



Técnicas de Machine Learning

UNIDADE 03

Tomando outro respiro

| TOMANDO OUTRO RESPIRO

É tudo deep learning e GPT, né?



Na última unidade, conversamos sobre a **aprendizagem não supervisionada** e sobre as bibliotecas em python. Nesse sentido, trago mais uma vez alguns tópicos que já comentamos extensivamente anteriormente, mas que vale lembrar de volta:

- Existem várias bibliotecas em python que adicionam novas funcionalidades ao python como, a manipulação de imagens e sons, a criação de gráficos e a criação de algoritmos de *machine learning* (ML).
 1. A instalação e atualização de bibliotecas em python ocorre primariamente pelo comando **pip**.
 2. Podemos ter diferentes conjuntos de bibliotecas instaladas dentro de um computador, cada uma delas contendo versões ou combinações de bibliotecas diferentes. Isso acontece porque ocasionalmente uma biblioteca pode ter certas incompatibilidades com outra ou, ainda, porque um determinado código nosso só funciona em uma versão específica de uma biblioteca. O nome destes conjuntos é *environment*.
- Em ML, algumas das principais bibliotecas são o **pandas**, o **scikit-learn**, o **NumPy**, o **SciPy**, o **Matplotlib** e o **seaborn**.
 1. Como cada uma das bibliotecas possui as suas particularidades, é imprescindível que você saiba navegar pela documentação de cada uma delas e entenda como adaptá-las para as suas necessidades.
- A aprendizagem não supervisionada pode ser dividida em algumas técnicas. As principais são:
 1. Seleção de atributos.
 2. Extração de atributos.
 3. Agrupamento (clusterização).
- Para funcionar corretamente, técnicas de normalização de dados podem ser úteis para que os algoritmos de ML aprendam também de uma maneira adequada.
- Técnicas de aprendizagem não supervisionada servem como um passo de **preparação dos dados** antes do treinamento de um algoritmo de ML em si.

1. Técnicas de aprendizagem não supervisionada **também** são ML – por outro lado, para o público em geral, ML é praticamente um sinônimo de algoritmos preditivos os quais, por sua vez, são representados pelas técnicas de aprendizagem supervisionada.

Isso posto, podemos prosseguir com o conteúdo desta unidade. Quando pegamos um novo *dataset*, primeiramente **preparamos** os dados com uma combinação de algumas técnicas que vimos anteriormente. Vamos pensar em um *dataset* fictício contendo os dados de todos os estudantes da PUCPR:

- Garantimos que o **comportamento** dos dados faz sentido:
 1. Usamos bibliotecas visuais como o **seaborn**, **Plotly** e **matplotlib** para ver se a distribuição dos dados faz sentido ou não (o **pairplot** pode ser um bom ponto de partida):
 - Se percebêssemos que os estudantes da PUCPR em sua maioria possuem entre 20 e 25 anos de idade seria um bom sinal. Ver que existem alguns casos raros com pessoas acima de 60 anos de idade também. Agora, se a idade do estudante mais novo fosse de 3 anos teria algo de errado, não é? Talvez tenha sido um erro de digitação e deveríamos **substituir** esse valor ou, ainda, **remover** esse caso – eles são o que chamamos de *outliers*.
 - Imagine que exista um atributo chamado “altura”, o qual possui a altura dos estudantes. Se percebêssemos que metade dos alunos possui menos de 2,2 de altura e a outra metade dos alunos possui mais de 150 m de altura poderíamos ter um erro de conversão: pode ser que parte dos estudantes tenham a sua altura em **centímetros** (150 cm ou mais) e a outra parte, em **metros** (2,2 m ou menos). Nesse caso, precisaremos **padronizar** este atributo.
 2. Usamos bibliotecas como **pandas**, **NumPy** e **scikit-learn** para tratar as colunas:
 - Pode ser que nem todos os estudantes divulgaram sua idade, e essa informação pode ser útil para nós. Se poucos não divulgaram esta informação poderíamos **imputar** os dados – seja adotando um valor-padrão ou um valor calculado, por exemplo. Se uma grande parte dos estudantes não divulgou a idade, pode ser mais interessante remover a coluna (infelizmente).
- Garantimos que os dados estão preparados para um modelo de aprendizagem supervisionada:
 1. Pode ser que precisemos **normalizar** os atributos utilizando bibliotecas como pandas, NumPy e scikit-learn para que os algoritmos aprendam corretamente.
 2. Pode ser que precisemos garantir que não tenhamos nenhum **dado faltante** – alguns algoritmos apresentam erros se informarmos a eles um *dataset* de treino com algum dado faltante.
 3. Pode ser que tenhamos **muitas colunas**, para isso, precisaremos selecionar apenas as que importam para o nosso caso.
 - Podemos empregar técnicas de seleção de atributos ou de extração de atributos.
 - Podemos, ainda, selecionar manualmente as colunas que são mais relevantes para nós ao executarmos uma análise multivariada de dados e uma análise de correlação dos dados.

Depois de prepararmos os dados, adentramos então o mundo das técnicas de aprendizagem supervisionada em suas principais vertentes: classificação, regressão e previsão de séries temporais. Já comentamos sobre essas três vertentes logo na primeira unidade – logo, agora vamos mais a fundo entendê-las. Para isso, trabalharemos não somente com o scikit-learn, mas também com o LightGBM e o XGBoost.

Figura 1 – Processo de ML



Fonte: Lenz, 2020 (adaptado).

CINCO TRIBOS

A ideia de mostrar essas cinco “tribos” é a de garantir que você entenda de forma resumida e concisa o funcionamento dos principais tipos de algoritmos. Ela foi concebida por Pedro Domingos, professor emérito da Universidade de Washington e divide as técnicas de ML em cinco grandes **tipos/escolas de pensamento/tribos**. Ao ler sobre essas tribos você saberá melhor o que diferencia um *random forest* de um SVM, por exemplo.

Ao entender as tribos, você naturalmente entenderá como as técnicas de aprendizagem supervisionada funcionam em linhas gerais. Se me permite a analogia, é como se estivéssemos explicando a um futuro cozinheiro qual é a diferença entre queijo e requeijão, entre vinho e vinagre, entre orégano e erva-doce, entre presunto e apresuntado, ou entre brócolis, couve-flor, repolho e couve-de-bruxelas (a propósito: sabia que esses quatro são a mesma espécie de planta?). Bom, entendeu a ideia e a importância dessa conversa, né?

Cada tribo tenta resolver de uma forma diferente um tipo de problema que é especial para si. Por essa razão, não existe a **melhor** tribo que resolve da melhor forma todos os casos: dependendo do problema que esteja tentando resolver, pode ser que outro tipo de algoritmo dê resultados melhores e/ou mais rápido (e, por extensão, com um custo menor). Por isso, não vá pensar que redes neurais ou *deep learning* seriam a solução mágica ou a bala de prata de todos os problemas de ML do mundo.

Para explicar como cada uma das tribos aborda um determinado tipo de problema vamos usar novamente o exemplo do *Iris dataset*, beleza?

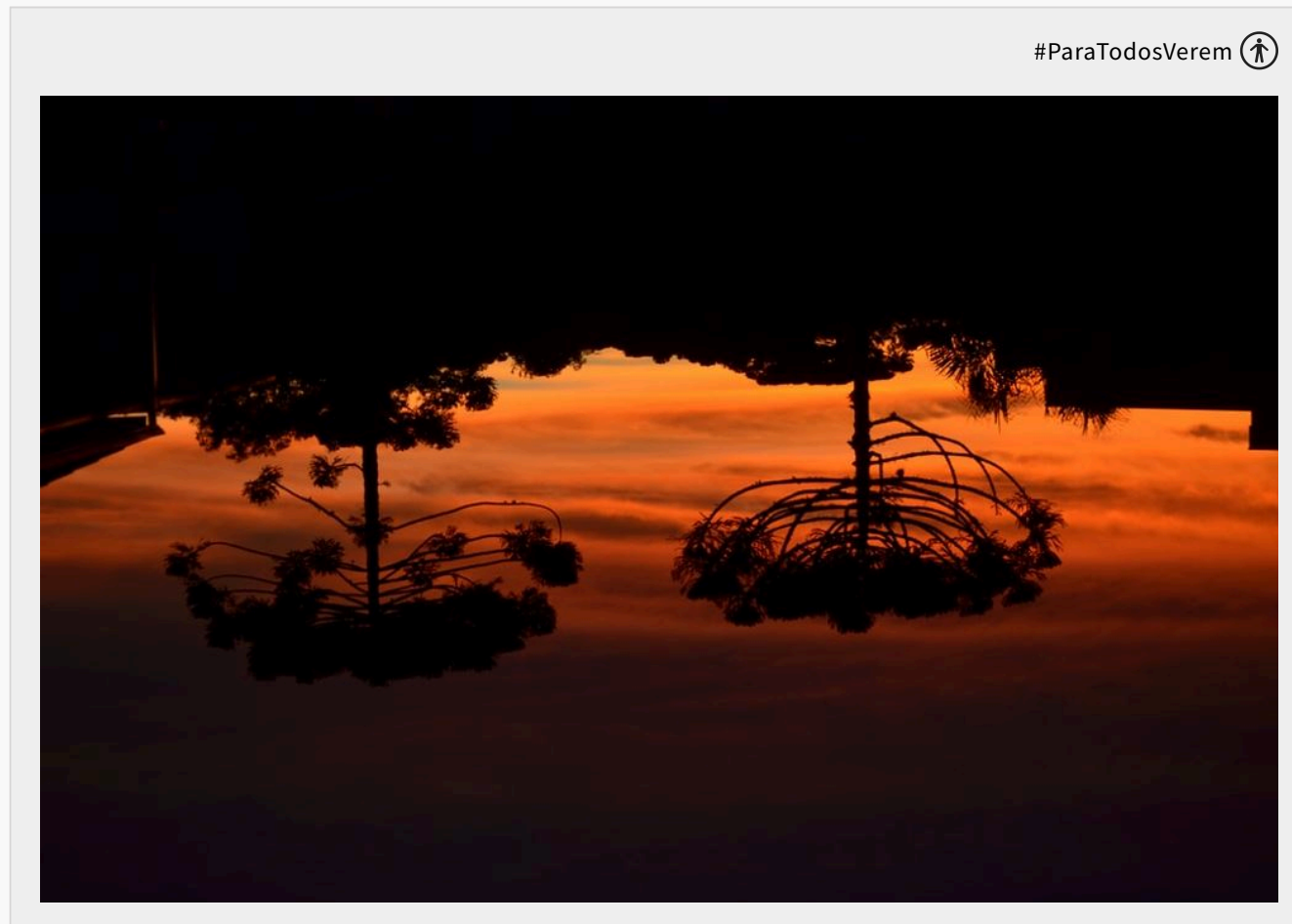
Vamos lá?

Symbolists (simbolistas)

1. **Foco:** lógica matemática
2. **Exemplos de algoritmos:** árvores de decisão, *random forests*.
3. **Pontos fortes:** facilmente interpretáveis por humanos; podem funcionar bem para *datasets* menores.
4. **Pontos fracos:** podem se confundir caso os dados tiverem problemas (ruídos) e/ou se não fizermos uma boa preparação da base de dados; podem não ser eficientes para grandes bases de dados.

Pesquisadores e algoritmos nessa área possuem como foco a lógica matemática. Esse era praticamente o sinônimo de IA até a década de 1980 e fazia-se o uso de um conjunto de **regras** rígidas para tomar uma decisão. Dependendo do problema que estamos tentando resolver, poderemos ter na casa dos milhões de regras. Algoritmos desse tipo são facilmente **interpretáveis**, mas não necessariamente **explicáveis**. Exemplos incluem os algoritmos de árvore de decisão.

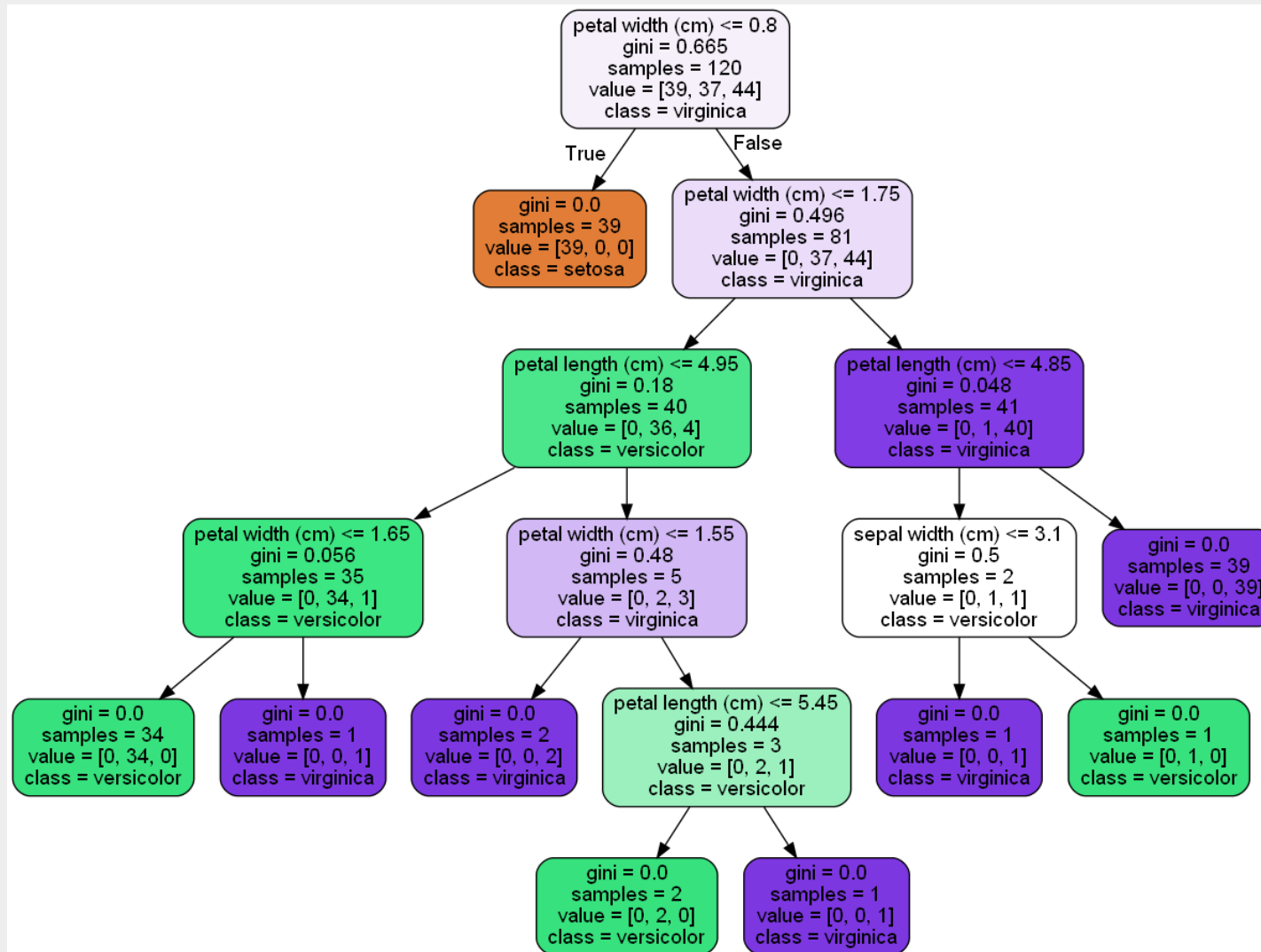
Figura 2 – Por algum motivo, em TI, as árvores sempre começam de cima para baixo. Acostume-se.



A imagem abaixo é um exemplo de uma árvore de decisão que foi treinada (ou seja, já é um exemplo de um algoritmo para prever **novos casos** do *Iris dataset*).

Podemos interpretar da seguinte maneira: começamos com uma regra principal (na parte superior) e vamos indo para a esquerda (se a resposta for sim) ou para a direita (se a resposta for não). Imaginemos uma flor com uma largura de pétala (*petal width*) de 5 cm: começamos no nível superior, em que a pergunta é “o *petal width* é menor ou igual que 0,8 cm?”. Como a resposta é “não”, vamos para a direita. A próxima pergunta é “o *petal width* é menor ou igual que 1,75 cm?”. Como a resposta é “não” mais uma vez, vamos para a direita. A nova pergunta é: “o *petal width* é menor ou igual que 4,85 cm?”. Como a resposta é de novo um “não”, terminamos: o algoritmo decidiu que a resposta é “*virginica*”. Consegue ver como ela funcionaria com outras combinações de flores?

Figura 3 – Árvore de decisão gerada para o Iris



Fonte: Autor (2021)

Existe também outro tipo de algoritmo que são os **random forests**. O *random forest* é aquilo que chamamos de **ensemble** – ou seja, um **comitê** de algoritmos. De uma forma bem simples, o *random forest* cria dezenas, centenas ou milhares de árvores parecidas com a árvore de decisão acima dentro de um modelo só. Cada árvore é organizada de forma diferente e, por consequência, possui regras diferentes. Cada uma das árvores pode chegar à uma decisão diferente. Assim, pegamos a decisão de todas as árvores e escolhemos a média de todos: é como se cada árvore tivesse um poder de voto e a maioria vencesse.

Analogizers (analogizadores)

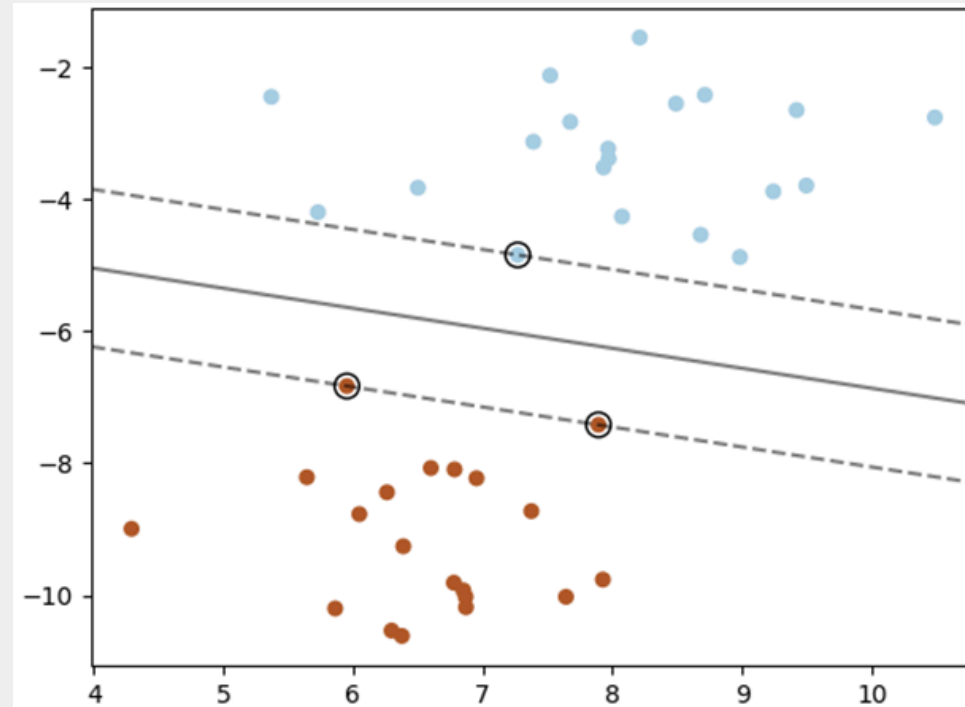
1. **Foco:** generalização de regras.
2. **Exemplos de algoritmos:** SVM e KNN.
3. **Pontos fortes:** consegue facilmente reconhecer similaridades entre vários casos e inferir, para novos casos, onde eles poderiam se encaixar.
4. **Pontos fracos:** podem sofrer bastante da maldição da dimensionalidade comentada na última unidade.

Algoritmos que entram nessa definição não criam um conjunto de regras como os *symbolists* – na verdade, buscam generalizar todos os dados de uma forma que seja possível estabelecer fronteiras. É como se pegássemos uma pessoa que riscasse o mapa do Brasil em uma linha reta, apontasse para a linha e falasse: “todo mundo que mora ao sul de São Paulo fala **bolacha** e não **biscoito** – *sem exceções!*”. Ter uma opinião tão pesada e crítica quanto essa pode trazer para essa pessoa um conjunto de problemas – ainda mais dependendo do contexto. Por outro lado, se precisássemos por algum motivo generalizar a tomada de decisão pensando em velocidade ou simplicidade, essa opinião poderia ser aceita. Isso se aplica aqui.

Os dois principais representantes dessa tribo são o KNN (às vezes escrito como kNN ou k-NN), o SVM e modelos lineares mais simples e encontrados normalmente na matemática, como a **regressão linear**. O KNN significa *k-nearest neighbors*, ou **k-vizinhos mais próximos**. O “k” é um número que você define, e que geralmente é algo entre 3 e 10. Esses algoritmos pegam um determinado *dataset* e, a partir dele, começam a calcular as k instâncias mais próximas de uma nova instância. A partir delas, ele prevê o resultado.

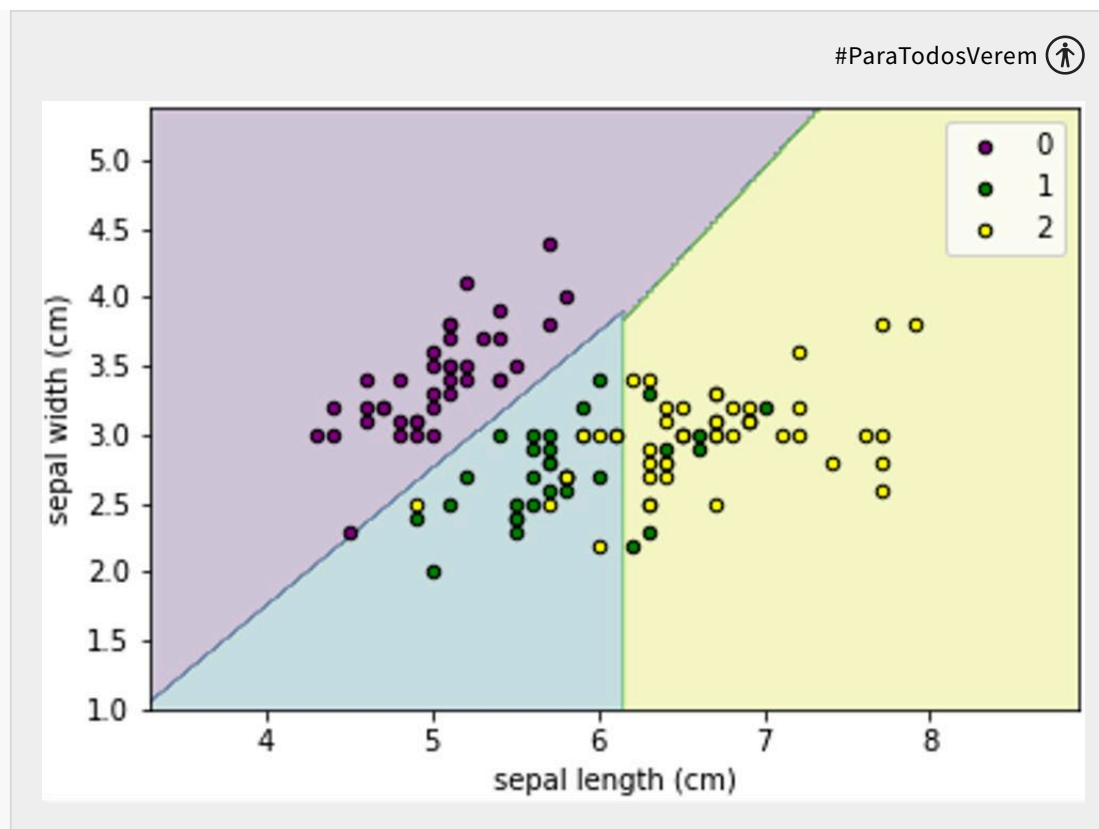
Um exemplo: vamos supor que queremos prever a espécie de uma nova flor *Iris*. Essa nova flor tem uma largura de pétala de 0,4 cm; comprimento de pétala de 1,3 cm; largura de sépala de 5,1 cm; e comprimento da sépala de 3,0 cm. Ora, pode ser que o KNN nunca tenha visto algo assim antes com essas mesmas métricas, mas ele viu que tem umas 4 flores que têm dimensões bem parecidas com essa, e todas elas são do tipo *Iris setosa*. Logo, por **similaridade**, ele prevê que essa também seria, por definição, uma *Iris setosa*.

Já o SVM, ou *support vector machine* (máquina de vetor de suporte) busca estabelecer “fronteiras” entre as instâncias. É como se ele tentasse criar uma linha de divisão entre uma classe e outra (na imagem abaixo ilustrada pela linha contínua, a qual chamamos de **hiperplano**) utilizando algumas instâncias para definir esse hiperplano (os **vetores de suporte**, ilustrados pelas bolinhas no meio das linhas tracejadas) as quais, por sua vez, são definidas por um parâmetro de regularização chamado “C” e que possui um efeito direto nas linhas tracejadas. Quanto maior o C, mais afastadas as linhas tracejadas ficam uma da outra, o que pode ter um impacto nos erros e na *performance* do modelo. Dependendo da literatura, vão dizer que um C maior implica em mais erros e, em outros casos, em menos erros.



Fonte: O autor (2021)

Observe o exemplo abaixo gerado para o *dataset Iris*. Apenas duas colunas foram utilizadas no treinamento pensando na visibilidade. Observe como o SVM tenta estabelecer uma faixa de corte entre cada uma das classes (ainda que isto não funcione muito bem para os casos mais próximos das fronteiras). Note também que as faixas de corte são retas: isso ocorre porque o *kernel* (motor matemático) usado é linear.



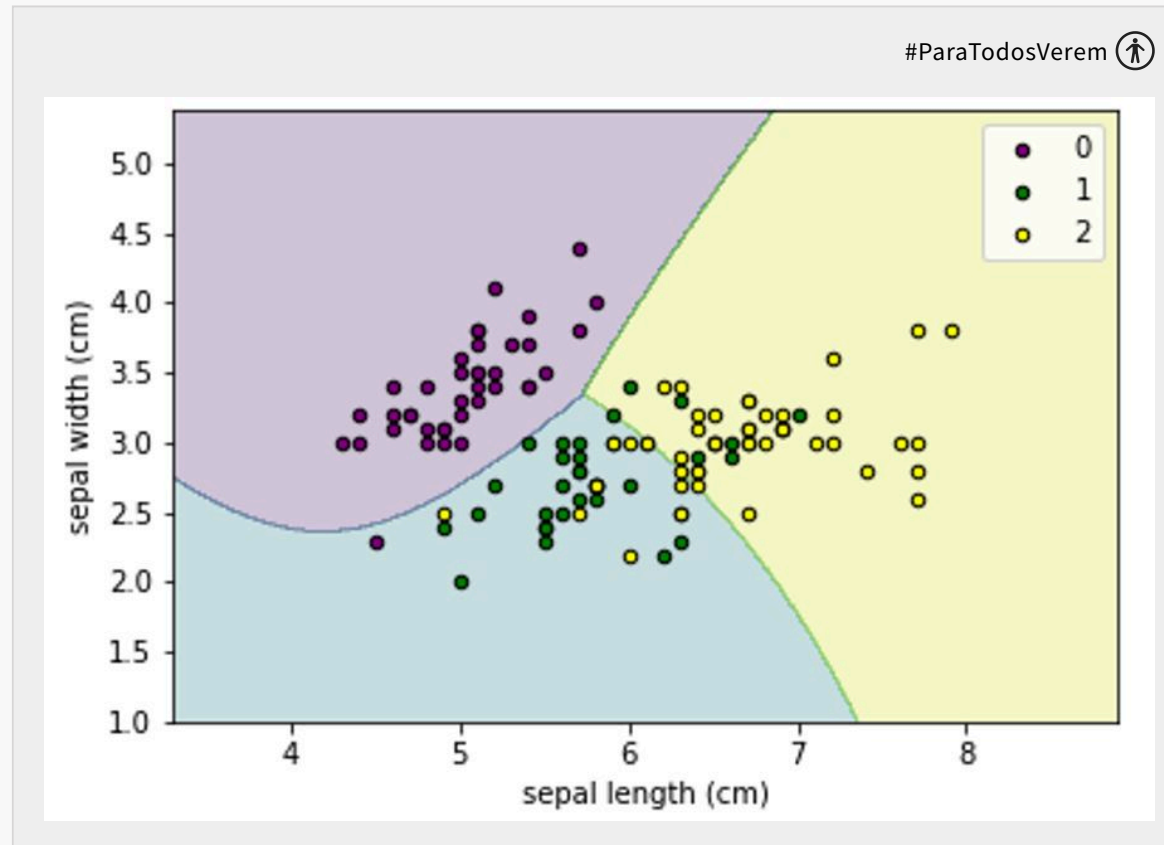
Fonte: O autor (2021)

Bayesians (bayesianos)

1. **Foco:** incerteza e uso de estatística.
2. **Exemplos de algoritmos:** Naïve Bayes.
3. **Pontos fortes:** fundado em regras estatísticas onde a lógica por si só poderia não funcionar corretamente – principalmente em casos nos quais temos muito ruído nos dados ou ambiguidade nos mesmos.
4. **Pontos fracos:** custo computacional caro (ou seja, pode exigir mais poder computacional como o processador para aprender um determinado *dataset*).

Algoritmos desta tribo são, geralmente, fundados no teorema de Bayes. Esse teorema vem da estatística e parte de uma premissa que a **probabilidade** de alguma coisa muda conforme vamos recebendo maiores evidências e informações. É como se soubéssemos que algo pode ser verdade (ou não) a partir de certos efeitos específicos (ou gerais) e da combinação entre eles.

No scikit-learn, o Naïve Bayes está representado em algumas implementações. O termo “*naïve*” ou “*naive*” significa “ingênuo” ou “inocente”. No nosso caso, esse tipo de algoritmo simplifica algumas relações ou combinações estatísticas de uma base de dados ao tornar todos os efeitos independentes, dada uma certa causa (ou seja, o que estamos tentando prever). Note como ficam os resultados da aplicação desse algoritmo com o mesmo *dataset* modificado do *Iris* que testamos com o SVM. Note que ele tenta adaptar os limites entre uma classe e outra de uma forma diferente:



Fonte: O autor (2021)[/legenda]

Connectionists (conexionistas)

1. **Foco:** aplicação de conceitos de neurociência/funcionamento do cérebro humano em algoritmos.
2. **Exemplos de algoritmos:** redes neurais (o que inclui *deep learning*).
3. **Pontos fortes:** pode aprender problemas muito complexos e com resultados impressionantes.

4. **Pontos fracos:** complexidade (pode requerer muito processador ou GPUs (placas de vídeo) para resolver um determinado problema); pode ser difícil de ser interpretado; suas soluções podem ser “caixa-preta”.

Algoritmos dessa tribo são representados pelas redes neurais ou *deep learning*. Se já pesquisou algo sobre isso, deve também ter se deparado com termos como **Tensorflow**, **Keras** e **pyTorch**. As redes neurais são algoritmos complexos compostos de camadas com “neurônios”. Essas camadas são conectadas entre si, e os neurônios são ativados/desativados dependendo dos estímulos enviados a eles – isto é, os atributos de um *dataset*. Dado o devido tempo, o algoritmo aprende e calibra quais os pesos que devem ser atribuídos para cada uma dessas conexões visando a ativá-los corretamente e a funcionarem bem para cada particularidade.

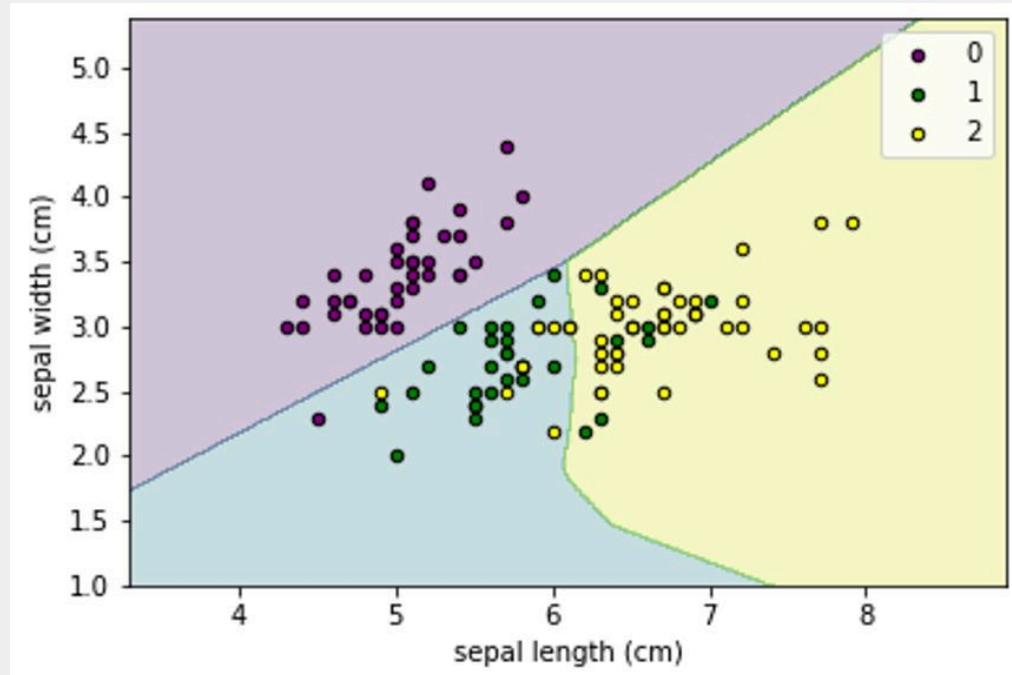
Parece legal, né? Se pesquisar mais sobre o assunto, verá que os tópicos redes neurais e *deep learning* (uma configuração mais complexa de redes neurais) é muitas vezes vendido como a melhor solução do universo. Não é bem assim: frequentemente, o emprego de redes neurais deve ser entendido como um canhão para matar uma formiga – nessa analogia, o canhão também pode errar a formiga. Ou seja: é algo que pode ser complexo demais, exigir um esforço tremendo e ser muito caro quando uma solução mais simples poderia muito bem resolver o mesmo problema (e até melhor)!

Além disso, outro problema é a falta de **explicabilidade** sobre como uma rede neural funciona: imagine ter que explicar para o seu gerente o porquê da sua rede neural hiper futurística ter errado uma decisão. Com uma árvore de decisão poderia ser fácil de entender, mas não com uma rede neural – ela se enquadra no que chamamos de “modelo caixa-preta” por não termos acesso fácil ao seu modo de funcionamento.



EXPERIMENTE

Se quiser entender como uma rede neural funciona na prática, sugiro brincar um pouco com o [playground do Tensorflow](#). Ele é mantido pela própria equipe do Tensorflow e serve para você ver como ele vai aprendendo na prática. O [scikit-learn também possui um algoritmo de redes neurais](#). Observe, na imagem abaixo, como fica o resultado da aplicação de redes neurais para o mesmo problema modificado do *Iris* que vimos para as tribos anteriores. Faça uma comparação visual desse resultado com os anteriores – principalmente para os casos extremos (isto é, mais próximos dos limites do gráfico).



Fonte: O autor (2021)

Evolutionaries (evolucionários)

1. **Foco:** aplicação de conceitos da biologia e genética como genes, reprodução, gerações, população e mutação.
2. **Exemplos de algoritmos:** algoritmos genéticos (GAs)
3. **Pontos fortes:** aprendem casos complexos ao empregar múltiplas soluções, adaptando-as até chegar ao(s) melhor(es) candidato(s)
4. **Pontos fracos:** podem acabar não explorando todas as alternativas, ou não encontrando a melhor alternativa possível. Também pode requerer um alto uso de recursos computacionais dependendo da complexidade do problema.

Algoritmos dessa tribo fazem uso da **programação genética** – isto é, utilizam conceitos relacionados à teoria da evolução para alcançar os melhores resultados. Esse tipo de algoritmo funciona muito bem em algumas áreas da Engenharia (como Controle e Otimização) e, por isso, acaba não sendo muito empregado em soluções mais genéricas como classificação e regressão. Outro motivo para isso é porque ele é mais próximo da área de aprendizagem por reforço, tema explicado na primeira unidade.

Um problema de otimização seria o seguinte: “encontre a rota mais próxima da sua casa até a praia”. Se pensássemos em todas as combinações de curvas, estradas, ruas, retornos, rotatórias e afins teríamos uma infinidade de alternativas. A ideia é que ele possa descobrir qual é a melhor rota, mas precisa entender o que é entendido por “melhor”. Se a definição de melhor for “rota mais rápida”, ele gerará aleatoriamente um certo número inicial de caminhos: cada rota seria um “indivíduo”.

Depois disso, o algoritmo descobre quais são as melhores rotas e gera “filhos” dessas rotas ao combinar algumas características das melhores rotas descobertas anteriores. Somados a esses filhos, ainda gera pequenas alterações (mutações) nas rotas para ver se isso não poderia melhorar ainda mais o resultado.

Nisso, é provável que alguns desses novos indivíduos (rotas) sejam ainda melhores do que os anteriores – essas novas rotas são então recombinadas e sofrem novas mutações, gerando, assim, mais rotas até que finalmente alcancemos às melhores rotas de fato.

| RESUMO DAS TRIBOS E EXEMPLOS COMPLEMENTARES

Vimos anteriormente que existem cinco grandes grupos/tipos/tribos de algoritmos de ML: *symbolists*, *analogizers*, *bayesians*, *connectionists* e *evolutionaries*. Cada um naturalmente possui as suas vantagens e desvantagens, como já explorado nas páginas anteriores. Além disso, como um curso em nível de tecnólogo possui como foco a aplicação em vez da teoria, buscamos trazer essas diferenciações de forma simplificada e ágil. Não raramente, o estudo de apenas uma dessas tribos ou somente de um desses algoritmos demanda anos de pesquisa científica.

Além disso, reforçamos para você que é **importante o conhecimento** desses diferentes algoritmos e tribos: pessoas que trabalham em equipes de ciência de dados necessitam muitas vezes conhecer diversas técnicas em vez de uma só: na maioria dos processos de contratação, espera-se conhecedores de técnicas de ML, e não apenas conhecedores de técnicas de redes neurais. Isso é importante, uma vez que a escolha da técnica dependerá de cada problema.

Isso, por sua vez, leva ao segundo ponto: a **definição do melhor algoritmo** para um determinado *dataset* é um processo construtivo: não raramente nos deparamos com pessoas que possuem um algoritmo favorito (principalmente o XGBoost, RandomForest, LightGBM e redes neurais), mas é entendido apenas como um **ponto de partida**: não há problema algum em você começar seus testes com RandomForest e depois testar outros algoritmos desde que você **teste** – desde que coloque a mão na massa. Não se restrinja jamais à uma única técnica: se desafie! Hoje, existem também soluções de **Automated ML** (ou Auto ML) as quais testam diferentes algoritmos, e que são geralmente usadas por empresas. Essas soluções testam, ao mesmo tempo, vários dos modelos que vimos acima – logo, para saber **interpretar** seus resultados, é importante que entenda as principais diferenças entre essas técnicas também.



EXEMPLO

Certo – passados esses pontos, preparamos uma coletânea contendo alguns exemplos de funcionamento das técnicas. São GIFs ou vídeos curtos os quais demonstram o funcionamento das diferentes técnicas, e servem como **complemento** ao conteúdo anterior. Entendo que é legal, uma vez que neste documento ficamos restritos às figuras estáticas. Vamos lá:

1. Analogizers

a. SVM

- i. <https://www.youtube.com/watch?v=UKqMQSKQFYI>
- ii. <https://www.youtube.com/watch?v=UFnjV1E615I>
- iii. https://raw.githubusercontent.com/ecs-vlc/torchbearer/master/docs/_static/img/svm_fit.gif

b. KNN

- i. https://www.youtube.com/watch?v=bmDdo_5IP2k
- ii. <https://machinelearningknowledge.ai/wp-content/uploads/2018/08/KNN-Classification.gif>

2. Bayesians

a. Naïve Bayes

1. https://upload.wikimedia.org/wikipedia/commons/b/b4/Naive_Bayes_Classifier.gif

3. Connectionists

4. Redes neurais/deep learning

- i. <https://playground.tensorflow.org>
- ii. <https://www.youtube.com/watch?v=fQ376G-Ek1E>

5. Evolutionaries

a. Algoritmos genéticos

| APRENDIZAGEM SUPERVISIONADA

Como comentamos anteriormente, vários dos algoritmos representados pelas tribos são empregados em aprendizagem supervisionada. Lembre-se de que a aprendizagem supervisionada é essencialmente dividida em dois tipos :

- Regressão (*regression*)
 1. **Objetivo:** queremos prever uma **quantidade** ou **valor**.
 2. **Exemplos:** cotação do dólar; porcentagem da inflação; número de seguidores no canal da Twitch; nota no final da disciplina; saldo bancário no final do mês; quantidade de gols marcados de um determinado jogador de futebol;
 3. **Subtipos:**
 - Séries temporais (*time-series forecasting*): queremos prever uma quantidade ou valor **ao longo do tempo**. Exemplo: cotação do dólar ao longo do próximo mês; litros de energético vendidos no supermercado ao longo da semana; evolução mês a mês do saldo bancário.
- Classificação (*classification*)
 1. **Objetivo:** queremos prever uma **classe/grupo/tipo**.
 2. **Exemplos:** empréstimo aprovado/reprovado; aluno aprovado/reprovado/direcionado ao exame final na disciplina; o *e-mail* é *spam*/não é *spam*; o time de futebol será rebaixado/não rebaixado/classificado para outro campeonato/campeão; a cotação do dólar cairá/não cairá.
 3. **Subtipos:**
 - Classificação binária (*binary classification*): quando estamos tratando somente de duas classes (sim/não; verdadeiro/falso; chove/não chove).
 - Classificação multiclasse (*multiclass classification*): quando estamos tratando de três ou mais classes (céu limpo/céu nublado/chuvoso; urgência baixa/média/alta; bandeira amarela/laranja/vermelha).

Técnicas incluídas no scikit-learn

Para resolver esses problemas, o scikit-learn possui algoritmos divididos em alguns tipos. A lista abaixo possui alguns exemplos. Nesta disciplina, seria interessante se você pudesse testar ao menos as árvores de decisão/random forests, SVM, regressão linear/logística e KNN.

- *Symbolists*

1. Regressão

- Árvore de decisão (DecisionTreeRegressor).
- Florestas de árvores de decisão (RandomForestRegressor).

2. Classificação

- Árvore de decisão (DecisionTreeClassifier).
- Florestas de árvores de decisão (RandomForestClassifier).

• *Analogizers*

1. Regressão

- SVM para regressão (SVR).
- KNN para regressão (KNeighborsRegressor).
- Regressão linear (LinearRegression).

2. Classificação

- SVM para classificação (SVC).
- KNN para classificação (KNeighborsClassifier).
- Regressão logística (LogisticRegression).

• *Bayesians*

1. Classificação

- Naive Bayes Gaussiano (GaussianNB).
- Naive Bayes para atributos categóricos (CategoricalNB).

• *Connectionists*

1. Regressão

- Multi-layer Perceptron (MLP) para regressão (MLPRegressor).

2. Classificação

- Multi-layer Perceptron (MLP) para classificação (MLPClassifier).

Ensembles

Além disso, existem os ***ensembles***, ou comitês de algoritmos. Lembra que falamos deles quando mencionamos os *random forest* lá atrás? Pois bem: existem outras configurações de *ensembles* as quais podem funcionar com outros algoritmos. Ou seja, da mesma forma que o *random forest* é um comitê de árvores de decisão, podemos também ter um comitê de SVMs, ou um comitê de algoritmos de regressão logística ou, ainda, um que misture vários desses algoritmos – até um comitê de *random forests* é possível!

A grande vantagem dos *ensembles* é combinar vários modelos que sozinhos não seriam tão eficazes em um conjunto robusto: é um exemplo clássico de “a união faz a força”. Os *ensembles*, hoje, são empregados para vários problemas nas empresas com uma boa *performance* e um baixo custo computacional.

No scikit-learn, existem vários tipos de *ensembles*, e eles são divididos em dois tipos: os que calculam uma média/votação (*bagging*) de vários modelos que servem de base; e os de *boosting*, em que o resultado de um pode servir de entrada para o outro: é como se um modelo servisse de trampolim para o outro, que serve de trampolim para o próximo, e assim sucessivamente. Por outro lado, gostaria que ficasse claro para você que provavelmente são somente dois tipos de *ensembles* que você precisa conhecer nesta disciplina:

- *Random forest (bagging)*, que já vimos.
- *Gradient boosting (boosting)*, representado principalmente por:
 1. XGBoost.
 2. LightGBM.

O XGBoost e LightGBM são outras duas bibliotecas (ou seja, não estão *dentro* do scikit-learn, mas seriam algo como suas “irmãs”) que implementam um algoritmo de *gradient boosting trees*: usando um conceito matemático de **gradiente**, no qual conseguimos criar um conjunto de árvores de decisão segundo uma lógica na qual cada nova árvore tem como objetivo reduzir o erro das árvores anteriores – ou seja, todas as árvores não são geradas aleatoriamente como no *random forest*, mas vão sendo criadas de uma forma mais “inteligente”.

Séries temporais

Para séries temporais, existem alguns métodos. Para ilustrar, vamos pensar em um *dataset* contendo somente o histórico da cotação do dólar. Abaixo encontra-se uma amostra de cinco valores. Note que o atributo “Cotacao” não possui cedilha e til – é uma boa prática evitarmos o uso de espaços e de acentuação no nome dos atributos:

Data	Cotacao
14.05.2024	5.273
13.05.2024	5.309
12.05.2024	5.306
11.05.2024	5.221

- 1. Utilizar bibliotecas específicas para a previsão de séries temporais, como o Prophet.
- 2. Utilizar técnicas da área da Estatística como o ARIMA.
- 3. Adaptar a base de dados para um problema de *regressão*: nesse caso, criamos atributos contendo tão somente as informações dos dias anteriores. No exemplo, criamos cinco atributos contendo a cotação dos cinco dias anteriores. Como o algoritmo precisa aprender para casos futuros e não poderia ter uma dependência específica de uma data, removemos aquela coluna. Esse *dataset* poderia então ser processado por um algoritmo de regressão.

Cotacao	Cotacao_-1	Cotacao_-2	Cotacao_-3	Cotacao_-4	Cotacao_-5
5.273	5.309	5.306	5.221	5.227	5.237
5.309	5.306	5.221	5.227	5.237	5.276
5.306	5.221	5.227	5.237	5.276	5.354
5.221	5.227	5.237	5.276	5.354	5.444
5.227	5.237	5.276	5.354	5.444	5.442

| HORA DO TREINO

Figura 4 – Se falamos de treino, falamos de hidratação. Já tomou água hoje?



Fonte: ©Nigel Msipa/ Unsplash

Retomando o passo a passo:

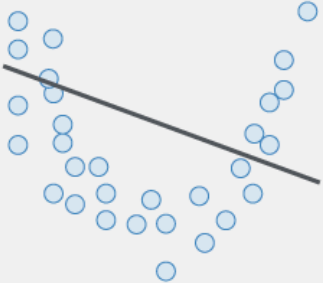
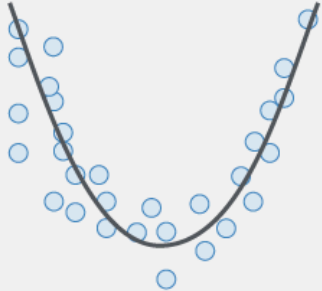
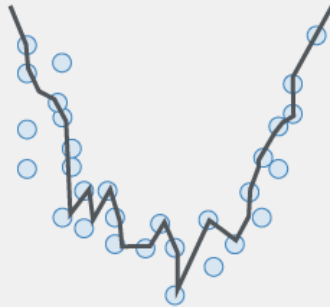
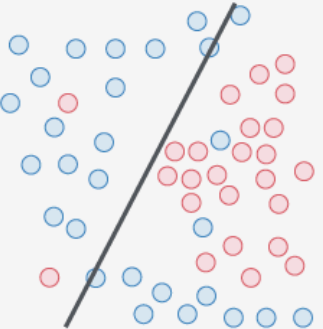
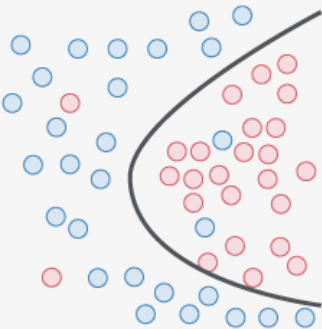
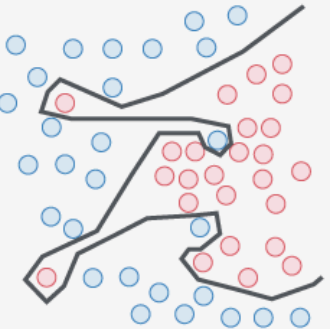
1. Selecionamos um **dataset**.
2. Fazemos uma preparação do **dataset**, removendo “sujeiras” dele, entendendo também quantas instâncias, atributos temos e identificando o que queremos **prever** (isto é, qual é o atributo que queremos prever).
3. Após identificar esse atributo, entendemos se é um problema de **regressão** (uma quantidade ou valor a ser previsto) ou **classificação** (um grupo ou tipo a ser previsto).
4. Dependendo do problema, escolhemos uma **técnica** de aprendizagem supervisionada para começar o treinamento (como o KNN, SVM e *random forest*).
5. Dividimos o **dataset**: algo em torno de 80% para o treinamento do algoritmo e os outros 20% separamos para simular dados novos a que o algoritmo nunca teve acesso durante o seu treinamento. Utilizamos esses 20% para validar a sua *performance* e para evitar o **overfit**.
6. Treinamos o modelo de ML escolhido no ponto 4 com o **dataset** de **treino** do ponto 5.

7. Avaliamos os resultados do algoritmo. Se as métricas não forem boas, poderemos retornar ao ponto 4 e tentar novamente.

Note que no ponto 5 falamos de **overfit**. Quando treinamos um algoritmo de ML, queremos que ele aprenda de uma forma *generalizável*: isto é, que consiga se adaptar bem às diferenças entre cada caso. Observe a imagem abaixo, em que mostramos um exemplo para classificação e outro para regressão. A linha preta refere-se ao que o modelo aprendeu. Note as diferenças entre o **underfitting** (no qual poderíamos dizer que ele, de fato, não aprendeu direito o comportamento da base de dados) com o **overfitting** (quando ele tenta reproduzir todas as particularidades da base de treinamento o que, por sua vez, implica que não conseguiria generalizar bem novos casos). O **mundo ideal** é a coluna do meio: note que ele busca generalizar todos os casos de uma forma suave.

É para garantir que estamos na direção certa que fazemos aquela divisão entre base de treinamento e de teste: se não o fizermos, não conseguiremos saber se estamos tendo um problema de *overfit* ou *underfit*. Imagine que o nosso algoritmo foi treinado e entendemos que ele possui um erro próximo de zero. Sensacional, não é? Ele está perfeito, afinal de contas! Aí, quando vamos cruzar com a base de testes que separamos anteriormente, vimos que ele tem um erro altíssimo! Ora, esse seria um sinal claro de *overfit*.

Em linhas gerais, sempre duvide quando o modelo acusar uma assertividade alta demais (ou, em outras palavras, um erro muito baixo).

	<i>Underfitting</i>	<i>Perfeito!</i>	<i>Overfitting</i>
Sintomas	<ul style="list-style-type: none">• Erro alto no treinamento.• O valor do erro do treinamento é próximo ao erro do teste.• Alto viés.	<ul style="list-style-type: none">• O valor do erro do treinamento é um pouco mais baixo do que o erro do teste.	<ul style="list-style-type: none">• O valor do erro do treinamento é muito baixo.• O erro do treinamento é muito mais baixo do que o erro do teste.• Alta variância.
Exemplo em regressão			
Exemplo em classificação			

Fonte: Adaptado de Victor Lavrenko, University of Edinburgh.

| MÃO NA MASSA

Note que existe um *notebook* em python na VM intitulado **Semana3_TecnicasMachineLearning_CincoTribos**. Esse *notebook* possui os códigos de treinamento dos modelos das tribos comentadas anteriormente utilizando scikit-learn. Note a sintaxe dos modelos – sempre temos um passo de **fit** para treinar e, depois, o **predict** para obter os resultados da predição. Também note que existe um passo de divisão entre uma base de treino e teste (**train_test_split**). Como sugestão, execute as células e veja o comportamento delas. Também busque identificar diferenças e semelhanças na **forma de uso** de cada um dos modelos.

Após isso, observe e execute o *notebook* **Semana3_TecnicasMachineLearning_ML**. Ele pega uma base de dados e cria modelos de classificação, regressão e previsão de séries temporais. Observe o código desenvolvido e o comportamento dele. Abra a documentação dos modelos e entenda os diferentes parâmetros – isso pode ajudá-lo bastante na compreensão dos modelos.

Para auxiliar na construção do seu entendimento, assista ao vídeo **É hora do show!**, no qual demonstramos o processo de treino e teste de um modelo de ML.

Depois de vermos os principais conceitos de ML vamos colocar a mão na massa e criar um modelo preditivo utilizando python e o Jupyter. Vamos lá?

É hora do show!



| CONCLUSÃO

Nesta unidade, adentramos o tema de aprendizagem supervisionada ao contextualizar os diferentes tipos de técnicas em cinco grandes “tribos”. Após isso, comentamos como os algoritmos se comportam de acordo com a tribo a qual eles pertencem, independentemente, de ser um algoritmo de classificação ou de regressão. Também comentamos sobre os *ensembles* – conjuntos de modelos de ML que conseguem dar resultados frequentemente mais assertivos.

Finalmente, citamos a importância de dividirmos o *dataset* em treino e teste para evitarmos o *overfit*.

| REFERÊNCIAS BIBLIOGRÁFICAS

HUYEN, C. **Projetando sistemas de *machine learning***: processo iterativo para aplicações prontas para produção. Rio de Janeiro: Editora Alta Books, 2024.

LENZ, M. L. *et al.* **Fundamentos de aprendizagem de máquina**. Porto Alegre: SAGAH, 2020.

RUSSELL, S.; NORVIG, P. **Inteligência artificial**. 4 ed. Rio de Janeiro: LTC, 2024.

