Aula 10 – Transformação e Criação de Novas Colunas

Desvendando Seus Dados: Transformação e Criação de Colunas para Análises Poderosas

Bem-vindo(a) à Aula 10 do nosso Curso de Análise Exploratória de Dados! Se você já se sentiu sobrecarregado(a) pela quantidade de informações em uma planilha ou banco de dados, saiba que não está sozinho(a). Muitas vezes, os dados brutos, por si só, não contam a história completa. Eles são como ingredientes em uma cozinha: deliciosos individualmente, mas o verdadeiro potencial surge quando os combinamos e transformamos em algo novo.

Nesta aula, vamos mergulhar no coração da manipulação de dados, aprendendo a moldar e aprimorar suas informações para extrair *insights* valiosos. Imagine que você tem uma lista de vendas, mas precisa calcular o lucro, ou talvez agrupar clientes por faixa etária. Essas são as transformações que nos permitem ir além do óbvio, revelando padrões e tendências que, de outra forma, permaneceriam ocultos.

Nosso objetivo principal é que, ao final desta jornada, você se sinta confiante para modificar colunas existentes, criar novas variáveis a partir de operações complexas e até mesmo transformar dados contínuos em categorias mais gerenciáveis. Você será capaz de preparar seus dados para análises mais profundas e visualizações impactantes, utilizando as ferramentas mais poderosas do mercado: Python e a biblioteca Pandas.

Para embarcar nesta aula, é útil que você já tenha uma familiaridade básica com estruturas de dados como DataFrames do Pandas e a lógica de programação em Python. Se você já sabe como carregar um arquivo CSV ou selecionar uma coluna, está no caminho certo! Prepare-se para elevar suas habilidades de análise de dados a um novo patamar, tornando-se um verdadeiro arquiteto da informação.

O Poder da Adaptação: Modificando Colunas Existentes e Alterando Tipos de Dados

No mundo real, os dados raramente chegam prontos para uso. Pense em um chef que recebe ingredientes frescos: alguns precisam ser lavados, outros picados, e alguns talvez precisem ser cozidos antes de serem combinados. Da mesma forma, seus dados podem vir com tipos incorretos, valores inconsistentes ou em formatos que não permitem a análise desejada. É aqui que entra a arte de modificar colunas existentes.

Imagine que você está analisando dados de vendas de uma loja online. Você tem uma coluna de "Preço" que, por algum motivo, foi importada como texto (string) em vez de número. Se tentar somar esses "preços", o Python vai tratá-los como palavras, e não como valores monetários, resultando em um erro ou em uma concatenação de textos, o que não faz sentido. Esse é um problema clássico e muito comum no dia a dia de um analista de dados.

> A solução para esse tipo de desafio é a **alteração de tipos de dados**, e o Pandas oferece uma ferramenta elegante para isso: o método .astype(). Ele permite que você converta uma coluna de um tipo para outro, como de texto para número, de número para data, ou até mesmo para tipos mais eficientes em termos de memória.

Essa é uma etapa fundamental na limpeza e preparação de dados, garantindo que suas operações matemáticas e lógicas funcionem como esperado. Ao dominar o .astype(), você ganha controle sobre a estrutura dos seus dados, garantindo que cada coluna esteja no formato ideal para a análise. É como garantir que cada ferramenta na sua caixa de ferramentas esteja pronta para a função certa: um martelo para pregos, uma chave de fenda para parafusos. Sem essa organização, o trabalho se torna muito mais difícil e propenso a erros.

.astype() em Ação: Transformando Dados para Análise

Vamos continuar com nosso exemplo da loja online. Suponha que você tenha um DataFrame chamado df_vendas e a coluna 'Preco_Unitario' está como object (o tipo genérico para strings no Pandas). Para realizar cálculos, precisamos que ela seja numérica.

A beleza do .astype() é sua simplicidade. Você seleciona a coluna e aplica o método, especificando o novo tipo de dado. Por exemplo, para converter para um número decimal (float), você faria `df_vendas['Preco_Unitario'] = df_vendas['Preco_Unitario'].astype(float)`. Se houvesse valores não numéricos, o Pandas levantaria um erro, o que é bom, pois indica a necessidade de limpeza prévia.

```
import pandas as pd

# Exemplo de DataFrame com coluna de preço como string
dados = {
    'Produto': ['Camisa', 'Calça', 'Sapato'],
    'Preco_Unitario': ['59.90', '120.50', '89.00'],
    'Quantidade': [2, 1, 3]
}
df_vendas = pd.DataFrame(dados)

print("Tipos de dados originais:")
print(df_vendas.dtypes)

# Convertendo a coluna 'Preco_Unitario' para float
df_vendas['Preco_Unitario'] = df_vendas['Preco_Unitario'].astype(float)

print("\nTipos de dados após conversão:")
print(df_vendas.dtypes)

# Agora podemos fazer operações matemáticas
df_vendas['Total_Venda'] = df_vendas['Preco_Unitario'] * df_vendas['Quantidade']

print("\nDataFrame com nova coluna 'Total_Venda':")
print(df_vendas)
```

Na prática profissional, a alteração de tipos de dados é uma das primeiras etapas em qualquer projeto de análise. Dados de data e hora, por exemplo, frequentemente vêm como strings e precisam ser convertidos para o tipo datetime para que você possa filtrar por períodos, extrair o dia da semana ou calcular durações. Sem essa conversão, operações de tempo seriam impossíveis. É a base para garantir a integridade e a funcionalidade dos seus dados.

Indo Além do Óbvio: Criando Novas Colunas a Partir de Operações

Depois de garantir que seus dados estão nos tipos corretos, o próximo passo é desbloquear seu verdadeiro potencial: criar novas informações. Pense em um detetive que, ao invés de apenas coletar fatos isolados, os combina para formar uma nova pista, uma nova teoria que o leva à solução do caso. No mundo dos dados, isso significa gerar novas colunas que são derivadas de outras já existentes.

Valor Total da Venda

Preço unitário × Quantidade vendida

Idade do Cliente

Data atual - Data de nascimento

Faixa Etária

Categorização baseada na idade

Muitas vezes, as colunas que você recebe não são as que você realmente precisa para responder às suas perguntas de negócio. Por exemplo, você pode ter o preço unitário e a quantidade vendida, mas o que realmente importa para a análise de desempenho é o "Valor Total da Venda". Ou talvez você tenha a data de nascimento dos clientes, mas para segmentá-los, precise da "Idade" ou da "Faixa Etária".

A criação de novas colunas é uma das técnicas mais poderosas na análise exploratória de dados, pois permite que você enriqueça seu conjunto de dados com variáveis que são diretamente relevantes para seus objetivos. Isso pode envolver operações matemáticas simples, como soma, subtração, multiplicação e divisão, ou até mesmo lógicas mais complexas, como condições "se-então" para categorizar dados.

Essa capacidade de engenharia de *features* (criação de novas características) é o que diferencia um analista de dados que apenas relata o que vê de um que realmente extrai *insights* acionáveis. É a sua chance de ser criativo e transformar dados brutos em conhecimento estratégico.

A Receita para Novas Variáveis: Operações e Lógica Condicional

Criar novas colunas no Pandas é surpreendentemente intuitivo. Para operações matemáticas, basta aplicar a operação diretamente entre as colunas. Por exemplo, para calcular o "Lucro", você subtrairia o "Custo" do "Preço de Venda": `df['Lucro'] = df['Preco_Venda'] - df['Custo']`. O Pandas realiza a operação elemento a elemento, de forma vetorizada e eficiente.

Para lógicas mais complexas, como categorizar dados com base em condições, podemos usar a função np.where da biblioteca NumPy (que geralmente é importada junto com Pandas). Imagine que você quer criar uma coluna 'Status_Venda' que seja 'Alta' se o 'Total_Venda' for maior que 100, e 'Baixa' caso contrário.

```
import numpy as np

# Continuamos com nosso df_vendas
# Adicionando uma coluna de custo para calcular o lucro
df_vendas['Custo_Unitario'] = [30.00, 80.00, 50.00]

# Calculando o Lucro por item
df_vendas['Lucro_Item'] = df_vendas['Preco_Unitario'] - df_vendas['Custo_Unitario']

# Calculando o Lucro Total da Venda
df_vendas['Lucro_Total'] = df_vendas['Lucro_Item'] * df_vendas['Quantidade']

# Criando uma coluna condicional 'Status_Venda'
df_vendas['Status_Venda'] = np.where(df_vendas['Total_Venda'] > 100, 'Alta', 'Baixa')

print("\nDataFrame com novas colunas de Lucro e Status:")
print(df_vendas)
```

> No ambiente profissional, essa capacidade é crucial para a **engenharia de *features***, um passo vital na preparação de dados para modelos de *machine learning*. Por exemplo, criar uma coluna 'Tempo_Desde_Ultima_Compra' pode ser um preditor muito mais forte para a próxima compra do que a data da última compra em si.

É a diferença entre ter dados e ter *dados inteligentes*.

O Canivete Suíço da Transformação: A Função .apply()

Até agora, vimos como modificar colunas existentes e criar novas a partir de operações diretas ou lógicas condicionais simples. Mas e se a transformação que você precisa for mais complexa? E se envolver uma lógica personalizada que não pode ser expressa por uma simples operação matemática ou um np.where? É aqui que a função .apply() entra em cena, atuando como um verdadeiro canivete suíço para suas transformações de dados.

01

Definir a Tarefa

Você cria uma função Python personalizada

02

Aplicar a Função

O .apply() executa sua função em cada elemento

03

Obter Resultados

Receba os dados transformados automaticamente

Pense no .apply() como um assistente pessoal que você pode instruir para realizar uma tarefa específica em cada item de uma lista, ou em cada linha/coluna de uma tabela. Você define a tarefa (uma função Python), e o .apply() se encarrega de executá-la para você, elemento por elemento, ou linha por linha, ou coluna por coluna. Isso é incrivelmente útil quando suas regras de negócio são muito específicas ou quando você precisa integrar lógicas de outras bibliotecas.

Embora operações vetorizadas (como as que usamos para somar colunas) sejam geralmente mais rápidas, o .apply() oferece uma flexibilidade incomparável. Ele permite que você use qualquer função Python – seja uma função anônima (lambda) ou uma função que você mesmo definiu – para processar seus dados. Essa capacidade de aplicar lógica customizada é o que o torna indispensável para cenários onde a complexidade da transformação excede as operações padrão do Pandas.

Dominar o .apply() significa que você não estará limitado pelas funções pré-definidas. Você terá a liberdade de implementar qualquer regra de negócio, por mais intrincada que seja, diretamente em seu DataFrame.

.apply() na Prática: Funções Customizadas para Seus Dados

A função .apply() pode ser usada em Series (colunas) ou DataFrames inteiros. Quando aplicada a uma Series, ela opera em cada elemento. Quando aplicada a um DataFrame, ela pode operar em cada linha (axis=1) ou em cada coluna (axis=0).

Vamos imaginar que você tem uma coluna de 'Nome_Completo' e precisa extrair apenas o primeiro nome e o sobrenome, ou talvez padronizar o formato de um texto. Ou, em um cenário mais complexo, você precisa calcular uma pontuação de risco para cada cliente com base em múltiplas colunas (idade, renda, histórico de crédito) usando uma fórmula específica.

```
# Exemplo de uso do .apply() para padronizar texto e calcular pontuação
dados_clientes = {
    'Nome_Completo': ['joao silva', 'MARIA OLIVEIRA', 'Pedro Souza'],
    'Idade': [30, 45, 22],
    'Renda': [5000, 8000, 3000]
}
df_clientes = pd.DataFrame(dados_clientes)

# Função para padronizar nomes (Primeira Letra Maiúscula)
def padronizar_nome(nome):
    return ' '.join([n.capitalize() for n in nome.split()])

# Aplicando a função na coluna 'Nome_Completo'
df_clientes['Nome_Padronizado'] = df_clientes['Nome_Completo'].apply(padronizar_nome)

# Função para calcular pontuação de risco (exemplo arbitrário)
def calcular_risco(row):
    # Quanto maior a idade, menor o risco (até certo ponto)
    # Quanto maior a renda, menor o risco
    risco = (100 - row['Idade']) + (10000 / row['Renda'])
    return round(risco, 2)

# Aplicando a função em cada linha do DataFrame (axis=1)
df_clientes['Pontuacao_Risco'] = df_clientes.apply(calcular_risco, axis=1)

print("\nDataFrame de Clientes com nomes padronizados e pontuação de risco:")
print(df_clientes)
```

> No contexto de **análise de dados reprodutível** e **storytelling com dados**, o .apply() é uma ferramenta valiosa. Ele permite que você encapsule lógicas de negócio complexas em funções claras, tornando seu código mais legível e fácil de manter.

Isso é crucial para que outros analistas (ou seu eu futuro) possam entender e replicar suas transformações, garantindo a transparência e a confiabilidade de suas análises.

Simplificando o Contínuo: Discretização de Variáveis (Binning)

Em muitos cenários de análise, temos variáveis contínuas, como idade, renda, temperatura ou tempo de serviço. Embora essas variáveis ofereçam granularidade, às vezes é mais útil agrupá-las em categorias ou "bins". Pense em como as pessoas são frequentemente categorizadas por faixas etárias (crianças, adolescentes, adultos, idosos) em vez de sua idade exata em anos, meses e dias. Essa técnica de agrupar dados contínuos em intervalos discretos é conhecida como **discretização** ou **binning**.

Simplifica a análise e visualização

É mais fácil identificar tendências em "clientes jovens" versus "clientes maduros" do que em idades individuais

Lida com outliers e ruído

Pode ajudar a reduzir o impacto de valores extremos nos dados

Melhora algoritmos de ML

Alguns algoritmos de *machine learning* funcionam melhor com variáveis categóricas

Imagine que você está analisando o desempenho de vendas e tem a "Idade do Cliente". Se você quiser ver se clientes mais jovens ou mais velhos compram mais, olhar para cada idade individualmente seria inviável. Agrupá-los em faixas etárias como "18-25", "26-40", "41-60" e "60+" torna a análise e a comunicação dos resultados muito mais claras e eficazes.

O Pandas oferece funções poderosas para realizar o binning de forma eficiente, como `pd.cut` e `pd.qcut`. Essas ferramentas permitem que você defina os limites dos seus bins ou que o Pandas os crie automaticamente com base na distribuição dos dados.

pd.cut e pd.qcut: As Ferramentas para o Binning

Existem duas abordagens principais para o binning no Pandas:

pd.cut

Permite que você defina os **limites dos bins** manualmente. É como cortar um bolo em fatias de tamanhos específicos que você predeterminou.

Útil quando: Você tem categorias predefinidas por regras de negócio (ex: faixas de renda para impostos, categorias de idade para marketing).

pd.qcut

Divide os dados em bins de **tamanhos iguais em termos de número de observações** (quantis). É como cortar o bolo de forma que cada pessoa receba a mesma quantidade.

Útil quando: Você quer garantir que cada categoria tenha um número similar de dados, balanceando grupos para análises.

Vamos ver um exemplo prático com pd.cut para criar faixas etárias:

```
# Exemplo de discretização de variáveis contínuas
dados_alunos = {
    'Nome': ['Ana', 'Bruno', 'Carla', 'Daniel', 'Eva', 'Felipe'],
    'Nota_Final': [75, 88, 62, 95, 70, 80],
    'Idade': [20, 25, 19, 30, 22, 28]
}
df_alunos = pd.DataFrame(dados_alunos)

# Definindo os limites das faixas etárias
bins_idade = [18, 25, 35, 50, 100]  # Limites: 18-25, 26-35, 36-50, 51-100
labels_idade = ['Jovem Adulto', 'Adulto', 'Meia Idade', 'Idoso']

# Criando a coluna 'Faixa_Etaria' usando pd.cut
df_alunos['Faixa_Etaria'] = pd.cut(df_alunos['Idade'],
                                   bins=bins_idade,
                                   labels=labels_idade,
                                   right=True)

# Exemplo com pd.qcut para Notas_Finais em 3 grupos (tercis)
df_alunos['Nivel_Nota'] = pd.qcut(df_alunos['Nota_Final'],
                                  q=3,
                                  labels=['Baixo', 'Médio', 'Alto'])

print("\nDataFrame de Alunos com faixas etárias e níveis de nota:")
print(df_alunos)
```

Conceito	Âmbito/Aplicação	Base/Origem	Exemplo
pd.cut	Agrupamento por intervalos definidos	Limites de corte pré-determinados	Faixas de renda (0-10k, 10k-20k, etc.)
pd.qcut	Agrupamento por quantis (número igual de dados)	Distribuição dos dados (percentis)	Divisão de clientes em 3 grupos de gastos

A discretização é uma técnica poderosa para o **storytelling com dados**. Ao transformar números complexos em categorias compreensíveis, você facilita a comunicação de *insights* para públicos não técnicos, tornando suas análises mais acessíveis e impactantes.

Consolidando o Conhecimento e Preparando para o Próximo Passo

Chegamos ao fim de uma jornada crucial na análise exploratória de dados! Nesta aula, você aprendeu que os dados brutos são apenas o ponto de partida. Vimos como a **transformação e criação de novas colunas** são essenciais para desbloquear o verdadeiro potencial de suas informações. Desde a simples alteração de tipos de dados com .astype(), passando pela criação de variáveis derivadas de operações e lógicas condicionais, até a aplicação de funções complexas com .apply() e a simplificação de dados contínuos com o **binning** (pd.cut e pd.qcut), você adquiriu um arsenal de ferramentas para moldar seus dados.

Em prática: Lembre-se que a manipulação de dados é um processo iterativo. Comece limpando e padronizando os tipos, depois explore a criação de novas *features* que respondam às suas perguntas de negócio, e por fim, considere a discretização para simplificar e visualizar melhor seus resultados. A prática constante com diferentes conjuntos de dados é a chave para a maestria.

Autoavaliação

1. Qual método do Pandas é mais adequado para converter uma coluna de texto contendo números para um tipo numérico (inteiro ou float)?
   - a) .convert_type()
   - b) .to_numeric()
   - c) .astype()
   - d) .change_data_type()
2. Você tem um DataFrame com colunas 'Preço' e 'Desconto'. Para criar uma nova coluna 'Preço_Final' que seja 'Preço' menos 'Desconto', qual a abordagem mais eficiente no Pandas?
   - a) Usar um loop for para iterar sobre as linhas e calcular.
   - b) Aplicar a função .apply() com uma função lambda.
   - c) Realizar a operação diretamente: df['Preço_Final'] = df['Preço'] - df['Desconto'].
   - d) Utilizar np.where com uma condição de subtração.
3. Quando você precisa aplicar uma lógica de negócio muito específica e complexa que envolve múltiplas colunas de um DataFrame, qual função do Pandas oferece a maior flexibilidade para isso?
   - a) .sum()
   - b) .mean()
   - c) .apply()
   - d) .groupby()
4. Qual a principal diferença entre pd.cut e pd.qcut ao realizar a discretização de variáveis contínuas?
   - a) pd.cut cria bins com número igual de observações, pd.qcut com limites definidos.
   - b) pd.cut é para variáveis categóricas, pd.qcut para contínuas.
   - c) pd.cut permite definir os limites dos bins, pd.qcut cria bins com número aproximadamente igual de observações.
   - d) Não há diferença significativa, são sinônimos.
5. Descreva um cenário real onde a criação de uma nova coluna a partir de dados existentes seria fundamental para uma análise de negócio. Explique qual coluna seria criada e por que ela seria importante.

Gabarito: 1. c) | 2. c) | 3. c) | 4. c)

Próxima Aula

Agora que você é um mestre na manipulação e transformação de dados, o próximo passo é aprender a comunicá-los de forma eficaz. Na **Aula 11 – Princípios do Design de Visualização de Dados**, exploraremos como transformar seus *insights* em gráficos e *dashboards* que contam uma história clara e impactante.

Recursos Adicionais:

- **Documentação oficial do Pandas**: Para aprofundar em cada função e seus parâmetros.
- **Livro "Python for Data Analysis" (Wes McKinney)**: Uma referência completa para manipulação de dados com Pandas.
- **Kaggle Notebooks**: Para ver exemplos práticos de aplicação dessas técnicas em projetos reais.

NOTA IMPORTANTE: As informações técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações ou novas funcionalidades das bibliotecas.