Fundamentos de Machine Learning com Python

Machine Learning é uma subárea da Inteligência Artificial que foca no desenvolvimento de sistemas capazes de aprender e melhorar com a experiência sem serem explicitamente programados.

O processo de aprendizado em Machine Learning inicia com a alimentação de dados para os modelos de aprendizado. Esses dados podem ser imagens, textos, registros de áudio

Com base na identificação desses padrões, o modelo é capaz de fazer previsões ou tomar decisões quando confrontado com novos conjuntos de dados.

As abordagens para Aprendizado de Máquina (Machine Learning) podem ser categorizadas em três principais métodos: aprendizado supervisionado, aprendizado não supervisionado e aprendizado por reforço.

Aprendizado Supervisionado
O algoritmo é treinado em um conjunto de dados que já contém as respostas desejadas, chamadas de “labels” ou etiquetas. O objetivo é que, após o treinamento, o modelo seja capaz de prever a etiqueta correta para novos conjuntos de dados não vistos anteriormente. Esse tipo de aprendizado é utilizado em tarefas como classificação (onde a saída é discreta, como identificar se um email é spam ou não) e regressão (onde a saída é contínua, como prever o preço de uma casa).

Aprendizado Não Supervisionado
Diferentemente do aprendizado supervisionado, o aprendizado não supervisionado lida com dados que não possuem etiquetas. O objetivo aqui é explorar a estrutura dos dados para extrair padrões, agrupamentos ou correlações sem a orientação de uma variável de saída específica. Essa abordagem é útil em situações onde se sabe pouco sobre os dados ou quando é difícil ou inviável etiquetar os dados manualmente. Exemplos de uso do aprendizado não supervisionado incluem a

segmentação de clientes em marketing, detecção de anomalias e redução de dimensionalidade.

Aprendizado por Reforço
Abordagem dinâmica em que o modelo, ou agente, aprende a tomar decisões através de tentativa e erro, interagindo com um ambiente. Em cada ação, o agente recebe uma recompensa ou penalidade, com o objetivo de maximizar as recompensas ao longo do tempo. Esse método é particularmente útil em situações que requerem uma sequência de decisões, como jogos ou navegação de robôs.

Diferenças
Enquanto o aprendizado supervisionado requer um grande volume de dados etiquetados, o que pode ser um desafio em si, o aprendizado não supervisionado e por reforço oferecem alternativas quando tais dados não estão disponíveis ou são difíceis de obter.

Machine Learning Aprendizado Supervisionado vs Aprendizado não Supervisionado

Aprendizado Supervisionado

Usa **conjuntos de dados rotulados durante o treinamento**, permitindo que algoritmos classifiquem dados com precisão ou prevejam resultados, se divide em tarefas de classificação e regressão ao analisar dados:

Problemas de classificação: envolvem algoritmos que atribuem dados de teste a categorias específicas com precisão, como separar maçãs de laranjas. Esse enfoque é útil em filtros de spam. **Algoritmos** comuns usados incluem **classificadores lineares, máquinas de vetor de suporte, árvores de decisão e florestas aleatórias.**

Tarefas de regressão: compreendem as relações entre variáveis independentes e uma variável dependente. Modelos preveem valores

numéricos baseados em pontos de dados, tornando-os ideais para prever projeções de receita de vendas para empresas. **Algoritmos** de **regressão populares incluem regressão linear, regressão logística e regressão polinomial.**

Aprendizado Não Supervisionado

O aprendizado não supervisionado analisa e agrupa **conjuntos de dados não rotulados de forma independente**, revelando padrões ocultos sem intervenção humana. Os modelos de aprendizado não supervisionado realizam tarefas de agrupamento, associação e redução de dimensionalidade:

Agrupamento: agrupa dados não rotulados de acordo com a semelhança ou diferença. Algoritmos de agrupamento como K-means atribuem pontos de dados semelhantes a grupos, determinados pelos valores K que representam tamanho e granularidade. As aplicações variam de segmentação de mercado a compressão de imagens.

Associação: descobre conexões entre variáveis dentro de um conjunto de dados usando regras.

Redução de dimensionalidade: reduz conjuntos de dados com muitos recursos para tamanhos gerenciáveis enquanto preserva a integridade.

Principais Diferenças Entre Aprendizado Supervisionado e Não Supervisionado

No aprendizado supervisionado, são usados dados de entrada e saída rotulados, enquanto modelos de aprendizado não supervisionado não requerem rótulos. Modelos de aprendizado supervisionado tendem a ser mais precisos porque aprendem de conjuntos de dados rotulados, mas exigem intervenção humana inicial para rotulagem adequada. Modelos de aprendizado não supervisionado descobrem estruturas inerentes aos dados

de forma independente, porém necessitam de validação para interpretação das variáveis de saída.

Escolhendo Entre Aprendizado Supervisionado e Não Supervisionado

Considere se seus dados de entrada são rotulados ou não, se especialistas podem apoiar rotulagem adicional, se há problemas recorrentes a serem resolvidos ou se é necessário prever problemas desconhecidos. Reveja as opções de algoritmos em relação aos requisitos de dimensionalidade, capacidades de manuseio de volume de dados e adequação estrutural.

Aprendizado por Reforço

Esta técnica não depende de dados rotulados ou não rotulados. Em vez disso, baseia-se na ideia de recompensa e penalidade, um agente aprende a tomar decisões por meio da interação com um ambiente. Cada ação tomada pelo agente resulta em uma recompensa ou uma penalidade, e o objetivo é **maximizar as recompensas** ao longo do tempo. Esse método é amplamente utilizado em sistemas que requerem uma sequência de decisões, como jogos, navegação de robôs e automação de veículos autônomos.

- **O Agente**: o modelo ou algoritmo que toma decisões.
- **O Ambiente**: o mundo com o qual o agente interage e onde ele realiza suas ações.
- **A Recompensa**: o feedback que o agente recebe após cada ação

Oferece uma opção dinâmica e adaptativa para problemas que envolvem decisões sequenciais e interações com ambientes que estão em constante mudança.

Desvendando o Ciclo de Vida de um Projeto de Machine Learning

1. Definição do Problema

Essa fase é crítica porque orienta a direção estratégica do projeto, definindo o escopo e os objetivos que o modelo de ML deve alcançar.

Entender o Contexto de Negócio: Esta etapa começa com uma imersão profunda nos objetivos estratégicos da empresa ou do setor. É crucial alinhar o problema de ML com as metas do negócio e os resultados desejados. A equipe deve fazer perguntas como: “Quais são os desafios de negócios que estamos tentando superar?” ou “Que processo ou resultado específico queremos melhorar com ML?”.

Formulação do Problema de ML: Após compreender as metas do negócio, o próximo passo é traduzir esses objetivos em um problema de ML bem definido. A problemática deve ser formulada de tal forma que seja clara, mensurável e viável.

Estabelecendo Metas e Métricas Claras: A definição do problema deve ser acompanhada pela determinação de métricas específicas que serão utilizadas para avaliar o sucesso do projeto.

Identificação dos Stakeholders: Nesta fase, é importante identificar todas as partes interessadas – internas e externas – que serão afetadas pelo projeto de ML ou que terão influência sobre ele.

Viabilidade Técnica e de Dados: Antes de prosseguir, a equipe deve avaliar se o problema definido é tecnicamente viável com os dados disponíveis ou se é possível adquirir os dados necessários. É crucial verificar se os dados

existentes são suficientemente informativos e se existe a infraestrutura técnica e a expertise necessária para desenvolver a solução de ML.

2. Coleta de Dados

Aqui, a atenção meticulosa é dada a reunir informações que serão a base para o treinamento do algoritmo.

Identificação das Fontes de Dados: A busca pelos dados corretos começa com a identificação das fontes apropriadas. Essas fontes podem variar de bancos de dados internos e arquivos de log, a conjuntos de dados públicos ou dados adquiridos de terceiros.

Avaliação da Qualidade dos Dados: Nem todos os dados são criados iguais. A equipe precisa avaliar se os dados coletados são de alta qualidade, o que envolve checar a precisão, a completude e a consistência das informações.

Volume e Variedade: O volume de dados necessário pode variar de acordo com o problema específico e a complexidade do modelo. Modelos mais sofisticados normalmente exigem quantidades maiores de dados.

Aspectos Legais e Éticos: Ao coletar dados, especialmente dados sensíveis ou pessoais, é fundamental considerar aspectos legais, como conformidade com a General Data Protection Regulation (GDPR) ou outras regulamentações de privacidade.

Coleta e Agregação: Uma vez identificadas as fontes de dados e avaliada sua qualidade, começa o processo de coleta e agregação. Ferramentas e sistemas de ETL (Extract, Transform, Load) são comumente utilizados para extrair dados das fontes, transformá-los conforme necessário e carregá-los em um repositório centralizado onde eles podem ser acessados e utilizados pelos modelos de ML.

Pré-Processamento Inicial: Embora o pré-processamento em profundidade ocorra na próxima fase do ciclo de vida, é importante realizar uma limpeza inicial dos dados durante a coleta. Isso pode incluir a remoção de duplicatas, o tratamento de valores ausentes e a identificação de outliers.

3. Preparação e Análise dos Dados

Aqui, os dados coletados são transformados e refinados para garantir que estão em um formato ideal para extração de padrões relevantes pelo algoritmo de aprendizado.

Limpeza de Dados: Esta subetapa envolve a correção ou remoção de dados incorretos, incompletos, duplicados ou irrelevantes.

Transformação dos Dados: Os dados podem precisar ser transformados para se ajustarem melhor aos requisitos do algoritmo. Isso inclui a normalização ou padronização para que os dados estejam na mesma escala, a conversão de variáveis categóricas em numéricas, ou a engenharia de features, que é o processo de criar novos atributos preditivos a partir dos dados existentes.

Análise Exploratória de Dados (EDA): A EDA é um componente crucial desta fase e envolve a exploração visual e quantitativa para identificar padrões, detectar outliers e testar hipóteses.

Seleção de Características: Nem todas as features dos dados são igualmente importantes para o modelo. A seleção de características é o processo de identificar as variáveis mais relevantes para o problema. Isso não só melhora o desempenho do modelo, mas também reduz a complexidade computacional e o risco de overfitting.

Divisão dos Dados: Geralmente, os dados são divididos em conjuntos de treino e teste (e às vezes um conjunto de validação). Essa prática é essencial para avaliar a capacidade do modelo de generalizar para novos dados.

Documentação: A documentação rigorosa durante a preparação e análise dos dados é fundamental. Isso inclui registrar as decisões tomadas, as técnicas utilizadas e as descobertas feitas.

4. Treinamento do Modelo

Esta fase é repleta de complexidades técnicas e requer uma estratégia metodológica para assegurar que o modelo final possa fazer previsões precisas.

Escolha do Algoritmo Apropriado: O ponto de partida do treinamento é selecionar um algoritmo adequado para o problema em mãos. O algoritmo escolhido depende da natureza dos dados, do tipo de problema (classificação, regressão, agrupamento, etc.) e dos objetivos específicos estabelecidos durante a definição do problema. Modelos comuns incluem regressão linear, árvores de decisão, redes neurais e máquinas de vetor de suporte

Configuração dos Hiperparâmetros: Antes de iniciar o treinamento, é necessário configurar os hiperparâmetros do modelo. Hiperparâmetros são configurações que governam o processo de aprendizado e podem ter um impacto significativo no desempenho do modelo.

Treinamento e Validação Cruzada: Com os dados preparados e os hiperparâmetros definidos, o modelo é treinado utilizando os conjuntos de treino. Durante este processo, técnicas como a validação cruzada são frequentemente empregadas para avaliar como o modelo generalizará para um conjunto de dados independente.

Minimização da Função de Perda: O objetivo do treinamento é minimizar a função de perda, uma métrica que quantifica a diferença entre as previsões do modelo e os verdadeiros resultados.

Evitar Sobreajuste: Um dos desafios cruciais durante o treinamento é evitar o sobreajuste, que ocorre quando o modelo aprende padrões específicos para o conjunto de treino, mas não generaliza bem para novos dados.

Monitoramento do Processo de Treinamento: Ao longo do treinamento, é importante monitorar o desempenho do modelo. Isso normalmente envolve acompanhar a função de perda e outras métricas de desempenho, como acurácia, ao longo do tempo.

5. Avaliação do Modelo

É crucial para entender a eficácia do modelo de Machine Learning após o treinamento. Ela serve como um indicativo de como o modelo irá se comportar na prática, quando exposto a dados que nunca viu.

Utilização de Conjuntos de Teste: Para avaliar a performance, utilizamos o conjunto de teste que foi mantido separado durante a etapa de treinamento do modelo.

Métricas de Avaliação: Dependendo do tipo de problema de ML (por exemplo, classificação ou regressão), diferentes métricas são utilizadas para avaliar o modelo. Para problemas de classificação, métricas como precisão, recall e a pontuação F1 são comuns, enquanto que para regressão, métricas como o erro quadrático médio (MSE) ou o coeficiente de determinação (R^2) são utilizados.

Análise de Erros: Um componente essencial da avaliação é a análise dos erros cometidos pelo modelo. Ao examinar os casos em que o modelo errou, podemos obter insights sobre possíveis melhorias tanto nos dados quanto no algoritmo utilizado.

Curvas de Aprendizado: As curvas de aprendizado são gráficos que mostram a evolução do desempenho do modelo em função da quantidade de dados de treinamento.

Matriz de Confusão: Em classificação, a matriz de confusão é uma ferramenta poderosa que fornece uma visão detalhada do desempenho do modelo.

Validação Cruzada Robusta: Diferente da validação cruzada usada durante o treinamento, que visava ajustar os parâmetros do modelo, aqui a validação cruzada é usada para confirmar a robustez do desempenho do modelo.

Feedback de Stakeholders: Eles podem oferecer uma perspectiva diferente sobre o desempenho do modelo, especialmente em relação a como ele atende aos objetivos do negócio definidos na primeira etapa.

6. Ajuste Fino e Otimização

Aqui, iteramos sobre o modelo com o intuito de melhorar seu desempenho, fazendo ajustes baseados nas informações coletadas durante a avaliação.

Tuning de Hiperparâmetros: Um dos primeiros passos para otimizar o modelo é o tuning, ou ajuste fino, dos hiperparâmetros. Esta é uma etapa delicada, pois envolve encontrar o equilíbrio correto entre o poder de generalização e a capacidade de capturar padrões nos dados. Técnicas como pesquisa em grade (grid search), pesquisa aleatória (random search) ou métodos Bayesianos são frequentemente usadas para automatizar e otimizar este processo.

Reengenharia de Características: Isso pode envolver a combinação de características existentes, a transformação de variáveis, ou a eliminação de características que não estão contribuindo para o desempenho do modelo.

Ensemble Learning: Isso pode aumentar a precisão e a robustez do sistema de Machine Learning, pois reduz o risco de erros devido a variações nos dados de treino.

Pruning (Poda) de Modelos: A técnica de pruning envolve a remoção seletiva de partes do modelo que têm pouco ou nenhum valor.

Validação em Diferentes Subconjuntos de Dados: O modelo otimizado é frequentemente validado em diferentes subconjuntos de dados. Isso assegura que o modelo mantém uma boa performance independente das variações dos dados.

Análise de Custo-Benefício: Alguns ajustes podem trazer melhorias mínimas no desempenho com custos computacionais ou temporais significativamente mais altos. Uma análise cuidadosa deve ser feita para evitar a otimização excessiva.

Automatização do Ajuste Fino: Ferramentas e plataformas de AutoML têm ganhado popularidade por sua capacidade de automatizar muitos dos processos de ajuste fino e otimização

7. Implantação em Produção

A última milha no desenvolvimento de um modelo de Machine Learning é a **Implantação em Produção**, onde o modelo bem testado e otimizado é finalmente colocado em uso real.

Seleção da Infraestrutura: A decisão sobre onde e como o modelo será implantado é crucial. Opções incluem servidores locais, na nuvem, ou até mesmo edge computing.

Integração com Sistemas Existentes: A implementação deve ser feita de forma que o modelo possa se integrar sem problemas com os sistemas já em uso na empresa. Isso pode envolver o desenvolvimento de APIs, microserviços, ou adaptações em sistemas de banco de dados.

Monitoramento Contínuo: Uma vez implantado, o modelo deve ser monitorado continuamente para garantir que mantenha seu desempenho e para identificar rapidamente qualquer degradação ou falha.

Atualização e Manutenção: A implantação do modelo não é o final da jornada. Modelos podem precisar ser reajustados ou re-treinados ao longo do tempo devido a mudanças nos padrões de dados.

Governança e Compliance: Uma vez implantado, o modelo deve ser monitorado continuamente para garantir que mantenha seu desempenho e para identificar rapidamente qualquer degradação ou falha. Métricas de desempenho são frequentemente usadas, além de logs de sistema e alertas automatizados.

Atualização e Manutenção: Modelos podem precisar ser reajustados ou re-treinados ao longo do tempo devido a mudanças nos padrões de dados.

Governança e Compliance: O cumprimento das regulamentações de dados é um aspecto significativo, especialmente para modelos que lidam com informações sensíveis.

Feedback Loop para Melhorias Contínuas: A implantação bem-sucedida de um modelo oferece a oportunidade para um feedback loop onde os dados coletados na produção podem ser utilizados para refinar ainda mais o modelo e sua performance.

Estratégias de Implantação: Existem diferentes estratégias para a implantação, como a implantação direta (ou “big bang”), implantação paralela, onde o modelo antigo e o novo funcionam simultaneamente, e a implantação em fases (ou “canary release”), onde o modelo é exposto a uma parte dos usuários antes de uma liberação completa.

8. Monitoramento e Manutenção

O monitoramento ajuda a detectar mudanças nos padrões dos dados que podem afetar o desempenho do modelo, enquanto a manutenção regular garante que o modelo continua a performar adequadamente ao longo do tempo.

9. Feedback e Iteração

A fase de **Feedback e Iteração** é vital para refinamentos contínuos e garantia de que o modelo permanece alinhado com as necessidades dos usuários e as dinâmicas do mercado.

Coleta de Feedback: O feedback dos usuários finais do modelo é uma mina de ouro para entender seu desempenho no mundo real. A

Análise do Feedback: O feedback coletado precisa ser sistematicamente analisado para identificar padrões, problemas recorrentes ou oportunidades de aprimoramento.

Iteração Contínua: Com base nesses insights, o modelo pode necessitar de iterações.

Aprendizado com os Dados: À medida que o modelo está em produção, ele gera novos dados que podem ser usados para aprendizado adicional.

Retreinamento do Modelo: Quando são identificadas mudanças significativas nos dados ou no desempenho do modelo, pode-se retreinar o modelo utilizando os dados mais recentes.

A/B Testing e Experimentação: A implementação de testes A/B, onde diferentes versões de um modelo são testadas em paralelo

Gestão do Ciclo de Vida do Modelo: Isso inclui saber quando é hora de aposentar um modelo ou quando investir em uma nova geração de soluções.

Documentação e Comunicação: Documentar cada iteração, os motivos para as mudanças e os resultados obtidos é importante tanto para o processo de aprendizado quanto para a comunicação com as partes interessadas.

NumPy

Conversão de Lista para ndarray: Você pode transformar uma lista Python em um ndarray usando a função `np.array()`
Para verificar o tipo de dados com `print(meu_array.dtype)`

Criação usando Funções do NumPy: O NumPy também oferece várias funções para criar arrays com conteúdo inicial específico. Por exemplo, `np.zeros()` ou `np.ones()` para criar arrays preenchidos com zeros ou uns

Quando você executa o comando `print(meu_array.shape)` no contexto da biblioteca NumPy e recebe a saída `(5,)`, isso indica que `meu_array` é um array unidimensional (1D) com 5 elementos.

- A **primeira parte** da saída, que é o número `5`, representa o número de elementos ao longo da primeira (e única, nesse caso) dimensão do array.
- O uso da **vírgula** seguida de um **parêntese fechado** é a notação do Python para indicar que se trata de uma **tupla** de um único elemento.

A Função linspace(): Outra função poderosa é `np.linspace(start, stop, num)`, que cria arrays com valores igualmente espaçados dentro de um intervalo especificado.

Seleção e Manipulação de Dados

Acessar Elementos Individuais: Para acessar um único elemento, utilize índices que correspondam à sua posição na matriz.

```
elemento_200 = array_2d[0, 1]
print(elemento_200)
```

Slicing: Para acessar um SUBCONJUNTO do array, utilizamos a técnica conhecida como slicing. O slicing pode ser aplicado em arrays 1D, 2D.

```
# Acessar a primeira linha inteira
```

```
primeira_linha = array_2d[0, :]
# Acessar a terceira coluna inteira
terceira_coluna = array_2d[:, 2]
```

antes do : é o início depois dele é o fim

por exemplo

```
teste = array_2d[2:, 1:4]
```

resultado => `[[7 8 9]]`

Quando não especificamos antes do : o python entende que é para começar do início do array, ou seja, indice 0. Quando não especificamos o depois do :, o python entende que é para ir até o fim.

- **array_2d[:, 2]** esse aqui indica que vamos pegar todas as linhas da coluna no índice 2, ou seja, a terceira coluna

Adição de Step ao Slice

O 'step' pode ser especialmente útil em arrays maiores, onde talvez você queira acessar elementos saltando um certo intervalo.

```
# Acessar elementos na segunda linha, saltando de dois em dois
linha_com_step = array_2d[1, ::2]
```

antes do : é o início, o meio dele é o fim, e último valor é o passo. Então:

inicio:fim:passo => é sintaxe do slice + step

Exemplo mais detalhado:

```
slice_step = segundo_array_2d[1, 0:4:2]
```

- **1**: Especificamos que queremos acessar a segunda linha do array.
- **0:4:2** é a expressão de fatiamento onde:
  - **0** é o índice inicial do slice.
  - **4** é o índice final do slice, não inclusivo, então fazemos 4 - 1, assim, indo até o índice 3.

- **2** é o ‘step', que nos diz que queremos selecionar elementos saltando de dois em dois.

Outro exemplo:

```
slice_2d = array_2d[1:3, ::2]
```

O primeiro par de índices `1:3` define quais linhas do array serão incluídas no slice. Neste caso, ele começa no índice 1 e vai até o índice 3.

A segunda parte, `::2`, é aplicada nas colunas. Aqui, o `::` informa que você deseja incluir todas as colunas, mas o `2` adicionado ao final indica que você quer fazer isso em passos de dois.

Explorando Métodos ndarray

métodos `sum()`, `mean()` e `reshape()`

O Método sum(): O método `sum()` é usado para somar todos os elementos de um array ou somar elementos ao longo de um determinado eixo.

```
array_2d = np.array([[1, 2], [3, 4], [5, 6]])

# Somando elementos de cada coluna (axis=0)
soma_colunas = array_2d.sum(axis=0)

# Somando elementos de cada linha (axis=1)
soma_linhas = array_2d.sum(axis=1)
```

axis=0 => é as linhas| **axis=1** => é às colunas

`soma_colunas` resultará em `[9, 12]` porque soma os elementos de cada coluna individualmente (1+3+5 e 2+4+6), `soma_linhas` dará `[3, 7, 11]`, somando os elementos linha por linha.

Método mean(): Já o método `mean()` calcula a média aritmética de um array. Quando não especificado, ele calcula a média de todos os elementos.

O Método reshape(): Com `reshape()`, você pode alterar a estrutura de um array sem alterar seus dados.

```
array_2x5 = array_1d.reshape(2, 5)
```

Passar `-1` em uma das dimensões instrui o NumPy a calcular automaticamente o tamanho dessa dimensão.

```
# Redimensionar para 2 linhas e calcular o número necessário de colunas
array_2d_auto = array_1d.reshape(2, -1)
# Redimensionar para um número desconhecido de linhas e 5 colunas
array_autox5 = array_1d.reshape(-1, 5)
```

No NumPy, `reshape()` pode ser utilizado tanto como **função** quanto como **método** de um objeto array NumPy.

Como função:

```
import numpy as np

# Cria um array com 6 elementos
array_original = np.array([1, 2, 3, 4, 5, 6])

# Usa a função np.reshape para alterar a forma do array
array_reshaped = np.reshape(array_original, (2, 3))
```

Funções Essenciais do NumPy

Concatenação com concatenate(): A função `concatenate()` é utilizada quando você precisa unir dois ou mais arrays NumPy, seja por linhas ou colunas.

No NumPy, `concatenate()` é uma **função** e não um método. Ela é usada para unir dois ou mais arrays ao longo de um eixo especificado.

Exemplo:

```
import numpy as np

# Cria dois arrays
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])

# Usa a função concatenate para unir os arrays
array_unido = np.concatenate((array1, array2))
```

Wrapper Functions no NumPy: Wrapper functions, ou funções de envolvimento, são aquelas que “embrulham” outras funções, possibilitando uma abstração adicional, manipulação de entradas ou saídas, e outros comportamentos úteis.

Funções vs Métodos: A principal **diferença** entre funções e métodos em NumPy é que os métodos são chamados a partir de um objeto ndarray (ou seja, são funções associadas a um objeto específico), enquanto as funções podem operar em uma variedade de inputs. Por exemplo, a função `np.sum()` pode somar elementos de qualquer array-like object, incluindo listas do Python, enquanto o método `.sum()` é específico para objetos ndarray.

```
# Usando uma função do NumPy
resultado_funcao = np.sum(array_2d_1)
```

```
# Usando um método de um objeto ndarray
resultado_metodo = array_2d_1.sum()
```

Pandas

Explorando a Criação de Séries com o Pandas

As séries do Pandas são estruturas que se assemelham a arrays unidimensionais e têm a flexibilidade de funcionar quase como uma coluna isolada de uma planilha. O grande diferencial de uma série é a possibilidade de associar um índice customizado a cada um de seus elementos, oferecendo uma variedade de formas para manipular e acessar os dados. Diferentemente de um índice puramente numérico, estes rótulos podem ser strings, datas, ou qualquer tipo imutável

- **Estrutura Unidimensional:** Uma série mantém seus valores alinhados em uma única dimensão linear.
- **Índices Associados:** Cada entrada em uma série tem um rótulo único que funciona como índice
- **Operações Verticais:** Processos como filtragem e agregação são feitos ao longo do eixo da série

```
import pandas as pd
idades = [25, 30, 35]
serie_idades = pd.Series(idades)
print(idades, end='\n\n')
print(serie_idades, end='\n\n')
# Associando idades a nomes
serie_nome_idades = pd.Series(idades, index=['Alice', 'Bob',
'Charlie'])
print(serie_nome_idades, end='\n\n')
```

Métodos Estatísticos Comuns

O Pandas facilita a obtenção de medidas estatísticas comuns, como média, mediana, desvio padrão

```
# Calculando a média dos valores
media = serie.mean()
# Encontrando a mediana
mediana = serie.median()

# Obtendo o desvio padrão
desvio_padrao = serie.std()
```

Operações de Comparação e Booleanas

```
# Verificando valores acima da média
valores_acima_media = serie > serie.mean()
print(valores_acima_media,'\n\n')
# Selecionando apenas valores acima de um determinado threshold
valores_selecionados = serie[serie > 50]
print(valores_selecionados, '\n\n')
```

A função `isnull()` é usada para criar uma série booleana onde cada posição reflete se o valor no dado correspondente da série original está faltando ou não. Isso é feito da seguinte maneira:

```
nulos = temperaturas.isnull()
print("\nValores Faltantes (True indica um valor faltante):")
print(nulos)
```

temperaturas.fillna(temperaturas.mean()) => Para preencher valores faltando

`temperaturas.dropna()=>` tirar os valores nulos usar o dropna

Para operações mais complexas ou personalizadas, o método `.apply()` é extremamente útil. Ele permite que você aplique uma função a cada item na série

Criando DataFrame no Pandas

Pense nele como uma tabela de dados multidimensional onde cada coluna pode conter tipos diferentes de dados, ou seja, ele é heterogêneo. Os DataFrames vêm com os dois eixos rotulados – as linhas (índice) e as colunas.

Para criação de DataFrame simples => você pode simplesmente passar uma lista de listas ou um array 2D.

Exemplo:

```
import pandas as pd

# Criando um DataFrame simples
dataframe = pd.DataFrame(data=[[1, 'John'], [2, 'Jane']],
columns=['ID', 'Nome'])
```

DataFrames a Partir de Dicionários de Listas

Criar DataFrames é através de um dicionário de listas. Cada chave do dicionário se torna uma coluna no DataFrame, e a lista associada contém os dados para essa coluna.

```
# DataFrame a partir de um dicionário
dados = {
    'ID': [1, 2, 3],
    'Nome': ['John', 'Jane', 'Jim'],
    'Idade': [22, 33, 44]
}
df_dicionario = pd.DataFrame(dados)
```

DataFrames com Índices Personalizados

Especificar os índices das linhas => útil quando os índices têm significado próprio:

```
# DataFrame a partir de um dicionário
dados = {
    'ID': [1, 2, 3],
    'Nome': ['John', 'Jane', 'Jim'],
    'Idade': [22, 33, 44]
}
df_indices = pd.DataFrame(data=dados, index=['linha1',
'linha2','linha3'])
print(df_indices)
```

Adicionando Colunas a um DataFrame Existente

```
# Adicionando uma nova coluna ao DataFrame
df_indices['Salário'] = [50000, 60000, 70000]
print(df_indices)
```

Criando DataFrames Complexos

```
import pandas as pd
import numpy as np
# DataFrame complexo com vários tipos de dados
df_complexo = pd.DataFrame({
    'A': pd.Series([1, 2, 3], index=['primeiro', 'segundo',
'terceiro']),
    'B': np.linspace(0, np.pi, 3),
    'C': pd.date_range(start='20210101', periods=3, freq='D')
})
print(df_complexo)
```

O DataFrame `df_complexo` é criado utilizando o construtor `pd.DataFrame()`, que organiza os dados em uma estrutura tabular de linhas e colunas.

Coluna 'A' => `pd.Series([1, 2, 3], index=['primeiro', 'segundo', 'terceiro'])`: Esta série representa a coluna 'A' do DataFrame. Uma Series do Pandas é um array unidimensional capaz de armazenar qualquer tipo de dado

Coluna 'B' => `np.linspace(0, np.pi, 3)`: Esta expressão utiliza a função `linspace` do NumPy para gerar três valores igualmente espaçados entre 0 e π (aproximadamente 3.14159).

Coluna 'C' => `pd.date_range(start='20210101', periods=3, freq='D')`: Esta expressão cria uma sequência de datas usando a função `date_range` do Pandas. O argumento `start='20210101'` define a data de início da sequência como 1º de janeiro de 2021. O `periods=3` especifica que a sequência deve conter três datas. O `freq='D'` indica que a frequência entre as datas é diária.

Usando read_csv() na Prática

A maneira mais básica de carregar um arquivo CSV com o Pandas é passando o caminho do arquivo para a função `read_csv()`:

```
import pandas as pd
 # Lendo um arquivo CSV em um DataFrame
df = pd.read_csv('caminho/para/seu/arquivo.csv')
```

Lidando com Cabeçalhos de Colunas

```
# Lendo um CSV com cabeçalho
df_com_cabecalho = pd.read_csv('caminho/para/seu/arquivo.csv', header=0)
```

Se o arquivo CSV não tiver uma linha de cabeçalho, você pode especificar `header=None` e fornecer os nomes das colunas usando o parâmetro `names`:

```
# Lendo um CSV sem cabeçalho
df_sem_cabecalho = pd.read_csv('caminho/para/seu/arquivo.csv', header=None, names=['Coluna1', 'Coluna2', 'Coluna3'])
```

Especificando Tipos de Dados

```
# Especificando tipos de dados de colunas
df_tipos = pd.read_csv('caminho/para/seu/arquivo.csv',
dtype={'Coluna1': int, 'Coluna2': float})
```

Tratando Dados Faltantes

```
# Tratando dados faltantes com o valor NaN padrão do Pandas
df_dados_faltantes = pd.read_csv('caminho/para/seu/arquivo.csv',
na_values=['NA', ''])
```

Manipulando Grandes Conjuntos de Dados

Para grandes conjuntos de dados, pode ser útil ler o arquivo em pedaços. O Pandas permite que você faça isso com o parâmetro `chunksize`:

```
# Lendo um CSV em pedaços
tamanho_do_chunk = 500
chunks = pd.read_csv('caminho/para/seu/arquivo.csv',
chunksize=tamanho_do_chunk)
for chunk in chunks:
    # faça algo com cada pedaço, como processamento ou análise
```

Parâmetros Adicionais

A função `read_csv()` vem com vários outros parâmetros que permitem personalizar como os dados são lidos, incluindo:

`usecols`- para selecionar quais colunas carregar.

`skiprows-` para pular um número específico de linhas no início do arquivo.

`nrows-` para carregar um número específico de linhas.

`parse_dates`- para analisar colunas como datas.

– `head()` para visualizar as primeiras linhas ou `describe()` para obter uma descrição estatística dos dados:

```
# Análise inicial
print(df.head())
print(df.describe())
```

Exportando Dados de um DataFrame Exportar seus dados é tão simples quanto carregá-los. Para salvar seu DataFrame em um arquivo CSV, você usará:

```
df.to_csv('caminho/para/seu/novo_arquivo.csv')
```

Carregando o Arquivo CSV

```
import pandas as pd
classData = pd.read_csv('pandas-sample-data.csv')
print(classData)
```

Análise Preliminar dos Dados

```
# Visualizando as primeiras 5 linhas
print(classData.head())

# Visualizando as últimas 5 linhas
print(classData.tail())
```

Tipos de Dados e Informações do DataFrame

```
# Verificando os tipos de dados
print(classData.dtypes)

# Informações sobre o DataFrame
print(classData.info())
```

Sumário Estatístico

```
# Resumo estatístico das colunas numéricas
print(classData.describe())

# Resumo estatístico das colunas categóricas
print(classData.describe(include=[object]))
```

Contagem de Valores Únicos

```
# Contagem de instrutores únicos
print(classData['Instrutor'].value_counts())

# Contagem de AE únicos
print(classData['AE'].value_counts())
```

Seleção e Filtragem

```
# Seleção de colunas específicas
df_avaliacao_inscritos = classData[['Avaliacao', 'Inscritos']]
print(df_avaliacao_inscritos, end='\n\n')

# Filtragem de linhas baseada em uma condição
df_filtrado = classData[classData['Avaliacao'] >= 4.5]
print(df_filtrado, end='\n\n')
```

Ordenando Dados

```
# Ordenando pela avaliação de forma descendente
df_ordenado = classData.sort_values(by='Avaliacao', ascending=False)
print(df_ordenado)
```

Agrupando e Agregando Dados

```
# Agrupando por instrutor e obtendo a média de inscritos
df_grupo_inscritos = classData.groupby('Instrutor')['Inscritos'].mean()
print(df_grupo_inscritos)
```

Trabalhando com Dados Faltantes

```
# Identificando dados faltantes
print(classData.isnull().sum())

# Preenchendo dados faltantes
df_preenchido = classData.fillna({'Avaliacao':
classData['Avaliacao'].mean()})
print(df_preenchido)
```

Plotando Dados

```
# Plotando a distribuição das avaliações
classData['Avaliacao'].plot(kind='hist')
```

Correlações

```
import pandas as pd
df = pd.read_csv('pandas-sample-data.csv')
# Correlação entre número de inscritos e avaliação do curso
print(df[['Inscritos', 'Avaliacao']].corr())
```

Alterando Lables de Linhas e Colunas no DataFrame

O Pandas facilita esse processo através do método `rename`, o qual nos permite modificar os rótulos das colunas e linhas para atender a diversas necessidades de análise e apresentação.

Mudando os Rótulos das Colunas

```
classData.rename(columns={
    'ID_Curso': 'Código_Curso',
    'Instrutor': 'Nome_Instrutor',
    'AE': 'Assistente_Ensino',
    'Inscritos': 'Total_Inscritos',
    'Avaliacao': 'Média_Avaliação'
}, inplace=True)
print(classData)
```

`inplace=True` **para garantir que a mudança afete o DataFrame original.**

Padronização e Limpeza dos Nomes de Colunas

Convertendo todos os nomes de colunas para letras minúsculas e substituir os espaços por underscores:

```
classData.rename(columns=lambda x: x.lower().replace(" ", "_"), inplace=True)

print(classData)
```

Renomeando as Linhas

```
classData.rename(index=lambda i: 'Linha_' + str(i), inplace=True)
print(classData)
```

Renomeando as colunas imediatamente após a leitura do arquivo CSV

```
classData = pd.read_csv('pandas-sample-data.csv')
classData.columns = ['Código_Curso', 'Nome_Instrutor',
'Assistente_Ensino', 'Total_Inscritos', 'Média_Avaliação']
print(classData)
```

Selecionar Colunas

```
import pandas as pd
classData = pd.read_csv('pandas-sample-data.csv')
# Selecionando a coluna 'Inscritos' usando colchetes
inscritos = classData['Inscritos']
print(inscritos)

# Selecionando a coluna 'Avaliacao' como um atributo
avaliacao = classData.Avaliacao
print(avaliacao)
```

selecionando múltiplas colunas

```
import pandas as pd
classData = pd.read_csv('pandas-sample-data.csv')
# Selecionando múltiplas colunas
df_selecionado = classData[['Instrutor', 'AE', 'Avaliacao']]
print(df_selecionado)
```

Utilizando Arrays Booleanos para Filtragem de Dados

```
import pandas as pd
classData = pd.read_csv('pandas-sample-data.csv')
# Criando um array booleano para cursos com Avaliacao maior que 4.7
array_booleano = classData['Avaliacao'] > 4.7
print(array_booleano,end='\n\n')

# Selecionando linhas que satisfazem a condição
cursos_top = classData[array_booleano]
print(cursos_top,end='\n\n')
```

Combinando Condições com Arrays Booleanos

Podemos combinar múltiplas condições utilizando operadores lógicos como `&` (e) e `|` (ou):

```
# Selecionando cursos com Avaliacao acima de 4.7 E menos de 50 Inscritos
condicao = (classData['Avaliacao'] > 4.7) & (classData['Inscritos'] < 50)
cursos_selecionados = classData[condicao]
print(cursos_selecionados, end='\n\n')
```

Invertendo Condições com o Operador ~

Também é possível inverter uma condição usando o operador `~`, o que é equivalente a dizer 'não':

```
# Selecionando cursos que NÃO têm Avaliacao de 4.8
cursos_nao_48 = classData[~(classData['Avaliacao'] == 4.8)]
```

Usando Arrays Booleanos com loc

```
import pandas as pd
classData = pd.read_csv('pandas-sample-data.csv')
# Usando 'loc' com array booleano para selecionar linhas e a coluna
'Instrutor'
instrutores_top_cursos = classData.loc[classData['Avaliacao'] > 4.7,
'Instrutor']
print(instrutores_top_cursos)
```

Combinando `loc` e `iloc` para Seleção Específica

O método `loc` permite selecionar com base nos rótulos das linhas e nomes das colunas, enquanto `iloc` trabalha com as posições numéricas (índices) de linhas e colunas.

```
# Selecionando a avaliação e o instrutor do curso MT101
avaliacao_instrutor_mt101 = classData.loc[classData['ID_Curso'] ==
'MT101', ['Instrutor', 'Avaliacao']]
```

Este comando nos dá todas as linhas onde o ID do curso é 'MT101', mas apenas as colunas 'Instrutor' e 'Avaliacao'.

Selecionando Intervalos com `iloc`

```
# Selecionando os primeiros 5 cursos e apenas as colunas de 'Instrutor'
e 'Avaliacao'
primeiros_cursos = classData.iloc[0:5, [1, 4]]
```

Filtrando por Condições Complexas e Selecionando Colunas

```
# Selecionando cursos com avaliação maior que 4.7 e as colunas
'Instrutor' e 'Avaliacao'
cursos_avaliacao_alta = classData.loc[classData['Avaliacao'] > 4.7,
['Instrutor', 'Avaliacao']]
```

Aqui, o resultado conterá apenas as linhas dos cursos com nota acima de 4.7, e das colunas, somente ‘Instrutor' e ‘Avaliacao' serão retornadas.

Selecionando Baseado em Múltiplos Critérios

```
# Selecionando cursos de 'FI' e que tenham mais de 45 inscritos
cursos_fi_45_inscritos =
classData.loc[(classData['ID_Curso'].str.contains('FI')) &
(classData['Inscritos'] > 45)]
```

Atualizando Valores Individuais

```
classData = pd.read_csv('pandas-sample-data.csv')
classData.loc[(classData['ID_Curso'] == 'MT102') &
(classData['Instrutor'] == 'Luiz Costa'), 'Inscritos'] = 59
```

Imputando Valores Faltantes

Nosso DataFrame possui algumas células vazias, representando dados faltantes.

```
import pandas as pd
df = pd.read_csv('pandas-sample-data.csv')
# Imputando a média de inscritos nos cursos onde falta essa informação
media_inscritos = df['Inscritos'].mean()
df['Inscritos'].fillna(value=media_inscritos, inplace=True)
```

Atualizações Baseadas em Condições

```
df.loc[df['Inscritos'] < 50, 'Avaliacao'] *= 1.1
```

Alterando o Tipo de Dados de Uma Coluna

```
import pandas as pd
df = pd.read_csv('pandas-sample-data.csv')
# Convertendo a coluna 'Inscritos' de float para int
df['Inscritos'] = df['Inscritos'].astype(int)
```

Adicionando Colunas Calculadas

```
import pandas as pd
df = pd.read_csv('pandas-sample-data.csv')
df['Relacao_Inscritos_Avaliacao'] = df['Inscritos'] / df['Avaliacao']
```

Removendo Colunas ou Linhas

```
import pandas as pd
df = pd.read_csv('pandas-sample-data.csv')
# Removendo a coluna 'AE' (Assistente de Ensino)
df.drop(columns='AE', inplace=True)
```

Renomeando Colunas em Massa

```
import pandas as pd
df = read_csv('pandas-sample-data.csv')
df.rename(columns={'Instrutor': 'Prof', 'Inscritos': 'Alunos',
'Avaliacao': 'Nota'}, inplace=True)
```

Reordenando Colunas

```
import pandas as pd
df = read_csv('pandas-sample-data.csv')
df = df[['ID_Curso', 'Prof', 'Alunos', 'Nota',
'Relacao_Inscritos_Avaliacao']]
```

Ajustando Índices Após Alterações

```
import pandas as pd
df = read_csv('pandas-sample-data.csv')
df.reset_index(drop=True, inplace=True)
```

Métodos Úteis no Pandas

`describe()`: Resumo Estatístico de Alto Nível

```
resumo_estatistico = df.describe()
```

Com `resumo_estatistico`, você obtém a contagem, média, desvio padrão, mínimo, percentis e máximo de todas as colunas numéricas do DataFrame.

`unique()`: Explorando a Diversidade de Valores

```
valores_unicos = df['Instrutor'].unique()
```

`groupby()`: Análise Agrupada por Categorias

O `groupby()` permite agrupar o DataFrame por uma ou mais colunas e aplicar funções de agregação:

```
media_por_curso = df.groupby('ID_Curso')['Avaliacao'].mean()
```

`merge()`: Unindo DataFrames por Informação em Comum

Imagine que você tem outro DataFrame `professores_df` que contém mais informações sobre os instrutores. Com `merge()`, você pode combinar estes DataFrames facilmente:

```
df_detalhado = df.merge(professores_df, on='Instrutor')
```

`pivot_table()`: Tabelas Dinâmicas para Análise Multidimensional

Às vezes, você precisa reestruturar seus dados para uma análise mais complexa, o `pivot_table()` é essencial para isso:

```
tabela_dinamica = df.pivot_table(values='Inscritos', index='ID_Curso', columns='Instrutor', aggfunc='sum')
```

Matplotlib

Plotagem de um Gráfico de Dispersão (Scatter Plot)

O gráfico de dispersão é útil para visualizar a relação entre duas variáveis:

```
import matplotlib
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [2, 3, 4, 5, 6]
plt.scatter(x, y)
plt.title("Gráfico de Dispersão Simples")
plt.xlabel("Eixo X")
plt.ylabel("Eixo Y")
plt.show()
```

O gráfico de dispersão, ou *scatter plot*, é uma ferramenta fundamental na visualização de dados e é essencialmente utilizado para explorar a relação ou correlação entre duas variáveis numéricas.

```
Plotando o Gráfico de Dispersão
import matplotlib.pyplot as plt
import numpy as np

#Declaração das variáveis com seus dados
horas_estudo = [1, 2, 3, 4, 5, 6, 7, 8]
pontuacao_teste = [50, 55, 60, 65, 70, 75, 80, 85]

# Criação do gráfico de dispersão
plt.scatter(horas_estudo, pontuacao_teste, alpha=0.6, edgecolors='w', s=100)

# Cálculo da linha de tendência
z = np.polyfit(horas_estudo, pontuacao_teste, 1)
p = np.poly1d(z)
plt.plot(horas_estudo, p(horas_estudo), "r--")

# Títulos e rótulos dos eixos
plt.title('Relação entre Horas de Estudo e Pontuação no Teste', fontsize=16)
plt.xlabel('Horas de Estudo', fontsize=12)
plt.ylabel('Pontuação no Teste', fontsize=12)

# Adição de uma grade para melhor leitura dos dados
plt.grid(True)

# Adição de uma grade para melhor leitura dos dados
plt.grid(True)
```

1. **Cálculo da linha de tendência:**
    - `z = np.polyfit(horas_estudo, pontuacao_teste, 1)`: A função `np.polyfit()` da biblioteca NumPy é utilizada para ajustar um polinômio de grau especificado (neste caso, 1, indicando um polinômio linear) aos dados fornecidos. Ela retorna os coeficientes do polinômio que melhor se ajusta aos dados, no sentido dos mínimos quadrados. Para um polinômio de grau 1, `z` conterá dois valores: a inclinação da linha (`slope`) e o intercepto y (`intercept`).
    - `p = np.poly1d(z)`: Após obter os coeficientes, `np.poly1d()` é utilizado para criar um objeto polinomial `p` a partir dos coeficientes `z`. Este objeto é uma função que pode ser chamada com um valor de `x` (neste caso, `horas_estudo`), retornando o valor de `y` correspondente na linha de tendência calculada.
2. Desenho da linha de tendência no gráfico:
    - `plt.plot(horas_estudo, p(horas_estudo), "r--")`: Esta linha utiliza a função `plt.plot()` para desenhar a linha de tendência no gráfico. `horas_estudo` é usado como o eixo x, e `p(horas_estudo)` calcula os valores correspondentes no eixo y usando a função polinomial `p` criada anteriormente. `"r--"` define o estilo da linha de tendência, onde `"r"` significa vermelho e `"--"` indica que a linha será tracejada.

Criação de um Gráfico de Barras (Bar Chart)

```
import matplotlib
import matplotlib.pyplot as plt

categorias = ['A', 'B', 'C', 'D']
valores = [3, 7, 2, 5]
plt.bar(categorias, valores)
plt.title("Gráfico de Barras Simples")
plt.show()
```

Elaboração de um Histograma (Histogram)

Um histograma é um tipo de gráfico que permite visualizar a distribuição de frequências de um conjunto de dados.

```
import matplotlib
import matplotlib.pyplot as plt
idades = [
    23, 29, 22, 35, 42, 39, 56, 48, 33, 36, 26, 24, 28, 30, 50, 45, 41, 31, 57, 55,
    52, 47, 63, 59, 60, 38, 37, 49, 44, 43, 53, 27, 25, 34, 32, 40, 46, 58, 61, 54,
    51, 64, 62, 65, 66, 67, 29, 21, 24, 28, 26, 30, 22, 35, 31, 48, 43, 38, 39, 36
]  # Idades dos funcionários
# Criação do histograma
plt.hist(idades, bins=[20, 30, 40, 50, 60, 70], color='dodgerblue', edgecolor='black')

# Título e rótulos dos eixos
plt.title('Distribuição de Idades no Local de Trabalho', fontsize=16)
plt.xlabel('Idade', fontsize=12)
plt.ylabel('Quantidade de Funcionários', fontsize=12)

# Adição de marcações no eixo X para as faixas etárias
plt.xticks([25, 35, 45, 55, 65])

# Exibição do histograma
plt.show()
```

Neste código Python a seguir, utilizamos a biblioteca `numpy` para gerar 24 dados que simulam os tempos de resposta do website em um período de 24 horas. Substituímos aleatoriamente 5 desses dados para representar tempos de resposta superiores a 1 segundo

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
# Gerar 24 dados aleatórios para simular os tempos de resposta médios a cada hora do dia
tempos_resposta = np.random.uniform(low=0.5, high=1.0, size=24)
# Substituir aleatoriamente alguns valores por tempos de resposta > 1 segundo
tempos_resposta[np.random.choice(np.arange(24), size=5, replace=False)] =
np.random.uniform(low=1.0, high=1.5, size=5)
# Criar o histograma
plt.hist(tempos_resposta, bins=10, color='skyblue', edgecolor='black')
# Adicionar títulos e rótulos
plt.title('Distribuição dos Tempos de Resposta do Website')
```

```
plt.xlabel('Tempo de Resposta (segundos)')
plt.ylabel('Frequência')
# Desenhar uma linha vertical em x=1 para facilitar a visualização dos tempos > 1 segundo
plt.axvline(x=1, color='red', linestyle='--', label='1 segundo')
# Adicionar legenda
plt.legend()
# Exibir o gráfico
plt.show()
```

A linha vertical vermelha em `x=1` é um marcador visual significativo; ela divide o gráfico entre tempos de resposta que são considerados aceitáveis (menos de 1 segundo) e aqueles que podem comprometer a experiência do usuário (mais de 1 segundo).

Desenho de um Gráfico de Linhas (Line Graph)

Definição dos Dados das Coordenadas X e Y:

```
x = range(10)
y = [x**2 for x in range(10)]
plt.plot(x, y)
```

Em machine learning, gráficos de linhas são frequentemente usados para avaliar o desempenho dos algoritmos ao longo de várias iterações de treinamento, e em análise de dados, para visualizar séries temporais e outras tendências contínuas nos dados.

Integração com Pandas para Visualização de Dados

```
import pandas as pd
# Exemplo com dados financeiros
dados_financeiros = {'Ano': [2015, 2016, 2017, 2018], 'Lucro': [15, 18, 20, 22]}
df = pd.DataFrame(dados_financeiros)
df.plot(x='Ano', y='Lucro', kind='line')
plt.show()
# Exemplo com dados climáticos
dados_climaticos = pd.DataFrame({'Temperatura': [22, 24, 19, 24]})
dados_climaticos.hist(bins=3)
plt.show()
```

-O método `plot()` do DataFrame é utilizado para gerar o gráfico de linhas. Especificamos que a coluna ‘Ano' deve ser usada para o eixo X e ‘Lucro' para o eixo Y. O argumento `kind='line'` informa que queremos um gráfico de linhas.
-Para a visualização da distribuição de temperaturas, utilizamos o método `hist()`, que cria um histograma para a coluna ‘Temperatura'. O parâmetro `bins=3` especifica que queremos que os dados sejam agrupados em 3 intervalos.

A Interface Orientada a Objetos do Matplotlib: Mais Controle de Seus Gráficos

```
import matplotlib
import matplotlib.pyplot as plt

# Dados para os gráficos
a = [1, 2, 3, 4]
b = [7, 3, 1, 4]
c = [4, 2, 3, 5]

# Criação de uma figura e dois subplots (axes) lado a lado
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

# Utilização do primeiro Axes (ax1) para um gráfico de dispersão
ax1.scatter(a, b, color='blue')
ax1.set_title('Gráfico de Dispersão')
ax1.set_xlabel('Eixo X')
ax1.set_ylabel('Eixo Y')

# Utilização do segundo Axes (ax2) para um gráfico de linha
ax2.plot(a, c, color='red')
ax2.set_title('Gráfico de Linha')
ax2.set_xlabel('Eixo X')
ax2.set_ylabel('Eixo Y')

# Ajuste do layout para evitar sobreposições indesejadas
fig.tight_layout()

# Exibição dos gráficos
plt.show()
```

Ao usar a função `plt.subplots(1, 2, figsize=(10, 4))`, nós efetivamente indicamos ao Matplotlib que queremos uma figura com uma linha e duas colunas de subplots, juntamente com um tamanho específico definido pelo

`figsize`. Isso nos dá uma tela dividida em duas áreas de gráfico distintas, referenciadas pelas variáveis `ax1` e `ax2`.

Com `ax1.scatter()` e `ax2.plot()`, cada gráfico é criado de forma independente dentro de sua respectiva área.

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

# Dados de exemplo
meses = np.array(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'])
vendas = np.array([20, 35, 30, 35, 27, 25])
satisfacao_cliente = np.array([70, 82, 73, 65, 90, 83])

# Criação de um objeto Figure e dois objetos Axes
fig, ax1 = plt.subplots()

# Plotando as vendas com um gráfico de barras no primeiro Axes
ax1.bar(meses, vendas, color='g', label='Vendas')
ax1.set_xlabel('Mês')
ax1.set_ylabel('Vendas', color='g')
ax1.tick_params('y', colors='g')

# Criando um segundo Axes que compartilha o mesmo eixo x
ax2 = ax1.twinx()

# Plotando a satisfação do cliente com um gráfico de linha no segundo Axes
ax2.plot(meses, satisfacao_cliente, color='b', label='Satisfação do Cliente')
ax2.set_ylabel('Satisfação do Cliente (%)', color='b')
ax2.tick_params('y', colors='b')

# Adicionar títulos e mostrar a legenda
fig.suptitle('Vendas e Satisfação do Cliente por Mês')
ax1.legend(loc='upper left')
ax2.legend(loc='upper right')

# Mostrar o gráfico
plt.show()
```

Neste exemplo, o objeto `ax1` é usado para plotar o gráfico de barras e o objeto `ax2` é criado com a chamada `ax1.twinx()`, o que significa que `ax2` é um novo conjunto de eixos que compartilha o eixo x com `ax1`, mas tem um eixo y independente. Isso é particularmente útil quando temos variáveis com diferentes escalas de medida.

No gráfico de barras (`ax1.bar`), as vendas são representadas como barras verdes, com seus valores no eixo y à esquerda. No gráfico de linha (`ax2.plot`), a satisfação do cliente é representada por uma linha azul, com seus valores no eixo y à direita.

O que é Scikit-Learn

É particularmente conhecida por sua implementação de algoritmos de classificação, regressão e clustering.

- **Classificação**: Identificar a qual categoria um objeto pertence.
- **Regressão**: Prever uma resposta contínua.
- **Clustering**: Agrupar pontos de dados semelhantes.
- **Redução de dimensionalidade**: Reduzir o número de variáveis aleatórias a serem consideradas.
- **Model selection**: Comparar, validar e escolher parâmetros e modelos.
- **Pré-processamento**: Transformar ou encodar dados.

Classes: O Coração dos Modelos de Machine Learning

As **classes** são essencialmente moldes para criar objetos que representam tanto os modelos de machine learning quanto as ferramentas de pré-processamento de dados. No Scikit-Learn, cada algoritmo de machine learning é implementado como uma **classe**. Essas classes são agrupadas logicamente em **módulos**, com base no tipo de problema de machine learning que elas resolvem.

- O módulo `linear_model` contém classes para algoritmos de regressão linear, como `LinearRegression` e `LogisticRegression`.
- O módulo `cluster` oferece classes para algoritmos de agrupamento, como `KMeans`.
- O módulo `decomposition` tem classes para redução de dimensionalidade, como `PCA`.

Métodos: As Ações dos Objetos

Os métodos mais comuns encontrados nos estimadores são:

- `.fit()`: Utilizado para treinar o modelo com os dados fornecidos.

- `.predict()`: Após o treinamento, é utilizado para fazer previsões com novos dados.
- `.transform()`: Usado em transformadores para alterar ou selecionar características dos dados.
- `.fit_transform()`: Uma combinação que realiza o treinamento e a transformação em um único passo, otimizando o processo.

Atributos: As Características dos Objetos

Os **atributos** são variáveis que armazenam informações sobre o estado do objeto. Após treinar um modelo, por exemplo, ele vai conter diversos atributos que fornecem informações úteis. Esses **atributos** podem incluir detalhes sobre os dados treinados, como parâmetros ajustados, importâncias de características ou coeficientes do modelo.

Por exemplo, um objeto `LinearRegression` terá atributos como:

- `.coef_`: Que armazena os coeficientes da regressão linear para cada característica.
- `.intercept_`: Que contém o termo independente da linha de regressão.

Exemplo de Uso de Classe, Método e Atributo

from sklearn.linear_model import LinearRegression
import numpy as np

```
# Dados de exemplo
X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
# y = 1 * x_0 + 2 * x_1 + 3
y = np.dot(X, np.array([1, 2])) + 3

# Instanciando a classe LinearRegression
modelo = LinearRegression()

# Treinando o modelo com os dados, usando o método .fit()
modelo.fit(X, y)

# Verificando os atributos coef_ e intercept_ após o treino
print("Coeficientes: ", modelo.coef_)
print("Intercept: ", modelo.intercept_)
```

2. **Preparação dos dados:**
– `X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])`: Cria um array `numpy` para as variáveis independentes (features), `X`. Cada sub-array representa um ponto de dado, com duas dimensões (ou features).

– `y = np.dot(X, np.array([1, 2])) + 3`: Gera o array de variáveis dependentes (target), `y`, usando uma operação de produto escalar (`np.dot`) entre `X` e um array de coeficientes `[1, 2]`, seguido pela adição de um intercepto `3`. A equação representada é `y = 1*x_0 + 2*x_1 + 3`, onde `x_0` e `x_1` são as variáveis independentes.

3. **Instanciando o modelo de Regressão Linear:**
– `modelo = LinearRegression()`: Cria uma instância da classe `LinearRegression`. Este objeto `modelo` será usado para treinar o modelo com os dados.

4. **Treinamento do modelo:**
– `modelo.fit(X, y)`: Treina o modelo de regressão linear usando os dados `X` (variáveis independentes) e `y` (variável dependente). O método `.fit()` ajusta o modelo linear aos dados, encontrando os coeficientes (pesos) das variáveis independentes e o intercepto que minimizam a soma dos quadrados dos resíduos, uma medida de erro entre os valores observados e os valores previstos pelo modelo.

5. **Verificação dos resultados do treinamento:**
– `print("Coeficientes: ", modelo.coef_)`: Imprime os coeficientes das variáveis independentes (`x_0` e `x_1` neste caso) encontrados pelo modelo após o treinamento. Estes valores devem estar próximos de `[1, 2]`, que são os coeficientes usados para gerar `y`.

– `print("Intercept: ", modelo.intercept_)`: Imprime o valor do intercepto adicionado ao modelo, que deve estar próximo de `3`

O objetivo é prever o valor da variável dependente `y` a partir das variáveis independentes `x`, baseando-se nos coeficientes (`coef_`) e no intercepto (`intercept_`) aprendidos pelo modelo durante o treinamento.

X = [1 1]
[1 2]
[2 2]
[2 3]

- A primeira coluna representa o número de itens no pedido.
- A segunda coluna representa a complexidade média dos itens, onde 1 indica baixa complexidade, 2 indica média complexidade e 3 indica alta complexidade.

y = np.dot(X, np.array([1, 2])) + 3

- `np.array([1, 2])` representa os coeficientes das características. O coeficiente 1 está associado ao número de itens e o coeficiente 2 está associado à complexidade média dos itens.
- O valor 3 representa o intercepto, que é o tempo base de preparo, independentemente do número de itens ou da complexidade.

Calculo o tempo total de preparo para cada pedido:

1. Pedido 1: (1 *1) + (1* 2) + 3 = 6 minutos
    - 1 item com baixa complexidade: 1 * 1 = 1
    - Complexidade média dos itens: 1 * 2 = 2
    - Tempo base de preparo: 3 minutos
2. Pedido 2: (1 *1) + (2* 2) + 3 = 8 minutos
    - 1 item com média complexidade: 1 * 1 = 1
    - Complexidade média dos itens: 2 * 2 = 4
    - Tempo base de preparo: 3 minutos
3. Pedido 3: (2 *1) + (2* 2) + 3 = 9 minutos
    - 2 itens com média complexidade: 2 * 1 = 2
    - Complexidade média dos itens: 2 * 2 = 4
    - Tempo base de preparo: 3 minutos
4. Pedido 4: (2 *1) + (3* 2) + 3 = 11 minutos
    - 2 itens com alta complexidade: 2 * 1 = 2
    - Complexidade média dos itens: 3 * 2 = 6
    - Tempo base de preparo: 3 minutos

Em termos da equação de regressão linear, podemos representar esse cálculo da seguinte forma:

```
y = X * [1, 2] + [3, 3, 3, 3]
```

Onde:

- `X` é a matriz de características
- `[1, 2]` são os coeficientes de características
- `[3, 3, 3, 3]` é um vetor coluna com o intercepto de 3 repetido para cada amostra de dados

Estimators, Transformers e Predictors: Pilares do Scikit-Learn

Os **Estimators** são a base para quase todos os algoritmos de machine learning implementados no Scikit-Learn. Eles são responsáveis por **estimar** parâmetros com base nos dados fornecidos. Para fazer isso, todos os **Estimators** possuem o método `.fit()`. Exemplo:

```
from sklearn.cluster import KMeans

# Dados de exemplo para o agrupamento

X = [[6, 7], [2, 1], [3, 2], [8, 9]]

# Instanciação de um Estimator KMeans

kmeans = KMeans(n_clusters=2, n_init=10)

# Treinando o modelo com .fit()

kmeans.fit(X)

# Os centróides do cluster podem ser acessados através de um atributo após o treino

print(kmeans.cluster_centers_)
```

`KMeans` é um **Estimator** que é usado para encontrar os centróides de um conjunto de dados dado que estamos tentando clusterizar em dois grupos.

Os centróides são os pontos centrais de cada cluster (grupo)

O processo do algoritmo KMeans pode ser resumido da seguinte forma:

1. Inicialização: O algoritmo começa escolhendo aleatoriamente K pontos do conjunto de dados como centróides iniciais.
2. Atribuição: Cada ponto de dados é atribuído ao centróide mais próximo com base em uma medida de distância (geralmente, a distância euclidiana).
3. Atualização: Após a atribuição de todos os pontos de dados aos centróides, os centróides são recalculados como a média (ou centro de massa) dos pontos de dados atribuídos a cada cluster.
4. Repetição: Os passos 2 e 3 são repetidos iterativamente até que os centróides não mudem mais significativamente ou até que um número máximo de iterações seja atingido.

Ao final do processo, cada centróide representa o centro do cluster correspondente. Os centróides são os pontos que minimizam a soma das distâncias quadradas entre os pontos de dados e seus respectivos centróides.

O parâmetro `n_clusters=2` especifica que queremos encontrar dois clusters. O parâmetro `n_init` especifica o número de vezes que o algoritmo KMeans será executado com diferentes inicializações aleatórias dos centróides.

- Após as 10 execuções, o algoritmo retornará a melhor solução encontrada, ou seja, a solução com a menor soma das distâncias quadradas entre os pontos de dados e seus respectivos centróides.

Após o treinamento do modelo com `kmeans.fit(X)`, os centróides encontrados podem ser acessados através do atributo `kmeans.cluster_centers_`. Esses centróides representam os pontos centrais dos dois clusters encontrados pelo algoritmo KMeans.

Os centróides são úteis para entender a estrutura dos clusters e podem ser usados para atribuir novos pontos de dados ao cluster mais próximo. Além disso, os centróides podem fornecer insights sobre as características comuns dos pontos de dados em cada cluster.

-os centróides são os pontos centrais dos clusters encontrados pelo algoritmo KMeans e representam a média ou o centro de massa dos pontos de dados atribuídos a cada cluster.

Transformers: Preparando os Dados

Os Transformers são ferramentas poderosas fornecidas pela biblioteca scikit-learn para realizar transformações nos dados antes de aplicá-los a um modelo de aprendizado de máquina.

1. Pré-processamento: Isso inclui tarefas como padronização, normalização, codificação de variáveis categóricas, preenchimento de valores ausentes, entre outras. O objetivo é garantir que os dados estejam limpos, consistentes e em um formato adequado para o modelo.
2. Redução de dimensionalidade: Quando lidamos com conjuntos de dados de alta dimensionalidade, ou seja, com muitas features (variáveis), pode ser benéfico reduzir a dimensionalidade para melhorar o desempenho do modelo e evitar a maldição da dimensionalidade. Os Transformers podem aplicar técnicas como PCA (Principal Component Analysis) ou t-SNE (t-Distributed Stochastic Neighbor Embedding)
3. Seleção de características: Em alguns casos, pode haver features irrelevantes ou redundantes no conjunto de dados. Os Transformers podem ser usados para selecionar as features mais importantes e descartar as menos relevantes, melhorando assim a eficiência e a interpretabilidade do modelo.

Os Transformers possuem dois métodos principais:

1. `.fit()`: Esse método é usado para aprender os parâmetros da transformação com base nos dados fornecidos. Ele analisa os dados e calcula as estatísticas necessárias para realizar a transformação. Por exemplo, no caso do StandardScaler, o método `.fit()` calcula a média e o desvio padrão de cada feature.
2. `.transform()`: Após o Transformer ser ajustado aos dados com o método `.fit()`, o método `.transform()` é usado para aplicar a transformação aos dados. Ele utiliza os parâmetros aprendidos durante o `.fit()` para transformar os dados de acordo com a lógica específica do Transformer.

Exemplo de uso de um Transformer:

```
from sklearn.preprocessing import StandardScaler

# Dados de exemplo para o pré-processamento
X_train = [[0, 15], [1, -10], [2, 0]]

# Instanciação do Transformer StandardScaler
scaler = StandardScaler()

# Aprendendo os parâmetros com .fit() e aplicando a transformação com .transform()
X_scaled = scaler.fit_transform(X_train)

print(X_scaled)
```

No código acima, o `StandardScaler` é um Transformer que padroniza os recursos, removendo a média e escalando-os para ter uma variância unitária.

Neste exemplo**, estamos usando o Transformer `StandardScaler` para padronizar os dados. O `StandardScaler` é um Transformer que remove a média e escala os dados para ter uma variância unitária. Isso é útil quando temos features com escalas diferentes e queremos que todas tenham a mesma influência no modelo.**

- Utilizamos o método `fit_transform()` para ajustar o `scaler` aos dados de treinamento (`X_train`) e aplicar a transformação de padronização em uma única etapa. Isso é equivalente a chamar `scaler.fit(X_train)` seguido de `scaler.transform(X_train)`.

- O resultado da transformação é atribuído à variável `X_scaled`, que contém os dados padronizados.

resultado:

[[-1.22474487 1.33630621]

 [0.   -0.26726124]

 [1.22474487 -1.06904497]]

Cada linha representa uma amostra padronizada, onde cada feature foi transformada para ter média zero e variância unitária.

X_scaled = scaler.fit_transform(X_train)

1. **Chama o método `fit()` internamente para ajustar o `scaler` aos dados de treinamento (`X_train`). Durante essa etapa, o `scaler` calcula a média e o desvio padrão de cada feature nos dados de treinamento.**
2. **Após o ajuste, o método `transform()` é chamado internamente para aplicar a transformação de padronização nos mesmos dados de treinamento (`X_train`). Ele subtrai a média e divide pelo desvio padrão de cada feature, utilizando os parâmetros aprendidos durante o `fit()`.**
3. **O resultado da transformação é retornado e atribuído à variável `X_scaled`.**

Portanto, a linha `X_scaled = scaler.fit_transform(X_train)` é equivalente a chamar `scaler.fit(X_train)` seguido de `scaler.transform(X_train)`

É importante ressaltar que o método `fit_transform()` é usado apenas nos dados de treinamento. Para transformar novos dados (como dados de teste ou dados futuros), você deve usar apenas o método `transform()`, pois o `scaler` já foi ajustado aos dados de treinamento anteriormente. Por exemplo:

```
X_test = [[3, 5], [4, 2]]
X_test_scaled = scaler.transform(X_test)
```

Neste caso, o `scaler` já foi ajustado aos dados de treinamento usando `fit_transform()`, então podemos aplicar a transformação nos dados de teste usando apenas o método `transform()`.

Predictors: Fazendo Previsões

Predictors são objetos em machine learning que, após serem treinados sobre um conjunto de dados, são capazes de fazer previsões sobre novos dados. Eles são a concretização de um modelo que aprendeu padrões nos dados de treinamento e agora pode aplicar esse aprendizado para prever resultados em dados não vistos anteriormente.

Método `.predict()`

Após o treinamento do modelo com um conjunto de dados (usando o método `.fit()`), o `.predict()` é usado para aplicar o modelo treinado a novos dados, gerando previsões baseadas no aprendizado anterior.

Exemplo

```
from sklearn.linear_model import LinearRegression

# Dados de exemplo
X_train = [[0, 0], [1, 1], [2, 2]]
y_train = [0, 1, 2]

# Instanciação de um Predictor LinearRegression
regressor = LinearRegression()

# Treinando o modelo com .fit()
regressor.fit(X_train, y_train)

# Fazendo previsões com .predict()
X_test = [[3, 3]]
y_pred = regressor.predict(X_test)

print(y_pred)
```

Neste caso, o `LinearRegression` é um Predictor. Ele é treinado com `.fit()` usando o conjunto de treinamento e, em seguida, faz previsões para novos dados com o método `.predict()`.

1. **Preparação dos Dados**: Inicialmente, os dados de treinamento (`X_train`, `y_train`) são definidos. Eles representam, respectivamente, as variáveis independentes e a variável dependente que o modelo tentará prever.

2. **Instanciação do Modelo**: Um objeto `LinearRegression` é instanciado. Este objeto é o Predictor, capaz de realizar a regressão linear, um método estatístico que visa modelar a relação entre uma variável dependente e uma ou mais variáveis independentes.
3. **Treinamento do Modelo**: O método `.fit()` é chamado com os dados de treinamento, permitindo que o modelo "aprenda" a relação entre `X_train` e `y_train`.
4. **Previsão**: Com o modelo treinado, novos dados (`X_test`) são introduzidos ao modelo através do método `.predict()`, que gera as previsões (`y_pred`) baseadas no aprendizado obtido durante o treinamento.

Pre-processamento de Dados com Scikit-Learn: Técnicas Essenciais e Exemplos Práticos

O Scikit-Learn oferece ferramentas robustas para preparar seus dados antes de treinar modelos.

Criação do Conjunto de Dados de Exemplo

import numpy as np
import pandas as pd

```
# Dados de exemplo
data = {
   'Idade': [25, 35, 45, np.nan, 55, 65, 75, np.nan, 85, 95],
   'Gênero': ['F', 'M', 'F', 'F', 'M', 'M', 'F', 'M', 'F', 'M'],
   'Colesterol': [190, np.nan, 240, 225, 210, np.nan, 180, 195, 220, np.nan],
   'Pressão Sanguínea': [70, 80, 78, 75, 72, 130, 60, 85, 90, 77],
   'Fumante': ['Não', 'Sim', 'Não', 'Não', 'Sim', 'Não', 'Sim', 'Não', 'Não', 'Sim'],
   'Risco': ['baixo', 'médio', 'alto', 'baixo', 'alto', 'médio', 'baixo', 'médio', 'baixo', 'alto']
}

df = pd.DataFrame(data)
```

Tratando Dados Faltantes (Handling Missing Data)

No Scikit-Learn, a classe `SimpleImputer` é utilizada para lidar com esses valores.
from sklearn.impute import SimpleImputer

```
# Imputação de dados faltantes na coluna 'Idade' e 'Colesterol'
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')

#aqui aplicamos o imputer nas colunas ‘Idade' e ‘Colesterol' do DataFrame df. Usamos o método fit_transform() para ajustar o imputer aos dados e transformá-los de uma só vez.

df['Idade'] = imputer.fit_transform(df[['Idade']])
df['Colesterol'] = imputer.fit_transform(df[['Colesterol']])
```

Aqui estamos lidando com o problema de dados faltantes nas colunas ‘Idade' e ‘Colesterol'.
`SimpleImputer` do módulo `sklearn.impute`. Essa classe é usada para preencher os valores faltantes de acordo com uma estratégia especificada.

`SimpleImputer`, como `'median'` (mediana) e `'most_frequent'` (valor mais frequente), que podem ser mais apropriadas dependendo das características dos seus dados.

Convertendo Dados Categóricos em Números (Encoding Categorical Data)

Dados categóricos devem ser convertidos para um formato numérico. Usamos `LabelEncoder` para transformar a coluna target e `OneHotEncoder` para as características categóricas.

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

# Encoding da coluna 'Gênero' e 'Fumante'
column_transformer = ColumnTransformer(
    [('encoder', OneHotEncoder(), ['Gênero', 'Fumante'])],
    remainder='passthrough'
)

# Aplicando o ColumnTransformer aos dados, excluindo a coluna 'Risco'
x = column_transformer.fit_transform(df.drop('Risco', axis=1))

# Encoding da coluna 'Risco' usando LabelEncoder
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(df['Risco'])
```

- `LabelEncoder` e `OneHotEncoder` do módulo `sklearn.preprocessing` para codificar as variáveis categóricas.

- `ColumnTransformer` do módulo `sklearn.compose` para aplicar transformações em colunas específicas do DataFrame.
- Criamos uma instância do `ColumnTransformer` chamada `column_transformer` para codificar as colunas ‘Gênero' e ‘Fumante' usando o `OneHotEncoder`:
    - O primeiro argumento é uma lista de tuplas, onde cada tupla especifica uma transformação a ser aplicada em um conjunto de colunas.
    - Nesse caso, temos apenas uma tupla: `('encoder', OneHotEncoder(), ['Gênero', 'Fumante'])`, que aplica o `OneHotEncoder` nas colunas ‘Gênero' e ‘Fumante'.
    - O parâmetro `remainder='passthrough'` indica que as colunas não especificadas devem ser mantidas inalteradas.

Escalonamento de Características (Feature Scaling)

O escalonamento de características é importante para muitos algoritmos de machine learning. No nosso exemplo, podemos usar `StandardScaler` para padronizar as features.

```
from sklearn.preprocessing import StandardScaler

# Instanciando o objeto StandardScaler
scaler = StandardScaler()

# Aplicando o escalonamento no nosso array de características 'X'
x_scaled = scaler.fit_transform(x)
```

O escalonamento de características é uma técnica de pré-processamento que visa transformar as variáveis numéricas para que tenham uma escala similar. Isso é importante porque muitos algoritmos de aprendizado de máquina são sensíveis à escala das features. Se uma feature tiver uma escala muito maior do que as outras, ela pode dominar o algoritmo e prejudicar o desempenho do modelo.

`StandardScaler` é uma classe do Scikit-Learn que padroniza as features, subtraindo a média e dividindo pelo desvio padrão de cada feature. Isso resulta em features com média zero e desvio padrão igual a um.

- Aplicamos o escalonamento no array de características `X` usando o método `fit_transform()` do `scaler`:
    - O método `fit_transform()` ajusta o `scaler` aos dados, calculando a média e o desvio padrão de cada feature, e então aplica a transformação nos dados.
    - O resultado é atribuído à variável `X_scaled`.

Exemplo:

Idade: [25, 35, 45, 55, 65]

Pressão Sanguínea: [120, 130, 140, 150, 160]

Ao aplicar o `StandardScaler`, ele calcula a média e o desvio padrão de cada feature:

- ‘Idade': média = 45, desvio padrão = 15.81
- ‘Pressão Sanguínea': média = 140, desvio padrão = 15.81

Em seguida, o `StandardScaler` subtrai a média de cada valor e divide pelo desvio padrão:

- ‘Idade' escalonada: [-1.26, -0.63, 0.0, 0.63, 1.26]
- ‘Pressão Sanguínea' escalonada: [-1.26, -0.63, 0.0, 0.63, 1.26]

É importante ressaltar que o escalonamento deve ser aplicado apenas nas features numéricas. No código que aplicamos como exemplo, as colunas “Gênero” e “Fumante”, que são categóricas, são transformadas em variáveis numériNo entanto, é importante notar que, embora essas colunas binárias passem pelo escalonamento, o impacto dessa transformação é limitado devido à natureza binária dos dados. O escalonamento pode ajustar essas colunas para que tenham média 0 e desvio padrão 1, mas, dado que os valores originais são 0 e 1, essa etapa não altera a interpretação dessas variáveis da mesma forma que altera para variáveis contínuas.

Além do `StandardScaler`, o Scikit-Learn oferece outras técnicas de escalonamento, como o `MinMaxScaler` (que escala as features para um intervalo específico, geralmente entre 0 e 1) e o `RobustScaler` (que é menos sensível a outliers nos dados).

Introdução aos Pipelines: Simplificando o Processo de Machine Learning

Um **Pipeline** no Scikit-Learn é uma ferramenta que nos auxilia a encadear vários passos de transformação e modelagem de maneira a simplificar esse processo.

O Exemplo Prático de um DataFrame

```
import numpy as np

import pandas as pd

data = {

    'Idade': [25, 35, 45, np.nan, 55, 65, 75, np.nan, 85, 95],

    'Gênero': ['F', 'M', 'F', 'F', 'M', 'M', 'F', 'M', 'F', 'M'],

    'Colesterol': [190, np.nan, 240, 225, 210, np.nan, 180, 195, 220, np.nan],

    'Pressão Sanguínea': [70, 80, 78, 75, 72, 130, 60, 85, 90, 77],

    'Fumante': ['Não', 'Sim', 'Não', 'Não', 'Sim', 'Não', 'Sim', 'Não', 'Não', 'Sim'],

    'Risco': ['baixo', 'médio', 'alto', 'baixo', 'alto', 'médio', 'baixo', 'médio', 'baixo', 'alto']

}

df = pd.DataFrame(data)
```

Simplificação com Pipeline

```
from sklearn.pipeline import Pipeline

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

from sklearn.linear_model import LogisticRegression

# Definição do pré-processador para o ColumnTransformer

preprocessor = ColumnTransformer(

    transformers=[

        ('num', Pipeline(steps=[

            ('imputer', SimpleImputer(strategy='mean')),  # Preenchendo valores ausentes

            ('scaler', StandardScaler())]),              # Escalonando as características numéricas

        ['Idade', 'Colesterol', 'Pressão Sanguínea']),

        ('cat', OneHotEncoder(), ['Gênero', 'Fumante'])  # Convertendo as características categóricas

    ]

)
```

```
# Criação do pipeline completo com o ColumnTransformer e o modelo

pipeline = Pipeline(steps=[

    ('preprocessor', preprocessor),

    ('classifier', LogisticRegression())  # Modelo final que será treinado

])

# Separando as features e o target

X = df.drop('Risco', axis=1)

y = df['Risco']

# Treinando o pipeline completo

pipeline.fit(X, y)
```

1. Importações necessárias:
    - `Pipeline` do módulo `sklearn.pipeline` para criar o pipeline.
    - `SimpleImputer` do módulo `sklearn.impute` para preencher valores ausentes.
    - `StandardScaler` e `OneHotEncoder` do módulo `sklearn.preprocessing` para escalonar as características numéricas e converter as características categóricas, respectivamente.
    - `ColumnTransformer` do módulo `sklearn.compose` para aplicar diferentes transformações em diferentes subconjuntos de colunas.
    - `LogisticRegression` do módulo `sklearn.linear_model` como o modelo final a ser treinado.
2. Definição do pré-processador usando o `ColumnTransformer`:

- O `ColumnTransformer` permite aplicar diferentes transformações em diferentes subconjuntos de colunas.
- Neste caso, temos dois transformadores:
  - `'num'`: Um pipeline que lida com as colunas numéricas (‘Idade', ‘Colesterol', ‘Pressão Sanguínea'). Ele consiste em duas etapas:
    - `SimpleImputer` com `strategy='mean'` para preencher os valores ausentes com a média da coluna.
    - `StandardScaler` para escalonar as características numéricas, subtraindo a média e dividindo pelo desvio padrão.
  - `'cat'`: Um `OneHotEncoder` para converter as colunas categóricas (‘Gênero', ‘Fumante') em variáveis binárias (one-hot encoding).

3. Criação do pipeline completo:
   - O pipeline completo é criado usando a classe `Pipeline` do Scikit-Learn.
   - Ele consiste em duas etapas:
     - `'preprocessor'`: O `ColumnTransformer` definido anteriormente, que aplica as transformações de pré-processamento nos dados.
     - `'classifier'`: O modelo final a ser treinado, neste caso, uma regressão logística (`LogisticRegression`).

Para fazer previsões usando o pipeline que você definiu, você pode simplesmente chamar o método `predict()` no pipeline treinado, passando os novos dados como argumento. Exemplo:

```
# Suponha que você tenha novos dados em um DataFrame chamado 'new_data'

new_data = pd.DataFrame({

    'Idade': [30, 40, 50],
```

```
    'Gênero': ['M', 'F', 'M'],
    'Colesterol': [200, 180, 220],
    'Pressão Sanguínea': [120, 110, 130],
    'Fumante': ['Não', 'Sim', 'Não']
})

# Fazendo previsões usando o pipeline treinado
predictions = pipeline.predict(new_data)

# Imprimindo as previsões
print(predictions)
```

Vantagens do Pipeline

A grande vantagem de utilizar o **Pipeline** é que ele encapsula todas as etapas de tratamento dos dados e modelagem, evitando que tenhamos que manualmente aplicar transformações a cada vez que manipulamos ou dividimos os dados em conjuntos de treino e teste. Além disso, o pipeline evita o vazamento de dados do conjunto de teste para o conjunto de treino, pois garante que o pré-processamento seja aplicado separadamente em cada fase. Essa abordagem não só simplifica o código, mas também fortalece a integridade do processo de modelagem.

Portanto, o **Pipeline** é um recurso poderoso que traz eficiência e segurança ao processo de construção de modelos de machine learning, ao mesmo tempo que mantém um código mais limpo, mais legível e com manutenção mais fácil.

ColumnTransformer

Embora os pipelines sejam extremamente úteis para sequenciar etapas de processamento e modelagem, o `ColumnTransformer` tem seu próprio poder: permite aplicar transformações distintas a colunas específicas dentro de um DataFrame.

Entendendo o ColumnTransformer

O `ColumnTransformer` é uma ferramenta do Scikit-Learn projetada para combinar várias transformações de pré-processamento para colunas distintas de um DataFrame. Enquanto um **pipeline** é ideal para definir uma sequência de passos que serão aplicados à totalidade dos dados, o `ColumnTransformer` permite que diferentes colunas ou grupos de colunas recebam tratamentos individuais.

A principal diferença entre o `ColumnTransformer` e os pipelines tradicionais é que, com o `ColumnTransformer`, podemos realizar ações como:

- Preencher valores faltantes apenas em algumas colunas.
- Escalar apenas as colunas numéricas.
- Codificar apenas as colunas categóricas.

Essas ações podem ser realizadas **simultaneamente**, mas de forma independente para cada grupo de colunas, o que seria mais complexo se estivéssemos utilizando um pipeline simples.

Exemplo:

```
from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.impute import SimpleImputer



# Instanciando o ColumnTransformer
```

```
column_transformer = ColumnTransformer(
    transformers=[
        ('num', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='mean')),  # Tratamento de dados numéricos faltantes
            ('scaler', StandardScaler())                # Escalonamento dos dados numéricos
        ]), ['Idade', 'Colesterol', 'Pressão Sanguínea']),
        ('cat', OneHotEncoder(), ['Gênero', 'Fumante'])  # Codificação de variáveis categóricas
    ],
    remainder='drop'  # Colunas não listadas serão descartadas
)


# Pré-processamento das colunas com transformações específicas
X_transformed = column_transformer.fit_transform(df)
```

-Pipeline permite encadear várias etapas de transformação e estimação em uma única objeto, tornando o fluxo de trabalho mais organizado e fácil de gerenciar. Ele é especialmente útil quando você tem uma sequência fixa de etapas que precisam ser aplicadas consistentemente aos dados, como pré-processamento, seleção de recursos e treinamento do modelo. O Pipeline garante que as etapas sejam executadas na ordem correta e permite que você trate o fluxo de trabalho como um único objeto, facilitando a aplicação de técnicas de validação cruzada e ajuste de hiperparâmetros.

-ColumnTransformer é uma ferramenta flexível que permite aplicar diferentes transformações em diferentes subconjuntos de colunas de um conjunto de dados. Ele é particularmente útil quando você tem dados heterogêneos, com colunas de diferentes tipos (por exemplo, numéricas, categóricas, de texto) que requerem tratamentos distintos. Permitindo especificar transformações específicas para cada subconjunto de colunas, como imputação de valores

ausentes, escalonamento, codificação one-hot, entre outros. Isso elimina a necessidade de pré-processar manualmente cada tipo de coluna e torna o código mais legível e mantível.

Classificação: acurácia, recall, precisão e métricas relacionadas

Acurácia

$$\text{Accuracy} = \frac{\text{correct classifications}}{\text{total classifications}} = \frac{TP + TN}{TP + TN + FP + FN}$$

TP = True Positive (Verdadeiro Positivo)
Exemplo: O modelo previu "sim" e a resposta real era "sim".

TN = True Negative (Verdadeiro Negativo)
Exemplo: O modelo previu "não" e a resposta real era "não".

FP = False Positive (Falso Positivo)
Exemplo: O modelo previu "sim", mas a resposta real era "não" (erro do tipo I).

FN = False Negative (Falso Negativo)
Exemplo: O modelo previu "não", mas a resposta real era "sim" (erro do tipo II).

Recall ou taxa de verdadeiro positivo

A **taxa de verdadeiro positivo (TVP)**, ou a proporção de todos os positivos reais que foram classificados corretamente como positivos, também é conhecida como **recall**.

$$\text{Recall (or TPR)} = \frac{\text{correctly classified actual positives}}{\text{all actual positives}} = \frac{TP}{TP + FN}$$

Falsos negativos são positivos reais que foram classificados incorretamente como negativos, e é por isso que eles aparecem no denominador. No exemplo de classificação de spam, o recall mede a *fração de e-mails de spam que foram classificados corretamente como spam*. Por isso, outro nome para recall é **probabilidade de detecção**: ele responde à pergunta "Qual fração de e-mails de spam é detectada por este modelo?"

Um modelo perfeito hipotético teria zero falsos negativos e, portanto, um recall (TPR) de 1,0, ou seja, uma taxa de detecção de 100%.

A false negative typically has more serious consequences than a false positive.

Taxa de falso positivo

A **taxa de falsos positivos (FPR)** é a proporção de todos os negativos reais que foram classificados *incorretamente* como positivos, também conhecida como **probabilidade de falso alarme**. Ela é definida matematicamente como:

$$\text{FPR} = \frac{\text{incorrectly classified actual negatives}}{\text{all actual negatives}} = \frac{FP}{FP + TN}$$

Falsos positivos são negativos reais que foram classificados incorretamente, por isso aparecem no denominador. No exemplo de classificação de spam, a FPR mede a *fração de e-mails legítimos que foram classificados incorretamente como spam* ou a taxa de falsos alarmes do modelo.

Um modelo perfeito teria zero falso positivo e, portanto, uma FPR de 0,0, ou seja, uma taxa de falso alarme de 0%.

Em um conjunto de dados desequilibrado em que o número de negativos reais é muito, muito baixo (por exemplo, 1 a 2 exemplos no total), pode ser útil

Precisão

Precisão é a proporção de todas as classificações positivas do modelo que são realmente positivas. Ela é definida matematicamente como:

$$\text{Precision} = \frac{\text{correctly classified actual positives}}{\text{everything classified as positive}} = \frac{TP}{TP + FP}$$

No exemplo de classificação de spam, a precisão mede a *fração de e-mails classificados como spam que realmente eram spam*.

Um modelo perfeito hipotético teria zero falso positivo e, portanto, uma precisão de 1,0.

Em um conjunto de dados desequilibrado em que o número de positivos reais é muito, muito baixo, digamos, 1 a 2 exemplos no total, a precisão é menos significativa e menos útil como uma métrica.

A precisão melhora à medida que os falsos positivos diminuem, enquanto o recall melhora quando os falsos negativos diminuem.

Métrica	Orientação
Acurácia	Use como um indicador aproximado do progresso/convergência do treinamento do modelo para conjuntos de dados equilibrados. Para a performance do modelo, use apenas em combinação com outras métricas. Evite para conjuntos de dados desequilibrados. Considere usar outra métrica.
Recall (taxa de verdadeiro positivo)	Use quando os falsos negativos forem mais caros do que os falsos positivos.
Taxa de falso positivo	Use quando os falsos positivos forem mais caros do que os falsos negativos.
Precisão	Use quando for muito importante que as previsões positivas sejam precisas.

Fonte:
https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall

Avaliação de Modelos com Scikit-Learn: Métricas para Classificação e Regressão

https://maykosilva.com/blog/introducao-ao-machine-learning-com-python-usando-scikit-learn/

Métricas de Classificação

Accuracy (Acurácia)

A acurácia é uma métrica que mede a proporção de previsões corretas realizadas por um modelo de classificação. No Scikit-Learn, podemos usar a função `accuracy_score()` para calcular facilmente essa taxa, passando como argumentos as listas dos valores verdadeiros e das previsões feitas pelo modelo.

EXEMPLO:

```
# Lista dos rótulos verdadeiros
y_true = ['spam', 'não spam', 'spam', 'não spam', 'spam', 'não spam', 'não spam', 'spam', 'spam', 'não spam']
# Lista das predições do modelo
y_pred = ['spam', 'não spam', 'spam', 'spam', 'não spam', 'não spam', 'não spam', 'spam', 'spam', 'spam']
```

Cálculo da Acurácia

```
from sklearn.metrics import accuracy_score

# Calculando a acurácia
accuracy = accuracy_score(y_true, y_pred)

# Imprimindo a acurácia
print(f"A acurácia do modelo de classificação é: {accuracy:.2f}")
```

Se obtermos uma acurácia de 0.70, ou 70%. Isso significa que o modelo acertou 70% das suas previsões.

Interpretação da Acurácia

O valor de 0.70 de acurácia indica que, em termos gerais, o modelo é razoavelmente bom em diferenciar e-mails ‘spam' de ‘não spam'. No entanto, é sempre importante considerar o contexto e a aplicação prática. Se, por exemplo, o custo de classificar um ‘não spam' como ‘spam' (perder um e-mail importante) for muito alto, outras métricas como a precisão e o recall devem ser avaliadas conjuntamente.

A função `accuracy_score()` é uma maneira eficiente de obter uma visão rápida da performance de um modelo de classificação. Essa métrica é útil em conjuntos de dados bem balanceados

Precisão e Recall

Precisão (Precision)

A **precisão** mede a exatidão do modelo quando ele faz uma previsão positiva. Em outras palavras, é a proporção de previsões positivas que foram efetivamente corretas. Essa métrica é particularmente importante em situações onde o custo de um falso positivo é alto. Por exemplo, enviar uma promoção para um cliente não interessado pode ser um desperdício de recursos.

Recall (Sensibilidade)

O **recall**, por sua vez, avalia a capacidade do modelo de encontrar todas as instâncias relevantes dentro de uma classe. É a proporção de positivos reais que foram identificados corretamente. Ele é essencial quando o custo de um falso negativo é elevado, como no diagnóstico de uma doença grave onde falhar em identificar um caso positivo pode ter consequências sérias.

Cálculo de Precisão e Recall

```
from sklearn.metrics import precision_score, recall_score

# Calculando a precisão e o recall
precision = precision_score(y_true, y_pred, pos_label='spam')
recall = recall_score(y_true, y_pred, pos_label='spam')

# Imprimindo os resultados
print(f"Precisão do modelo: {precision:.2f}")
print(f"Recall do modelo: {recall:.2f}")
```

Balanceamento entre Precisão e Recall

Em muitos casos, existe uma troca entre precisão e recall. Melhorar um geralmente reduz o desempenho do outro. Uma forma de buscar um equilíbrio é através da métrica **F1-Score**, que é a média harmônica entre precisão e recall.

A **F1-Score** é uma métrica que combina precisão e recall em um único número, oferecendo uma visão equilibrada do desempenho de um

classificador. É particularmente útil em situações onde é preciso balancear a importância de minimizar falsos positivos e falsos negativos.

Cálculo da F1-Score

```
from sklearn.metrics import f1_score

# Supondo que 'y_true' contém os rótulos verdadeiros e 'y_pred' as previsões do modelo
y_true = ['spam', 'não spam', 'spam', 'não spam', 'spam', 'não spam', 'não spam', 'spam', 'spam', 'não spam']
y_pred = ['spam', 'não spam', 'spam', 'spam', 'não spam', 'não spam', 'não spam', 'spam', 'spam', 'spam']

# Calculando a F1-Score
f1 = f1_score(y_true, y_pred, pos_label='spam')

# Imprimindo o resultado
print(f"A F1-Score do modelo é: {f1:.2f}")
```

Um F1-Score de 0.80 indica um bom equilíbrio entre precisão e recall, considerando que o valor pode variar de 0 (pior) a 1 (melhor).

Métricas de Regressão

Erro Quadrático Médio (MSE)

O MSE é calculado tomando a média das diferenças ao quadrado entre os valores previstos pelo modelo e os valores reais. Esse procedimento penaliza mais os erros grandes, devido ao quadrado, tornando essa métrica bastante sensível a outliers no conjunto de dados.

```
from sklearn.metrics import mean_squared_error

# Supondo que temos os valores reais 'y_true' e as previsões 'y_pred' do modelo de regressão
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]

# Calculando o MSE
mse = mean_squared_error(y_true, y_pred)

# Apresentando o MSE
print(f"O Erro Quadrático Médio (MSE) do modelo é: {mse:.2f}")
```

O RMSE (Root Mean Squared Error) é uma métrica comumente usada para avaliar o desempenho de modelos de regressão no contexto de machine learning. O Scikit-Learn, uma popular biblioteca de machine learning em

Python, fornece uma função chamada `mean_squared_error` que nos permite calcular facilmente o RMSE.

```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# Gerar dados sintéticos para regressão
X, y = make_regression(n_samples=100, n_features=1, noise=10, random_state=42)

# Dividir os dados em conjuntos de treinamento e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Criar e treinar o modelo de regressão linear
model = LinearRegression()
model.fit(X_train, y_train)

# Fazer previsões no conjunto de teste
y_pred = model.predict(X_test)

# Calcular o RMSE
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("RMSE:", rmse)
```

O valor de RMSE deve sempre ser interpretado no contexto dos dados. Por estar na mesma unidade dos valores observados, o RMSE oferece uma compreensão direta do erro típico que o modelo comete em suas previsões. Quanto **menor o RMSE, melhor o modelo** é em prever sem grandes erros, e um RMSE de zero indicaria previsões perfeitas.

Coeficiente de Determinação (R^2 Score)

O Coeficiente de Determinação, também conhecido como R^2 (R-squared) ou R^2 Score, é outra métrica comumente usada para avaliar o desempenho de modelos de regressão no contexto de machine learning. O Scikit-Learn fornece a função `r2_score` para calcular facilmente o R^2 Score.

O R^2 Score mede a proporção da variância na variável dependente que é explicada pelas variáveis independentes no modelo de regressão. Ele fornece uma indicação de quão bem o modelo se ajusta aos dados. O valor do R^2 varia entre 0 e 1, onde:

- Um R² de 0 indica que o modelo não explica nenhuma variação nos dados.
- Um R² de 1 indica que o modelo explica toda a variação nos dados.

Exemplo:

```
from sklearn.datasets import make_regression

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import r2_score

# Gerar dados sintéticos para regressão

X, y = make_regression(n_samples=100, n_features=1, noise=10, random_state=42)

# Dividir os dados em conjuntos de treinamento e teste

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Criar e treinar o modelo de regressão linear

model = LinearRegression()

model.fit(X_train, y_train)

# Fazer previsões no conjunto de teste

y_pred = model.predict(X_test)

# Calcular o R² Score

r2 = r2_score(y_test, y_pred)

print("R² Score:", r2)
```

- Ele não indica se os coeficientes do modelo são estatisticamente significativos.
- Ele não considera a complexidade do modelo, ou seja, um modelo com mais variáveis independentes tende a ter um R² maior, mesmo que essas variáveis não contribuam significativamente para a previsão.

Modelos de Seleção com Scikit-Learn

Train Test Split

O Train Test Split é uma abordagem simples e direta para dividir um conjunto de dados em duas partes: uma parte para treinar o modelo e outra parte para testar seu desempenho. Essa divisão nos permite avaliar como o modelo generaliza para dados não vistos durante o treinamento.

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_california_housing
from sklearn.ensemble import RandomForestRegressor

# Carregando o California Housing dataset
california = fetch_california_housing()
X, y = california.data, california.target

# Dividindo o dataset em 80% treino e 20% teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Criando o modelo de regressão com Random Forest
model = RandomForestRegressor()

# Treinando o modelo com os dados de treino
model.fit(X_train, y_train)

# Avaliando o desempenho do modelo com os dados de teste
score = model.score(X_test, y_test)
print(f"Desempenho do Modelo (R² Score): {score:.2f}")
```

Neste exemplo, carregamos o conjunto de dados California Housing usando a função `fetch_california_housing()`. Em seguida, dividimos o conjunto de dados em 80% para treinamento e 20% para teste usando a função `train_test_split()`. Criamos um modelo de regressão Random Forest e o treinamos com os dados de treinamento.

Kfold Cross-Validation

Embora o Train Test Split seja uma técnica útil, ele pode não fornecer uma estimativa robusta do desempenho do modelo, especialmente quando lidamos com conjuntos de dados pequenos ou quando queremos uma avaliação mais abrangente.
Na Kfold Cross-Validation, dividimos o conjunto de dados em “k” partes (geralmente 5 ou 10) e realizamos “k” iterações de treinamento e teste. Em cada iteração, uma parte diferente é usada como conjunto de teste, enquanto as partes restantes são usadas para treinamento.
Ao final, temos “k” resultados de desempenho do modelo, que podem ser combinados para obter uma estimativa mais confiável.

Exemplo:

```
from sklearn.model_selection import cross_val_score
# O modelo RandomForestRegressor já foi instanciado no exemplo anterior
# Executando a validação cruzada Kfold com 10 folds
scores = cross_val_score(model, X, y, cv=10)
# Mostrando o R² Score para cada fold
print("R² Score de cada fold:", scores)
# E a média dos R² Scores
print("Média dos R² Scores:", scores.mean())
```

- `model` é o modelo Random Forest que já foi instanciado anteriormente.
- `X` são os recursos (features) do conjunto de dados completo.
- `y` são os valores alvo (target) do conjunto de dados completo.
- `cv=10` especifica o número de folds (dobras) a serem usados na validação cruzada. Neste caso, estamos usando 10 folds.

A validação cruzada Kfold divide o conjunto de dados em 10 partes (folds) e realiza 10 iterações de treinamento e teste. Em cada iteração, uma parte diferente é usada como conjunto de teste, enquanto as outras 9 partes são usadas para treinamento. O modelo é treinado e avaliado em cada iteração, e no final, temos 10 pontuações de desempenho (R² Scores) do modelo.

PARA ML COM PEQUENO CONJUNTO DE DADOS

https://www.it.exchange/blog/machine-learning-with-small-data-when-big-data-is-not-available/

Algoritmos de Machine Learning com Python

Machine Learning, também conhecido como aprendizado de máquina, é um subcampo da inteligência artificial que emprega algoritmos estatísticos para fazer com que os computadores aprendam a partir dos dados e, então, façam previsões ou tomem decisões.

O que é Regressão?

A regressão é uma técnica que tenta prever um valor contínuo (por exemplo, o preço de uma casa) com base em um ou mais valores de entrada (por exemplo, o tamanho da casa, a localização, o número de quartos, etc.). Em outras palavras, ela tenta estabelecer uma relação entre variáveis independentes (entrada) e uma variável dependente (saída).

Regressão Linear: Conceitos fundamentais

Na regressão linear, assumimos que a saída depende linearmente das variáveis de entrada.

Podemos expressar essa relação na forma:

$$y = a0 + a1x1 + a2x2 + a3x3 + \ldots + anxn$$

Aqui, y é a variável alvo e x1, x2, ..., xn são as variáveis preditoras ou características. A regressão linear envolvendo apenas uma característica é conhecida como **regressão linear simples**, enquanto a que envolve múltiplas características é conhecida como **regressão linear múltipla**. Os parâmetros a0, a1, a2, ..., an (também conhecidos como coeficientes ou pesos) do modelo são o que queremos determinar com o algoritmo de regressão linear.

Exemplo:

Peso: [50, 60, 70, 80, 90]

Altura: [152, 167, 170, 175, 192]

A Regressão Linear faz isso através de uma equação matemática. No nosso caso, a equação seria assim:

Altura = a0 + a1*Peso + ϵ

Na equação, “a0” e “a1” são os chamados coeficientes. Eles são os números que queremos descobrir para que a nossa equação possa prever a altura de uma pessoa baseada apenas no seu peso.

Substituindo o peso e a altura de cada pessoa em nossa equação de regressão linear, teríamos as seguintes cinco equações (uma para cada observação):

152 = a0 + a150 + ϵ1 167 = a0 + a160 + ϵ2 170 = a0 + a170 + ϵ3 175 = a0 + a180 + ϵ4 192 = a0 + a1*90 + ϵ5

As incógnitas em nosso sistema de equações são os coeficientes a0 (o intercepto) e a1 (a inclinação). O objetivo do algoritmo de regressão linear é encontrar os valores de a0 e a1 que minimizam a soma dos quadrados dos erros (ϵ). Este é um problema de otimização que pode ser resolvido com várias técnicas, incluindo gradientes descendentes e equações normais.

Mas como encontrar os valores de “a0” e “a1” que fazem a nossa equação funcionar da melhor maneira possível?

É aqui que entra o chamado “Erro Quadrático Médio” (MSE, do inglês “Mean Squared Error”). O MSE é uma maneira de calcular o quão errada está a nossa equação. Ele faz isso comparando os valores que nossa equação prevê com os valores reais que temos (no nosso caso, as alturas das pessoas). Quanto menor for o MSE, melhor é a nossa equação.

Então, o que a Regressão Linear faz é testar muitos valores diferentes para “a0” e “a1” e escolher os que fazem o MSE ser o menor possível. Em outras palavras, ela escolhe os valores que fazem a nossa equação prever as alturas das pessoas da maneira mais precisa possível.

Essas técnicas incluem resolver uma equação matricial conhecida como equação normal ou usar uma técnica chamada **gradiente descendente** para testar diferentes conjuntos de parâmetros iterativamente.

Regressão Linear com Scikit-Learn

```
from sklearn.datasets import load_diabetes

from sklearn.linear_model import LinearRegression

from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt

# Carregando o conjunto de dados diabetes

diabetes = load_diabetes()

# Separando as variáveis independentes (X) e a variável alvo (y)

X = diabetes.data[:, 2]  # Usando apenas uma característica para fins de visualização

y = diabetes.target

# Dividindo os dados em conjuntos de treinamento e teste

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Criando o modelo de regressão linear

model = LinearRegression()

# Treinando o modelo com os dados de treinamento

model.fit(X_train.reshape(-1, 1), y_train)
```

```
# Fazendo previsões com os dados de teste

y_pred = model.predict(X_test.reshape(-1, 1))

# Plotando o gráfico de dispersão dos dados de teste

plt.scatter(X_test, y_test, color='blue', label='Dados de Teste')

# Plotando a linha de regressão

plt.plot(X_test, y_pred, color='red', linewidth=2, label='Linha de Regressão')

# Adicionando rótulos e título ao gráfico

plt.xlabel('Característica')

plt.ylabel('Progressão da Doença')

plt.title('Regressão Linear - Diabetes')

plt.legend()

# Exibindo o gráfico

plt.show()
```

Regressão Polinomial: Conceitos Fundamentais

Ás vezes a relação entre as variáveis de entrada e saída não é linear. Nesses casos, a regressão polinomial pode ser uma melhor opção. A regressão polinomial tenta encontrar a melhor curva, e não uma linha reta, que se ajusta aos dados.

Regressão Polinomial: Conceitos Fundamentais

A regressão polinomial é uma extensão da regressão linear que pode ajudar quando a relação entre as variáveis de entrada e saída não é linear.

Exemplo:

Tempo (em minutos): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Velocidade (em km/h): [10, 20, 30, 40, 50, 60, 65, 70, 70, 70]

Na regressão polinomial, a equação não é uma linha reta, mas uma curva. A equação se parece com isto:

Velocidade = a0 + a1*Tempo + a2*Tempo^2 + … + an*Tempo^n + ϵ

Aqui, “a0” é o termo constante, “a1” é o coeficiente do tempo, “a2” é o coeficiente do tempo ao quadrado, e assim por diante, até “an”, que é o coeficiente do tempo elevado à potência “n”. O termo “ϵ” é o erro, que é a diferença entre o valor real da velocidade e o valor previsto pela nossa equação.

```
# Importando as bibliotecas necessárias

from sklearn.preprocessing import PolynomialFeatures

from sklearn.linear_model import LinearRegression

import numpy as np

import matplotlib.pyplot as plt

# Nosso conjunto de dados

Tempo = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).reshape(-1, 1)

Velocidade = np.array([10, 20, 30, 40, 50, 60, 65, 70, 70, 70])

# Transformando os dados para um formato polinomial

poly = PolynomialFeatures(degree = 2) # Vamos considerar um polinômio de grau 2

Tempo_poly = poly.fit_transform(Tempo)

# Criando o modelo de regressão polinomial

model = LinearRegression()
```

```
# Treinando o modelo

model.fit(Tempo_poly, Velocidade)

# Agora vamos criar um gráfico

plt.scatter(Tempo, Velocidade, color = 'blue') # Pontos originais

plt.plot(Tempo, model.predict(Tempo_poly), color = 'red') # Linha da regressão

plt.title('Regressão Polinomial')

plt.xlabel('Tempo')

plt.ylabel('Velocidade')

plt.show()
```

Por favor, note que esse é um exemplo simplificado. Em um cenário real, você precisaria dividir seus dados em conjuntos de treinamento e teste, ajustar os hiperparâmetros do modelo, avaliar o desempenho do modelo, etc.

Regressão Polinomial com Scikit-Learn

```
# Importando as bibliotecas necessárias

from sklearn import datasets

from sklearn.preprocessing import PolynomialFeatures

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error

import matplotlib.pyplot as plt

import numpy as np

# Carregando o conjunto de dados de diabetes
```

```
diabetes = datasets.load_diabetes()

# Vamos usar apenas uma característica para simplificar

X = diabetes.data[:, np.newaxis, 2]

y = diabetes.target

# Dividindo os dados em conjunto de treinamento e teste

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Transformando os dados para um formato polinomial

poly = PolynomialFeatures(degree=2)

X_train_poly = poly.fit_transform(X_train)

X_test_poly = poly.transform(X_test)

# Criando o modelo de regressão polinomial

model = LinearRegression()

# Treinando o modelo

model.fit(X_train_poly, y_train)

# Fazendo previsões

y_pred = model.predict(X_test_poly)

# Calculando o erro quadrático médio

mse = mean_squared_error(y_test, y_pred)

print('Erro Quadrático Médio: ', mse)

# Agora vamos criar um gráfico

plt.scatter(X_test, y_test, color='blue')  # Pontos de teste
```

```
plt.scatter(X_test, y_pred, color='red')  # Pontos previstos

plt.title('Regressão Polinomial com Scikit-Learn')

plt.xlabel('Medição de índice de massa corporal')

plt.ylabel('Progressão da doença')

plt.show()
```

Pipeline em Machine Learning

Aqui está um exemplo de como você pode criar um pipeline para um projeto de regressão:

```
from sklearn.datasets import load_diabetes

from sklearn.model_selection import train_test_split

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler, PolynomialFeatures

from sklearn.linear_model import LinearRegression



# Carregando os dados de diabetes

diabetes = load_diabetes()



# Separando as variáveis independentes (X) e a variável dependente (y)

X = diabetes.data

y = diabetes.target
```

```
# Dividindo os dados em conjuntos de treinamento e teste

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)



# Criando o pipeline

pipeline = Pipeline([

    ("scaler", StandardScaler()),

    ("poly_features", PolynomialFeatures(degree=2)),

    ("regressor", LinearRegression())

])



# Treinando o pipeline com os dados de treinamento

pipeline.fit(X_train, y_train)



# Fazendo previsões com os dados de teste

predictions = pipeline.predict(X_test)
```

Validação Cruzada com Pipelines e Scikit-Learn Python

A validação cruzada é uma técnica essencial em aprendizado de máquina que nos permite avaliar o desempenho de um modelo de forma mais robusta e confiável. Ela envolve dividir o conjunto de dados em subconjuntos menores, treinar e testar o modelo em cada subconjunto e, em seguida, combinar os resultados para obter uma estimativa mais precisa do desempenho do modelo.

```
from sklearn import datasets

from sklearn.preprocessing import PolynomialFeatures

from sklearn.model_selection import train_test_split, cross_val_score

from sklearn.linear_model import LinearRegression

from sklearn.pipeline import make_pipeline

import numpy as np

# Carregando o conjunto de dados de diabetes

diabetes = datasets.load_diabetes()

# Vamos usar apenas uma característica para simplificar

X = diabetes.data[:, np.newaxis, 2]

y = diabetes.target

# Dividindo os dados em conjunto de treinamento e teste

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Criando o pipeline para a regressão polinomial

model = make_pipeline(PolynomialFeatures(degree=2), LinearRegression())

# Treinando o modelo com validação cruzada

scores = cross_val_score(model, X_train, y_train, cv=5)

print("Scores da validação cruzada: ", scores)
```

Ao executar o código fonte acima, você obterá o seguinte resultado:

Scores da validação cruzada: [0.34951055 0.20118732 0.37641365 0.48463914 0.18011347]

Os números impressos são os escores R^2 para cada uma das 5 divisões (folds) da validação cruzada. A métrica R^2 (também conhecida como coeficiente de determinação) é uma medida de quão bem as previsões do nosso modelo de regressão se ajustam aos dados reais. O R^2 varia de 0 a 1, onde 1 significa que nosso modelo explica completamente a variância dos dados alvo

A variação nos escores vai de aproximadamente 0.18 a 0.48. Isso é um indicativo de que nosso modelo está sofrendo de algum grau de variância, pois seu desempenho muda dependendo do subconjunto específico de dados usado para treiná-lo. Pode ser útil experimentar ajustar os hiperparâmetros do modelo, utilizar mais recursos dos dados, ou tentar uma técnica diferente de pré-processamento dos dados para melhorar o desempenho do modelo.

É importante notar que, embora esses escores sejam úteis para avaliar a capacidade de nosso modelo de generalizar para novos dados, eles não nos fornecem a imagem completa. Para uma análise mais completa, você poderia considerar a visualização das previsões reais vs esperadas, ou a verificação de outras métricas de desempenho, como erro médio absoluto ou erro médio quadrático.

A validação cruzada é importante porque nos fornece uma estimativa mais confiável do desempenho do modelo em dados não vistos. Ao treinar e testar o modelo em diferentes subconjuntos dos dados, podemos avaliar sua capacidade de generalização e identificar possíveis problemas de overfitting (quando o modelo se ajusta bem demais aos dados de treinamento, mas tem desempenho ruim em novos dados).

O que são Algoritmos de Classificação?

Os algoritmos de classificação são uma categoria de algoritmos de Machine Learning supervisionado que têm como objetivo atribuir rótulos ou classes a novos dados com base em um conjunto de dados de treinamento.

Decision Tree (Árvore de Decisão)

A Decision Tree é um algoritmo de classificação que constrói uma estrutura de árvore para tomar decisões com base nos atributos dos dados. Cada nó interno da árvore representa um teste em um atributo, cada ramo representa o resultado desse teste, e cada nó folha representa uma classe ou rótulo.

O Gini Impurity é uma medida utilizada para avaliar a qualidade de uma divisão em um nó da árvore de decisão. Ele mede a impureza ou a mistura de classes em um nó. Quanto menor o valor do Gini Impurity, mais pura é a divisão, ou seja, mais homogêneas são as classes resultantes.

Exemplo de codigo:

```
import numpy as np

from sklearn.tree import DecisionTreeClassifier

from sklearn.tree import plot_tree

import matplotlib.pyplot as plt

# Dados de exemplo

data = [

    ['A', 8, 'Sim', 'Baixo', 'Boa'],

    ['B', 6, 'Não', 'Alto', 'Ruim'],
```

```
    ['C', 7, 'Sim', 'Moderado', 'Regular'],
    ['D', 5, 'Não', 'Alto', 'Ruim'],
    ['E', 9, 'Sim', 'Baixo', 'Boa'],
    ['F', 6, 'Sim', 'Moderado', 'Regular'],
    ['G', 7, 'Não', 'Baixo', 'Regular'],
    ['H', 8, 'Sim', 'Baixo', 'Boa'],
    ['I', 5, 'Não', 'Alto', 'Ruim'],
    ['J', 7, 'Sim', 'Moderado', 'Regular']
]

# Convertendo os dados para um array numpy
data_array = np.array(data)

# Separando os atributos (X) e a classe (y)
X = data_array[:, 1:-1]
y = data_array[:, -1]

# Convertendo os valores categóricos para numéricos
X_encoded = []
for row in X:
    hours = int(row[0])
    exercise = 1 if row[1] == 'Sim' else 0
    stress = {'Baixo': 0, 'Moderado': 1, 'Alto': 2}[row[2]]
    X_encoded.append([hours, exercise, stress])
```

```
# Criando e treinando a Decision Tree

clf = DecisionTreeClassifier()

clf.fit(X_encoded, y)

# Plotando a árvore de decisão

plt.figure(figsize=(12, 8))

plot_tree(clf, filled=True, feature_names=['Horas de Sono', 'Exercício Físico', 'Estresse'],
class_names=['Boa', 'Regular', 'Ruim'])

plt.show()

# Fazendo previsões com novos dados

new_data = [[7, 'Sim', 'Baixo'], [6, 'Não', 'Alto'], [8, 'Sim', 'Moderado']]

new_data_encoded = []

for row in new_data:

    hours = int(row[0])

    exercise = 1 if row[1] == 'Sim' else 0

    stress = {'Baixo': 0, 'Moderado': 1, 'Alto': 2}[row[2]]

    new_data_encoded.append([hours, exercise, stress])

predictions = clf.predict(new_data_encoded)

print("Previsões:", predictions)
```

O algoritmo de Árvore de Decisão é um método de aprendizado supervisionado que constrói uma árvore binária para tomar decisões com base nos atributos dos dados. Ele seleciona recursivamente o melhor atributo para dividir os dados em cada nó, buscando maximizar a separação das classes. A árvore resultante pode ser usada para fazer previsões em novos dados, seguindo os caminhos da árvore com base nos valores dos atributos.

A Árvore de Decisão é um algoritmo simples e interpretável, capaz de lidar com dados categóricos e numéricos. No entanto, pode ser propenso a overfitting se a árvore se tornar muito profunda. Para mitigar esse problema, técnicas como a poda da árvore podem ser aplicadas.

Random Forest (Floresta Randômica)

A Random Forest é um algoritmo que combina várias Decision Trees para fazer previsões mais robustas e precisas. Cada árvore na floresta é treinada em um subconjunto aleatório dos dados de treinamento e usa um subconjunto aleatório dos atributos para tomar decisões.

A ideia por trás da Random Forest é que, ao combinar as previsões de várias árvores treinadas de forma independente, é possível reduzir a variância e melhorar a generalização do modelo. Durante a previsão, cada árvore na floresta faz sua própria previsão, e a classe final é determinada por votação majoritária.

Exemplo:

```
import numpy as np

from sklearn.ensemble import RandomForestClassifier

from sklearn.tree import plot_tree

import matplotlib.pyplot as plt

# Dados de exemplo

data = [
```

```
    ['A', 8, 'Sim', 'Baixo', 'Boa'],
    ['B', 6, 'Não', 'Alto', 'Ruim'],
    ['C', 7, 'Sim', 'Moderado', 'Regular'],
    ['D', 5, 'Não', 'Alto', 'Ruim'],
    ['E', 9, 'Sim', 'Baixo', 'Boa'],
    ['F', 6, 'Sim', 'Moderado', 'Regular'],
    ['G', 7, 'Não', 'Baixo', 'Regular'],
    ['H', 8, 'Sim', 'Baixo', 'Boa'],
    ['I', 5, 'Não', 'Alto', 'Ruim'],
    ['J', 7, 'Sim', 'Moderado', 'Regular']
]

# Convertendo os dados para um array numpy
data_array = np.array(data)

# Separando os atributos (X) e a classe (y)
X = data_array[:, 1:-1]
y = data_array[:, -1]

# Convertendo os valores categóricos para numéricos
X_encoded = []
for row in X:
    hours = int(row[0])
    exercise = 1 if row[1] == 'Sim' else 0
```

```
    stress = {'Baixo': 0, 'Moderado': 1, 'Alto': 2}[row[2]]
    X_encoded.append([hours, exercise, stress])
# Criando e treinando o Random Forest
clf = RandomForestClassifier(n_estimators=100)  # Usando 100 árvores
clf.fit(X_encoded, y)
# Escolhendo uma árvore específica para visualização, por exemplo, a primeira árvore
estimator = clf.estimators_[0]
# Plotando a árvore de decisão escolhida
plt.figure(figsize=(20, 10))
plot_tree(estimator, filled=True, feature_names=['Horas de Sono', 'Exercício Físico', 'Estresse'],
class_names=['Boa', 'Regular', 'Ruim'])
plt.show()
# Fazendo previsões com novos dados
new_data = [[7, 'Sim', 'Baixo'], [6, 'Não', 'Alto'], [8, 'Sim', 'Moderado']]
new_data_encoded = []
for row in new_data:
    hours = int(row[0])
    exercise = 1 if row[1] == 'Sim' else 0
    stress = {'Baixo': 0, 'Moderado': 1, 'Alto': 2}[row[2]]
    new_data_encoded.append([hours, exercise, stress])
predictions = clf.predict(new_data_encoded)
```

```
print("Previsões:", predictions)
```

Em resumo, a Floresta Aleatória produz uma série de árvores de decisão com estruturas potencialmente diferentes, o que pode levar a diferentes previsões para os mesmos dados. Esta é uma das razões pelas quais a Floresta Aleatória é muitas vezes mais poderosa do que uma única Árvore de Decisão, já que ela pode capturar uma gama mais ampla de padrões nos dados.

Decision Tree e Random Forest com Scikit-Learn e Dataset load_iris()

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score

from sklearn.tree import plot_tree

import matplotlib.pyplot as plt

# Carregando o conjunto de dados Iris

iris = load_iris()

X = iris.data

y = iris.target

# Dividindo os dados em conjuntos de treinamento e teste

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Criando e treinando a Decision Tree

dt_clf = DecisionTreeClassifier()

dt_clf.fit(X_train, y_train)

# Criando e treinando a Random Forest

rf_clf = RandomForestClassifier(n_estimators=100)

rf_clf.fit(X_train, y_train)

# Fazendo previsões nos dados de teste

dt_predictions = dt_clf.predict(X_test)

rf_predictions = rf_clf.predict(X_test)

# Calculando a acurácia das previsões

dt_accuracy = accuracy_score(y_test, dt_predictions)

rf_accuracy = accuracy_score(y_test, rf_predictions)

print("Acurácia da Decision Tree:", dt_accuracy)

print("Acurácia da Random Forest:", rf_accuracy)

# Plotando a árvore de decisão

plt.figure(figsize=(12, 8))

plot_tree(dt_clf, filled=True, feature_names=iris.feature_names, class_names=iris.target_names)

plt.show()
```

Ao executar esse código, você verá uma representação visual da árvore de decisão treinada. Cada nó na árvore representa uma condição de divisão com base em um atributo, e as folhas representam as classes previstas.

Lembre-se de que, para conjuntos de dados maiores e árvores mais complexas, a visualização pode ficar muito grande e difícil de interpretar. Nesses casos, você pode optar por visualizar apenas uma parte da árvore ou usar outras técnicas de interpretação, como a importância dos atributos.

Entendendo a Máquina de Vetores de Suporte (SVM)