.11 Curva ROC e Valor AUC

Entre todas as teorias vistas para Ciência de Dados este é um dos conceitos mais simples e ao mesmo tempo mais complicado (acho que é equivalente a Matriz de Confusão e inclusive depende dela).

Na teoria **ROC** (*Receiver Operating Characteristic*, algo como Característica de Operação do Receptor) é uma curva de probabilidade. É criada ao traçar a taxa de verdadeiro-positivo (TPR - true positive rate) contra a taxa de falsos-positivos (FPR - false positive rate). Que taxas são essas? Voltando ao conceito de Matriz de Confusão a TPR e FPR são calculadas pelas fórmulas:

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{FPR} = \frac{\text{FP}}{\text{TN} + \text{FP}}$$

Figura 2.16: Fórmulas para o cálculo da ROC

Ou seja, número de vezes que o classificador acertou a predição contra o número de vezes que errou. **AUC** (*Area Under the Curve*) representa a área da ROC, considera-se como o grau ou medida de separabilidade. Quanto maior o valor, melhor é o modelo em prever ou (por exemplo) em distinguir entre pacientes com e sem uma determinada doença.

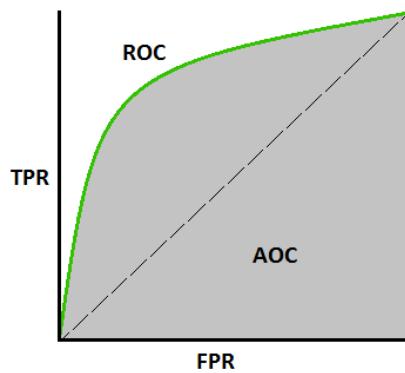


Figura 2.17: Fórmulas para o cálculo da ROC

Vejamos um simples exemplo para entendermos como esse processo funciona:

```
import pandas as pd
from sklearn import metrics
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_score
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
%matplotlib inline
```

Apenas para entendermos como aquilo tudo que foi escrito no início da seção funciona, usar a função *make_classification* para produzir alguns dados aleatórios:

```
X, y = make_classification(n_samples = 10000, n_features=10, n_classes = 2, flip_y =
0.5)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = .25)
```

Foram geradas 10.000 amostras com 10 campos e 2 classes (traduzindo para o português isso significa valores 0 e 1), separamos 25% para testar nosso modelo. Usar o Modelo de Regressão Logística para analisar esses dados:

```
model = LogisticRegression(solver='liblinear', penalty='l2', C=0.1)
model.fit(X_train, y_train)
print('Acurácia', model.score(X_test, y_test))
```

Como os dados são randômicos o resultado pode variar, mas chegamos em torno de 70%. E agora podemos calcular o **AUROC** (*Area Under the Receiver Operation Characteristics*):

```
y_prob = model.predict_proba(X_test)[:,1]
fpr, tpr, _ = metrics.roc_curve(y_test, y_prob)
auroc = float(format(metrics.roc_auc_score(y_test, y_prob), '.8f'))
print(auroc)
```

Existe 72% de área preenchida. Qual o motivo da diferença? Estamos levando em consideração as taxas TPR e FPR, não apenas acertou/errou. **AUC** resume a curva **ROC** num único valor que é o cálculo da “área sob a curva”, porém apresentar a informação assim não tem graça, colocar de forma gráfica:

```
plt.plot(fpr, tpr, label='Curva ROC (area = %.2f)' % auroc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('Taxa de Falso Positivo')
plt.ylabel('Taxa de Verdadeiro Positivo')
plt.title('Exemplo do ROC')
plt.legend(loc="lower right")
plt.show()
```

E obtemos como resultado:

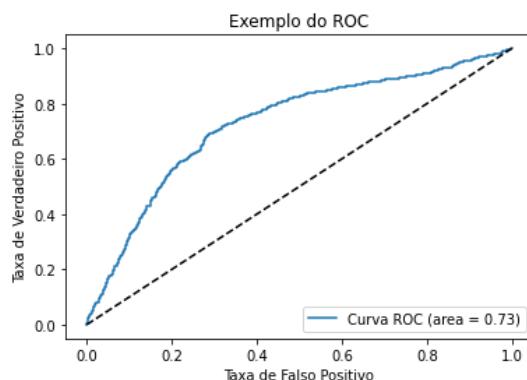


Figura 2.18: Visão da ROC

2.11.1 Na prática

Voltamos ao nossos dados sobre Diabetes, ler os dados:

```
colnames = ['gest', 'glic', 'sang', 'skin', 'insul', 'mass', 'familia', 'idade',
           'conf']
df = pd.read_csv('pima-indians-diabetes.data.csv', names=colnames)
df.head()
```

Deixar somente os atributos que nos interessam:

```
df = df.drop(columns=['sang'], axis=1)
df = df.drop(columns=['insul'], axis=1)
df = df.drop(columns=['gest'], axis=1)
df = df.drop(columns=['skin'], axis=1)
df.head()
```

Isolar o atributo alvo e retirá-lo dos dados.

```
target = df['conf']
df = df.drop(columns=['conf'], axis=1)
```

Separar as observações em treino e teste:

```
X_train, X_test, y_train, y_test = train_test_split(df, target, test_size = .25)
```

Testamos dois modelos de agrupamento para saber qual o melhor comportamento com esses dados. Para facilitar nossa vida, criar um método que retorna a acurácia de um determinado modelo:

```
def score(mdl, Xtrn, Xtst, ytrn, ytst):
    mdl.fit(Xtrn, ytrn)
    return float(format(mdl.score(Xtst, ytst), '.8f'))
```

Recebemos o modelo a ser treinado, e o conjunto de atributos separados em X e y (tanto para treino como teste) e devolvemos a acurácia do modelo. De modo semelhante:

```
def auroc(ytst, yprob):
    fpr, tpr, _ = metrics.roc_curve(ytst, yprob)
    auc = float(format(metrics.roc_auc_score(ytst, yprob), '.8f'))
    return fpr, tpr, auc
```

Esse outro método recebe o y de teste e o de probabilidades e retorna o **FPR**, **TPR** e **AUC**. Os modelos de agrupamento para realizarmos nosso teste são **Regressão Logística** e **Floresta Aleatória**. Para cada um desses, basicamente, são os mesmos comandos.

Regressão Logística:

```
clfRL = LogisticRegression(max_iter=1000)
print("Acurácia RL:", score(clfRL, X_train, X_test, y_train, y_test))
y_probRL = clfRL.predict_proba(X_test)[:,1]
```

```
fprRL, tprRL, aucRL = auroc(y_test, y_probRL)
print("AUC RL", aucRL)
```

Floresta Aleatória:

```
clfRF = RandomForestClassifier(n_estimators=1000)
print("Acurácia RF:", score(clfRF, X_train, X_test, y_train, y_test))
y_probRF = clfRF.predict_proba(X_test)[:,1]
fprRF, tprRF, aucRF = auroc(y_test, y_probRF)
print("AUC RF:", aucRF)
```

Ambos os modelos trabalham com grupos de 1.000, sendo que para a **Regressão Logística** isso é considerado de iterações realizadas enquanto que para a **Floresta Aleatória** o número de árvores de decisão utilizadas.

Dica 2.6: Como esses modelos trabalham?. Não devemos nos preocupar nesse momento como esses modelos processam, em capítulos subsequentes trataremos separadamente cada um deles. Aqui veremos apenas os conceitos que serão necessários para a melhor escolha e avaliação dos modelos.

E obtemos o seguinte resultado, que pode variar de acordo com o que foi separado de treino e teste: A **Regressão Logística** atingiu um resultado melhor tanto na acurácia (73,95%) quanto na curva com quase 80% de área ocupada enquanto que a **Floresta Aleatória** ficou com quase 71% de acurácia e 77,35% de área. Às vezes o resultado de acurácia pode até ser similar, mas raramente a área ocupada será igual:

```
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fprRL,tprRL,label="RL " + str(aucRL))
plt.plot(fprRF,tprRF,label="RF " + str(aucRF))
plt.xlabel('Taxa de Falso Positivo')
plt.ylabel('Taxa de Verdadeiro Positivo')
plt.title('Detecção da Diabetes')
plt.legend(loc="lower right")
plt.show()
```

E obtemos como resultado:

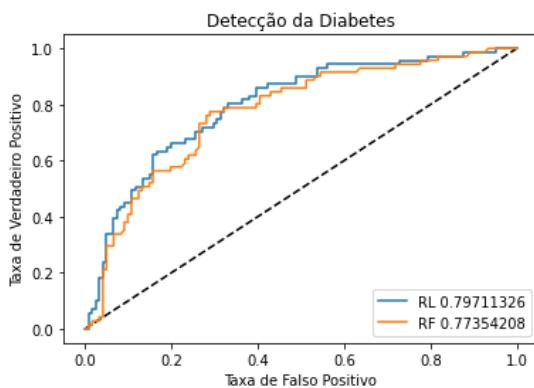


Figura 2.19: Performance dos Modelos

2.12 Terminamos?

Como conceitos sim, mas como prática permita-me deixar um exercício. A **Scikit-Learn** nos oferece uma base sobre de cistos (Câncer de Mama) encontrados em pacientes de *Wisconsin* com classificação se é malígnio (212 amostras) ou benigno (357 amostras). Para usá-la importar a biblioteca:

```
from sklearn.datasets import load_breast_cancer
```

Carregamos os dados:

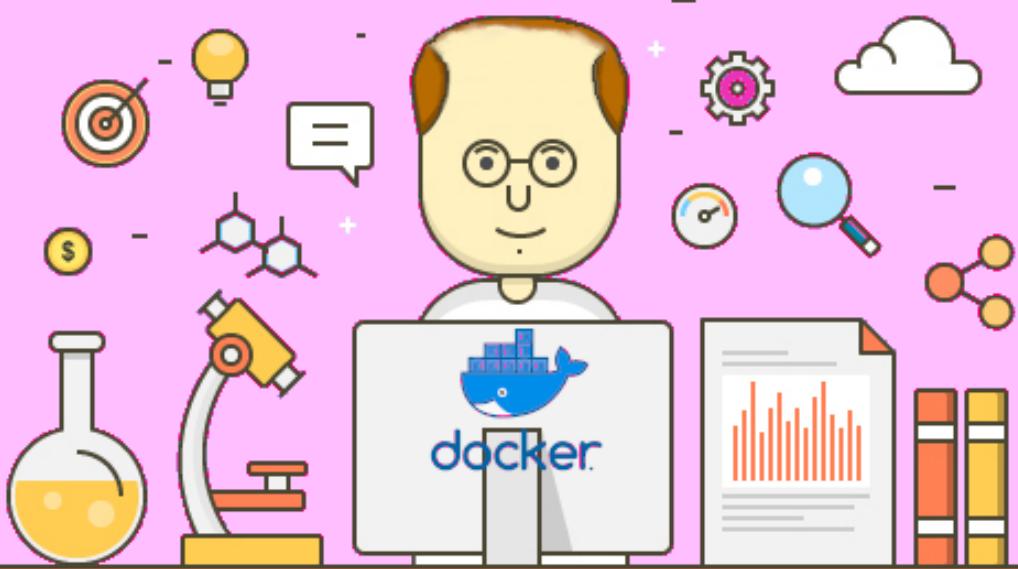
```
cancer = load_breast_cancer()
```

E obtemos um objeto *Bunch* da Scikit-Learn com os 569 casos registrados. Para criar as nossas bases de treino e teste usar o comando:

```
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
    test_size = .25)
```

A variável *data* contém os atributos preditores enquanto que *target* se a paciente teve um tipo malígnio ou não (atributo dependente). Pronto, agora é com você. Aplicar os conhecimentos que vimos nesse capítulo para definir quais são os melhores atributo a usar e o modelo colhendo todos os resultados possíveis.

No endereço https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html está disponibilizada a documentação sobre esta base.



3. EDA

F Nós torturamos os dados até eles confessarem. (Ricardo Cappra - Cientista de Dados)

3.1 Passos da EDA

EDA é fundamental para entender qualquer conjunto de observações. É aqui podemos obter informações e fazer descobertas. Aqui colocamos o conhecimento para trabalhar. Acrônimo para *Exploratory Data Analysis* (Análise Exploratória de Dados), desempenha um papel crítico na compreensão do quê? por que? e como? na declaração do problema.

É a primeira ação realizada na ordem das operações que um Cientista de Dados deve executar ao receber uma nova fonte de observações e a declaração de problema. Tratamos EDA como uma série de técnicas utilizadas como forma de entendermos os diversos aspectos que temos para trabalhar.



Figura 3.1: Passos da EDA

A preparação dos dados para análise é inevitável, e a maneira como fazemos isso define a sua qualidade. Na prática o que faremos nesse capítulo é compreendermos o que temos a nossa disposição para trabalhar.

Normalmente as observações se dividem em atributos **Preditores** (Entradas) e **Alvo** (saída). Uma vez que os localizemos, devemos identificar seu tipo e categoria.

3.2 Passo 1 - Entender os Dados

Nesta fase precisamos compreender o que temos a nossa disposição. Começamos o processo com a importação das bibliotecas:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Vamos trabalhar com três bibliotecas básicas, que como já mencionamos devemos conhecê-las a fundo: **Pandas** para análise, **MatPlotLib** e **SeaBorn** para mostrar em forma gráfica.

Agora precisamos dos dados, para isso usaremos o arquivo *StudentsPerformance.csv*:

```
df = pd.read_csv('StudentsPerformance.csv')
```

Nessa fase compreendemos melhor o que temos na nossa mão, **Pandas** é ideal para essa tarefa. Seu funcionamento é como um "Editor de Planilha", dessa forma que devemos encarar essa biblioteca, sua diferença básica é a nomenclatura de como o *DataFrame* (e não Planilha) é visualizado:

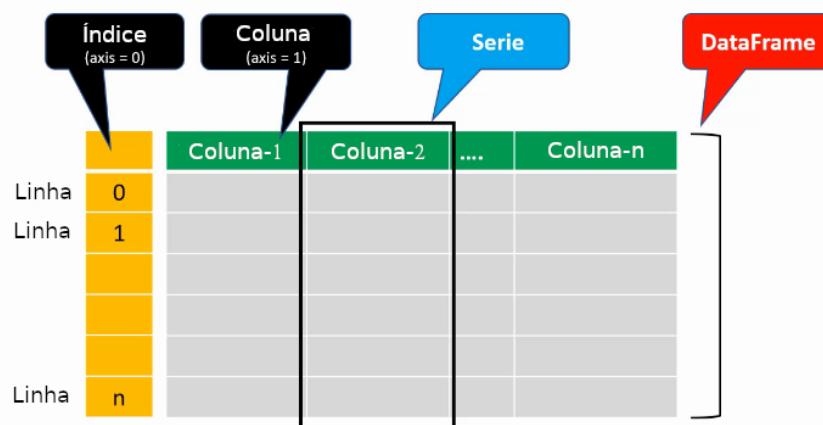


Figura 3.2: Visão Pandas

Uma coluna aqui é vista como uma *Serie* e *index* é o que mantém a "cola" das series juntas. Dois comandos são básicos para visualizarmos o *DataFrame*:

```
df.head()
```

Porém nesse livro não utilizaremos o termo "coluna" e sim "atributo" (consideremos ambos como sinônimos). Mostra as primeiras observações, como parâmetro podemos passar a quantidade. E:

```
df.tail()
```

Mostra as últimas observações e também como parâmetro podemos passar a quantidade. O que temos até o momento? Sabemos é uma base sobre estudantes e as linhas são: gênero, etnicidade, nível de escolaridade dos pais, forma de alimentação, realizou um teste de preparação do curso, nota de matemática, nota de leitura e nota de escrita.

Então só com esses dois comandos já podemos saber sobre qual assunto iremos tratar: estudantes que

realizaram provas e em quais condições. Quantos registros temos a nossa disposição? Ou quais são os nomes dos atributos?

```
print("Tamanho: ", df.shape)
print("Nome dos Atributos: ", df.columns)
```

As variáveis *shape* e *columns* do *DataFrame* respondem aos questionamentos. De forma mais completa podemos usar:

```
df.info()
```

Nos mostra inclusive o tipo de cada atributo e se contém ou não elementos nulos. Temos 3 atributos que são do tipo inteiro (*int64*) e podemos analisá-los com o comando:

```
df.describe()
```

Nos fornece as informações estatísticas básicas como média, desvio padrão, menor valor, máximo, 1º quartil (25%), 2º quartil ou mediana (50%), 3º quartil (75%) e o maior valor. Ou seja, as informações para a montagem de um **BoxPlot**. Vamos montá-lo para melhor visualizar as observações:

```
fig, axes = plt.subplots(1, 3, figsize=(10,4))
axes[0].boxplot(df['math score'])
axes[0].set_title("Matemática")
axes[1].boxplot(df['reading score'])
axes[1].set_title("Leitura")
axes[2].boxplot(df['writing score'])
axes[2].set_title("Escrita")
plt.show()
```

E obtemos como resultado:

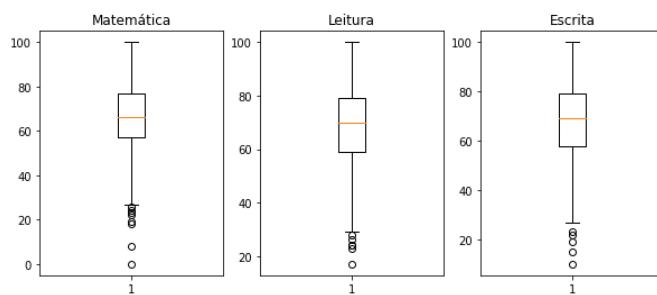


Figura 3.3: BoxPlot das Notas

Boxplot¹ é um gráfico que avalia a distribuição das observações. É formado exatamente com os atributos que mostramos na função *describe()*. Porém suas hastas (inferiores e superiores) se estendem do quartil inferior (ou superior) até o menor valor não inferior (ou superior) ao limite. São calculados da seguinte forma:

¹Diagrama de Caixa se prefere, foi atribuída ao matemático **John W. Tukey** (1915 –2000), curiosamente algumas literaturas chamam de "Tukey BoxPlot", mas se realizar uma pesquisa ninguém sabe ao certo quem criou realmente esse diagrama.

- Limite inferior: $Q_1 - 1,5 \times (Q_3 - Q_1)$.
- Limite superior: $Q_3 + 1,5 \times (Q_3 - Q_1)$.

Resumidamente é formado da seguinte maneira:

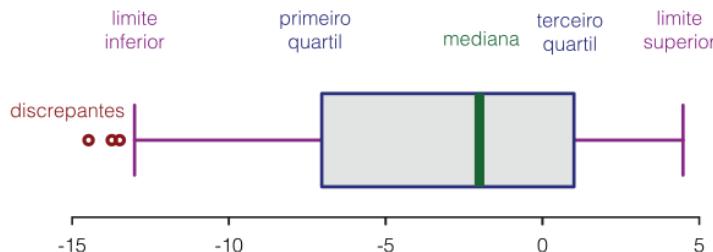


Figura 3.4: Estrutura do BoxPlot

Esses pontos "discrepantes" podem ocorrer acima ou abaixo dos limites, são chamados de *Outliers*. Não é necessariamente um erro, podemos classificá-lo como uma anomalia curiosa e que merece nossa atenção.

3.2.1 Localizar os Outliers

Para achar esses *Outliers* isolamos os três atributos numéricos:

```
X = df.iloc[:, 5:8].values
```

E criamos um novo *DataFrame* somente com a modificação de nome por um número:

```
pd.options.display.float_format = '{:.1f}'.format
xDF = pd.DataFrame(X)
```

Para quê isso serve? A função *describe()* cria um *DataFrame*, podemos percorrê-lo, porém fica muito mais simples se cada atributo for um numeral, pois assim podemos usar um comando *for* para isso:

```
z = xDF.describe()
for t in z:
    iqr = z[t][6] - z[t][4]
    extMenor = z[t][4] - (iqr * 1.5)
    extMaior = z[t][6] + (iqr * 1.5)
    print('Para o índice %d valores devem estar abaixo de %.2f e acima de %.2f' %
          (t, extMenor, extMaior))
```

E obtemos o seguinte resultado:

Para o índice 0 valores devem estar abaixo de 27.00 e acima de 107.00

Para o índice 1 valores devem estar abaixo de 29.00 e acima de 109.00

Para o índice 2 valores devem estar abaixo de 25.88 e acima de 110.88

Pelo BoxPlot todos os valores estão abaixo, então para localizá-los:

```
matOutliers = (X[:,0] < 27)
df[matOutliers]
```

E assim mostramos todas as observações que a nota de matemática (índice 0) é abaixo do valor 27. Proceder de mesmo modo para as notas de leitura (índice 1) e escrita (índice 2) e assim desvendar quais são os *Outliers*.

Podemos também analisar graficamente e visualizar a Distribuição Normal de cada atributo, por exemplo para a nota de Escrita:

```
sns.kdeplot(df['writing score'], shade=True)
plt.show()
```

Obtemos como resultado:

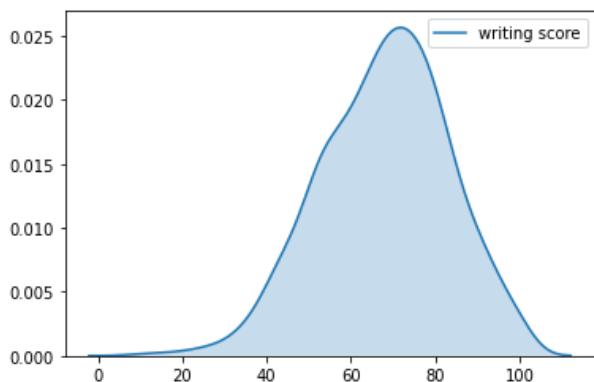


Figura 3.5: Distribuição das observações para Nota de Escrita

E assim verificamos como cada atributo numérico se comporta.

3.2.2 Tratar Atributos Categóricos

Sabemos que os primeiros cinco atributos do *Dataframe* são categóricos, porém conforme a função *info()* o tipo delas estás *object*. É interessante mudarmos para o tipo caractere para evitarmos quaisquer problemas futuros.

```
df['gender'] = df['gender'].astype(pd.StringDtype())
df['race/ethnicity'] = df['race/ethnicity'].astype(pd.StringDtype())
df['parental level of education'] = df['parental level of
    education'].astype(pd.StringDtype())
df['lunch'] = df['lunch'].astype(pd.StringDtype())
df['test preparation course'] = df['test preparation
    course'].astype(pd.StringDtype())
```

E ao aplicarmos uma nova chamada a função *info()* vemos que os tipos agora estão corretos. Quantos tipos únicos existem para cada atributo?

```
df.nunique()
```

Mostra a quantidade de valores não repetidos de cada atributos (inclusive os numéricos). E agora sabemos que temos: 2 gêneros, 5 etnicidades, 6 níveis de escolaridade dos pais, 2 formas de alimentação e 2 tipos para teste de preparação do curso. Mas quem são?

```
print("Gênero: ", df['gender'].unique())
print("Etnicidade: ", df['race/ethnicity'].unique())
print("Escolaridade dos Pais: ", df['parental level of education'].unique())
print("Refeição: ", df['lunch'].unique())
print("Realizou Preparatório: ", df['test preparation course'].unique())
```

3.3 Passo 2 - Limpar os Dados

A limpeza dos dados trata de muitos problemas como informação repetida, valores faltantes (que podem ser descobertos por associação) e inconsistentes. Para esse último tipo o pior caso são os nulos. (In)felizmente essa base está horrível para essa fase e assim pegamos um outro arquivo **titanic.csv**:

```
df = pd.read_csv('titanic.csv')
df.head()
```

Repetimos todo o processo da fase anterior para descobrirmos de que se tratam as observações e descobrimos que são os passageiros (sobreviventes ou não - atributo *Survived* - sendo este atributos alvo) do famoso **RMS Titanic**, este foi pensado para ser o navio mais luxuoso e seguro de sua época e supostamente "inafundável". Como sabemos em sua viagem inaugural de *Southampton* para *Nova Iorque* afundou no dia 14 de abril de 1912 com mais de 1.500 pessoas a bordo. Porém esta base contém apenas 891 registros.

Ao aplicarmos a função `info()` percebemos que os atributos *Age* (idade), *Cabin* (número da cabine) e *Embarked* (local de Embarque) possuem valores faltantes. Que valores são esses?

```
print(df.isnull().sum())
```

Sabemos que faltam: 177 em **Age**, 687 em **Cabin** e 2 em **Embarked**. Também podemos mostrar exclusivamente os que faltam, isso é útil para quando temos muitos atributos no modelo:

```
null_value_stats = df.isnull().sum(axis=0)
null_value_stats[null_value_stats != 0]
```

Ou ainda criar uma função personalizada que retorna um *Dataframe* com a informação mais completa o possível (inclusive com seu percentual):

```
def mostrarNulos(data):
    null_sum = data.isnull().sum()
    total = null_sum.sort_values(ascending=False)
    percent = ((null_sum /
        len(data.index))*100).round(2).sort_values(ascending=False)
    df_NULL = pd.concat([total, percent], axis=1, keys=['Tot.Nulo', 'Perc.Nulo'])
    df_NULL = df_NULL[(df_NULL.T != 0).any()]
    return df_NULL
```

E ao chamá-la:

```
df_Age = mostrarNulos(df)
df_Age.head()
```

Obtemos como resultado:

	Tot.Nulo	Perc.Nulo
Cabin	687	77.10
Age	177	19.87
Embarked	2	0.22

Figura 3.6: Nulos e Perceitual da Base Titanic

Lidar com esses tipos de nulos é complicado pois não temos como consultar e o máximo que podemos fazer é podá-los da nossa base ou atribuir um valor genérico que não afete nosso resultado (como o caso de *Embarked*). Porém **Número da Cabine** é um dado relevante? Essa é a principal pergunta que nos devemos fazer, por exemplo existe algum modelo preditivo que possa nos dizer que se estivéssemos em determinada cabine no navio sobreviveríamos ou não? Entretanto **Idade** é um dado relevante (lembra da frase: mulheres e crianças primeiro), então essa é uma característica que pode ser essencial.

Criar uma função com um gráfico para mostrar, por idade, como estão as observações:

```
def executarGrafico():
    try:
        sns.distplot([df['Age']])
        plt.show()
    except ValueError as err:
        print(err)
```

Agora a cada vez que chamarmos essa função obtemos como resultado:

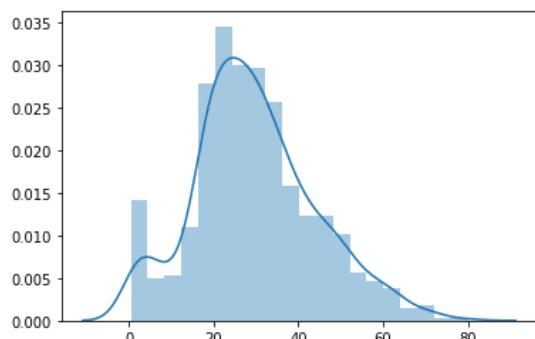


Figura 3.7: Gráfico de Idade do Titanic

Dica 3.1: Imputação ou retirada de valores. Como tratamos de adicionar ou retirar elementos na base a cada vez devemos ler novamente as observações contidas no arquivo CSV.

Porém em algumas versões da *SeaBorn* este pode apresentar erro devido a presença dos nulos, é ideal que os retiremos do *DataFrame* para evitarmos problemas. Em muitas biografias encontramos algo do tipo: "atribuir um valor (preferencialmente *outlier*) para estes tipos". Tentaremos essa técnica com os seguintes comandos:

```
df['Age'].fillna(-25, inplace=True)
executarGrafico()
```

Obtemos como resultado:

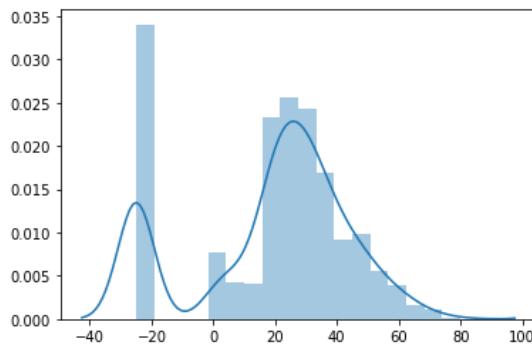


Figura 3.8: Gráfico de Idade do Titanic com Outliers

Nosso gráfico de idade ganhou uma nova barra, que sabemos com valores não existentes, também podemos atribuir qualquer outro valor como por exemplo a média:

```
df['Age'] = df['Age'].fillna(df['Age'].mean())
executarGrafico()
```

Ou a mediana (função `median()`) que resultaria em um gráfico completamente esquisito. Sendo assim vamos cortar esses valores:

```
df = df.dropna(axis=0)
executarGrafico()
```

E teremos a seguinte situação:

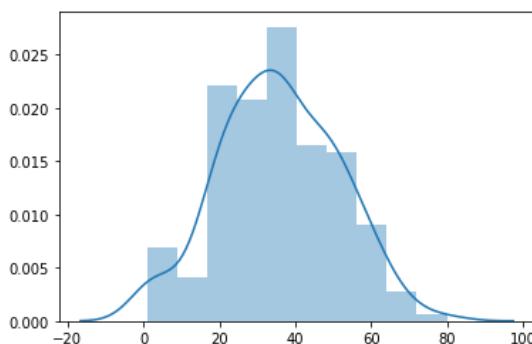


Figura 3.9: Gráfico de Idade do Titanic sem nulos

O que aconteceu? O comando executado eliminou todas as linhas que possuíam valores nulos, e o atributo *Cabin* interferiu e nos deixou, conforme pode ser mostrado com a função *info()*, somente 183 registros no total. Ou seja, o corte que devemos aplicar deve ser cirúrgico e somente no atributo que representa a idade.

```
df['Age'] = df['Age'].dropna(axis=0)
executarGrafico()
```

O que nos resulta no mesmo gráfico mostrado no início desta e 891 registros. Como citamos, podemos retirar o atributo *Cabin* para que este não interfira mais em futuras análises:

```
df = df.drop(['Cabin'], axis=1)
```

Dica 3.2: Ferramenta para Limpeza dos Dados. Conhece o **OpenRefine?** é uma ferramenta gratuita dedicada a limpeza e tratamento das observações, baixe uma apostila gratuitamente na minha página do Academia.edu (<https://iesbpreve.academia.edu/FernandoAnselmo>).

3.4 Passo 3 - Relacionamento entre os Atributos

Vamos retomar nossa base de **Estudantes** e verificarmos como os atributos se relacionam:

```
df = pd.read_csv('StudentsPerformance.csv')
df.corr()
```

E temos um valor que corresponde ao grau de relacionamento, um intervalo de -1 a 1, sendo quanto mais próximo do mínimo menor é seu grau de relacionamento. Porém é muito mais fácil de visualizarmos esse resultado com um Mapa de Calor:

```
rel = df.corr()
sns.heatmap(rel, xticklabels=rel.columns, yticklabels=rel.columns, annot=True)
plt.show()
```

Obtemos como resultado:



Figura 3.10: Mapa de Calor dos Relacionamentos

Vemos que as notas de Escrita e Leitura possuem um forte grau de relacionamento, como se uma fosse a responsável pela outra. Já a de matemática interfere mais na nota de leitura.

Curiosamente se aplicarmos isso na base do **Titanic** vemos que os atributos mais importantes para *Fare* (sobreviveu) que é nosso alvo são: *Fare* que é o valor pago pela passagem e *Parch* que se refere a quantidade de pais. Ou seja, os mais ricos e se a criança tinha ou não os pais a bordo de modo a colocá-las no bote salva vidas.

Outra maneira de visualizarmos, também de forma gráfica, é através da dispersão de valores:

```
sns.pairplot(df)
plt.show()
```

Obtemos como resultado:

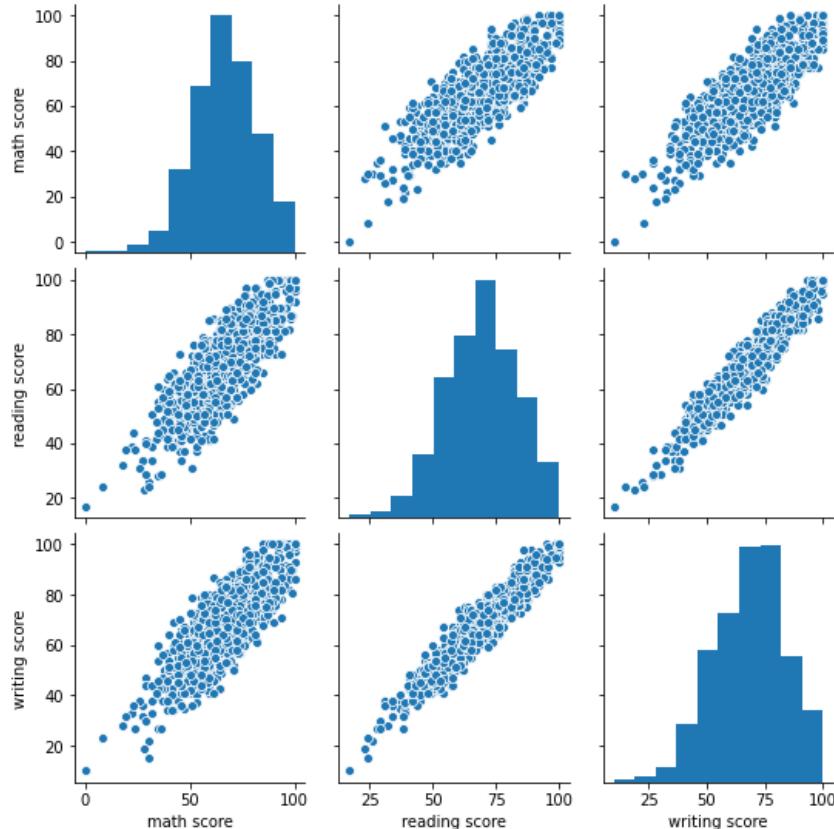


Figura 3.11: Dispersão Associada

Quanto mais juntos aparecem os pontos mais relacionadas estão. Podemos isolar as notas de Escrita e Leitura em um único gráfico, por exemplo:

```
sns.regplot(x='writing score', y='reading score', data=df)
plt.show()
```

Esta função executa um ajuste e plotagem simples com base no modelo de Regressão Linear. E obtemos como resultado:

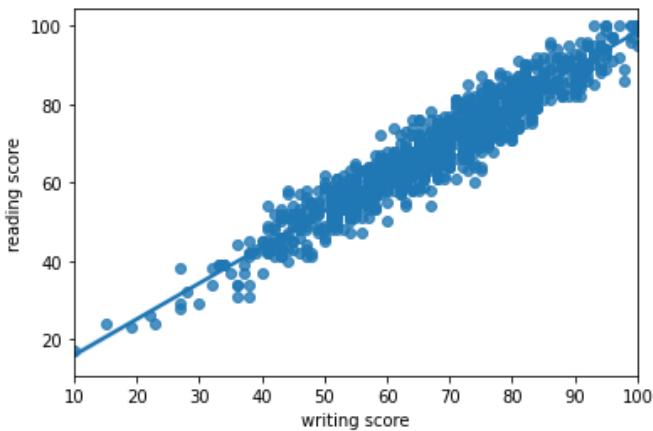


Figura 3.12: Notas de Leitura e Escrita

Porém o mais interessante é colorir os pontos de forma diferente com base em um atributo categórico que pode ser uma causa (para uma nota alta ou baixa), por exemplo o quanto a alimentação interferiu na nota:

```
sns.lmplot(x='writing score', y='reading score', hue='lunch', data=df)
plt.show()
```

A função *lmplot()* combina *regplot()* com a classe **FacetGrid**. Esta auxilia visualizar a distribuição de um determinado atributos, bem como o relacionamento entre os vários separadamente dentro de subconjuntos das observações. Obtemos como resultado:

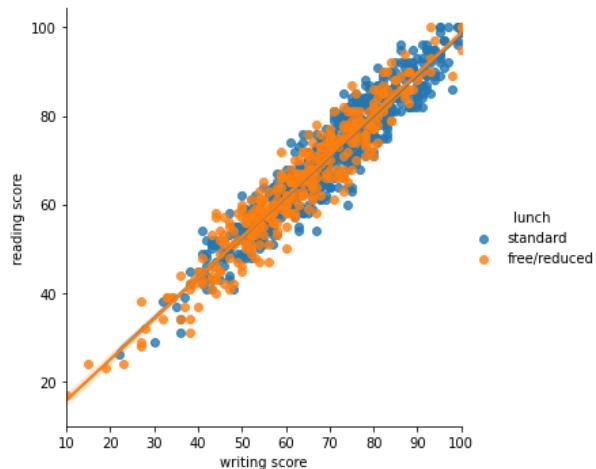


Figura 3.13: Nota associada a Alimentação - RegPlot

Uma melhor forma de visualizar é usar a função *relplot()* que fornece acesso a várias funções diferentes no nível de eixos que mostram o relacionamento entre dois atributos com mapeamentos semânticos de subconjuntos:

```
sns.relplot(x='writing score', y='reading score', hue='lunch', data=df)
plt.show()
```

Obtemos como resultado:

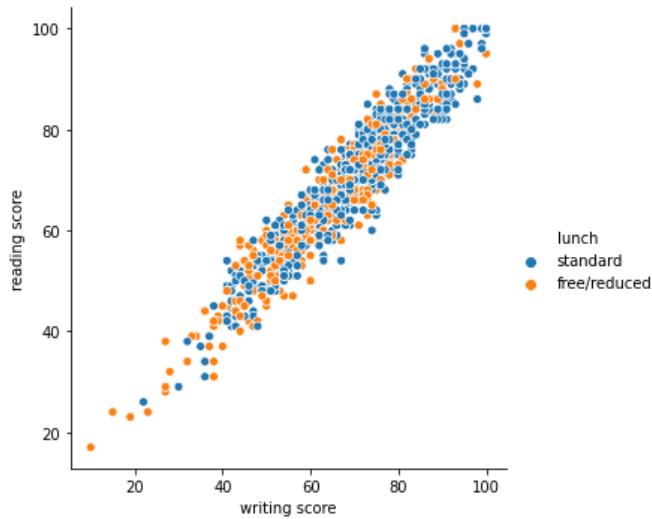


Figura 3.14: Nota associada a Alimentação - RelPlot

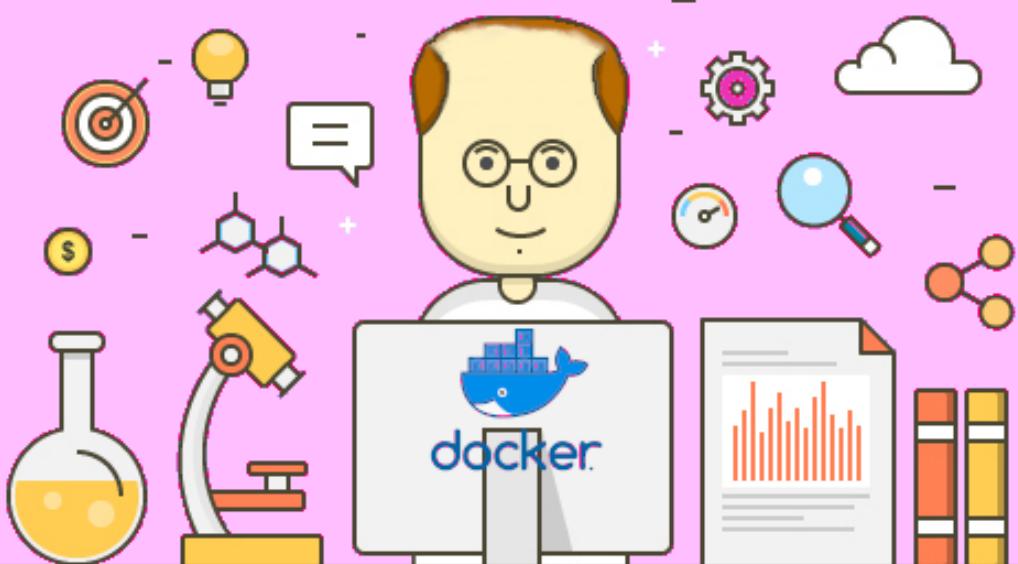
Ou seja, podemos responder várias perguntas apenas com a verificação do relacionamento entre os atributos. Como forma de fixar o conhecimento procure realizar o mesmo teste com outros atributos categóricos e descobrir como se comportam em relação a nota, se existe ou não interferência.

3.5 Conclusão

Mantemos em mente que EDA é um aspecto central da *Data Science*, que às vezes é esquecido. O primeiro passo de qualquer ação que tomemos é conhecer as observações: entendê-las e familiarizar-se. Quais são as respostas que estamos tentando obter? Quais são os atributos e o que significam? Como é a aparência de uma perspectiva estatística? As observações estão formatadas corretamente? Possuem valores ausentes? duplicados? E quanto aos *outliers*? Conhecemos eles? Ou seja, devemos responder a esses questionamentos.

É necessário muito trabalho de preparação, pois no mundo real dados raramente são limpos e homogêneos. Costumamos dizer que 80% do tempo valioso em um Cientista de Dados é utilizado com a localização, limpeza e organização das observações. Os 20% restantes são destinados a realizar as análises.

Agora estamos prontos para começarmos a explorar diversas observações com a utilização dos modelos.



4. Modelos Iniciais

F Na vida, não existe nada a temer, mas a entender. (Marie Curie - Cientista e Vencedora 2 vezes do Prêmio Nobel)

4.1 K-Means

Acredito que K-Means seja o modelo mais simples para começarmos, este é um algoritmo de Aprendizado Não Supervisionado, ou seja, não necessita de atributos alvo para agir, sua função é de separar as observações em grupos de modo que possamos observar melhor os dados.

Sendo assim, nosso problema para usar esse algoritmo é exatamente achar esse k ideal de modo que os grupos sejam separados coerentemente. Para isso existe uma técnica interessante chamada "Técnica do Cotovelo"(Elbow Technique).

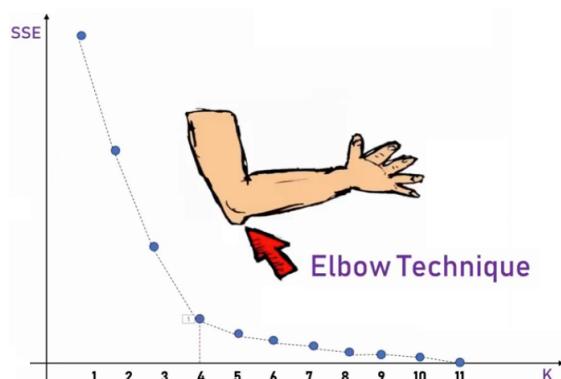


Figura 4.1: Técnica do Cotovelo

Exatamente na posição 4 existe uma "quebra" para passar ao próximo valor, usamos para definir essa quebra o SSE¹ (*Sum Squared Error*).

¹Soma Residual dos Quadrados, é a soma dos resíduos elevado por 2. É uma medida da discrepância entre os dados e um modelo de estimativa. Um valor pequeno SSE indica um ajuste apertado do modelo aos dados.

4.2 Aplicação da Técnica

Para achar o k ideal ativamos nosso JupyterLab personalizado que criamos com o Docker e na primeira célula importamos as bibliotecas necessárias:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import scale
from sklearn.cluster import KMeans
from matplotlib import pyplot as plt

%matplotlib inline
```

Importamos a biblioteca Pandas e a Numpy para manipularmos os dados, a Scikit-Learn para usarmos o modelo K-Means e Matplot para vermos o resultado em um gráfico. A última linha é utilizada para mostrar os gráficos no Jupyter. Próximo passo consiste em ler os dados, baixamos o arquivo **gameML.csv** e na posição do nosso arquivo **.ipynb** criamos uma subpasta chamada **bases** e nesta colocamos o arquivo.

```
df = pd.read_csv('bases/gameML.csv', delimiter=';')
df.head()
```

E como resultado da execução dessa célula devemos ter:

	Nome	Idade	Salário
0	Daenerys Targaryen	27	70000
1	Jon Snow	29	90000
2	Gregor Ciegane	29	61000
3	Arya Stark	28	60000
4	Tyrion Lannister	42	150000

Figura 4.2: Idades e Salários da Empresa GameML

No arquivo existem 3 campos: nome do funcionário, idade e salário, se plotarmos os dados entre idade e salário em gráfico:

```
plt.scatter(df['Idade'], df['Salário'])
```

Obtemos como resultado:

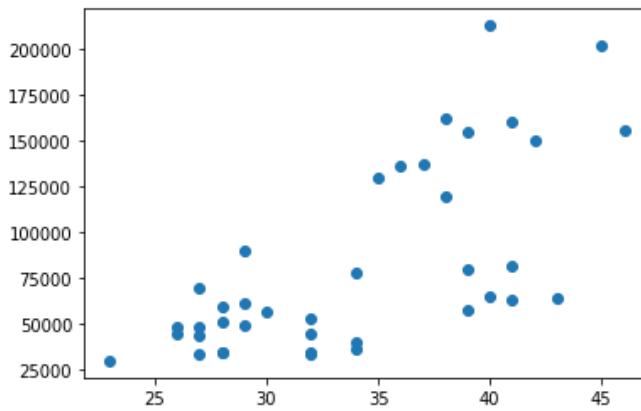


Figura 4.3: Idades e Salários da Empresa GameML

Quantos grupos de dados podemos distinguir? Para localizarmos a quantidade ideal aplicamos a técnica do cotovelo que consiste de:

```
k_rng = range(1,10)
sse = []
for k in k_rng:
    km = KMeans(n_clusters=k)
    km.fit(df[['Idade','Salário']])
    sse.append(km.inertia_)
plt.xlabel('K')
plt.ylabel('SSE (Sum Squared Error)')
plt.plot(k_rng, sse)
```

Criar um range de 1 a 10 (um simples número máximo de possíveis *clusters*), para cada valor treinamos o modelo com as variáveis e obtemos o valor do atributo **inércia**. O algoritmo agrupa dados e procura separar amostras em n grupos de igual variação, minimizando um critério conhecido como inércia ou **RSS** dentro do *cluster*. O que estamos fazendo na prática é colocar o valor 1 para o **K** e guardar esse valor, em seguida o valor 2 e assim sucessivamente. Por fim plotamos esse valor em um gráfico e obtemos como resultado:

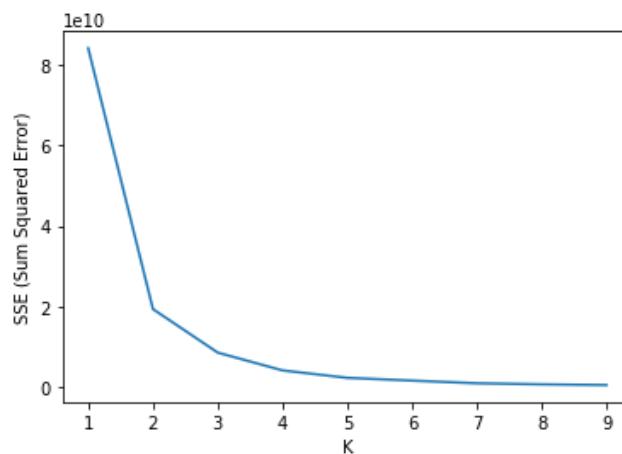


Figura 4.4: Gráfico com os valores de Inércia

E vemos nosso "cotovelo" da curva bem na posição **3**, marcando assim o número ideal de clusters.

4.3 Plotagem do Resultado do Modelo

Um detalhe interessante que para usarmos o algoritmo K-Means, devemos colocar os dados em "escala", vamos tentar usar o modelo sem proceder dessa forma:

```
km = KMeans(n_clusters=3)
y_predict = km.fit_predict(df[['Idade', 'Salário']])
df['ypred'] = y_predict
df.head()
```

Já sabemos que o valor de 3 clusters é o ideal, então realizamos o treinamento com os atributos Idade e Salário para montarmos um novo atributo com o resultado dessa predição (somente para que o gráfico apareça separado por cores). E plotamos o gráfico:

```
cores = np.array(['green', 'red', 'blue'])
plt.scatter(x=df['Idade'],
            y=df['Salário'],
            c=cores[df['ypred']], s=50)
```

Obtemos como resultado:

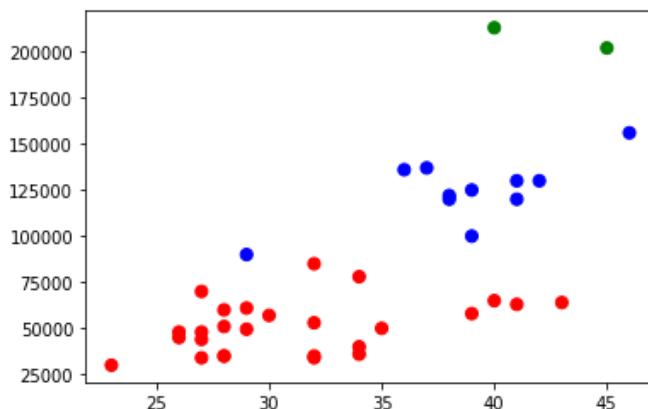


Figura 4.5: Separados por Grupo

E parece que obtemos algo bem errado com alguns *outliers* aparecendo, observamos o ponto azul no meio dos vermelhos e um outro azul isolado perto dos verdes. Então antes de treinarmos esse algoritmo devemos colocar os dados na mesma escala, isso é feito assim:

```
df['Salário'] = scale(df.Salário)
df['Idade'] = scale(df.Idade)
df.head()
```

Os atributos **idade** e **salário** possuem valores bem diferentes e distantes e isso gera problemas para nosso resultado final, colocar em escala e aproximar (sem modificar o resultado final) os valores seria algo criar

um modelo de um prédio porém mantendo as mesmas proporções do prédio original.

A função da **Scikit-Learn** que realiza este processo é chamada *scale()* e colocamos em escala os atributos se visualizarmos nossos dados agora veremos que o atributo **idade** possui valores entre -2 e 2 enquanto que **salário** entre -1.5 e 3 (são diferentes exatamente para manter a proporcionalidade). Retornamos ao mesmo processo de treinamento:

```
km = KMeans(n_clusters=3)
y_predict = km.fit_predict(df[['Idade', 'Salário']])
df['ypred'] = y_predict
df.head()
```

Plotamos novamente o gráfico e agora como resultado obtemos:

```
cores = np.array(['green', 'red', 'blue'])
plt.scatter(x=df['Idade'],
            y=df['Salário'],
            c=cores[df.ypred], s=50)
```

E obtemos como resultado:

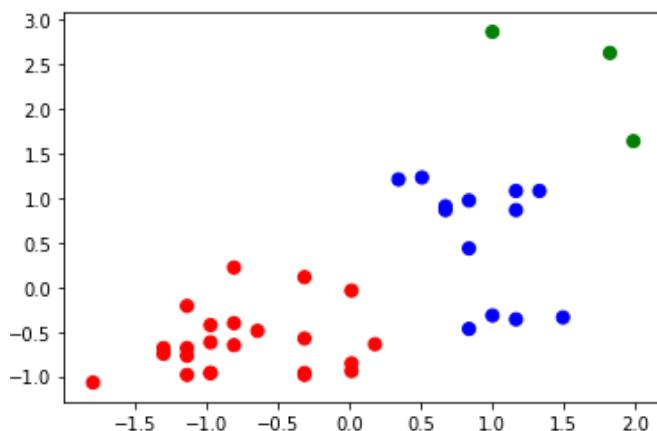


Figura 4.6: Separados por Grupo em Escala

Que é um resultado bem mais coerente.

4.4 K-Nearest Neighbors

Ou simplesmente KNN. Modelos assim existem pois muitas pessoas pensam que separar em *clusters* não auxilia na predição, pois bem nosso próximo modelo é um supervisionado e destinado a Predição por Clusterização (ou se prefere por proximidade dos grupos). KNN que normalmente é usado para a predição de imagens como: Isso é um Gato? Ou não é um Gato? Porém ao invés de imagens, vamos usar uma base bem conhecida chamada **Flores Íris** para entendermos seu comportamento.

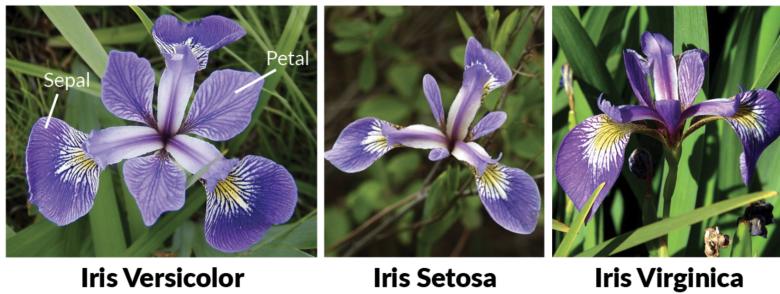


Figura 4.7: Flores Iris

Nessa base existem três espécies separadas: Versicolor, Setosa e Virgíñica. E para distingui-las utilizamos 2 medidas da sépala e da pétala (largura e altura de cada). O problema é que algumas espécies causam as maiores confusões em nossos modelos. Para realizarmos uma predição sobre essa base importamos nossas bibliotecas:

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn import neighbors

%matplotlib inline
```

Usamos a **NumPy** para gerenciamento dos dados. **Matplotlib** para plotarmos os gráficos. Da **Scikit-Learn** obtemos os nossos dados através do pacote **datasets** e para separar uma massa de teste contamos com o *train_test_split*. E a *neighbors* contém o nosso algoritmo. O próximo passo consiste na preparação dos dados:

```
iris = datasets.load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=1234)
```

O método *load_iris()* traz a nossa base em uma matriz de dados. Nossa base está dividida em *data* que contém os *features* preditores (tamanho e largura da sépala e tamanho e largura da pétala, que colocaremos em *X*) e *target*, *feature* que contém a definição da espécie (0 representa **Setosa**, 1 para **Versicolor** e 2 para **Virgíñica** que colocaremos em *y*). Usamos o método *train_test_split* para retirar 20% dos dados como amostra de teste e obtemos quatro agrupamentos:

- **X_train**, com os dados para treino do algoritmo.
- **X_test**, com os dados para teste.
- **y_train**, com o resultado para o treino.
- **y_test**, com o resultado para o teste.

Com nossos dados preparados vamos treinar o modelo:

```
clf = neighbors.KNeighborsClassifier()
clf.fit(X_train, y_train)
```

```
print(clf.score(X_test, y_test))
```

E conseguimos uma boa acurácia com incríveis 96% de precisão, agora é vermos na prática como isso funciona.

4.4.1 Predição com K-Nearest Neighbors

Primeiro vamos mostrar os dados:

```
cores = np.array(['green', 'red', 'blue'])
subplt1 = plt.scatter(x=x[:, 0], y=x[:, 1], c=cores[y], s=50)
```

Pegamos as duas primeiras variáveis tamanho e largura da sépala e obtemos como resultado:

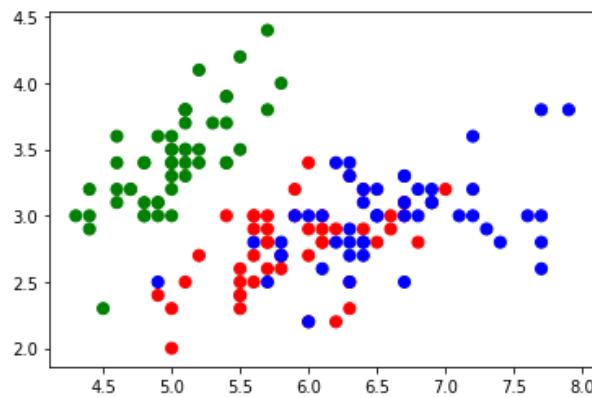


Figura 4.8: Comparar tamanho e largura da Sépala

Não nos perdemos nas cores **Verde** é Setosa, **Vermelho** é Versicolor e **Azul** é Virgínica. Agora vamos pensar em um ponto qualquer nesse espaço, por exemplo:

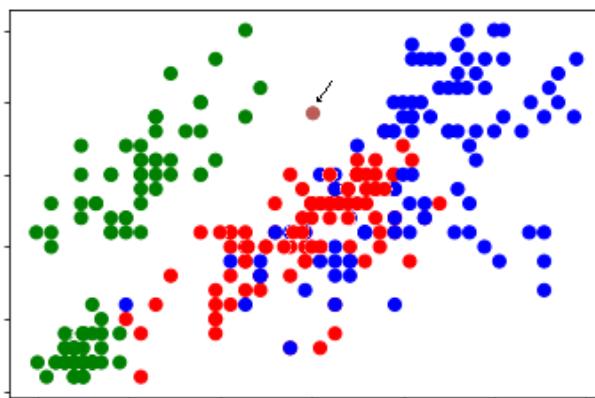


Figura 4.9: Localizar o Ponto Roxo

O ponto roxo fica na interseção do 4º valor de X e y qual cor real ele seria? Observamos no gráfico anterior que os pontos são 6,0 e 4,0 porém nos falta o valor para mais dois atributos tamanho e largura da pétala:

```
cores = np.array(['green', 'red', 'blue'])
subplt1 = plt.scatter(x=X[:, 2], y=X[:, 3], c=cores[y], s=50)
```

E obtemos como resultado:

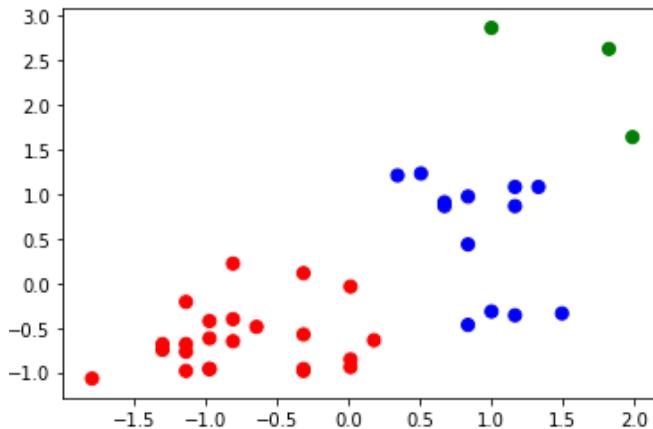


Figura 4.10: Comparar tamanho e largura da Pétala

E verificamos que na interseção do 4º valor de X e y obtemos os valores 4,0 e 2,0. Agora que obtemos os quatro valores podemos realizar uma predição:

```
predicao = clf.predict([[6.0, 4.0, 4.0, 2.0]])
print(predicao)
```

E resulta que o modelo prevê que é do tipo [1], ou seja, um ponto vermelho da espécie **Versicolor**.

4.5 Análise de Cluster

Então sabemos agora que ambos modelos K-Means e KNN trabalham utilizando *clusters* (agrupamentos) sendo que o primeiro é do tipo não supervisionado destinado a separação com base em um número de centroides (k) presentes e os valores médios mais próximos (isso representa uma distância Euclidiana entre as observações). Porém é necessário colocar os dados em escala para verificar se não ocorre nenhuma perturbação nesse centroide. Vamos importar algumas bibliotecas para realizarmos mais testes:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.preprocessing import scale
from sklearn.cluster import KMeans
from sklearn.metrics import classification_report

%matplotlib inline
```

Já passamos por todas e não desejo ser repetitivo porém dessa vez vamos utilizar a Pandas para manipular os dados e a classe *metrics* da SciKit-Learn para mostrar o comportamento do nosso modelo. Iremos continuar usando a base Iris e construímos um *DataFrame* somente com os dados dos atributos preditores porém guardaremos o atributo alvo para verificar como nosso modelo se comportou:

```
iris = datasets.load_iris()
X = scale(iris.data)
y = pd.DataFrame(iris.target)
y.columns = ['Targets']
variable_names = iris.feature_names
iris_df = pd.DataFrame(iris.data)
iris_df.columns = variable_names
iris_df.head()
```

E nosso *DataFrame* se apresenta da seguinte maneira:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Figura 4.11: DataFrame com os dados dos Atributos Preditores

O próximo passo é construir e treinar nosso modelo:

```
clustering = KMeans(n_clusters=3, random_state=5).fit(X)
```

Normalmente para treinar um modelo passamos dois conjuntos de dados, porém o K-Means só recebe um único conjunto, exatamente por não realizar previsões precisa apenas dos dados para separá-los em conjuntos. Mas como será que foi seu comportamento? Descobrimos isso comparando dois gráficos:

```
cores = np.array(['green', 'red', 'blue'])
relabel = np.choose(clustering.labels_, [1, 0, 2]).astype(np.int64)
plt.figure(figsize = [15, 5])

plt.subplot(1, 4, 1)
plt.scatter(x=iris_df['petal length (cm)'],
y=iris_df['petal width (cm)'],
c=cores[iris.target], s=50)
plt.title('Real (Pétala)')

plt.subplot(1, 4, 2)
plt.scatter(x=iris_df['petal length (cm)'],
y=iris_df['petal width (cm)'],
c=cores[relabel], s=50)
plt.title('KMeans (Pétala)')

plt.subplot(1, 4, 3)
plt.scatter(x=iris_df['sepal length (cm)'],
y=iris_df['sepal width (cm)'],
c=cores[iris.target], s=50)
```

```

plt.title('Real (Sépala)')

plt.subplot(1, 4, 1)
plt.scatter(x=iris_df['sepal length (cm)'],
y=iris_df['sepal width (cm)'],
c=cores[relabel], s=50)
plt.title('KMeans (Sépala)')

plt.show()

```

Usamos os mesmos conjuntos de cores para cada espécie, obtemos quatro gráficos comparativos: 1º largura e altura da Pétala e a cor será mostrada com base em nosso atributo alvo (ou seja o valor real), 2º o que o modelo achou que seria o correto, 3º largura e altura da Sépala e o 4º novamente como o modelo separou. E obtemos como resultado:

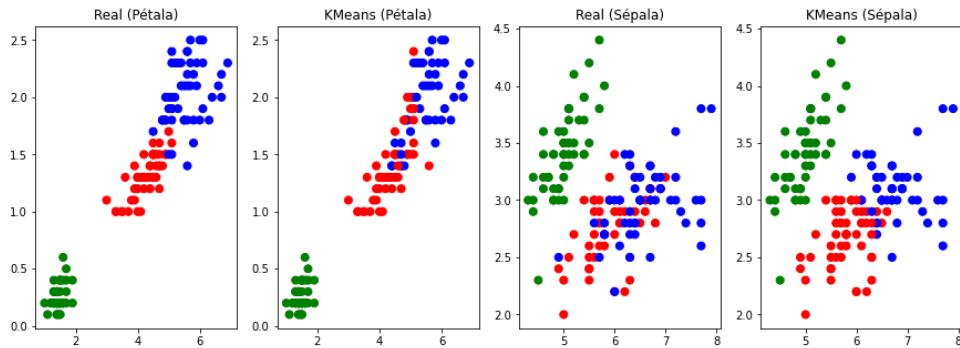


Figura 4.12: Comparativo entre o Real e o KMeans

Para pétala o **K-Means** quase acertou a posição de cada espécie, porém para Sépala aconteceram as maiores confusões, isso se deve ao fato do centroide. Para melhor avaliarmos nosso modelo precisamos de mais medidas: *Precision* (precisão) é a medida de relevância do modelo, *Recall* (revocação ou sensibilidade) se trata da medida de completude do modelo e *F1 Score* se trata de uma média ponderada entre *precision* e *recall*. Podemos obtê-las da seguinte forma:

```

metricas = classification_report(y, relabel)
print(metricas)

```

Obtemos como resultado:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	50
1	0.74	0.78	0.76	50
2	0.77	0.72	0.74	50
accuracy			0.83	150
macro avg	0.83	0.83	0.83	150
weighted avg	0.83	0.83	0.83	150

Figura 4.13: Relatório de Performance do K-Means

Precision - É a razão entre as observações positivas previstas corretamente e o total de observações positivas previstas. Calculada com a fórmula: $TP \div (TP + FP)$.

Recall - É a razão entre as observações positivas previstas corretamente e todas as observações da classe real. Calculada com a fórmula: $TP \div (TP + FN)$

F1 Score - Essa pontuação leva em consideração tanto os falsos positivos quanto os negativos. Intuitivamente, não é tão fácil entender como precisão, mas F1 é geralmente mais útil que *precision*, especialmente se estivermos com uma distribuição de classe desigual. $2 \times (recall \times precision) \div (recall + precision)$

Acurácia funciona melhor se os falsos positivos e negativos tiverem um custo semelhante. Se o custo for muito diferente, é melhor olharmos essas métricas.

4.6 Clusterização Hierárquica

Este é um modelo alternativo ao particionamento de *cluster* no conjunto de dados, pode ser aplicado para encontrar a distância entre cada ponto e seus vizinhos mais próximos e conectá-lo de forma ideal. Podemos mostrar o número de subgrupos com o auxílio de um Dendrograma².

É útil pois não existe necessidade de especificar o número de *clusters* (ou K) antes da análise e o dendrograma fornece uma representação visual desses. Vamos trazer para o conjunto de bibliotecas visto anteriormente mais três:

```
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import accuracy_score
```

Para este exemplo vamos utilizar outra base que está contida no arquivo **mtcars.csv** (trazer essa para a subpasta **/base**). E carregamos os dados do seguinte modo:

```
carros = pd.read_csv('bases/mtcars.csv')
carros.columns = ['name', 'mpg', 'cil', 'disp', 'hp', 'drat', 'wt', 'qsec', 'vs',
    'am', 'gear', 'carb']
X = carros[['mpg', 'disp', 'hp', 'wt']].values
y = carros['am'].values
```

Essa base contém 32 modelos de carros com os seguintes atributos: Nome, autonomia em milhas por galão, número de cilindros, deslocamento (medida de poder do carro em polegada cúbica), cavalos de força, relação do eixo traseiro, peso (em libras), eficiência do gasto de combustível (por 1/4 milha), motor (0 = V-shaped, 1 = straight), câmbio (0 = automática, 1 = manual), total de marchas e carburadores. Porém para não trabalharmos com tantos atributos vamos usar somente: consumo de gasolina (mpg), deslocamento (disp), cavalos de força (hp) e peso (wt) e o nosso objetivo e descobrir se o carro possui um câmbio manual ou automático.

Podemos montar o dendrograma do seguinte modo:

```
z = linkage(X, 'ward')
dendrogram(z, truncate_mode='lastp', p=12, leaf_rotation=45, leaf_font_size=15,
```

²É um gráfico em formato de árvore que mostra visualmente os relacionamentos entre as observações.

```

show_contracted=True)

plt.title('Dendograma')
plt.xlabel('Tamanho do Cluster')
plt.ylabel('Distância')
plt.axhline(y=500)
plt.axhline(y=150)
plt.show()

```

Obtemos como resultado:

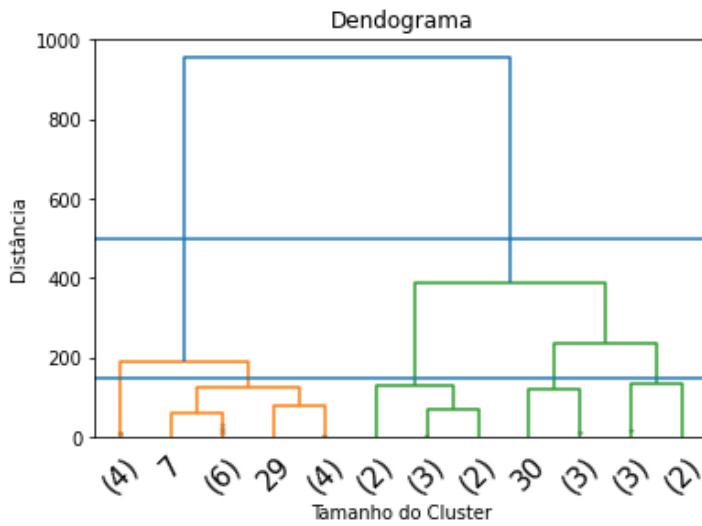


Figura 4.14: Dendrograma dos tamanhos do Cluster

O dendrograma mostra como cada *cluster* é composto e desenha um link em forma de U entre cada cluster e seus filhos. A parte superior indica uma mesclagem. Cada perna indica quais foram mesclados. O comprimento das pernas e do U representa a distância entre os filhos.

Para mesclar recursivamente o par de *clusters* e aumentar minimamente a distância de ligação utilizamos a função `AgglomerativeClustering()`. Essa possui dois parâmetros básicos: *affinity* e *linkage*.

affinity: métrica utilizada para calcular a ligação. Possui as seguintes opções:

- *euclidean* - é o único que aceita o parâmetro *linkage* como *ward*. Refere-se a distância euclidiana que pode ser provada pela aplicação repetida do teorema de Pitágoras.
- *l1* - critério de erro absoluto.
- *l2* - critério de erros quadrados (lembremos do RSS).
- *manhattan* - distância euclidiana ao quadrado.
- *cosine* - também chamada de Similaridade do Cosseno. É a distância do cosseno entre duas variáveis.
- *precomputed* - necessita de uma matriz de distância (em vez de similaridade) como entrada para o método de ajuste, pois X será considerado uma matriz.

linkage: define qual o critério de ligação usar. Determina qual distância usar entre os conjuntos de observação. Possui as seguintes opções:

- *ward* - minimiza a variação dos *clusters* que estão sendo mesclados.
- *average* - média das distâncias de cada observação dos conjuntos.
- *complete* - distâncias máximas entre todas as observações dos dois conjuntos.
- *single* - mínimo das distâncias entre todas as observações dos dois conjuntos.

Como escolher os parâmetros ideais? Fácil, testemos várias combinações e veremos qual possui uma melhor acurácia para os dados que estamos tratando:

```

hclusters1 = AgglomerativeClustering(n_clusters=2, affinity='euclidean',
                                      linkage='ward').fit(X)
print('Método 1:', accuracy_score(y, hclusters1.labels_))

hclusters2 = AgglomerativeClustering(n_clusters=2, affinity='euclidean',
                                      linkage='complete').fit(X)
print('Método 2:', accuracy_score(y, hclusters2.labels_))

hclusters3 = AgglomerativeClustering(n_clusters=2, affinity='euclidean',
                                      linkage='average').fit(X)
print('Método 3:', accuracy_score(y, hclusters3.labels_))

hclusters4 = AgglomerativeClustering(n_clusters=2, affinity='manhattan',
                                      linkage='single').fit(X)
print('Método 4:', accuracy_score(y, hclusters4.labels_))

hclusters5 = AgglomerativeClustering(n_clusters=2, affinity='manhattan',
                                      linkage='complete').fit(X)
print('Método 5:', accuracy_score(y, hclusters5.labels_))

hclusters6 = AgglomerativeClustering(n_clusters=2, affinity='manhattan',
                                      linkage='average').fit(X)
print('Método 6:', accuracy_score(y, hclusters6.labels_))

hclusters7 = AgglomerativeClustering(n_clusters=2, affinity='cosine',
                                      linkage='single').fit(X)
print('Método 7:', accuracy_score(y, hclusters7.labels_))

hclusters8 = AgglomerativeClustering(n_clusters=2, affinity='cosine',
                                      linkage='complete').fit(X)
print('Método 8:', accuracy_score(y, hclusters8.labels_))

hclusters9 = AgglomerativeClustering(n_clusters=2, affinity='cosine',
                                      linkage='average').fit(X)
print('Método 9:', accuracy_score(y, hclusters9.labels_))

```

Obtemos como resultado:

```

Método 1: 0.78125
Método 2: 0.4375
Método 3: 0.78125
Método 4: 0.625
Método 5: 0.71875
Método 6: 0.71875
Método 7: 0.3125
Método 8: 0.28125

```

```
Método 9: 0.1875
```

Assim para esse caso *Euclidian/Ward* ou *Manhattan/Complete* são os que melhor responderam ao nosso conjunto de dados com uma acurácia de 78,12%. Podemos inclusive tirar um relatório mais completo (como já vimos):

```
print(classification_report(y, hclusters1.labels_))
```

Obtemos como resultado:

	precision	recall	f1-score	support
0	0.88	0.74	0.80	19
1	0.69	0.85	0.76	13
accuracy			0.78	32
macro avg	0.78	0.79	0.78	32
weighted avg	0.80	0.78	0.78	32

Figura 4.15: Relatório de Performance da Clusterização Hierárquica

Só que ficou uma pergunta no ar, esse método se comporta melhor que um modelo de clusterização preditivo como o KNN?

4.6.1 Clusterização Hierárquica versus K-Nearest Neighbors

Para verificar como o KNN se comporta com os dados dos carros adicionamos mais quatro bibliotecas:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
```

Como já obtemos nossos dados, vamos apenas separá-los em bases de treino e teste:

```
X = preprocessing.scale(X)
X_treino, X_teste, y_treino, y_teste = train_test_split(X, y, test_size=.20,
    random_state=17)
```

Porém devemos sempre lembrar que os modelos de clusterização trabalham melhor quando os dados estão em escala, assim acertamos os atributos preditores antes de realizar a separação de 80% dos dados para treino e 20% para teste.

```
clf = KNeighborsClassifier()
clf.fit(X_treino, y_treino)
```

Treinamos nosso modelo e podemos avaliar o resultado:

```
y_predito = clf.predict(X_teste)
print(classification_report(y_teste, y_predito))
```

Obtemos como resultado:

	precision	recall	f1-score	support
0	0.80	1.00	0.89	4
1	1.00	0.67	0.80	3
accuracy			0.86	7
macro avg	0.90	0.83	0.84	7
weighted avg	0.89	0.86	0.85	7

Figura 4.16: Relatório de Performance do K-Nearest Neighbors

E na média percebemos que este se comporta melhor pois atinge resultados acima dos 80%.

4.7 Regressão Linear

A regressão linear tenta modelar o relacionamento entre dois atributos, através de ajustes sob uma equação linear dos dados observados. Um atributo é considerado **explicativo** e a outro **dependente**. Para simplificar um pouco, é uma técnica que utiliza valores de entrada para predizer os de saída (como por exemplo, prever o crescimento da população de um País) através da aplicação dos coeficientes (também chamados de peso) da equação linear.

Comecemos com a importação das bibliotecas que necessitamos:

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.linear_model import LinearRegression

%matplotlib inline
```

A classe *Linear Model* da *Scikit-Learn* o método *LinearRegression* para realizar nosso trabalho. Baixar a base de dados **PopBrasil.csv** que contém as observações de Crescimento da População Brasileira.

```
df = pd.read_csv('bases/PopBrasil.csv')
df.head()
```

Obtemos como resultado:

	Ano	Populacao
0	1960	72179226
1	1961	74311343
2	1962	76514328
3	1963	78772657
4	1964	81064571

Figura 4.17: Dados da População Brasileira

Devemos saber que o modelo trabalha com a relação entre atributos numéricos: explanatórios X e

dependentes y . Utiliza somente esse tipo devido aos ajustes matemáticos que são realizados e os pesos criados conforme a função minimiza os erros. Nossas observações são bem simples: obtemos atributos numéricos, "Ano" e "População". Para entendermos o relacionamento entre os atributos, plotamos esses em um gráfico:

```
plt.xlabel('Ano')
plt.ylabel('Quantidade da População')
plt.scatter(df.Ano, df.Populacao, color='red', marker='+')
```

Obtemos como resultado:

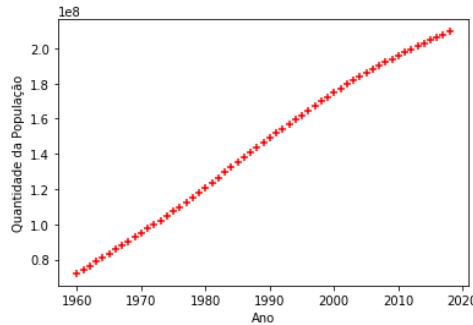


Figura 4.18: Dados da População Brasileira

E esta é a parte mais importante na execução desse modelo, a medida que alteramos o valor de "Ano" o valor de "População" também é afetado, ou seja, existe um relacionamento linear. Essa é a premissa básica para se usar este algoritmo, o relacionamento forte entre os atributos deve existir.

4.7.1 Aplicar a Regressão Linear

Agora que obtemos nossos atributos conferidos, basta treinarmos nosso modelo e obtermos nossa previsão:

```
reg = LinearRegression()
reg.fit(df[['Ano']], df.Populacao)
prev = reg.predict([[2020]])
print("Previsão 2020 é: %d" % prev)
```

E obtemos a predição da população brasileira para o ano de 2020, que é 221.322.254 de habitantes. Como a magia acontece? Pura matemática que é fornecida pela seguinte fórmula:

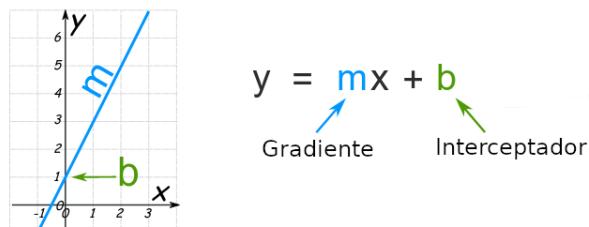


Figura 4.19: Base da Regressão Linear

Dica 4.1: Para saber mais. Se deseja conhecer mais sobre o assunto, visite a página: <https://www.mathsisfun.com/algebra/linear-equations.html> onde se obtém uma explicação mais completa.

E podemos reproduzir esse resultado pois o objeto treinado nos fornece tanto o valor do Gradiente (`coef_[0]`) quanto do Interceptador (`intercept_`). Então:

```
m = reg.coef_[0]
b = reg.intercept_
prev2020 = m * 2020 + b
print("Previsão 2020 é: %d" % prev2020)
```

E obtemos exatamente o mesmo resultado. Podemos traçar a "Reta da Regressão Linear", pois o modelo consegue predizer os resultados de cada ano:

```
plt.xlabel('Ano')
plt.ylabel('Quantidade da População')
plt.scatter(df.Ano, df.Populacao, color='red', marker='+')
plt.plot(df.Ano, reg.predict(df[['Ano']]), color='blue')
```

Obtemos como resultado:

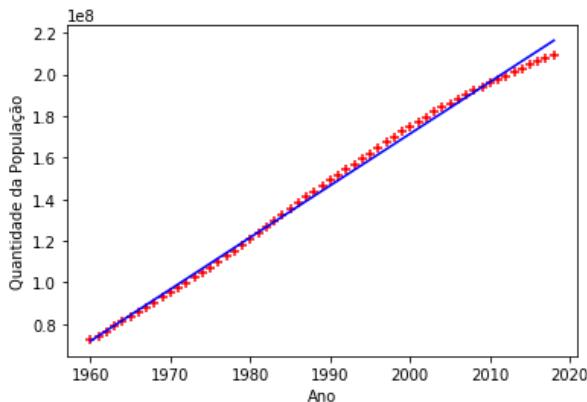


Figura 4.20: Dados da População Brasileira com a Previsão

Vamos praticar nossos novos "poderes de futurólogo", junto a essa base encontramos outra chamada ExpecVida.csv, com ela, tente prever qual será a Expectativa de Vida do brasileiro no ano de 2020.

4.7.2 Regressão Linear com mais de um Preditor

Vimos como usar o modelo de Regressão Linear, porém apenas a título de facilitação do entendimento, somente um atributo preditor. Mas o que acontece quando o alvo é influenciado por mais de um preditor? Vamos entender na prática como isso acontece.

Pensemos em um caso do Varejo, vamos utilizar um conjunto de observações chamado **marketSales.csv** que como o nome sugere, são compostos por transações de vendas. Sabemos que várias coisas influenciam

a saída de um determinado produto, tais como, o grau de visibilidade, peso, se possui muita ou pouca quantidade de gordura, tamanho do mercado ou outros.

Começamos com a importação da bibliotecas necessárias:

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt

%matplotlib inline
```

E ler nossa base de dados:

```
df = pd.read_csv('bases/marketSales.csv')
df.head()
```

Até o momento nada de novo, nosso problema começa ao repararmos nas observações:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year
0	FDA15	9.30	Low Fat	0.016047	Dairy	249.8092	OUT049	1999
1	DRC01	5.92	Regular	0.019278	Soft Drinks	48.2692	OUT018	2009
2	FDN15	17.50	Low Fat	0.016760	Meat	141.6180	OUT049	1999
3	FDX07	19.20	Regular	0.000000	Fruits and Vegetables	182.0950	OUT010	1998
4	NCD19	8.93	Low Fat	0.000000	Household	53.8614	OUT013	1987

Figura 4.21: Observações sobre Vendas de Produtos

Sabemos que os modelos de regressão só trabalham com tipos numéricos, muito pior existe o caso de nulos entre algumas outras inconsistências nessas 14.204 observações.

4.7.3 Regressão Linear e Limpeza dos Dados

Sejamos francos, maior parte de trabalho do Cientista de Dados é arrumar os dados que sofridamente conseguiu para realizar o trabalho, então começaremos a compreender como uma parte disso funciona. Primeiro detalhe vamos tratar os atributos indesejáveis, nulos e que não contribuem em absolutamente em nada para o aumento/diminuição das vendas. Atributos como o código identificador do produto (*Item_Identifier*) e do mercado (*Outlet_Identifier*) - por esse motivo que o Cientista de Dados deve entender do negócio.

Ao verificarmos a função *info()* descobrimos ainda que o atributo alvo (*Item_Outlet_Sales*) que indica a quantidade de produtos vendidos possui dados nulos (ou seja, também não servem para previsão).

```
df = df.drop(df[df['Item_Outlet_Sales'].isnull()].index)
df = df.drop(columns=['Item_Identifier', 'Outlet_Identifier'], axis=1)
```

Cuidado pois se aplicamos um corte seco como: `df.dropna(how='any', inplace=True)` obtemos somente 4.650 observações (devido a eliminação dos valores nulos contidos em outros atributos) - ou seja

perdemos quase 10.000 observações. Lembrar que o tratamento dos nulos deve ser cirúrgico e criterioso. Ao aplicar o corte corretamente somente do atributo alvo ficamos com 8.523 observações. Além disso removemos os preditores que não serviam.

Nosso próximo problema com nulos é nos atributos: peso do item (*Item_Weight*) e tamanho da loja (*Outlet_Size*). Em um caso de dados real devemos procurar preencher esses valores solicitando a informação necessária aos responsáveis, porém para fins desse trabalho iremos remover essa colunas também.

```
df = df.drop(columns=['Item_Weight', 'Outlet_Size'], axis=1)
```

Não obtemos mais a presença de nulos, mas ainda existem problemas, precisamos verificar os atributos não numéricos das observações, isto é: conteúdo de gordura (*Item_Fat_Content*), tipo do item (*Item_Type*), localização da loja (*Outlet_Location_Type*) e tipo da loja (*Outlet_Type*). Para isso:

```
print("Gordura:", df['Item_Fat_Content'].unique())
print("Tipo:", df['Item_Type'].unique())
print("Loc. Loja:", df['Outlet_Location_Type'].unique())
print("Tipo Loja:", df['Outlet_Type'].unique())
```

O atributo *Item_Fat_Content* possui uma faixa com os seguintes valores: '*LF*', '*Low Fat*', '*Regular*', '*low fat*' ou '*reg*'. Obviamente só existem dois tipos: '*Low Fat*' e '*Regular*' os outros três são variações desses valores. Para corrigir isso e realizar sua conversão:

```
df['Item_Fat_Content'] = df['Item_Fat_Content'].map({'LF': 1, 'Low Fat': 1, 'low fat': 1, 'reg': 2, 'Regular': 2})
df['Item_Fat_Content'] = df['Item_Fat_Content'].astype(pd.Int64Dtype())
df['Outlet_Location_Type'] = df['Outlet_Location_Type'].map({'Tier 1': 1, 'Tier 2': 2, 'Tier 3': 3})
df['Outlet_Location_Type'] = df['Outlet_Location_Type'].astype(pd.Int64Dtype())
df['Outlet_Type'] = df['Outlet_Type'].map({'Supermarket Type1': 1, 'Supermarket Type2': 2, 'Supermarket Type3': 3, 'Grocery Store': 4})
df['Outlet_Type'] = df['Outlet_Type'].astype(pd.Int64Dtype())
```

Criamos um dicionário com as faixas, repetimos os mesmos valores para os tipos que são semelhantes e realizamos a troca dos elementos no *DataFrame*. Aplicamos também a mesma prática para a localização e tipo da loja que possui poucos valores. Vamos agora com o caso de tipo do item que devemos trocá-lo para uma forma diferente (é ideal quando existem muitos valores diferentes).

Cada atributo tem um tipo determinado, por exemplo, *float* aceita números com pontos decimais, *int* numéricos inteiros, *string* caracteres, além disso *Python* trabalha com um tipo especial denominado *category*. Corresponde a uma determinada faixa de valores. Converter tipo do item em atributo categórico:

```
df['Item_Type'] = df.Item_Type.astype('category')
```

Uma vez realizado esse processo podemos "codificá-lo":

```
le_Item_Type = LabelEncoder()
df['Item_Type'] = le_Item_Type.fit_transform(df['Item_Type'])
df.head()
```

Para cada valor categorizado é atribuído um valor numérico (ou seja o mesmo trabalho que tivemos para o mapa). Usamos as funções `info()` e `describe()` e podemos partir para a próxima etapa sem quaisquer problemas com os dados, pois agora são todos numéricos e não possuem qualquer valor nulo.

4.7.4 Separação e treino

Separar em treino e teste (para avaliarmos nosso modelo) e remover o atributo alvo:

```
target = df['Item_Outlet_Sales']
df = df.drop(columns=['Item_Outlet_Sales'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(df, target, test_size = .2)
print('Amostra de Treino:', X_train.shape)
print('Amostra de Teste:', X_test.shape)
```

Usamos um valor de 20% para nosso teste e obtemos: 6.818 observações para treino e 1.705 de teste. Treinamos nosso modelo e verificamos seu resultado:

```
clf = LinearRegression()
clf.fit(X_train, y_train)
print('Acurácia: ', clf.score(X_test, y_test))
```

E obtemos uma acurácia aproximada de 42% e qual o motivo dessa discrepância tão grande? Simples, estamos cada vez mais perto da realidade e podemos verificar que realizar previsões com altos scores e poucas ações não existe. Pois se fosse assim: Corramos para treinar um modelo que nos dará os seis números da MegaSena. Ou ao menos nos dizer quando vai chover corretamente com muito pouco trabalho. Por fim podemos ver como os dados estão bem discrepantes em relação ao que foi predito e o real:

```
y_pred = model.predict(X_test)
plt.plot(y_test, y_test)
plt.scatter(y_test, y_pred, c = 'red', marker='+')
plt.ylabel('Real')
plt.xlabel('Predito')
plt.show()
```

Obtemos como resultado:

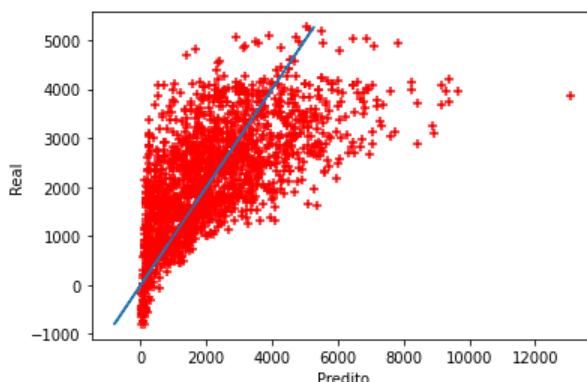


Figura 4.22: Regressão Linear aplicada a vários atributos

Em vermelho são a relação entre o valor real e o que foi predito, a linha azul mostra a Reta da Regressão. Verificamos a presença de um ponto bem isolado? Pode ser um *outlier*? Exatamente por esse motivo que passamos um bom tempo em EDA.

4.8 Árvore de Decisão

Se existe uma unanimidade entre os modelos que todo o Cientista de Dados conhece *Decision Tree* seria provavelmente o grande campeão (ou estaria entre os primeiros colocados), é um modelo com base em perguntas e respostas. Vamos começar com a importação das nossas bibliotecas básicas:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn import tree
```

Além da Pandas e NumPy, que serão básicas para todos os modelos que vemos, necessitamos também da *LabelEncoder* que responde as manipulações no DataFrame e do objeto *tree* por conter a classe para executar o algoritmo. Próximo passo é buscarmos os nossos dados:

```
df = pd.read_csv('bases/salaries.csv')
df.head()
```

Essa base de salários é composta por 3 empresas: Google, Facebook e ABC Pharma. Temos ainda Cargo, Grau de Ensino e Salário (Anual). E chegamos na parte essencial por adotar esse modelo: **O que desejamos responder?**, exatamente isso o modelo responde a uma pergunta que deve ser formulada de forma binária, ou seja que tal na empresa X, com um cargo Y e curso Z conseguimos ganhar um salário maior que 100 mil anuais?

Precisamos preparar nossa base de dados e adicionar uma resposta a nossa pergunta, assim criamos a variável **desejo**:

```
desejo = pd.Series(np.where(df['salary'] >= 100000, 1, 0))
print(desejo)
```

Observamos que desejo possui somente os valores 0 e 1 que indica falso ou verdadeiro para cada um dos itens da nossa tabela original. Por exemplo: trabalhar na Google, com um curso de Bacharel no cargo Executivo de Vendas não fará ganhar um salário de 100 mil anuais, mas no cargo Gerente de Negócios sim.

Para criarmos nossa Árvore de Decisão, precisamos que todos os campos (que participarão de sua composição) sejam variáveis numéricas categóricas, sua formação se dará da seguinte forma:

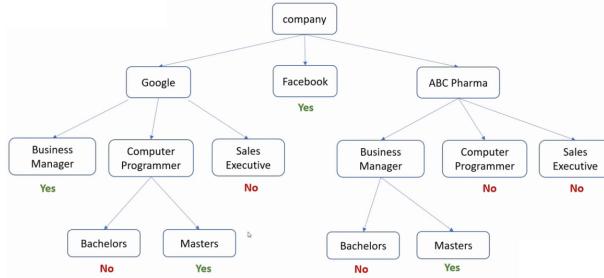


Figura 4.23: Estrutura da Árvore Montada

A função da biblioteca LabelEncoder é fazer exatamente isso, ou seja, transformar uma variável caractere, criar uma categoria e atribuir uma valor para ela, então:

```

le_company = LabelEncoder()
le_job = LabelEncoder()
le_degree = LabelEncoder()

df['company_n'] = le_company.fit_transform(df['company'])
df['job_n'] = le_company.fit_transform(df['job'])
df['degree_n'] = le_company.fit_transform(df['degree'])
df.head()
  
```

Criamos 3 novas séries e adicionamos ao nosso DataFrame original, vemos que foi atribuído um valor numérico para cada categoria, por exemplo Google é 2, Business Manager (Gerente de Negócios) é 0. Dessa forma que o algorítimo monta nossa árvore, a partir dessas três variáveis. Porém precisamos de um *DataFrame* composto somente por elas:

```

entradas = df.drop(['company', 'job', 'degree', 'salary'], axis='columns')
entradas.head()
  
```

De um modo mais simples, criamos um novo *DataFrame* (entradas) sem as variáveis descritivas (Empresa, Cargo e Grau de Ensino) e a numérica (Salário). Tudo está pronto, com esses dois novos objetos (**desejo** e **entradas**) podemos treinar nosso algorítimo:

```

model = tree.DecisionTreeClassifier()
model.fit(entradas, desejo)
  
```

E podemos realizar nossas perguntas, por exemplo, alguém da **Google** (2) que trabalha no cargo **Executivo de Vendas** (2) e fez **Mestrado** (1) recebe um salário maior que 100 mil anuais?

```

model.predict([[2, 2, 1]])
  
```

E obtemos como resposta o valor **0** indicando que não. Este é um dos principais modelos utilizados em Ciência de Dados tornou-se a porta de entrada para muitos outros. Podemos nos enganar com o pensamento que bastaria dar uma olhada nos dados para respondermos a pergunta, em uma base pequena realmente isso pode até ser viável, porém em uma base com muitos dados seria impraticável.

4.8.1 Critérios Gini e Entropia

Uma árvore de decisão se enquadra nos Algoritmos para Aprendizado de Máquina supervisionados e pode ser usada tanto para classificação quanto regressão - embora principalmente para o primeiro tipo. Como vimos este modelo obtém uma instância, atravessa uma árvore e compara recursos importantes com uma determinada declaração condicional. Se ele desce para o ramo filho esquerdo ou direito depende do resultado. Normalmente, os recursos mais importantes estão próximos a raiz.

Existem dois critérios que são comumente passados, são consideradas métricas padrão para calcular "impureza" ou "nível de informação". Orientam a divisão de um nó na árvore de decisão apenas com base nas informações existentes nesse nó. São calculados conforme as seguintes fórmulas:

- Gini: $1 - \sum_{j=1}^c (p_j)^2$
- Entropy: $-\sum_{j=1}^c p_j \times \log(p_j)$

Gini é a probabilidade de uma amostra aleatória ser classificada incorretamente se escolhermos aleatoriamente um *label* de acordo com a distribuição em um ramo. **Entropia** é uma medida de informação (ou melhor, a falta dela). Ao calcular o ganho de informação sobre uma divisão. Qual é a diferença em entropias. Isso mede como se reduz a incerteza sobre um *label*.

Vamos começar um outro projeto, porém veremos algo bem diferente, não iremos nos preocupar em explanar os dados que exploraremos. Vamos iniciar pelas bibliotecas:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
```

A grande diferença está aqui: criamos um conjunto de funções para realizar todo o trabalho de leitura, separação e execução do modelo. Recomendo esse método quando for criar seus projetos, pois simplifica muito a manutenção dos mesmos:

```
def obterDados(base):
    balance_data = pd.read_csv(base, sep = ',', header = None)
    print('Tamanho:', balance_data.shape)
    print(balance_data.head())
    return balance_data

def separarDados(dados):
    x = dados.values[:,1:5]
    y = dados.values[:,0]
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3,
        random_state = 100)
    return x, y, x_train, x_test, y_train, y_test

def treinar(x_train, y_train, criterio):
    clf = DecisionTreeClassifier(criterion = criterio, random_state = 100, max_depth =
        3, min_samples_leaf = 5)
    clf.fit(x_train,y_train)
```

```

return clf

def predizer(x_test, clf):
    y_pred = clf.predict(x_test)
    print(f"Valores Preditos: {y_pred}")
    return y_pred

def acuracia(y_test, y_pred):
    print("Acurácia:", accuracy_score(y_test, y_pred, average='weighted',
        zero_division=1) * 100)
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))

```

O *obterDados()* é responsável por ler e devolver as informações da nossa base de dados, *separarDados()* por dividir corretamente nossa base em treino e teste, *treinar()* realiza o treinamento da base conforme determinado critério, *predizer()* obtém as previsões com base na massa de teste em conjunto com o classificador e *acuracia()* verifica os resultados obtidos.

Com algumas mudanças podemos adaptar esses métodos e começar a pensar em criar uma biblioteca de modo a facilitar nossos projetos futuros. Veja como é mais simples com seu uso.

Para ler os dados:

```
data = obterDados('bases/balance-scale.data')
```

Para separar os dados:

```
x, y, x_train, x_test, y_train, y_test = separarDados(data)
```

Treinar com o critério **Gini**:

```
clf = treinar(x_train, y_train, 'gini')
y_pred_gini=prediction(x_test, clf)
```

Obter o resultado:

```
cal_accuracy(y_test, y_pred_gini)
```

E obtemos como resultado:

```

Acurácia: [100.          72.82608696  73.95833333]
[[ 0   6  7]
 [ 0 67 18]
 [ 0 19 71]]
      precision    recall  f1-score   support
      B       0.00     0.00     0.00      13
      L       0.73     0.79     0.76      85
      R       0.74     0.79     0.76      90
      accuracy                           0.73      188
      macro avg       0.49     0.53     0.51      188
      weighted avg      0.68     0.73     0.71      188

```

Figura 4.24: Resultado para o Critério Gini

Treinar com o critério **Entropia**:

```
clf = treinar(x_train, y_train, 'entropy')
y_pred_entr = prediction(x_test, clf)
```

Obter o resultado:

```
acuracia(y_test, y_pred_entr)
```

E obtemos como resultado:

		Acurácia: [100. 70.78651685 70.70707071]			
		[[0 6 7] [0 63 22] [0 20 70]]			
		precision	recall	f1-score	support
B		0.00	0.00	0.00	13
L		0.71	0.74	0.72	85
R		0.71	0.78	0.74	90
accuracy				0.71	188
macro avg		0.47	0.51	0.49	188
weighted avg		0.66	0.71	0.68	188

Figura 4.25: Resultado para o Critério Entropia

E observamos que neste caso, Gini possui uma melhor resposta aos dados.

4.9 Floresta Aleatória

Random Forest (por caridade *Florest* não existe na língua inglesa) é um modelo supervisionado que pode ser considerado como um aprimoramento da Árvore de Decisão, vamos recapitular seu funcionamento.

Através de um conjunto definido de regras, um novo valor passará por uma série de perguntas, os chamados ramos da árvore, para prevermos em qual melhor resposta se encaixa. Agora vamos pensar que podemos separar os dados em várias partes definidas aleatoriamente e usarmos uma árvore para cada uma dessas partes e para tomarmos uma decisão apenas computamos qual foi a resposta mais votada?

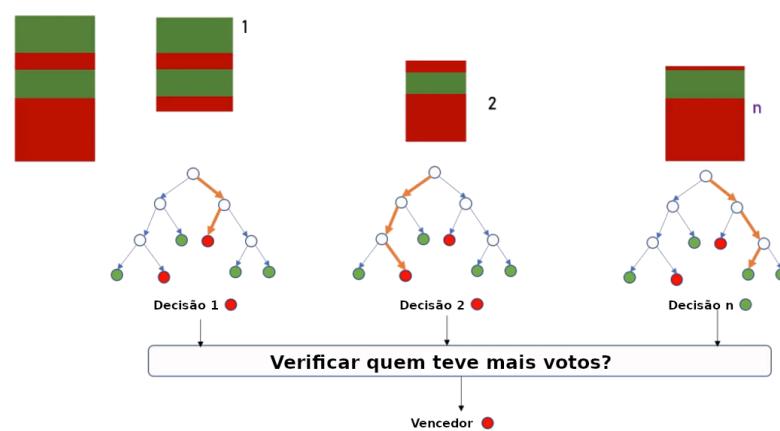


Figura 4.26: Funcionamento da Floresta Aleatória

E esse é o conceito da **Floresta Aleatória**, para aprendermos como é esse modelo na prática. Começamos com a importação das bibliotecas:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_digits
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sn

plt.figure(figsize=(10,7))
%matplotlib inline
```

A única aquisição nova é o método *RandomForestClassifier* que vem da ensemble contida na biblioteca *Scikit-learn*. Como fonte de dados usamos os dados de *load_digits* (isso mesmo, finalmente vamos trabalhar com imagens). Lemos nossos dados:

```
digits = load_digits()
```

E verificamos um determinado dígito, por exemplo o 8:

```
plt.matshow(digits.images[8])
plt.show()
```

E obtemos como resultado:

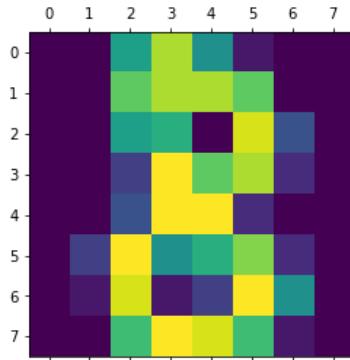


Figura 4.27: Número 8 da load_digits

Uma imagem é formada por vários pontos que denominamos de pixels, cada um desses possuem valor em uma escala de cores, no caso dessa base obtemos 64 colunas identificando cada um desses pontos. Vamos criar um DataFrame com esses dados:

```
df = pd.DataFrame(digits.data)
df.head()
```

E obtemos como resultado:

	0	1	2	3	4	5	6	7	8	9	...	54	55	56	57	58	59	60	61	62	63
0	0.0	0.0	5.0	13.0	9.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	6.0	13.0	10.0	0.0	0.0	0.0
1	0.0	0.0	0.0	12.0	13.0	5.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	11.0	16.0	10.0	0.0	0.0
2	0.0	0.0	0.0	4.0	15.0	12.0	0.0	0.0	0.0	0.0	...	5.0	0.0	0.0	0.0	0.0	3.0	11.0	16.0	9.0	0.0
3	0.0	0.0	7.0	15.0	13.0	1.0	0.0	0.0	0.0	8.0	...	9.0	0.0	0.0	0.0	7.0	13.0	13.0	9.0	0.0	0.0
4	0.0	0.0	0.0	1.0	11.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	2.0	16.0	4.0	0.0	0.0

Figura 4.28: Dados das Imagens

O problema é que não existe como coluna o resultado e precisamos colocá-lo, esse valor está no elemento target (não pense que existem apenas 9 imagens, são 1.997):

```
df['valor'] = digits.target
```

Separamos nossos dados na massa de treino e teste (20% do total):

```
X_train, X_test, y_train, y_test = train_test_split(df.drop(['valor']),
    axis='columns'), df['valor'], test_size = .2)
print(len(X_train), len(X_test))
```

E treinamos nosso modelo:

```
clf = RandomForestClassifier()
clf.fit(X_train, y_train)
```

Observamos no resultado o parâmetro "n_estimators=100", isso significa que são 100 árvores na nossa floresta, podemos aumentar ou diminuir (sem pensar em desmatamento) o número dessas árvores e observar se conseguimos aumentar nosso resultado. Verificamos a acurácia:

```
clf.score(X_test, y_test)
```

E obtemos significativos 98%, ou seja, nosso modelo erra apenas 2% das vezes, o que pode ser visto na Matriz de Confusão :

```
y_predicted = clf.predict(X_test)
cm = confusion_matrix(y_test, y_predicted)
sn.heatmap(cm, annot=True, cmap=plt.cm.Blues)
plt.xlabel('Predito')
plt.ylabel('Verdadeiro')
plt.show()
```

Obtemos como resultado:

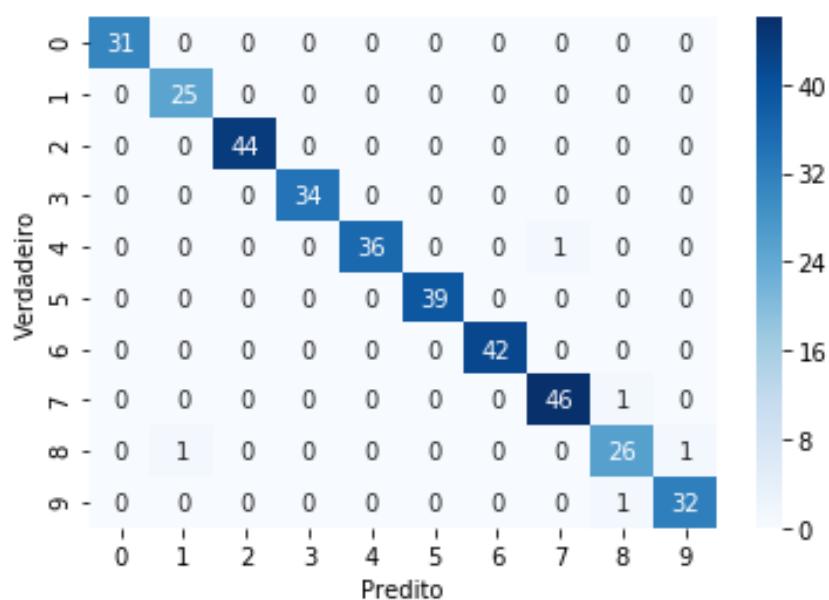
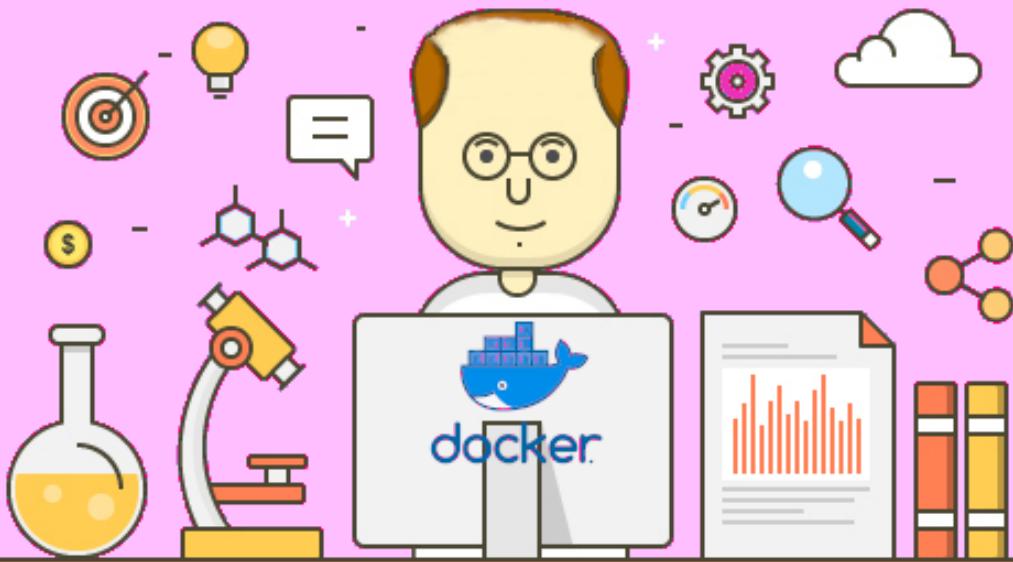


Figura 4.29: Matriz de Confusão no Seaborn



5. Modelos Preditivos

F O que sabemos é uma gota; o que ignoramos é um oceano. (Isaac Newton - Astrônomo)

5.1 Naïve Bayes

As pessoas pensam que o Cientista de Dados possui Contatos do Além e podem prever números da MegaSena ou um Desastre de Avião quando se fala em "Predição", sinto não somos charlatões, podemos pegar todos resultados da MegaSena e demonstrar com base em modelos quais os números que mais ou menos saíram, ou analisar os dados físicos de um avião, cruzar isso com todos os voos que já se acidentaram e tentar traçar uma característica. Em ambos os casos não espere certeza, existe o que chamamos: "Probabilidade de Acontecer".

Naïve Bayes é o tipo de algorítimo que dá medo em muito Cientista novato, talvez seja porque aparenta ser o mais matemático possível, sua descrição precisa seria "classificador probabilístico baseado no Teorema de Bayes, o qual foi criado por Thomas Bayes (1701 - 1761)". Para não assustar ninguém, devemos saber que trata-se de um Algorítimo Supervisionado destinado a probabilidades. Resumidamente: Qual a probabilidade de um evento A ocorrer dado que um evento B ocorreu? Qual a probabilidade de um e-mail ser spam dado que chegou um e-mail? Ativamos nosso JupyterLab personalizado que criamos com o Docker e na primeira célula importamos as bibliotecas necessárias:

```
import numpy as np
import pandas as pd

from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import BernoulliNB
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

Além das bibliotecas tradicionais como Pandas e NumPy para manipulação dos dados, temos todo o pacote da Naive_Bayes para os três tipos do modelo:

- Multinomial - realizar predições quando as variáveis (categóricas ou contínuas) descrevem contas discretas de frequência.
- Bernoulli - realizar predições para variáveis binárias.

- Gaussiano - realizar previsões para campos de distribuição normal.

Por fim e não menos importante a biblioteca para separação das amostras de treino e testes além da biblioteca para medir a acurácia dos nossos modelos e ver a que melhor atende. Agora vamos baixar o arquivo **spambase.data** acessar nossos dados:

```
dataset = np.loadtxt('bases/spambase.data', delimiter=',')
print(dataset[0])
```

Nessa base em cada observação existem 48 variáveis explanatórias com valores numéricos classificados e a 49ª é nossa variável resposta que indica se esse e-mail é ou não um Spam. Assim precisamos separar nossa massa de treino e teste:

```
X = dataset[:, 0:48]
y = dataset[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = .33,
    random_state = 17)
```

Nossa amostra para teste será formada por 33% do total, o que nos permite avaliar muito bem a performance de cada um dos tipos do modelo.

5.1.1 Tipo Multinomial

Basicamente a codificação não difere muito entre usar um tipo ou outro:

```
MultiNB = MultinomialNB() # 1
MultiNB.fit(X_train, y_train) # 2
print(MultiNB)
y_expect = y_test # 3
y_pred = MultiNB.predict(X_test) # 4
print(accuracy_score(y_expect, y_pred))
```

Quatro passos básicos para cada um deles, (1) criamos um objeto do tipo selecionado, (2) executamos nossa massa de treino, (3) guardamos nosso resultado de teste, realizamos a previsão com o uso da massa de teste (4) como resultado teremos a acurácia do modelo em acertar. Que neste caso gerou um valor de 0.8736010533245556, que significa em 87% das vezes o modelo foi eficaz.

5.1.2 Tipo Bernoulli

Nosso próximo teste será feito para o tipo Bernoulli:

```
BernNB = BernoulliNB(binarize = 0.0)
BernNB.fit(X_train, y_train)
print(BernNB)
y_expect = y_test
y_pred = BernNB.predict(X_test)
print(accuracy_score(y_expect, y_pred))
```

Comparando os códigos veremos basicamente as mesmas coisas com a diferença do tipo selecionado e

um parâmetro chamado "Binarização", esse é um tipo de modelo que "ama" variáveis binárias quanto mais tiver nos dados melhor, é um parâmetro de valor numérico flutuante (por padrão o valor é 0,0) e como resultado temos um valor de 0.8130348913759052, ou seja, 81% e se paramos para pensar resultado pior que o tipo anterior, pois bem façamos uma mudança e troquemos o parâmetro de entrada:

```
BernNB = BernoulliNB(binarize = 0.1)
```

E com essa mudança nosso resultado muda para o valor 0.8953258722843976. Não pense erradamente que agora virou festa e quanto mais aumentarmos o valor do parâmetro melhor será nossa acurácia. Realize testes e constate que pode obter resultados piores. Então tudo é uma questão de ajustarmos os parafusos (e esse é o real trabalho do Cientista de Dados em relação aos modelos).

5.1.3 Tipo Gaussiano

E novamente voltamos ao básico:

```
GausNB = GaussianNB()
GausNB.fit(X_train, y_train)
print(GausNB)
y_expect = y_test
y_pred = GausNB.predict(X_test)
print(accuracy_score(y_expect, y_pred))
```

Mas como resultado temos uma acurácia baixa de 0.8130348913759052, se for comparada obviamente aos outros resultados.

5.1.4 Conclusão

Então constatamos que para essa massa de dados o Naïve Bayes tipo Bernoulli com o parâmetro definido para o valor 0,1 possui uma melhor acurácia de 89%. Agora é sentar e realizar testes de como outra massa de dados se comporta.

5.2 Apriori

Quando vamos no supermercado, normalmente encontramos o sal grosso perto da carne de churrasco (ou do carvão), ou quando uma loja começa a indicar ofertas de coisas que realmente precisamos? Isso não é sorte (nem Magia Negra) é chamado de ARM ou "Mineração por Regras de Associação" e esse próximo modelo faz exatamente isso.

O ARM (Associate Rule Mining) é uma das técnicas importantes em ciência de dados. No ARM, a frequência de padrões e associações no conjunto de dados é identificada entre os conjuntos de itens usados para prever o próximo item relevante no conjunto. Principalmente nas decisões em negócios de acordo com os gostos dos clientes. Porém precisamos instalar sua biblioteca com o comando:

```
!pip install apyori
```

Apriori é um algoritmo de aprendizagem não supervisionado que é utilizado na modalidade ARM. Sua

função é procurar por uma série de conjuntos frequentes em itens de dados e se baseia em associações e correlações entre esses conjuntos de itens. É o algoritmo por trás do "Você também pode gostar de..." que costumamos encontrar nas plataformas de recomendação.

Tudo pronto e podemos importar as bibliotecas necessárias:

```
import pandas as pd
from apyori import apriori
```

Diferente dos alguns outros modelos, esse algoritmo não se encontra sobre a Scikit-Learn mas sobre a biblioteca apyori. E estamos usando a Pandas somente para facilitar o trabalho de ler um arquivo CSV e depois retornar os dados de forma fácil. Baixamos a base de dados **store_data.csv** e a lemos com o comando:

```
store_data = pd.read_csv('bases/store_data.csv', header=None)
store_data.head()
```

O detalhe aqui é que usaremos o cabeçalho como uma linha de dados válido, pois o modelo usa para criar as associações. A biblioteca Apriori exige que nosso conjunto de dados esteja na forma de uma lista de listas, onde todo o conjunto de dados é uma grande lista e cada transação no conjunto de dados é uma lista interna da grande lista externa.

```
records = []
for i in range(0, 7501):
    records.append([str(store_data.values[i,j]) for j in range(0, 20)])
```

O primeiro valor para o laço de repetição (7501) vem do número de linhas que contém nosso modelo enquanto que o seguindo valor do laço (20) é o número de colunas.

Assim temos os dados prontos para trabalharmos. Um detalhe importante, este modelo se utiliza de três variáveis:

- Suporte (support): O suporte do item I é definido como a razão entre o número de transações que contêm o item I pelo número total de transações.
- Confiança (confidence): Isso é medido pela proporção de transações com o item I1, nas quais o item I2 também aparece. A confiança entre dois itens I1 e I2, em uma transação, é definida como o número total de transações contendo os itens I1 e I2 dividido pelo número total de transações contendo I1.
- Lift: uma tradução para a palavra seria "Aumento", porém esta é a razão entre a confiança e o suporte.

E assim criamos nossas listas de regras com base nos valores das 3 variáveis:

```
regras = list(apriori(records, min_support=0.0045, min_confidence=0.2, min_lift=3,
                      min_length=2))
```

Temos como resultado 48 regras, apenas para facilitar a leitura vamos usar a seguinte codificação para visualizar os resultados:

```
for item in regras:
```

```

items = [x for x in item[0]]
print("Regra: " + items[0] + " -> " + items[1])
print("Suporte: " + str(item[1]))
print("Confiança: " + str(item[2][0][2]))
print("Lift: " + str(item[2][0][3]))
print("====")

```

Para entendermos como funciona, vamos ver a primeira regra:

```

Regra: light cream -> chicken
Suporte: 0.004532728969470737
Confiança: 0.29059829059829057
Lift: 4.84395061728395

```

Figura 5.1: Primeira regra obtida

Entre o "Frango" e o "Creme Light". O valor de suporte é 0,0045 (este número é calculado dividindo o número de transações que contêm o primeiro produto dividido pelo número total de transações). A confiança de 0,2905 significa que de todas as transações que contêm "Creme Light", 29,05% também foi comprado "Frango". *Lift* mostra que o "Creme Light" tem 4,84 vezes mais chances de ser comprado pelos clientes que compram "Frango", em comparação com a probabilidade de venda do "Creme Light".

Para fixarmos vejamos a próxima regra:

```

Regra: mushroom cream sauce -> escalope
Suporte: 0.005732568990801226
Confiança: 0.3006993006993007
Lift: 3.790832696715049

```

Figura 5.2: Segunda regra obtida

Entre o "Molho de Creme de Cogumelos" e o "Escalope". O suporte é de 0,0057. A confiança de 0,3006 significa que de todas as transações que contêm o "Molho de Creme de Cogumelos" 30,06% também contêm "Escalope". *Lift* mostra que o "Escalope" tem 3,79 mais chances de ser comprado pelos clientes que compram o "Molho de Creme de Cogumelos", em comparação com a probabilidade de venda do "Escalope".

Pronto, agora já podemos ler todas as outras 46 regras criadas entre os produtos e aumentar as vendas do nosso negócio.

5.3 SVM

Support Vector Machine é um algoritmo supervisionado capaz de classificar, regredir e também encontrar outliers. Para entendê-lo vamos considerar a seguinte figura:

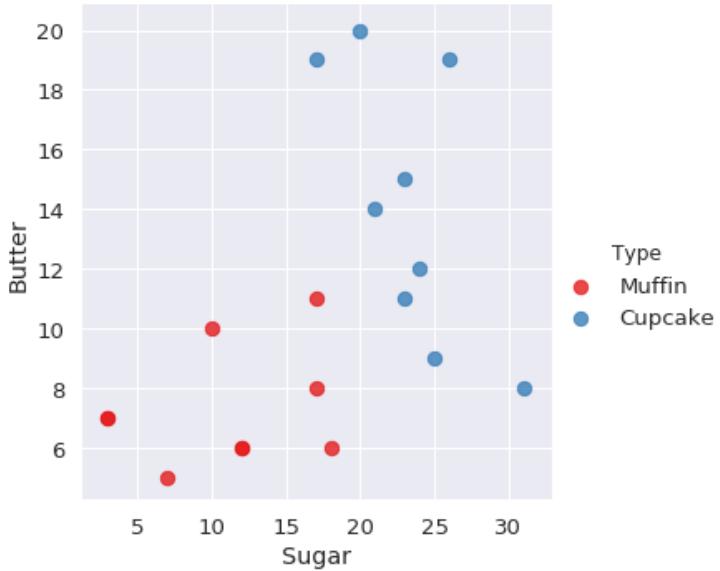


Figura 5.3: Comparação Muffin e Cupcake

Devemos nos perguntar aonde poderíamos colocarmos uma reta que separaria os dois grupos da melhor forma possível? Compreendendo isso matamos esse algoritmo. A nossa base será formada por percentuais com várias receitas de Muffin e Cupcake.

A função que daremos ao algoritmo será de identificar se trata de um ou outro. Vamos começar com a importação das bibliotecas necessárias:

```
import pandas as pd
import numpy as np
from sklearn import svm
from sklearn import datasets
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns; sns.set(font_scale=1.2)

%matplotlib inline
```

O algoritmo da SVM está na Scikit-Learn, porém precisamos entender que existem 3 classes distintas para se trabalhar com Classificação: **SVC**, **NuSVC** e **LinearSVC**. SVC e NuSVC são métodos semelhantes, mas possuem alguns parâmetros com algumas pequenas diferenças bem como formulações matemáticas. Por outro lado, o LinearSVC é outra implementação do Modelo para o caso de um kernel linear. Trabalharemos aqui com a SVC ou *C-Support Vector Classification*.

Próximo trabalho é baixar a base **muffins_cupcakes.csv** ler:

```
recipes = pd.read_csv('bases/muffins_cupcakes.csv')
recipes.head()
```

Basicamente essa base contém as proporções dos seguintes ingredientes: Farinha (*Flour*), Leite (*Milk*), Açúcar (*Sugar*), Manteiga (*Butter*), Ovo (*Egg*), Fermento (*Baking Powder*), Essência de Baunilha (*Vanilla*)

e Sal (*Salt*). Vamos separar Açúcar e Manteiga para treinar nosso modelo:

```
ingrediente = recipes[['Sugar', 'Butter']].values
tipo = np.where(recipes['Type']=='Muffin', 0, 1)
```

E como é um modelo supervisionado, precisamos também indicar do que se trata a receita. Treinamos nosso modelo:

```
model = svm.SVC(kernel='linear', decision_function_shape=None)
model.fit(ingrediente, tipo)
```

Uma vez feito isso, vamos criar dois pontos para definir uma linha (apenas para mostrar) como nossos dados são separados:

```
w = model.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(5, 30)
yy = a * xx - (model.intercept_[0]) / w[1]
```

Agora basta colocarmos tudo em um gráfico:

```
sns.lmplot('Sugar', 'Butter', data=recipes, hue='Type', palette='Set1',
           fit_reg=False, scatter_kws={"s": 70});
plt.plot(xx, yy, linewidth=2, color='black');
```

E obtemos o seguinte resultado:

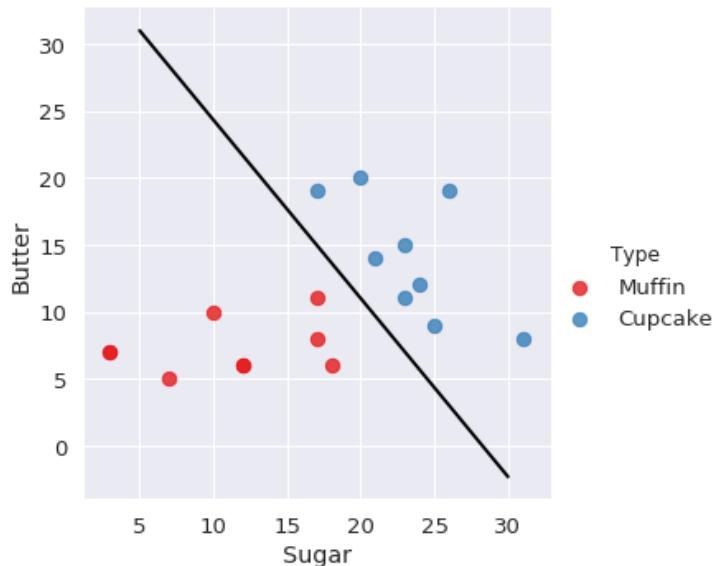


Figura 5.4: Comparação Muffin e Cupcake

Mas como essa linha é encontrada? Isso é feito com base em "margens". Criamos mais 2 linhas para entendermos o processo:

```
b = model.support_vectors_[0]
yy_down = a * xx + (b[1] - a * b[0])
b = model.support_vectors_[-1]
yy_up = a * xx + (b[1] - a * b[0])
```

E ao colocar novamente tudo em um gráfico:

```
sns.lmplot('Sugar', 'Butter', data=recipes, hue='Type', palette='Set1',
            fit_reg=False, scatter_kws={"s": 70})
plt.plot(xx, yy, linewidth=2, color='black')
plt.plot(xx, yy_down, 'k--')
plt.plot(xx, yy_up, 'k--')
plt.scatter(model.support_vectors_[:, 0], model.support_vectors_[:, 1], s=80,
            facecolors='none');
```

E obtemos o seguinte resultado:

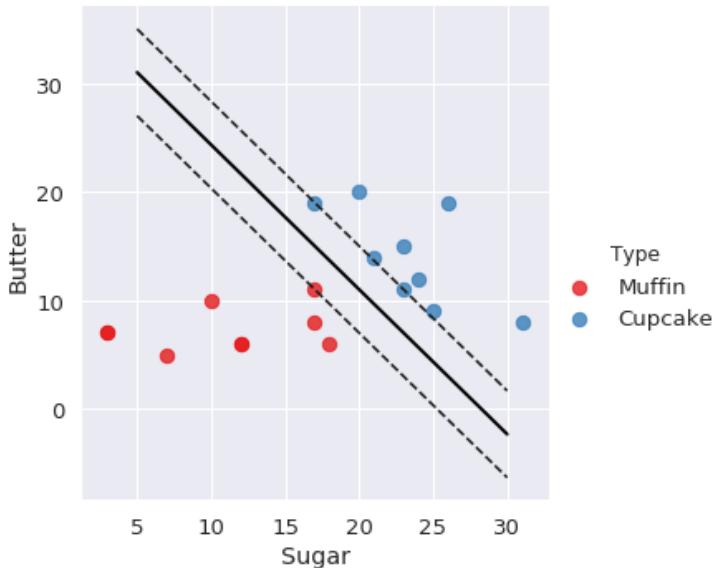


Figura 5.5: Comparação Muffin e Cupcake

E é exatamente a partir da melhor "margem" que a linha é criada. O mais interessante que esse é um algoritmo de previsão, então vamos criar um pequeno método:

```
def muffin_ou_cupcake(sugar, butter):
    if (model.predict([[sugar, butter]])) == 0:
        print('Isto é uma receita de Muffin')
    else:
        print('Isto é uma receita de Cupcake')
```

E podemos fazer algumas verificações:

```
muffin_ou_cupcake(20, 10)
```

E isso nos diz que provavelmente trata-se de um Muffin!

5.3.1 Retorno a Íris

Vamos retornar as nossas flores para entendermos mais algumas coisas sobre esse modelo:

```
iris = datasets.load_iris()
df = pd.DataFrame(data=np.c_[iris['data'], iris['target']],
                   columns=iris['feature_names'] + ['Species'])
df.head()
```

Separamos nossos dados em duas massas:

```
X = df.iloc[:, :4]
y = df.Species
```

Lembramos que as 4 primeiras informações são as variáveis explanatórias e a última define qual a espécie. Vamos então treinar nosso algoritmo:

```
clf = svm.SVC()
clf.fit(X, y)
```

E verificamos o resultado sobre a Matriz de Confusão:

```
array([[50,  0,  0],
       [ 0, 48,  2],
       [ 0,  2, 48]])
```

Figura 5.6: Resultado da Matriz de Confusão

Ou seja, 4 espécies foram classificadas erradas. Podemos melhorar esse resultado com a adição de alguns parâmetros:

- **C**: parâmetro de regularização
- **kernel**: que pode ter um dos seguintes valores linear, poly, rbf, sigmoid, precomputed.

Após alguns testes chegamos na seguinte configuração para treino do nosso modelo:

```
clf = svm.SVC(C=1, kernel='linear')
clf.fit(X, y)
```

E agora o resultado da Matriz de Confusão é:

```
array([[50,  0,  0],
       [ 0, 49,  1],
       [ 0,  0, 50]])
```

Figura 5.7: Resultado Final da Matriz de Confusão

E temos uma excelente resposta desse modelo.

5.4 Regressão Logística

Já vimos sobre a **Regressão Linear** e sobre sua capacidade "Preditiva" sobre valores contínuos, pensemos em Preços de Imóveis, Preço de Ações ou mesmo Previsão do Tempo. Porém em se tratando de variáveis categóricas, do tipo, isso é um email ou não? Essa pessoa pode ou não querer um seguro? Em qual partido votará? Precisamos ter uma outra forma de previsão. Esse é o objetivo da Regressão Logística, que é utilizada como uma das técnicas de classificação (o modelo anterior também era um classificador).

Os problemas de classificação se envolvem em dois tipos: Binários - o cliente quer ou não um seguro? e Multiclasse - em qual partido votará? Vamos começar com o tipo Binário e futuramente veremos o outro. Iniciamos com a importação das nossas bibliotecas:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from scipy.special import expit

%matplotlib inline
```

Se compararmos com o Modelo de Regressão Linear temos a mesma classe Linear Model da Scikit-Learn porém com o método LogisticRegression para realizar nosso trabalho. E o método expit que utilizaremos para traçar a "sigmóide". Agora baixamos a base **insurance_data.csv** e a lemos:

```
df = pd.read_csv('bases/insurance_data.csv')
df.head()
```

Os dados para o "Seguro de Vida" que utilizamos, contém duas variáveis: a idade e se possui (valor 1) ou não (valor 0) um seguro. Próximo passo é treinar nosso modelo com esses dados:

```
reg = LogisticRegression()
reg.fit(df[['age']], df.bought_insurance)
```

E realizamos nossas previsões:

```
reg.predict([[43], [25]])
```

Alguém com idade de 43 ou 25 anos desejará adquirir um seguro? E teremos como resposta: array([1, 0]), indicando sim para 43 anos e não para 25.

5.4.1 Função Sigmóide

Como a mágica acontece? Através de uma função chamada sigmoid, também chamada "função de achatamento". Matematicamente falando, essa função é assim: $f(x) = \frac{1}{1+e^{-x}}$. Podemos implementá-la da seguinte forma:

```
from scipy.optimize import curve_fit
import numpy as np

def funcao_sigmoid(x, x0, k):
    y = 1.0 / (1 + np.exp(-np.dot(k, x-x0)))
    return y

popt, pconv = curve_fit(funcao_sigmoid, df['age'], df.bought_insurance)
sigmoid = funcao_sigmoid(df['age'], *popt)
```

Porém a função expit (da biblioteca SciPy) já realiza toda essa implementação, sendo assim, tudo isso pode ser substituído por:

```
sigmoid = expit(df['age'] * reg.coef_[0][0] + reg.intercept_[0])
```

O que facilita bastante nossa codificação, plotamos os dados:

```
plt.plot(df['age'], sigmoid)
plt.scatter(df['age'], df.bought_insurance, c=df.bought_insurance, cmap='rainbow',
            edgecolors='b')
plt.show()
```

E o resultado final será esse:

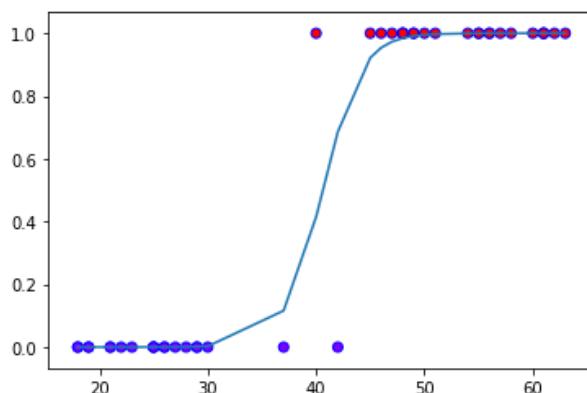


Figura 5.8: Curva da Sigmóide para nossos dados

5.4.2 Com mais Variáveis

A base de dados que usaremos é bem interessante, mostra os dados de empréstimos concedidos a estudantes, baixar o arquivo **emprestimo.csv**. Devemos prever se um potencial aquisitor do empréstimo será ou não um bom pagador.

Iniciamos com a importação das bibliotecas necessárias:

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from matplotlib import pyplot as plt

%matplotlib inline
```

Para em seguida ler os dados:

```
data = pd.read_csv('bases/emprestimo.csv')
data.show()
```

E temos os seguintes dados à nossa disposição com um total de 614 linhas:

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoaapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Property_Area	Loan_Status
0	LP001002	Male	No	0	Graduate	No	5849	0.0	NaN	360.0	1.0	Urban	Y
1	LP001003	Male	Yes	1	Graduate	No	4583	1508.0	128.0	360.0	1.0	Rural	N
2	LP001005	Male	Yes	0	Graduate	Yes	3000	0.0	66.0	360.0	1.0	Urban	Y
3	LP001006	Male	Yes	0	Not Graduate	No	2583	2358.0	120.0	360.0	1.0	Urban	Y
4	LP001008	Male	No	0	Graduate	No	6000	0.0	141.0	360.0	1.0	Urban	Y

Figura 5.9: Dados de Empréstimo

A variável explanatória chama-se "Loan_Status" o que indica se o estudante é ou não um bom pagador. Primeiro problema que temos é transformar a variável explanatória em 0 ou 1, pois o modelo que estamos esperando esse tipo de variável:

```
encode = LabelEncoder()
data.Loan_Status = encode.fit_transform(data.Loan_Status)
```

Na segunda parte desse tratamento eliminamos as variáveis que não contribuem para o resultado final (neste caso somente a Loan_ID) e qualquer linha que possua o valor nulo:

```
# Retirar o campo Loan_ID
data = data.drop(columns=['Loan_ID'], axis=1)
# Limpeza dos valores nulos
data.dropna(how='any', inplace=True)
```

E isso nos faz perder 22% dos dados e temos agora 480 linhas. Lembre-se sempre que o corte dos nulos deve ser realizado de forma cirúrgica e não com um machado¹. Nosso próximo passo é separar nossa amostra em treino e teste:

```
X_train, X_test, y_train, y_test =
train_test_split(data.drop(columns=['Loan_Status']), data['Loan_Status'],
```

¹o programa que utilizei para realizar esse tratamento é o *OpenRefine* mas como não é o tema aqui sugiro que veja uma apostila que publiquei sobre este software

```
test_size = .2)
```

Normalmente deixamos 20% para teste e verificação da acurácia do modelo esse número pode ser aumentado ou diminuído conforme seus dados, não existe uma regra definida. Modelos de regressão trabalham com variáveis numéricas, então vamos transformar nossas descritivas:

```
X_train = pd.get_dummies(X_train)
X_test = pd.get_dummies(X_test)
```

O método *get_dummies* transformará todas as variáveis em dados lógicos. Por exemplo para gênero temos duas: Gender_Female e Gender_Male. Assim é realizado sucessivamente para todas as outras. Executamos o modelo e verificamos a acurácia:

```
clf = LogisticRegression(max_iter=200)
clf.fit(X_train, y_train)
print('Acurácia:', clf.score(X_test, y_test))
```

E verificamos 75% de precisão, um outro meio de obtermos a acurácia seria pelo método *accuracy_score()*:

```
y_pred = clf.predict(X_test)
print('Acurácia:', accuracy_score(y_test, y_pred))
```

Como trabalhamos com um modelo de Regressão Logística não existe maneira de colocar todas as variáveis (até existe, mas vira uma bela confusão) em um único gráfico. Assim analisamos separadamente:

```
plt.rcParams['figure.figsize'] = (10,8)
df = pd.DataFrame({'ApplicantIncome': X_test['ApplicantIncome'], 'Correto': y_test,
                   'Predito': y_pred})
df = df.sort_values(by=['ApplicantIncome'])
plt.scatter(df['ApplicantIncome'], df['Correto'], color='red', marker='+')
plt.plot(df['ApplicantIncome'], df['Predito'], color='blue', linewidth=2)
plt.show()
```

E obtemos o seguinte gráfico:

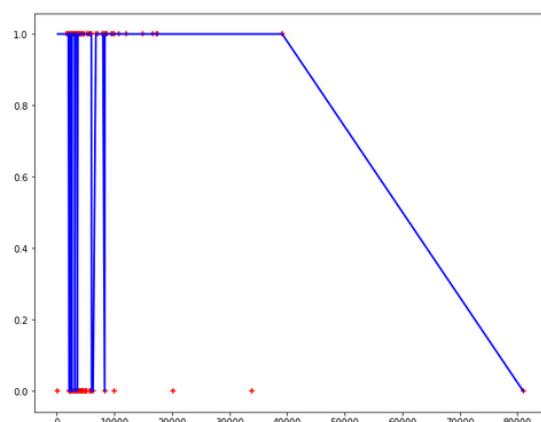


Figura 5.10: Gráfico sobre pagadores

E observamos que até o valor de \$ 10.000 existe uma "indecisão" sobre se o estudante é ou não um bom pagador. Acima disso torna-se um bom pagador (com 2 exceções). Ou seja, precisamos das outras variáveis para ver como o modelo realmente se comportam:

```
plt.rcParams['figure.figsize'] = (25,20)
for i, e in enumerate(X_test.columns):
    df = pd.DataFrame({e: X_test[e], 'Correto': y_test, 'Predito': y_pred})
    df = df.sort_values(by=[e])
    ax = plt.subplot(5, 4, i+1)
    ax.scatter(df[e], df['Correto'], color='red', marker='+')
    ax.plot(df[e], df['Predito'], color='blue', linewidth=2)
    ax.set_title(str(e))
plt.show()
```

Usamos um laço for para percorrer toda a lista de variáveis e obtemos o seguinte resultado:

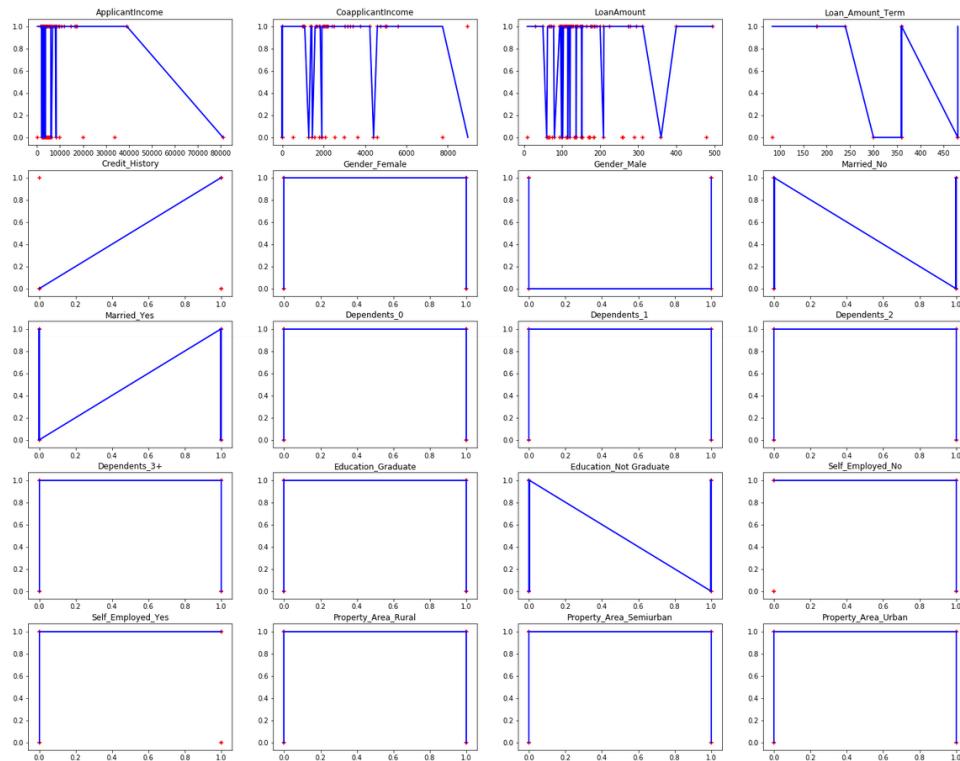
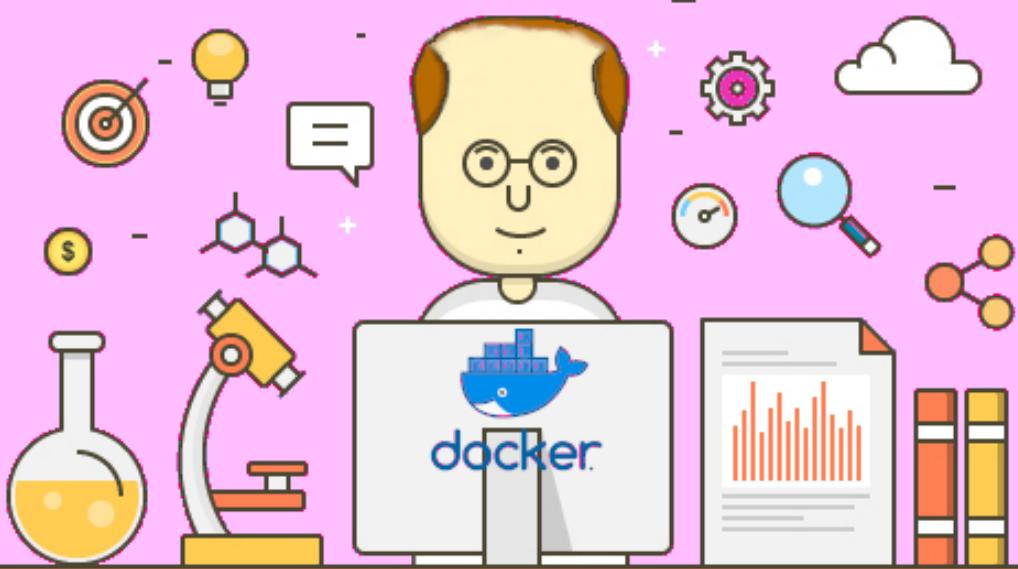


Figura 5.11: Gráficos aplicado a todas as variáveis

E podemos verificar como todas se comportam. Não espere encontrar sigmóides perfeitas em seus dados, isso é uma exceção e não um comportamento padrão.



6. Modelos Complementares

F Há verdadeiramente duas coisas diferentes: saber e crer que se sabe. A ciência consiste em saber; em crer que se sabe reside a ignorância. (Hipócrates - Filósofo Grego e Pai da Medicina)

6.1 DBScan

Todos os modelos vistos anteriormente são com certeza os mais utilizados, porém nem por isso são os únicos. Nesse último capítulo vamos investigar os modelos, digamos, não tão comuns mas que podem ser classificados como Complementares, não imagine que são largados ao acaso ou que podem ser esquecidos.

Esse é o mais contraditório dos modelos, calma que me explico. O nome *DBScan* parece bem curto, mas é apenas uma sigla para *Density-Based Spatial Clustering of Applications with Noise*. É um algoritmo não supervisionado destinado a clusterização que é excelente para identificar *outliers*.

Basicamente esse modelo trabalha com duas informações: EPS é a distância máxima entre 2 amostras para formar um cluster de mesmo tipo (neighborhood). minPts é o número mínimo de uma amostra na vizinhança para ser classificada como "*Core Point*" (um pouco acima que os minPts na EPS). E nesse ponto entra em cena um *noise* (traduzido literalmente para ruído ou barulho incômodo) e que para nós será tudo o que não for um *Core Point* ou *Border Point* (que são pontos um pouco abaixo que os minPts na EPS).

Esquecendo esse negócio de conceito, vejamos como isso funciona na prática, utilizamos a base Íris pois será bem mais fácil quando já conhecemos bem os dados. Ativamos nosso JupyterLab personalizado que criamos com o Docker e na primeira célula importamos as bibliotecas necessárias:

```
import pandas as pd
import numpy as np
from sklearn import datasets
from sklearn.cluster import DBSCAN
from matplotlib import pyplot as plt
from sklearn.datasets import make_moons

%matplotlib inline
```

As bibliotecas são Pandas e NumPy para organizar nossos dados, da Scikit-Learn vamos trazer os dados

pela datasets e o algorítimo pela DBSCAN e a MatPlotLib para mostrá-los de modo gráfico.

6.2 Explicação do noise

Antes de começarmos precisamos entender bem como ocorre um "distúrbio nos dados", para isso usaremos a base *make_moons* para produzir alguns dados:

```
X, label = make_moons(n_samples=200, noise=0.01)
X[0:5]
```

O mais importante aqui é observamos o parâmetro *noise*. Começamos com um valor bem baixo, ou seja, quase nenhum ruído será produzido nos dados. Identificamos uma variável *label* que corresponde a resposta dos dados, e é nela que devemos ficar atentos, verificamos seu valor:

```
print(label)
```

E observamos que basicamente temos uma série de 0s e 1s, isso indica que os dados contém 2 clusters. Ao treinarmos nosso modelo:

```
model = DBSCAN(eps=0.25, min_samples=10)
model.fit(X)
```

Descobriremos que ao visualizarmos os *labels*:

```
print(model.labels_)
```

O resultado da variável resposta é igual ao original, mas isso fica mais claro se visualizarmos de forma gráfica, então:

```
# Original
fig, ax = plt.subplots(figsize=(6,5))
ax.scatter(X[:,0], X[:,1], c=label)
fig.show()

# Predito
fig, ax = plt.subplots(figsize=(6,5))
ax.scatter(X[:,0], X[:,1], c=model.labels_)
fig.show()
```

E temos o seguinte resultado:

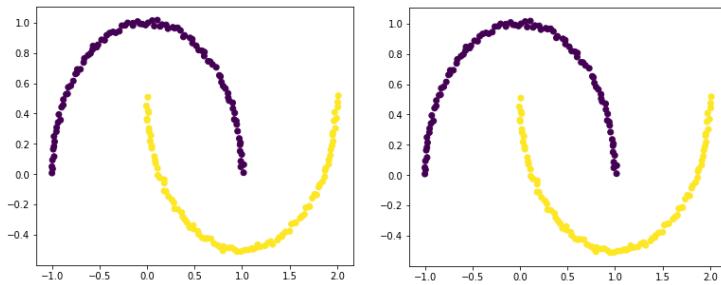


Figura 6.1: Visualização dos Resultados em Noise com 0,01

Observamos que são duas curvas iguais (não se preocupe pois as cores podem variar) o importante nesse desenho é repararmos dois detalhes: primeiro são 2 clusters de dados e os pontos estão bem próximos.

A medida que aumentamos o parâmetro noise, os pontos vão se afastando (façamos alguns teste e aumentando gradativamente). Ao chegarmos no valor 0,1 temos a seguinte situação:

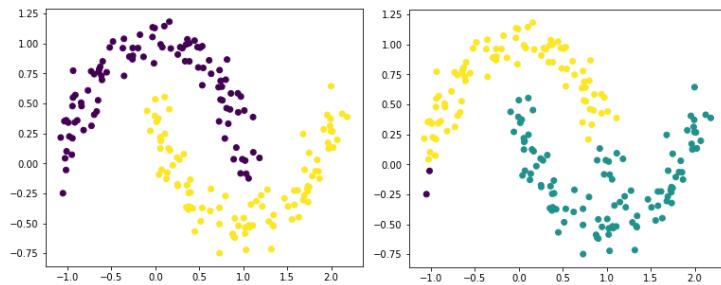


Figura 6.2: Visualização dos Resultados em Noise com 0,1

E é essa imagem que buscamos, pois ao invés de 2 clusters foi predito um 3, e esse terceiro é formado por dois pontos (no segundo gráfico em roxo). Ou seja, foi lançado tanto ruído que o modelo se atrapalhou. Além disso, ao olharmos a variável `model.labels_` veremos que existem valores -1.

6.3 Atrás dos outliers da Íris

Agora que já sabemos o que é um ruído podemos voltar para a base Íris. Inicialmente carregamos a base de dados:

```
iris = datasets.load_iris()
df = pd.DataFrame(data=np.c_[iris['data'], iris['target']],
                   columns=iris['feature_names'] + ['Species'])
df.head()
```

Agora precisamos treinar o modelo:

```
model = DBSCAN(eps=0.8, min_samples=19).fit(df)
print(model)
```

Porque 0,8 no EPS? Simples tentativa e erro, assim com o número mínimo de amostras em 19, não existe uma regra aplicável a qualquer base. Esse é um dos trabalhos em ser um Cientista de Dados. Apenas para facilitar, vamos pegar a fatia de dados que são suspeitas de outliers:

```
df[model.labels_==1]
```

E descobrimos que nessa situação temos 7 pontos suspeitos de serem outliers. Colocando-os em um gráfico:

```
fig = plt.figure()
ax = fig.add_axes([.1, .1, 1, 1])
colors = model.labels_
ax.scatter(df.iloc[:,2], df.iloc[:,1], c=colors, s=120)
ax.set_xlabel('Tam Pétala')
ax.set_ylabel('Lrg Sépala')
plt.title('DBSCAN para Detecção de Outliers')
plt.show()
```

Temos o seguinte resultado:

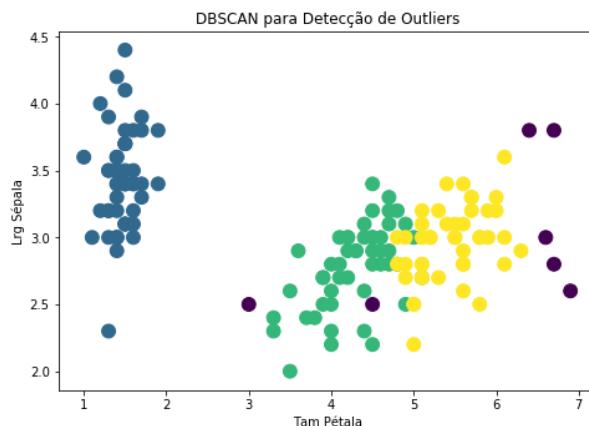
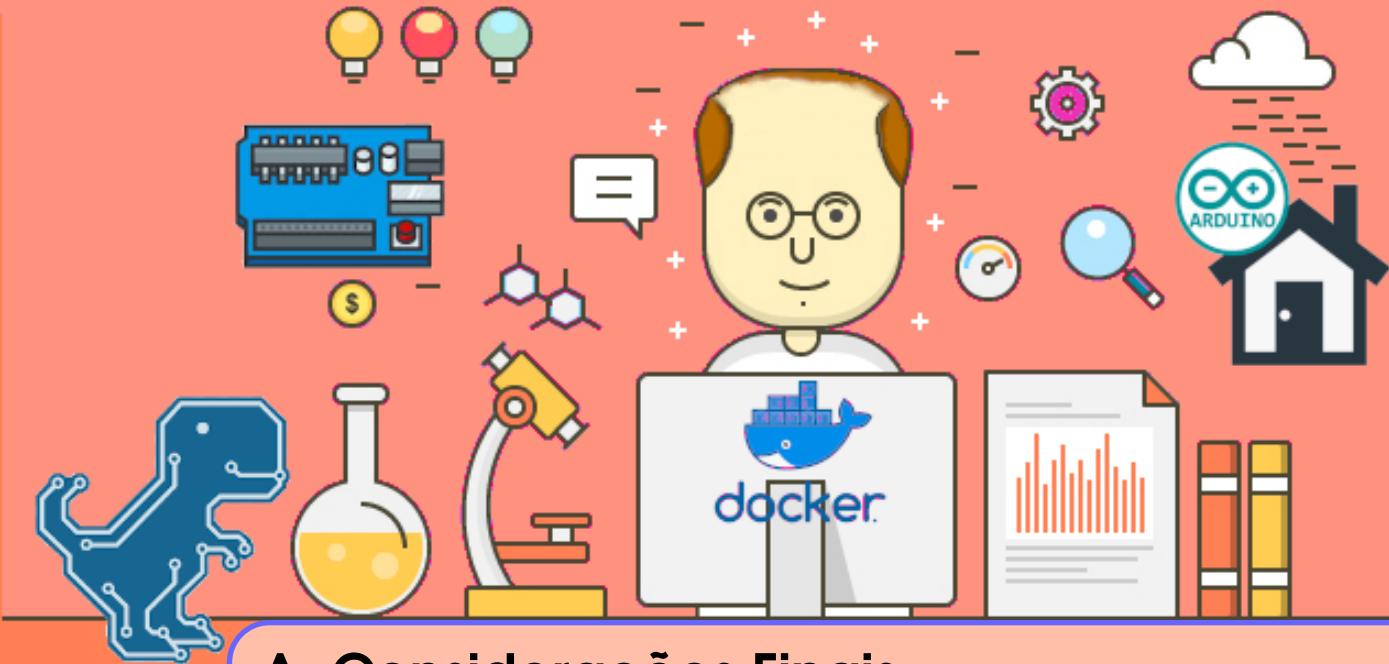


Figura 6.3: Visualização dos Outliers da Iris

Agora já temos a faca e o queijo na mão, devemos lembrar da nossa primeira forma de descobrir *outliers* (usando BoxPlots) não foi muito bem sucedida com o modelo KNN, talvez com alguns desses pontos conseguimos verificar se as espécies são realmente coerentes segundo nosso classificador.



A. Considerações Finais

F Você não pode ensinar nada a ninguém, mas pode ajudar a pessoas a descobrirem por si mesmas.
(Galileu Galilei - Físico)

Os artigos deste livro foram selecionados das diversas publicações que fiz no Linkedin e encontradas em outros sites que foram nesta obra explicitamente citadas. Acredito que apenas com a prática podemos almejar o cargo de Cientista de Dados, então segue uma relação de boas bases que podemos encontrar na Internet:

- 20BN-SS-V2: <https://20bn.com/datasets/something-something>
- Actualitix: <https://pt.actualitix.com/>
- Banco Central do Brasil: <https://www3.bcb.gov.br>
- Banco Mundial: <http://data.worldbank.org>
- Censo dos EUA (População americana e mundial): <http://www.census.gov>
- Cidades Americanas: <http://datasf.org>
- Cidade de Chicago: <https://data.cityofchicago.org/>
- CIFAR-10: <https://www.cs.toronto.edu/~kriz/cifar.html>
- Cityscapes: <https://www.cityscapes-dataset.com/>
- Criptomoedas: <https://pro.coinmarketcap.com/migrate/>
- Dados da União Europeia: <http://open-data.europa.eu/en/data>
- Data 360: <http://www.data360.org>
- Datahub: <http://datahub.io/dataset>
- DBpedia: <http://wiki.dbpedia.org/>
- Diversas áreas de negócio e finanças: <https://www.quandl.com>
- Diversos assuntos: <http://www.firebaseio.com>
- Diversos países (incluindo o Brasil): <http://knoema.com>
- Fashion-MNIST: <https://www.kaggle.com/zalando-research/fashionmnist>
- Gapminder: <http://www.gapminder.org/data>
- Google Finance: <https://www.google.com/finance>
- Google Trends: <https://www.google.com/trends>
- Governo do Brasil: <http://dados.gov.br>
- Governo do Canadá (em inglês e francês): <http://open.canada.ca>
- Governo dos EUA: <http://data.gov>
- Governo do Reino Unido: <https://data.gov.uk>
- ImageNET: <http://www.image-net.org/>

- IPEA: <http://www.ipeadata.gov.br>
- IMDB-Wiki: <https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki/>
- Kinetics-700: <https://deepmind.com/research/open-source/kinetics>
- Machine Learning Databases: <https://archive.ics.uci.edu/ml/machine-learning-databases/>
- MEC Microdados INEP: <http://inep.gov.br/microdados>
- MS coco: <http://cocodataset.org/#home>
- MPII Human Pose: <http://human-pose.mpi-inf.mpg.de/>
- Músicas: <https://aws.amazon.com/datasets/million-song-dataset>
- NASA: <https://data.nasa.gov>
- Open Data Monitor: <http://opendatamonitor.eu>
- Open Data Network: <http://www.opendatanetwork.com>
- Open Images: <https://github.com/openimages/dataset>
- Portal de Estatística: <http://www.statista.com>
- Públicos da Amazon: <http://aws.amazon.com/datasets>
- R-Devel: <https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html>
- Reconhecimento de Faces: <http://www.face-rec.org/databases>
- Saúde: <http://www.healthdata.gov>
- Statsci: <http://www.statsci.org/datasets.html>
- Stats4stem: <http://www.stats4stem.org/data-sets.html>
- Stanford Large Network Dataset Collection: <http://snap.stanford.edu/data>
- Vincent Rdatasets: <https://vincentarelbundock.github.io/Rdatasets/datasets.html>
- Vitivinicultura Embrapa: <http://vitibrasil.cnpuv.embrapa.br/>

Esse não é o fim de uma jornada acredito ser apenas seu começo. Espero que este livro possa lhe servir para criar algo maravilhoso e fantástico que de onde estiver estarei torcendo por você.

A.1 Sobre o Autor

Fortes conhecimentos em linguagens de programação Java e Python. Especialista formado em Gestão da Tecnologia da Informação com forte experiência em Bancos Relacionais e não Relacionais. Possui habilidades analíticas necessárias para encontrar a providencial agulha no palheiro dos dados recolhidos pela empresa. Responsável pelo desenvolvimento de dashboards com a capacidade para análise de dados e detectar tendências, autor de 15 livros e diversos artigos em revistas especializadas, palestrante em seminários sobre tecnologia. Focado em aprender e trazer mudanças para a organização com conhecimento profundo do negócio.

- Perfil no Linkedin: <http://www.linkedin.com/pub/fernando-anselmo/23/236/bb4>
- Endereço do Git: <https://github.com/fernandoans/machinelearning>

Machine Learning na Prática

ESTE LIVRO PODE E DEVE SER DISTRIBUÍDO LIVREMENTE

Fernando Anselmo