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.