

# Machine Learning



**Aprofundando Algoritmos  
(Códigos)**

## **Bem-vindo ao nosso guia digital exclusivo!**

Este eBook tem como objetivo fornecer insights profundos e estratégias práticas para ajudá-lo a alcançar seus objetivos. Através de uma linguagem clara e acessível, buscamos descomplicar temas complexos, tornando o aprendizado mais dinâmico e eficaz.

Ao longo das próximas páginas, você terá acesso a um conteúdo cuidadosamente elaborado por especialistas da área, garantindo que cada capítulo seja rico em informações valiosas e dicas aplicáveis. Estamos comprometidos em oferecer a você as melhores práticas do setor, estudos de caso relevantes e ferramentas indispensáveis para o seu desenvolvimento pessoal ou profissional.

Siga as instruções, aproveite os exercícios propostos e esteja pronto para se transformar com o conhecimento que está prestes a adquirir. Boa leitura!

## **Aprofundando: Machine Learning com Python (Códigos)**

Para a prática é necessário desenvolver todas as etapas da metodologia de projetos de ciência de dados:

- Definição do problema/objetivos (Observatório de Dados);
- Coleta e entendimento dos dados (Bases alvos);
- Tratamento/preparação dos dados (Pré-Processamento);
- Análise exploratória (Mineração);
- Modelagem do algoritmo (Mineração);
- Análises do algoritmo (Análise dos Resultados);
- Otimização do modelo (tuning) e (Análise dos Resultados);
- Deploy (não necessariamente deploy em alguma plataforma, mas simular como executar o modelo em novos dados);



# Tópicos Aprofundados

- Algoritmos supervisionados
- Algoritmos de regressão
- Regressão simples
- Regressão múltipla
- Métricas de avaliação
- Métricas de avaliação
- Feature importances
- Execução do modelo
- Algoritmos de classificação
- Regressão logística
- Árvores de decisão
- Random forest
- Métricas de avaliação
- Feature importances
- Execução do modelo
- Algoritmos não-supervisionados
- Algoritmos de cluster
- Kmeans
- Automatizando escolha do K grupos
- Métricas(visualização) dos resultados - TSNE/PCA/UMAP
- Execução do modelo
- Executando modelos
- Salvando modelos em objetos
- JOBLIB (opção ou outro recurso)
- Simulando modelo em produção
- Carregar modelo, carregar dados(tratamento - pipeline), executar modelo, salvar resultados (csv).



# Algoritmos supervisionados

Neste exemplo, os dados consistem em uma variável independente (X), e uma variável dependente (y). Em seguida, os dados são divididos em conjuntos de treinamento e teste usando **train\_test\_split**. Um modelo de regressão linear é criado e treinado com os dados de treinamento usando **LinearRegression()** e **fit()**. As previsões são feitas no conjunto de teste usando **predict()**, e a métrica de erro médio quadrado (**MSE**) é calculada usando **mean\_squared\_error()** para avaliar o desempenho do modelo.

```
1  from sklearn.linear_model import LinearRegression
2  from sklearn.model_selection import train_test_split
3  from sklearn.metrics import mean_squared_error
4
5  # Dados de exemplo
6  X = [[1], [2], [3], [4], [5]] # Variável independente
7  y = [2, 4, 6, 8, 10] # Variável dependente
8
9  # Dividir os dados em conjunto de treinamento e conjunto de teste
10 X_train, X_test, y_train, y_test = train_test_split(X, y,
11 test_size=0.2, random_state=42)
12
13 # Criar e treinar o modelo de regressão linear
14 model = LinearRegression()
15 model.fit(X_train, y_train)
16
17 # Fazer previsões no conjunto de teste
18 y_pred = model.predict(X_test)
19
20 # Avaliar o desempenho do modelo usando a métrica de erro médio
21 quadrado (MSE)
22 mse = mean_squared_error(y_test, y_pred)
23 print("Erro médio quadrado (MSE):", mse)
```



# Algoritmos de regressão

Neste exemplo, os dados consistem em uma variável independente (X), e uma variável dependente com classes binárias (y). Os dados são divididos em conjuntos de treinamento e teste usando **train\_test\_split**. Um modelo de regressão logística é criado e treinado com os dados de treinamento usando **LogisticRegression()** e **fit()**. As previsões são feitas no conjunto de teste usando **predict()**, e a acurácia é calculada usando **accuracy\_score()** para avaliar o desempenho do modelo.

```
1  from sklearn.linear_model import LogisticRegression
2  from sklearn.model_selection import train_test_split
3  from sklearn.metrics import accuracy_score
4
5  # Dados de exemplo
6  X = [[1], [2], [3], [4], [5]] # Variável independente
7  y = [0, 0, 1, 1, 1] # Variável dependente (classes binárias)
8
9  # Dividir os dados em conjunto de treinamento e conjunto de teste
10 X_train, X_test, y_train, y_test = train_test_split(X, y,
11 test_size=0.2, random_state=42)
12
13 # Criar e treinar o modelo de regressão logística
14 model = LogisticRegression()
15 model.fit(X_train, y_train)
16
17 # Fazer previsões no conjunto de teste
18 y_pred = model.predict(X_test)
19
20 # Avaliar o desempenho do modelo usando a acurácia
21 accuracy = accuracy_score(y_test, y_pred)
22 print("Acurácia:", accuracy)
```



# Regressão simples

Neste exemplo, os dados de exemplo consistem em uma variável independente (X), e uma variável dependente (y). Um modelo de regressão linear é criado e treinado com os dados usando a classe **LinearRegression()** e o método **fit()**. Em seguida, são feitas previsões para novos valores de X usando o método **predict()**. As previsões são impressas para cada novo valor de X.

```
1 from sklearn.linear_model import LinearRegression
2 import numpy as np
3
4 # Dados de exemplo
5 X = np.array([[1], [2], [3], [4], [5]]) # Variável independente
6 y = np.array([2, 4, 6, 8, 10]) # Variável dependente
7
8 # Criar e treinar o modelo de regressão linear
9 model = LinearRegression()
10 model.fit(X, y)
11
12 # Fazer previsões para novos valores
13 new_X = np.array([[6], [7], [8]])
14 predictions = model.predict(new_X)
15
16 # Imprimir as previsões
17 for i in range(len(new_X)):
18     print("Para X =", new_X[i][0], "a previsão é y =",
19           predictions[i])
```

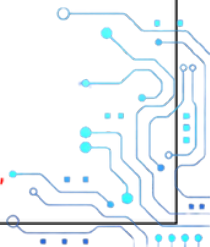


# Regressão múltipla

Neste exemplo, os dados de exemplo consistem em duas variáveis independentes (X) e uma variável dependente (y). Um modelo de regressão linear múltipla é criado e treinado com os dados usando a classe **LinearRegression()** e o método **fit()**. Em seguida, são feitas previsões para novos valores de X usando o método **predict()**. As previsões são impressas para cada novo valor de X.

Vale ressaltar que, para a regressão múltipla, os dados de entrada (X) devem ser uma matriz com várias colunas, cada uma representando uma variável independente diferente. O número de colunas em X deve corresponder ao número de coeficientes no modelo de regressão. A diferença entre regressão simples e regressão múltipla está relacionada ao número de variáveis independentes (ou preditoras) utilizadas para fazer a previsão da variável dependente (ou resposta). Na regressão múltipla, temos duas ou mais variáveis independentes sendo usadas para prever a variável dependente. Por exemplo, podemos ter um modelo de regressão múltipla que use a quantidade de horas de estudo e a idade do aluno como variáveis independentes para prever a nota na prova. Nesse caso, a relação entre as variáveis é modelada por um hiperplano em um espaço de dimensões mais altas.

```
1 from sklearn.linear_model import LinearRegression
2 import numpy as np
3
4 # Dados de exemplo
5
6 #Variável independente
7 X = np.array([[1, 3],[2, 6],[3, 9],[4, 12],[5, 15]])
8 #Variável dependente
9 y = np.array([2, 4, 6, 8, 10])
10
11 # Criar e treinar o modelo de regressão linear
12 model = LinearRegression()
13 model.fit(X, y)
14
15 # Fazer previsões para novos valores
16 new_X = np.array([[6], [7], [8]])
17 predictions = model.predict(new_X)
18
19 # Imprimir as previsões
20 for i in range(len(new_X)):
21     print("Para X =", new_X[i][0], "a previsão é y =",
22           predictions[i])
```



# Métricas de avaliação

Neste exemplo, os dados de exemplo consistem em uma variável independente (X), e uma variável dependente (y). Um modelo de regressão linear é criado e treinado com os dados usando a classe **LinearRegression()** e o método **fit()**. As previsões são feitas para os mesmos valores de X usando o método **predict()**.

Em seguida, as métricas de avaliação são calculadas usando as funções **mean\_absolute\_error()**, **mean\_squared\_error()**, **np.sqrt()** para calcular a raiz quadrada do **MSE** e **r2\_score()** para calcular o **R<sup>2</sup>**. Essas métricas são armazenadas em variáveis (**mae**, **mse**, **rmse**, **r2**) e são impressas na saída.

```
1  from sklearn.linear_model import LinearRegression
2  import numpy as np
3
4  # Dados de exemplo
5  X = np.array ([[1, 3],[2, 6],[3, 9],[4, 12],[5, 15]]) # Variável
6  independente
7  y = np.array ([2, 4, 6, 8, 10]) # Variável dependente
8
9  # Criar e treinar o modelo de regressão linear
10 model = LinearRegression()
11 model.fit(X, y)
12
13 # Fazer previsões para os mesmos valores de X
14 y_pred = model.predict(X)
15
16 # Calcular as métricas de avaliação
17 mae = mean_absolute_error(y, y_pred)
18 mse = mean_squared_error(y, y_pred)
19 rmse = np.sqrt(mse)
20 r2 = r2_score(y, y_pred)
21
22 # Imprimir as métricas de avaliação
23 print("MAE:", mae)
24 print("MSE:", mse)
25 print("RMSE:", rmse)
26 print("R²:", r2)
```





# Feature importances

Os dados de exemplo consistem em uma matriz de features (X) com 3 colunas e uma variável dependente (y). Um modelo de regressão de floresta aleatória é criado e treinado com os dados usando a classe **RandomForestRegressor()** e o método **fit()**.

As importâncias das features são obtidas utilizando o atributo **feature\_importances\_** do modelo treinado. Essas importâncias são armazenadas em um vetor (importances) e, em seguida, são impressas na saída.

Vale ressaltar que o cálculo das importâncias das features pode variar dependendo do algoritmo e do modelo utilizado. No exemplo abaixo, foi utilizado o algoritmo de floresta aleatória (Random Forest), mas outras técnicas e algoritmos também possuem métodos específicos para calcular a importância das features.

## Execução do modelo

```
1 from sklearn.ensemble import RandomForestRegressor
2 import numpy as np
3
4 # Dados de exemplo
5 X = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) # Matriz de
6 features
7 y = np.array([10, 20, 30]) # Variável dependente
8
9 # Criar e treinar o modelo de regressão de floresta aleatória
10 model = RandomForestRegressor()
11 model.fit(X, y)
12
13 # Obter as importâncias das features
14 importances = model.feature_importances_
15
16 # Imprimir as importâncias das features
17 for i, importance in enumerate(importances):
18     print("Feature", i+1, "importance:", importance)
```



# Algoritmos de classificação

Neste exemplo, utilizamos o conjunto de dados Iris, que é um conjunto de dados de exemplo amplamente utilizado para problemas de classificação. Primeiro, carregamos os dados usando a função **load\_iris()** da biblioteca Scikit-learn. Em seguida, dividimos os dados em conjuntos de treino e teste usando a função **train\_test\_split()**.

Em seguida, criamos o modelo de Árvore de Decisão usando a classe **DecisionTreeClassifier()** e o método **fit()** para treinar o modelo com os dados de treino.

Após treinar o modelo, fazemos previsões no conjunto de teste usando o método **predict()**. Em seguida, calculamos a acurácia do modelo comparando as previsões com os valores reais usando a função **accuracy\_score()**.

Por fim, imprimimos a acurácia do modelo na saída.

```
1  from sklearn.ensemble import RandomForestClassifier
2  from sklearn.metrics import accuracy_score
3  from sklearn.datasets import load_iris
4  from sklearn.model_selection import train_test_split
5
6  # Carregar o conjunto de dados de exemplo (Iris dataset)
7  data = load_iris()
8  X = data.data # Features
9  y = data.target # Classes
10
11 # Dividir o conjunto de dados em treino e teste
12 X_train, X_test, y_train, y_test = train_test_split(X, y,
13 test_size=0.2, random_state=42)
14
15 # Criar e treinar o modelo de Random Forest
16 model = RandomForestClassifier()
17 model.fit(X_train, y_train)
18
19 # Fazer previsões no conjunto de teste
20 predictions = model.predict(X_test)
21
22 # Calcular a acurácia do modelo
23 accuracy = accuracy_score(y_test, predictions)
24
25 # Imprimir a acurácia do modelo
26 print("Acurácia:", accuracy)
```



# Métricas de avaliação

## Acurácia:

```
1  from sklearn.ensemble import RandomForestClassifier
2  from sklearn.metrics import accuracy_score
3  from sklearn.datasets import load_iris
4  from sklearn.model_selection import train_test_split
5
6  # Carregar o conjunto de dados de exemplo (Iris dataset)
7  data = load_iris()
8  X = data.data # Features
9  y = data.target # Classes
10
11 # Dividir o conjunto de dados em treino e teste
12 X_train, X_test, y_train, y_test = train_test_split(X, y,
13 test_size=0.2, random_state=42)
14
15 # Criar e treinar o modelo de Random Forest
16 model = RandomForestClassifier()
17 model.fit(X_train, y_train)
18
19 # Fazer previsões no conjunto de teste
20 predictions = model.predict(X_test)
21
22 # Calcular a acurácia do modelo
23 accuracy = accuracy_score(y_test, predictions)
24
25 # Imprimir a acurácia do modelo
26 print("Acurácia:", accuracy)
```

---




# Matriz de Confusão

```
1 from sklearn.ensemble import RandomForestClassifier
2 from sklearn.metrics import confusion_matrix
3 from sklearn.datasets import load_iris
4 from sklearn.model_selection import train_test_split
5
6 # Carregar o conjunto de dados de exemplo (Iris dataset)
7 data = load_iris()
8 X = data.data # Features
9 y = data.target # Classes
10
11 # Dividir o conjunto de dados em treino e teste
12 X_train, X_test, y_train, y_test = train_test_split(X, y,
13 test_size=0.2, random_state=42)
14
15 # Criar e treinar o modelo de Random Forest
16 model = RandomForestClassifier()
17 model.fit(X_train, y_train)
18
19 # Fazer previsões no conjunto de teste
20 predictions = model.predict(X_test)
21
22 # Calcular a matriz de confusão do modelo
23 confusion = confusion_matrix(y_test, predictions)
24
25 # Imprimir a matriz de confusão
26 print("Matriz de Confusão:")
27 print(confusion)
```

# Relatório de Classificação

```
1 from sklearn.ensemble import RandomForestClassifier
2 from sklearn.metrics import classification_report
3 from sklearn.datasets import load_iris
4 from sklearn.model_selection import train_test_split
5
6 # Carregar o conjunto de dados de exemplo (Iris dataset)
7 data = load_iris()
8 X = data.data # Features
9 y = data.target # Classes
10
11 # Dividir o conjunto de dados em treino e teste
12 X_train, X_test, y_train, y_test = train_test_split(X, y,
13 test_size=0.2, random_state=42)
14
15 # Criar e treinar o modelo de Random Forest
16 model = RandomForestClassifier()
17 model.fit(X_train, y_train)
18
19 # Fazer previsões no conjunto de teste
20 predictions = model.predict(X_test)
21
22 # Gerar o relatório de classificação do modelo
23 report = classification_report(y_test, predictions)
24
25 # Imprimir o relatório de classificação
26 print("Relatório de Classificação:")
27 print(report)
```



# Feature importances

Abaixo é demonstrado como obter as importâncias das features em um modelo de Random Forest. No primeiro exemplo, obtemos as importâncias para uma única árvore da Random Forest usando `model.estimators_[0].feature_importances_`. No segundo exemplo, calculamos as importâncias médias das features em todas as árvores da Random Forest usando uma compreensão de lista e `np.mean()`.

## Obtendo as importâncias das features em uma única árvore da Random Forest:

```
1  from sklearn.ensemble import RandomForestClassifier
2  from sklearn.datasets import load_iris
3  import numpy as np
4
5  # Carregar o conjunto de dados de exemplo (Iris dataset)
6  data = load_iris()
7  X = data.data # Features
8  y = data.target # Classes
9
10 # Criar e treinar o modelo de Random Forest
11 model = RandomForestClassifier(n_estimators=100)
12 model.fit(X, y)
13
14 # Obter as importâncias das features para uma única árvore
15 tree_importances = model.estimators_[0].feature_importances_
16
17 # Imprimir as importâncias das features
18 for i, importance in enumerate(tree_importances):
19     print(f"Feature {i+1}: {importance}")
```

## Obtendo as importâncias das features em uma única árvore da Random Forest:

```
1  from sklearn.ensemble import RandomForestClassifier
2  from sklearn.datasets import load_iris
3  import numpy as np
4
5  # Carregar o conjunto de dados de exemplo (Iris dataset)
6  data = load_iris()
7  X = data.data # Features
8  y = data.target # Classes
9
10 # Criar e treinar o modelo de Random Forest
11 model = RandomForestClassifier(n_estimators=100)
12 model.fit(X, y)
13
14 # Obter as importâncias médias das features em todas as árvores
15 average_importances = np.mean([tree.feature_importances_ for tree
16 in model.estimators_], axis=0)
17
18 # Imprimir as importâncias das features
19 for i, importance in enumerate(average_importances):
20     print(f"Feature {i+1}: {importance}")
```



# Execução do modelo

Nesse exemplo, estamos usando o conjunto de dados Iris para ilustrar a execução de um modelo de Random Forest. O conjunto de dados é dividido em conjuntos de treino e teste usando a função **train\_test\_split**. Em seguida, criamos um objeto de modelo de Random Forest com **RandomForestClassifier** e o treinamos usando os dados de treino.

Após o treinamento, é feito as previsões no conjunto de teste usando **predict** e calculamos a acurácia do modelo usando **accuracy\_score**. Por fim, imprimimos a acurácia do modelo na saída.

```
1  from sklearn.ensemble import RandomForestClassifier
2  from sklearn.datasets import load_iris
3  from sklearn.model_selection import train_test_split
4  from sklearn.metrics import accuracy_score
5
6
7  # Carregar o conjunto de dados de exemplo (Iris dataset)
8  data = load_iris()
9  X = data.data # Features
10 y = data.target # Classes
11
12 # Dividir o conjunto de dados em treino e teste
13 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
14
15 # Criar e treinar o modelo de Random Forest
16 model = RandomForestClassifier(n_estimators=100)
17 model.fit(X_train, y_train)
18
19 # Fazer previsões no conjunto de teste
20 predictions = model.predict(X_test)
21
22 # Calcular a acurácia do modelo
23 accuracy = accuracy_score(y_test, predictions)
24
25 # Imprimir a acurácia do modelo
26 print("Acurácia:", accuracy)
```



# Algoritmos não-supervisionados

Nesse exemplo, estamos usando a função **make\_blobs** do Scikit-learn para gerar um conjunto de dados de exemplo com 200 pontos e 4 clusters. Em seguida, criamos um objeto de modelo de **K-Means** com **KMeans** especificando o número desejado de clusters.

Após criar o modelo, treinamos ele usando o método **fit** com os dados de entrada **X**. Em seguida, usamos o método **predict** para obter as previsões de cluster para cada ponto de dado.

Finalmente, usamos o **matplotlib** para plotar os pontos de dados, colorindo-os de acordo com o cluster atribuído. Essa visualização nos permite observar a separação dos clusters pelo algoritmo de clustering K-Means.

```
1 from sklearn.cluster import KMeans
2 from sklearn.datasets import make_blobs
3 import matplotlib.pyplot as plt
4
5 # Gerar um conjunto de dados de exemplo
6 X, y = make_blobs(n_samples=200, centers=4, random_state=0)
7
8 # Criar o modelo de K-Means com 4 clusters
9 model = KMeans(n_clusters=4, random_state=0)
10
11 # Treinar o modelo
12 model.fit(X)
13
14 # Obter as previsões dos clusters
15 predictions = model.predict(X)
16
17 # Plotar os pontos de dados coloridos por cluster
18 plt.scatter(X[:, 0], X[:, 1], c=predictions)
19 plt.title("K-Means Clustering")
20 plt.xlabel("Feature 1")
21 plt.ylabel("Feature 2")
22 plt.show()
```





# Algoritmos de cluster

Nesse exemplo, estamos usando a função **make\_blobs** do Scikit-learn para gerar um conjunto de dados de exemplo com 100 pontos e 3 clusters. Em seguida, criamos um objeto de modelo de clustering hierárquico aglomerativo com **AgglomerativeClustering**, especificando o número desejado de clusters.

Após criar o modelo, ajustamos ele aos dados usando o método **fit** com os dados de entrada X. Em seguida, usamos o atributo **labels\_** para obter os rótulos dos clusters atribuídos a cada ponto de dado.

Finalmente, imprimimos os rótulos dos clusters para visualizar a atribuição dos pontos de dados aos clusters pelo algoritmo de clustering hierárquico aglomerativo.

```
1  from sklearn.cluster import AgglomerativeClustering
2  from sklearn.datasets import make_blobs
3
4  # Gerar um conjunto de dados de exemplo
5  X, y = make_blobs(n_samples=100, centers=3, random_state=42)
6
7  # Criar o modelo de clustering hierárquico aglomerativo com 3
8  clusters
9  model = AgglomerativeClustering(n_clusters=3)
10
11 # Ajustar o modelo aos dados
12 model.fit(X)
13
14 # Obter os rótulos dos clusters
15 labels = model.labels_
16
17 # Imprimir os rótulos dos clusters
18 print("Rótulos dos clusters:")
19 print(labels)
```





# Automatizando escolha do K grupos

Neste exemplo, usamos a função **make\_blobs** do Scikit-learn para gerar um conjunto de dados de exemplo com 200 pontos e 6 clusters. Em seguida, criamos um loop para testar diferentes valores de K, variando de 1 a 9.

Dentro do loop, criamos um modelo de **K-Means** para cada valor de K e ajustamos aos dados usando o método **fit**. Em seguida, calculamos o valor da métrica de avaliação "inércia" usando o atributo **inertia\_** do modelo e o armazenamos na lista **inertia\_values**.

Finalmente, plotamos a curva de cotovelo, onde o eixo x representa o número de grupos (K) e o eixo y representa a inércia. O ponto de inflexão na curva é usado para determinar o número adequado de grupos.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.cluster import KMeans
4 from sklearn.datasets import make_blobs
5
6 # Gerar um conjunto de dados de exemplo
7 X, y = make_blobs(n_samples=200, centers=6, random_state=42)
8
9 # Lista vazia para armazenar as métricas de avaliação
10 inertia_values = []
11
12 # Testar diferentes valores de K
13 k_values = range(1, 10)
14 for k in k_values:
15     # Criar o modelo de K-Means com K clusters
16     model = KMeans(n_clusters=k, random_state=42)
17     model.fit(X)
18
19     # Obter o valor da métrica de avaliação (inércia)
20     inertia_values.append(model.inertia_)
21
22 # Plotar a curva de cotovelo
23 plt.plot(k_values, inertia_values, marker='o')
24 plt.xlabel('Número de grupos (K)')
25 plt.ylabel('Inércia')
26 plt.title('Método do Cotovelo para Escolha de K')
27 plt.show()
```



# Métricas(visualização) dos resultados - TSNE/PCA/UMAP

TSNE, PCA e UMAP são consideradas métricas de visualização porque permitem mapear dados de alta dimensão para espaços de menor dimensão, facilitando a visualização e a compreensão das relações e estruturas presentes nos dados. Ao reduzir a dimensionalidade dos dados, elas ajudam a tornar os padrões e agrupamentos mais evidentes, permitindo uma análise visual mais clara e intuitiva.

## **TSNE (t-Distributed Stochastic Neighbor Embedding):**

É uma técnica de redução de dimensionalidade utilizada para visualização de dados complexos em espaços bidimensionais ou tridimensionais, preservando as relações de similaridade entre os pontos.

## **PCA (Principal Component Analysis):**

É uma técnica clássica de redução de dimensionalidade que busca encontrar as direções principais de maior variância nos dados. É usado para visualização, remoção de redundância e identificação das principais características dos dados.

## **UMAP (Uniform Manifold Approximation and Projection):**

É uma técnica de redução de dimensionalidade que preserva a estrutura dos dados em um espaço de menor dimensão. É especialmente útil para preservar estruturas complexas e relações de vizinhança local, sendo amplamente usado em visualização e análise exploratória de dados.



# Execução do modelo

Neste exemplo, estamos usando a função **make\_blobs** do Scikit-learn para gerar um conjunto de dados de exemplo com 100 pontos e 3 clusters. Em seguida, criamos um modelo de K-Means com 3 clusters e ajustamos aos dados.

Em seguida, usamos as técnicas de redução de dimensionalidade para reduzir os dados de alta dimensionalidade para 2 dimensões. O TSNE é aplicado usando a classe **TSNE**, o PCA é aplicado usando a classe **PCA**, e o UMAP é aplicado usando a classe **UMAP**.

Depois de reduzir a dimensionalidade, plotamos os resultados em três subplots diferentes, um para cada técnica de redução de dimensionalidade. Cada ponto de dado é colorido de acordo com o cluster atribuído pelo algoritmo de clustering.

Dessa forma, podemos visualizar os dados em um espaço de duas dimensões e analisar a estrutura dos clusters formados pelo algoritmo de clustering.

## Executando modelos

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from sklearn.datasets import make_blobs
4  from sklearn.cluster import KMeans
5  from sklearn.manifold import TSNE
6  from sklearn.decomposition import PCA
7  from umap import UMAP
8
9  # Gerar um conjunto de dados de exemplo
10 X, y = make_blobs(n_samples=100, centers=3, random_state=42)
11
12 # Criar o modelo de K-Means com 3 clusters
13 model = KMeans(n_clusters=3, random_state=42)
14 model.fit(X)
15
16 # Reduzir a dimensionalidade com TSNE
17 tsne = TSNE(n_components=2, random_state=42)
18 X_tsne = tsne.fit_transform(X)
19
20 # Reduzir a dimensionalidade com PCA
21 pca = PCA(n_components=2, random_state=42)
22 X_pca = pca.fit_transform(X)
23
24 # Reduzir a dimensionalidade com UMAP
25 umap = UMAP(n_components=2, random_state=42)
26 X_umap = umap.fit_transform(X)
```



```

28 # Plotar os resultados
29 plt.figure(figsize=(12, 4))
30
31 plt.subplot(1, 3, 1)
32 plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=model.labels_)
33 plt.title("TSNE")
34 plt.xlabel("Componente 1")
35 plt.ylabel("Componente 2")
36
37 plt.subplot(1, 3, 2)
38 plt.scatter(X_pca[:, 0], X_pca[:, 1], c=model.labels_)
39 plt.title("PCA")
40 plt.xlabel("Componente 1")
41 plt.ylabel("Componente 2")
42
43 plt.subplot(1, 3, 3)
44 plt.scatter(X_umap[:, 0], X_umap[:, 1], c=model.labels_)
45 plt.title("UMAP")
46 plt.xlabel("Componente 1")
47 plt.ylabel("Componente 2")
48
49 plt.tight_layout()
50 plt.show()

```

## Salvando modelos em objetos

Para salvar modelos em objetos, você pode usar a biblioteca **pickle** em Python. A biblioteca pickle permite serializar e salvar objetos Python em arquivos, preservando sua estrutura e estado.

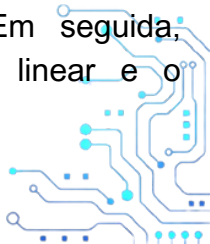
Aqui está um exemplo de como salvar um modelo em um objeto usando **pickle**:

```

1  import pickle
2  from sklearn.linear_model import LinearRegression
3
4  # Criar um modelo de regressão linear
5  model = LinearRegression()
6
7  # Treinar o modelo com dados de treinamento
8
9  # Salvar o modelo em um arquivo
10 with open('modelo.pkl', 'wb') as file:
11     pickle.dump(model, file)

```

Nesse exemplo, primeiro importamos a biblioteca pickle e o modelo LinearRegression da biblioteca sklearn. Em seguida, criamos uma instância do modelo de regressão linear e o treinamos com os dados de treinamento.



Depois disso, usamos a função **pickle.dump()** para salvar o modelo em um arquivo chamado "**modelo.pkl**". O argumento 'wb' indica que o arquivo será aberto para escrita em modo binário.

Agora, o modelo foi salvo como um objeto serializado no arquivo "**modelo.pkl**". Você pode carregar o modelo novamente em outro momento usando a biblioteca pickle da seguinte maneira:

```
1 import pickle
2
3 # Carregar o modelo a partir do arquivo
4 with open('modelo.pkl', 'rb') as file:
5     loaded_model = pickle.load(file)
6
7 # Usar o modelo carregado para fazer previsões
```

Nesse exemplo, usamos a função **pickle.load()** para carregar o modelo a partir do arquivo "**modelo.pkl**". O argumento 'rb' indica que o arquivo será aberto para leitura em modo binário.

Agora, você pode usar o modelo carregado para fazer previsões ou realizar outras tarefas desejadas.

Lembre-se de que, ao usar o **pickle**, é importante garantir que os arquivos sejam armazenados e recuperados de maneira segura, pois os objetos salvos podem conter informações sensíveis ou executáveis.



## JOBLIB (opção ou outro recurso)


Neste exemplo, temos uma função chamada **process\_data** que executa algum processamento intensivo em um conjunto de dados. A função **Parallel** é usada para paralelizar a execução dessa função, e a função **delayed** é usada para encapsular a chamada da função **process\_data** em cada elemento do conjunto de dados.

O argumento **n\_jobs** define o número de processos em paralelo a serem executados. O valor **-1** indica que o Joblib deve utilizar todos os núcleos de CPU disponíveis para processamento paralelo.

Ao executar esse código, a função **process\_data** será chamada em paralelo para cada elemento do conjunto de dados **input\_data**. Os resultados processados serão armazenados na lista **processed\_results**. No final, os resultados são exibidos.

É importante notar que o Joblib também oferece a possibilidade de usar a função **Memory** para armazenar resultados intermediários em cache, mas isso envolve um uso mais avançado e configuração adicional.

```
1  from joblib import Parallel, delayed
2
3  # Função a ser executada em paralelo
4  def process_data(data):
5      # Processamento intensivo
6      # ...
7      return processed_data
8
9  # Dados de entrada
10 input_data = [1, 2, 3, 4, 5]
11
12 # Executando a função process_data em paralelo
13 processed_results =
14 Parallel(n_jobs=-1)(delayed(process_data)(data) for data in
15 input_data)
16
17 # Exibindo os resultados
18 print(processed_results)
```



# Simulando modelo em produção

Segue um exemplo passo a passo de como você pode criar um modelo de Machine Learning em Python usando o JupyterLab. Vamos usar a biblioteca Scikit-learn para criar um modelo de classificação usando o conjunto de dados Iris.

## Preparação do ambiente:

- Certifique-se de ter o Python instalado em seu sistema. Recomenda-se usar uma distribuição como Anaconda, que já inclui muitas bibliotecas populares de Machine Learning.
- Instale o JupyterLab em seu ambiente Python usando o gerenciador de pacotes, como pip ou conda.

## Importação das bibliotecas:

- Abra o JupyterLab em seu navegador.
- Crie um novo notebook Python.
- Importe as bibliotecas necessárias para Machine Learning, como scikit-learn, pandas e numpy. Você pode usar comandos como `import sklearn`, `import pandas as pd`, `import numpy as np`, entre outros.

```
1 # Passo 1: Importação das bibliotecas
2 import pandas as pd
3 from sklearn.model_selection import train_test_split
4 from sklearn.linear_model import LogisticRegression
5 from sklearn.metrics import accuracy_score
```

## Carregamento e pré-processamento dos dados:

- Carregue os dados que serão usados para simular o modelo em produção. Isso pode ser feito lendo um arquivo CSV ou qualquer outra fonte de dados adequada.
- Realize o pré-processamento necessário nos dados, como limpeza, tratamento de valores ausentes, normalização, codificação de variáveis categóricas, etc. Use as funções e métodos apropriados das bibliotecas utilizadas.



# Simulando modelo em produção

```
1 # Passo 2: Carregamento e pré-processamento dos dados
2 # Vamos utilizar a base de dados "iris" embutida no scikit-learn
3 from sklearn.datasets import load_iris
4 iris = load_iris()
5 # Convertemos os dados em um dataframe do pandas para facilitar a
  manipulação
6 df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
7 df['target'] = iris.target
```

## Treinamento do modelo:

- Divida os dados em conjuntos de treinamento e teste usando a função `train_test_split()` do scikit-learn.
- Escolha um algoritmo de Machine Learning adequado para seu problema e crie uma instância desse modelo.
- Ajuste o modelo aos dados de treinamento usando o método `fit()`.

```
1 # Passo 3: Divisão dos dados em treinamento e teste
2 X = df.drop('target', axis=1)
3 y = df['target']
4
5 X_train, X_test, y_train, y_test = train_test_split(X, y,
6 test_size=0.2, random_state=42)
7
8 # Treinamento do modelo
9 model = LogisticRegression(max_iter=1000)
10 model.fit(X_train, y_train)
```

## Simulação do modelo em produção:

- Carregue dados de simulação que representem novos exemplos que seu modelo encontrará em produção. Certifique-se de pré-processar esses dados da mesma forma que fez durante o treinamento.
- Use o modelo treinado para fazer previsões nos dados de simulação usando o método `predict()`.
- Analise e interprete os resultados obtidos pelas previsões. Pode ser útil visualizar as previsões ou calcular métricas de avaliação para avaliar o desempenho do modelo.

```
1 # Passo 4: Simulação do modelo em produção
2 y_pred = model.predict(X_test)
```



# Simulando modelo em produção

## Avaliação dos resultados e salvando resultados:

- Após calcular a acurácia do modelo, crie um dataframe chamado "results" que contém as labels verdadeiras e as labels preditas.
- Salve os resultados da simulação em um formato adequado, como um arquivo CSV ou qualquer outro formato relevante para sua aplicação.
- Assim, a avaliação dos resultados é realizada e os resultados são salvos em um arquivo CSV para posterior análise ou uso.

```
1 # Passo 5: Avaliação dos resultados e salvando resultados
2 accuracy = accuracy_score(y_test, y_pred)
3 print("Acurácia do modelo:", accuracy)
4
5 # Salvando resultados em um arquivo CSV
6 results = pd.DataFrame({'True Labels': y_test, 'Predicted
7 Labels': y_pred})
8 results.to_csv('resultados.csv', index=False)
```

## Carregar modelo, carregar dados(tratamento - pipeline), executar modelo, salvar resultados (csv)

### Carregar modelo:

- Importe a biblioteca necessária para carregar o modelo (por exemplo, scikit-learn ou TensorFlow).
- Carregue o arquivo do modelo treinado usando a função apropriada do framework ou biblioteca. Por exemplo, para carregar um modelo treinado com o scikit-learn, você pode usar `joblib.load()` ou `pickle.load()`.
- Certifique-se de ter o modelo treinado salvo em um arquivo adequado.



# **Carregar modelo, carregar dados(tratamento - pipeline), executar modelo, salvar resultados (csv)**

## **Carregar dados e aplicar tratamento (pipeline):**

- Importe as bibliotecas necessárias para manipulação de dados (por exemplo, pandas, numpy).
- Carregue os dados em um formato adequado (como um dataframe do pandas ou uma matriz numpy).
- Aplique quaisquer etapas de pré-processamento ou tratamento de dados necessárias para preparar os dados para o modelo. Isso pode incluir transformações, limpeza, normalização, codificação de variáveis categóricas, entre outros.
- Certifique-se de que as etapas de pré-processamento sejam consistentes com as etapas aplicadas durante o treinamento do modelo.

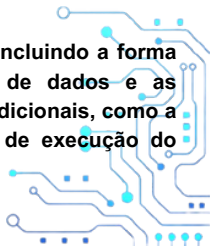
## **Executar o modelo e obter resultados:**

- Passe os dados pré-processados para o modelo carregado e execute-o para obter as previsões ou resultados desejados.
- Guarde os resultados em uma variável.

## **Salvar resultados em um arquivo CSV:**

- Importe a biblioteca necessária para manipulação de arquivos (por exemplo, pandas).
- Crie um dataframe do pandas ou uma estrutura de dados adequada para armazenar os resultados obtidos.
- Salve o dataframe ou estrutura de dados em um arquivo CSV usando a função `to_csv()` do pandas ou uma função semelhante.
- Especifique o caminho e o nome do arquivo onde deseja salvar os resultados.

É importante adaptar essas etapas de acordo com o seu caso específico, incluindo a forma como o modelo foi treinado, as necessidades de pré-processamento de dados e as bibliotecas/frameworks utilizados. Além disso, você pode adicionar etapas adicionais, como a validação dos resultados ou o tratamento de erros durante o processo de execução do modelo.



# Conclusão das práticas

Ao final deste eBook, consolidamos uma compreensão significativa das práticas abordadas, evidenciando seu impacto e relevância em nosso campo de estudo ou contexto de trabalho. As técnicas detalhadas não apenas promovem uma eficiência operacional melhorada, mas também fortalecem a base sobre a qual construímos estratégias inovadoras e soluções sustentáveis.

As metodologias aplicadas demonstraram a importância de uma abordagem metódica e bem fundamentada, a qual é crucial para alcançar resultados precisos e confiáveis. Através das práticas discutidas, identificamos pontos-chave para otimização de processos, ressaltando a necessidade de uma constante avaliação e adaptação às novas descobertas e tecnologias emergentes.

Este eBook serviu não apenas para instruir técnicas avançadas, mas também para inspirar um pensamento crítico e uma reflexão contínua sobre as melhores práticas em nossa área. O conhecimento adquirido aqui é um alicerce para futuras investigações e experimentações, incentivando a persistente busca pela excelência e inovação.

Em suma, as práticas executadas e discutidas proporcionam um forte entendimento teórico e prático que serve como uma ferramenta vital para futuras aplicações. À medida que continuamos a avançar em nossas respectivas áreas, a implementação consciente e criteriosa destas práticas será indispensável para o progresso e o sucesso contínuo.

