



Técnicas de Machine Learning

UNIDADE 06

Canalizando o conhecimento: pipelines de ML

| PREPARANDO-SE ANTES DE INSTALAR O ENCANAMENTO

Tem um jeito melhor de fazermos o nosso trabalho?



Até o momento, entendemos como se constrói de forma lógica um algoritmo de ML. Percebemos, por exemplo, que precisamos criar código para trabalhar com o nosso *dataset* (utilizando algoritmos como PCA, RobustScaler, MinMaxScaler, remoção de *outliers*, preenchimento de dados nulos, entre outros). Também percebemos que esse código facilmente se transforma em um conjunto de linhas que se agrupam entre várias células dentro de um *notebook* em python.

Não sei se você já se sentiu incomodado por isso, mas durante todo esse processo você teve que chamar várias funções repetidas vezes. Vários *fit*, *transform*, *predict* e afins. Várias regras a serem inseridas e gerenciadas. Vários botões *Run* sendo clicados no Jupyter Notebook (ou o seu atalho, Shift+F5).

Imagine fazendo isso **todo dia**, ou **toda vez** que chegam novas instâncias para prevermos o resultado!

E se eu te dissesse que existe uma forma de organizar o código para que só precisemos usar o *fit* ou o *predict* uma única vez e todos esses passos seriam executados de forma automática? É sobre isso que falaremos nesta semana. Essa forma se chama ***pipeline***, ou um encanamento. Imagine todo o processo de dados ocorrendo de forma fluida como um encanamento que transporta líquidos. Legal, né?

Além disso, falaremos sobre a **otimização de hiperparâmetros**. Lembra quando reforçamos para que você se ambiente com a documentação das bibliotecas? Recordemos, por exemplo, que o RandomForestRegressor possui um parâmetro chamado `n_estimators` (isto é, o número de árvores de decisão empregado). O padrão é 100 árvores (`n_estimators = 100`). Quem te garantiria que o seu algoritmo não seria melhor se tivesse 50 árvores? Ou 500? Ou 1000? Como testaríamos todas essas combinações?

| PIPELINES

Os *pipelines* são formas de **agrupar** todos os passos empregados para treinar um algoritmo de ML. Com frequência, nos deparamos com casos nos quais precisamos fazer várias transformações dos dados antes de treinar o modelo. E, com frequência, a “cara” desses dados nada tem a ver com a sua forma original quando eles chegam para um modelo de aprendizagem supervisionada. Vamos exemplificar, beleza?

Vamos supor que você possui a tarefa de construir um algoritmo para prever se uma pessoa terá o seu cartão de crédito aprovado. Nisso, temos algumas premissas. A primeira é a de que alguém nos informará uma base de dados histórica contendo várias informações de pessoas que pediram cartões de crédito no passado e que foram aprovadas ou reprovadas. A segunda é a de que se espera que os dados futuros trabalhem exatamente com esses mesmos dados (afinal de contas, de nada adianta criarmos um algoritmo que tem um erro baixíssimo e que necessita obrigatoriamente da renda da pessoa se para os dados futuros não tivermos essa informação disponível para nós, não é?). A terceira é a de que certamente teremos que manipular (sanear) a base de dados para o treinamento e que teremos que aplicar as mesmas manipulações para novos dados.

Ainda pensando nesta base, vamos supor que os dados seriam:

1. CPF da pessoa.
2. Data de nascimento.
3. Data de processamento da solicitação.
4. Sexo.
5. Salário.
6. Cargo.
7. Estado civil.
8. Cidade.
9. UF.
10. Solicitação aprovada/reprovada (classe).

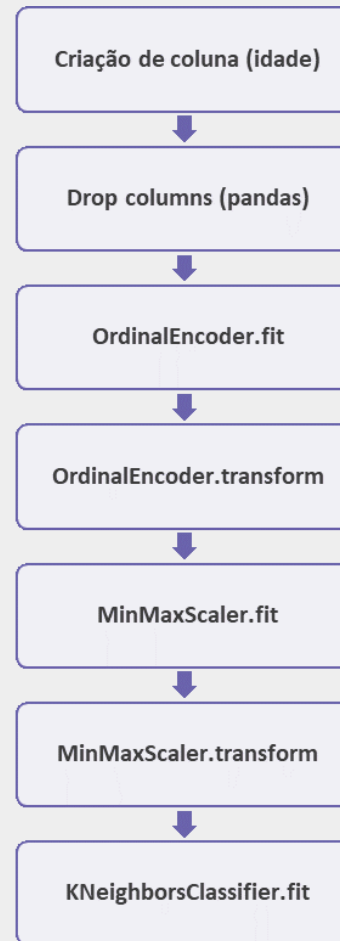
Olha... Não sei você, mas entendo que não poderíamos simplesmente pegar esses dados e colocar em um algoritmo de aprendizagem supervisionada. Vou listar alguns motivos:

1. O CPF precisará ser removido – não queremos que um algoritmo tente aprender algum padrão com o CPF, até porque o CPF é um identificador.
2. A data de nascimento precisará ser convertida para uma idade. Um algoritmo aprende a partir de números e, ainda, o que acontece com a vida de uma pessoa muda com próprio tempo: alguém nascido em 2000 poderá ter uma situação financeira desfavorável em 2018 (por estar iniciando a sua carreira), mas ter uma situação bem mais estável em 2020 (por ter um emprego fixo por dois anos). Logo, com a data de nascimento e a data de solicitação é possível saber a idade da pessoa quando ela solicitou o cartão.
3. O sexo, cargo, estado civil, cidade e UF precisariam ser convertidos para uma escala numérica com algo como o `OrdinalEncoder` ou o `OneHotEncoder`.
4. O salário e as demais colunas que foram convertidas para uma escala numérica poderiam ser também padronizadas com um *scaler* (como o `RobustScaler` ou `MinMaxScaler` que vimos nas unidades anteriores) e/ou com o uso de técnicas como o PCA ou `SelectKBest`.

Perceba que até o momento conduzimos todo esse processo no Jupyter ao longo de múltiplas células. No final das contas, treinávamos um modelo de aprendizagem supervisionada a partir do *dataset* manipulado (isto é, após passar pelos itens acima). Agora, pense comigo: chegaram novos dados, e não estou falando do `train_test_split`, mas, sim, de uma base de dados completamente nova. Como você organizaria o código para sempre fazer essas transformações para novos dados?

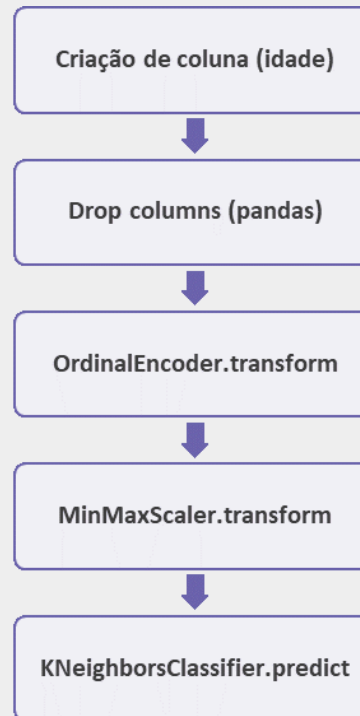
Os *pipelines* **agrupam** esses diferentes passos em uma única variável. Assim, você só precisa chamar a função *fit* uma única vez (ou a *predict* uma única vez). Isto ajuda muito na gestão de modelos prontos que podem ser utilizados para novos dados (como novas pessoas solicitando cartões de crédito: melhor fazer todo o processo com somente uma ou duas linhas de código do que repetir várias células, não é?).

Dessa forma, o que tínhamos até o momento era algo assim:



Fonte: O autor (2021)

Para novos casos, nós teríamos:




Fonte: O autor (2021)

Com os *pipelines*, teríamos:

Pipeline.fit

- ColumnTransformer.fit
- OrdinalEncoder.fit
- [classe customizada para datas].fit
- MinMaxScaler.fit
- KNeighborsClassifier.fit

E, para novos casos:

#ParaTodosVerem 

Pipeline.predict

- ColumnTransformer.predict
- OrdinalEncoder.predict
- [classe customizada para datas].predict
- MinMaxScaler.predict
- KNeighborsClassifier.predict

O scikit-learn possui uma ampla seção [explicando sobre pipelines](#) e [sobre o seu uso](#). Observando os exemplos anteriores, note que só fazemos o *fit* uma vez ou o *predict* uma vez. Dentro do pipeline é possível perceber que este *fit* ou *predict* é replicado para os passos dentro dele, mas de uma forma mais automática.

Resumidamente, vamos supor que esse processo de ML fosse comparável a transportar água de um lugar A até um lugar B. Dessa forma, se o que fizemos até agora era parecido com isto:



Fonte: © 鄧南光/Wikimedia Commons

Agora, com os *pipelines* seria parecido, com isto:



Fonte: ©Mike Benna/Unsplash

| OTIMIZAÇÃO DE HIPERPARÂMETROS

Se você analisou os notebooks em Python que disponibilizamos até o momento deve ter se deparado com vários momentos nos quais incentivávamos você a ler a documentação das bibliotecas (ex.: pandas, scikit-learn, LightGBM, XGBoost e outros). A ideia não é a de ler sem um direcionamento, mas, sim, a de compreender quais são as possibilidades.

Vamos fazer um pequeno exercício de pensamento: você comenta para mim que descobriu que o melhor algoritmo que você conseguiu para resolver um determinado problema foi o KNN (KNeighborsClassifier ou KNeighborsRegressor, ambos do scikit-learn). Você também comenta que não mexeu em nada nas configurações do scikit-learn.

Agora, te pergunto o seguinte: “Será que seria melhor com quatro vizinhos em vez de cinco? Ou seis vizinhos? Ou dez vizinhos? Tentou mudar o cálculo dos pesos de um valor uniforme para a distância inversa? Ou uma distância de Manhattan em vez de uma distância euclidiana?”.

E a sua resposta poderia ser: “Quê?”.

Se entrarmos na documentação do `KNeighborsClassifier` (ou qualquer outra função) você notará uma nomenclatura parecida com esta:

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski',
metric_params=None, n_jobs=None, **kwargs)
```

Como interpretamos isso? Ora, existem alguns parâmetros os quais já possuem valores-padrão e que poderemos alterar **se quisermos**. Aqui, temos o seguinte: `n_neighbors`, `weights`, `algorithm`, `leaf_size`, `p`, `metric`, `metric_params` e `n_jobs`. Observe que esses são os parâmetros de entrada dessas funções (lembra dos parâmetros de entrada em funções quando tivemos o primeiro contato com python?). Se **não informarmos** nada para esses parâmetros, serão esses os valores adotados (5 para `n_neighbors`, **uniform** para `weights`, e assim sucessivamente). Em ML nós chamamos os parâmetros que afetam o treinamento do modelo de **hiperparâmetros**.

Olhando a própria página da documentação você verá o significado de cada um desses hiperparâmetros, e eles variam de técnica para técnica e de biblioteca para biblioteca. Logo, é importante ler a documentação para ter uma breve noção sobre o significado desses hiperparâmetros.

Dito isso, voltemos à pergunta anterior: como testar diferentes combinações de hiperparâmetros para um determinado problema? O que seria melhor? `n_neighbors = 5` ou `6`? Ou uma mudança no `leaf_size`? Ou ambos, ao mesmo tempo?

Testar na mão não é muito amigável, mas existem formas de testarmos automaticamente diferentes combinações e escolher a melhor. Essa melhor combinação pode ser então usada em um *pipeline* ou de qualquer outra forma. Os dois métodos mais conhecidos são *grid search* e *randomized search*. Em ambas as formas o funcionamento é o mesmo: listamos todas as combinações que queremos testar para todos os parâmetros. Exemplos:

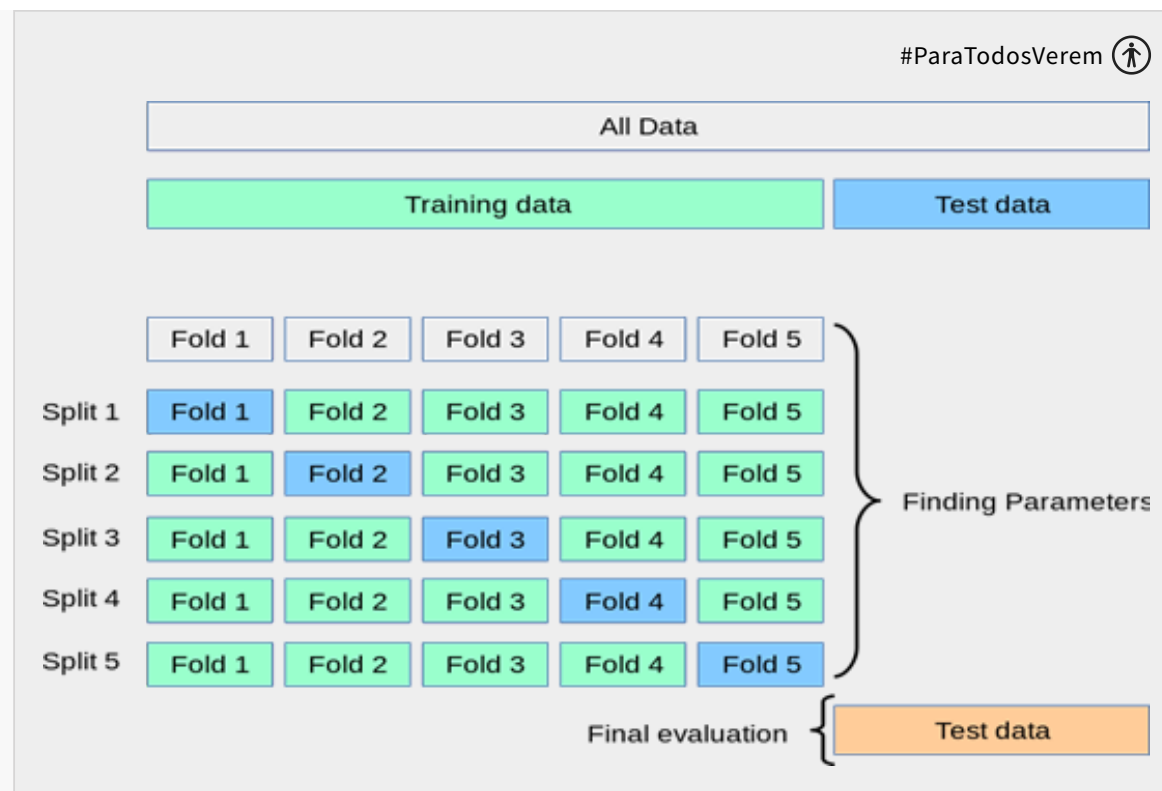
1. Queremos testar o modelo com `n_neighbors` igual a 4, 5, 6 ou 7. São 4 combinações diferentes (4 valores diferentes para um mesmo parâmetro).

2. Queremos testar o modelo com `n_neighbors` igual a 4, 5, 6 ou 7; e `leaf_size` igual a 20, 30 ou 40. São 12 combinações diferentes (`n_neighbors=4` e `leaf_size=20`; `n_neighbors=4` e `leaf_size=30`, `n_neighbors=4` e `leaf_size=40`; `n_neighbors=5` e `leaf_size=20`; e assim sucessivamente).
3. Queremos testar o modelo com `n_neighbors` igual a 4, 5, 6 ou 7; `leaf_size` igual a 20, 30 ou 40; e `weights` igual a “uniform” ou “distance”. São 24 combinações diferentes (4 valores do `n_neighbors` * 3 valores do `leaf_size` * 2 valores do `weights`).

A diferença entre o *grid search* e o *randomized search* é a forma na qual ambos atacam o problema. O *grid search* é o que chamamos de **busca exaustiva** porque ele testa todas as combinações. Isso pode ser útil quando queremos garantir que testamos todas as alternativas possíveis e garantir que pegamos a melhor opção. O problema pode ser no tempo: testar todas as combinações pode demorar muito tempo. O *randomized search* seleciona aleatoriamente somente algumas dessas combinações. Por um lado, ele pode ser mais rápido (já que não testa todas as combinações possíveis). Por outro, ele não necessariamente escolheria a melhor alternativa (pelo mesmo motivo).

No scikit-learn, temos ambas as funções implementadas pelo `GridSearchCV` e `RandomizedSearchCV`. O **CV** de ambas as funções significa *cross-validation* ou validação cruzada. Você se lembra da divisão que fazemos de um *dataset* entre uma base de treinamento e uma de testes com o `train_test_split`? O *cross-validation* pega a base de treinamento e a divide em partes menores (geralmente, 5 blocos menores). Então, ele executa o treinamento utilizando 4 dos blocos menores e valida os resultados no bloco remanescente. O algoritmo faz isso para todos os blocos, garantindo que todas as partes sejam utilizadas no treinamento e na validação. Esta é uma técnica utilizada para reduzir as chances de *overfit* (que comentamos nas unidades anteriores). Observe a imagem abaixo, o resultado do nosso `train_test_split` é o *training data* (em verde) e o *test data* (em azul). Perceba que o algoritmo dividiu a base de treinamento em cinco partes menores (*folds*), e que utilizou essas partes menores em cinco execuções separadas entre si (*splits*). No primeiro *split*, os *folds* 2, 3, 4 e 5 foram usados para treinamento e o algoritmo foi testado com o *fold* 1. No segundo *split*, o *fold* 1 passou a fazer parte do treinamento no lugar do *fold* 2, que agora passou a ser usado no teste. A mesma lógica foi aplicada nos *splits* posteriores.

Tanto o `GridSearchCV` quanto o `RandomizedSearchCV` utilizam essa técnica para cada uma das combinações de parâmetros para tentar chegar ao melhor resultado para você.



Fonte: https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation.

| MÃO NA MASSA

Para demonstrar como a criação de pipelines e a otimização de hiperparâmetros funcionam na prática criei para você um *notebook* (que surpresa!) no Jupyter com o nome **Semana7_PipelineHiperparametros**. Usamos um *dataset* já empregado anteriormente. É possível vermos a aplicação de diferentes configurações de *pipelines* para um mesmo *dataset* e, assim, conseguimos comparar os resultados.

Além disso, também temos uma aplicação do RandomizedSearchCV e outra com o GridSearchCV. Perceba que o GridSearchCV é mais lento, mas encontra uma solução que é melhor. Sugiro que tente refazer os mesmos testes e de criar pipelines diferentes com outros algoritmos que você já tenha criado no passado. É mais simples do que você pensa.

Também lhe convido para assistir ao vídeo **Abrindo o encanamento de casa**. Esse vídeo passa pelo passo –a passo empregado no *notebook* acima e o auxilia a entender melhor a lógica por trás da construção de um *pipeline*.

Abrindo o encanamento de casa



| Abrindo o encanamento de casa

Preocupamo-nos até então com a criação de um modelo, mas não com a organização do passo-a-passo. Neste vídeo, contextualizaremos a organização das técnicas de ML em um formato de *pipeline* do scikit-learn. Vamos lá?

| CONCLUSÃO

Nesta unidade nos aprofundamos em dois tópicos: o ajuste de hiperparâmetros para que tenhamos uma melhor *performance* a partir das métricas que vimos na última unidade e, finalmente, a criação de pipelines para uma execução mais organizada (*streamlined*) das técnicas de ML.

Em aplicações práticas, os *pipelines* são então agrupados em um arquivo do tipo *pickle* e utilizados em servidores/containers do Docker. Por outro lado, esse tópico em específico já chega à fronteira da nossa disciplina.



Fonte: Lenz, 2020 (adaptado).

| REFERÊNCIAS BIBLIOGRÁFICAS

FACELI, K. *et al.* **Inteligência artificial**: uma abordagem de aprendizado de máquina. 2 ed. Rio de Janeiro: LTC, 2021.

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

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

