

Aula 5 - Classificação - Aprendizado Supervisionado com Scykit-Learn

Antes de usar aprendizado supervisionado

- Requisitos:
 - Nenhum valor ausente
 - Dados em formato numérico
 - Dados armazenados em um DataFrame do pandas ou em um array do NumPy
- Realize primeiro uma Análise Exploratória de Dados (EDA)

Sintaxe do scikit-learn

```
from sklearn.module import Model # Uma instância do modelo é criada. model
= Model() # O modelo é treinado usando os dados X (features) e y (rótulo
s). model.fit(X, y) # O modelo faz previsões com base em novos dados (X_ne
w). predictions = model.predict(X_new) # As previsões são impressas no for
mato de um array NumPy. print(predictions)
```

Saída:

```
array([0, 0, 0, 0, 1, 0])
```

O **scikit-learn** mantém uma estrutura previsível para diferentes algoritmos de aprendizado supervisionado. O modelo é treinado com um conjunto de dados (**x** , **y**), faz previsões com novos exemplos (**x_new**) e retorna um array de resultados. No caso de um **classificador binário**, o retorno contém **0s** e **1s** , indicando a classificação de cada observação.

Classificação Binária

Existem dois tipos de aprendizado supervisionado: **classificação** e **regressão**.

A **classificação binária** é usada para prever uma variável-alvo que possui **apenas dois rótulos**, normalmente representados numericamente como **0 ou 1**.

Abaixo está a exibição das primeiras linhas (`.head()`) do conjunto de dados

`churn_df` . Você pode esperar que o restante dos dados tenha valores semelhantes:

account_length	total_day_charge	total_eve_charge	total_night_charge	total_intl_charge	customer_...
101	45.85	17.65	9.64	1.22	3
73	22.30	9.05	9.98	2.75	2
86	24.62	17.53	11.49	3.13	4
59	34.73	21.02	9.66	3.24	1
129	27.42	18.75	10.11	2.59	1

Observação sobre o conjunto de dados:

Observando esses dados, qual coluna poderia ser a variável-alvo para **classificação binária**?

- ☐ "customer_service_calls"
- ☐ "total_night_charge"
- ☐ "churn"
- ☐ "account_length"

O desafio da classificação

- O aprendizado supervisionado usa **rótulos**.
- Vamos aprender a construir um **modelo de classificação** para prever rótulos de novos dados.

Classificando dados não vistos

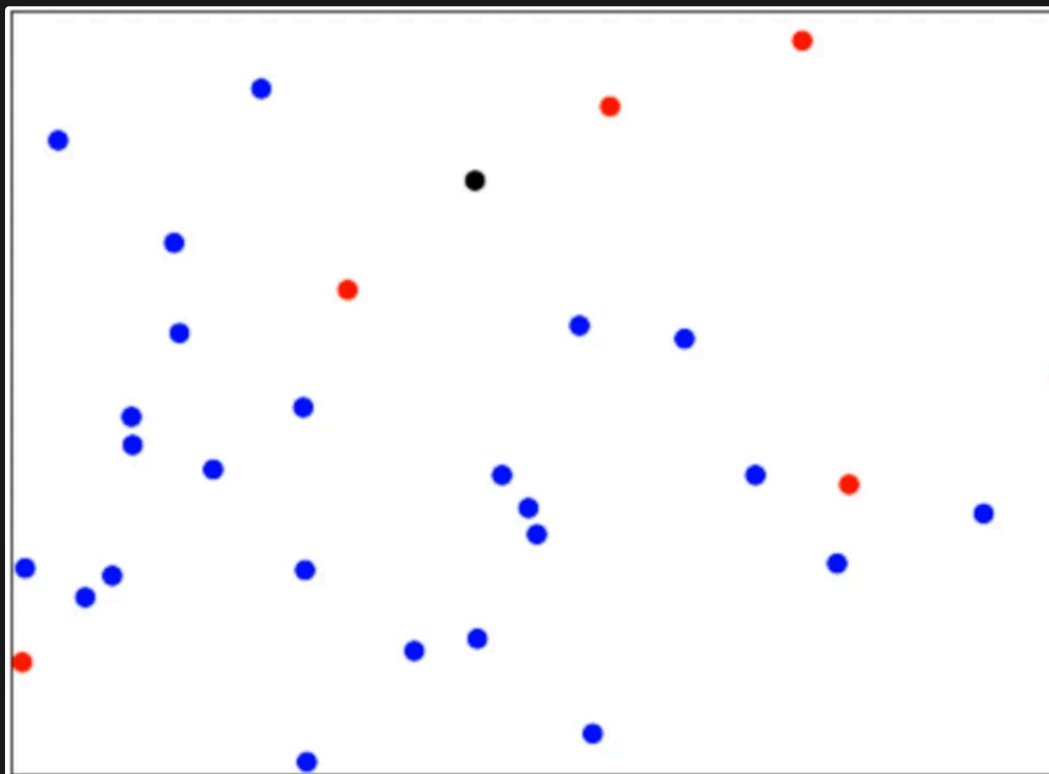
- Quatro passos da classificação:
 1. Construímos um **classificador** que aprende a partir dos dados rotulados.
 2. Passamos ao modelo dados **não rotulados**.
 3. O classificador **prevê os rótulos** desses novos dados.
 4. O conjunto de dados rotulados usados no aprendizado é chamado de **dados de treinamento**.

k-Nearest Neighbors (KNN)

- Primeiro modelo: **k-Nearest Neighbors (KNN)**, um algoritmo popular para classificação.
- **Ideia central:** prever o rótulo de um dado com base nos **k vizinhos mais próximos**.
- Exemplo: se $k = 3$, o rótulo é determinado pelo **voto da maioria** entre os três vizinhos mais próximos.

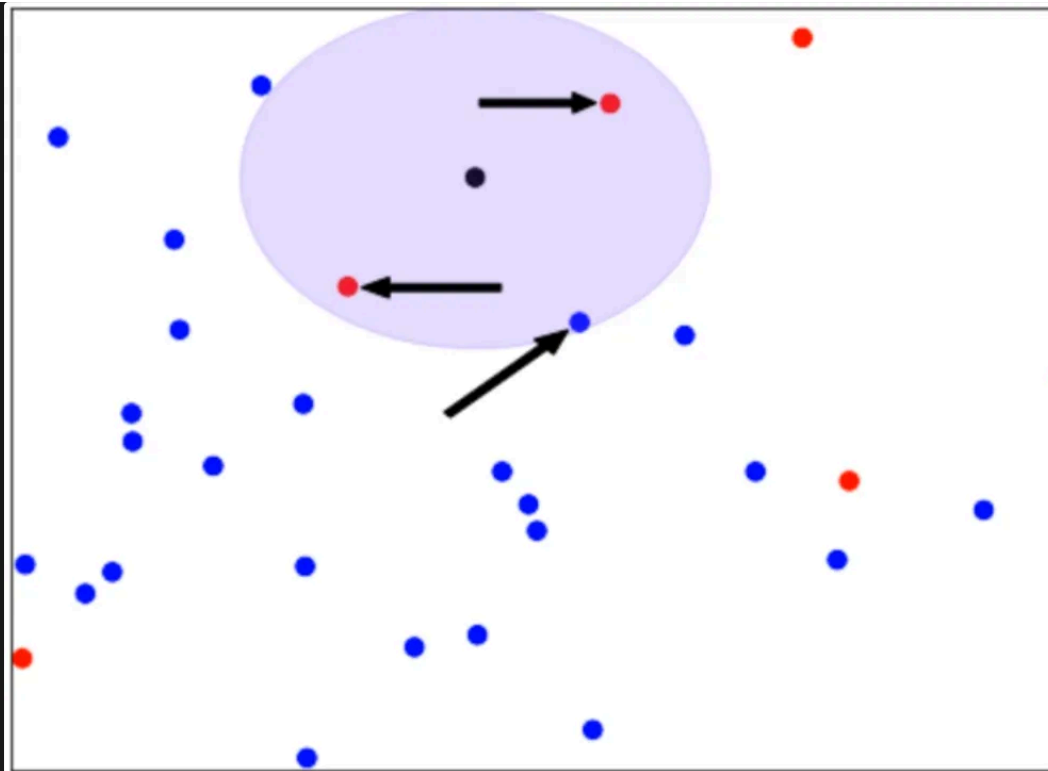
k-Nearest Neighbors (KNN) na prática

- Exemplo de classificação: Como classificar a observação preta no gráfico



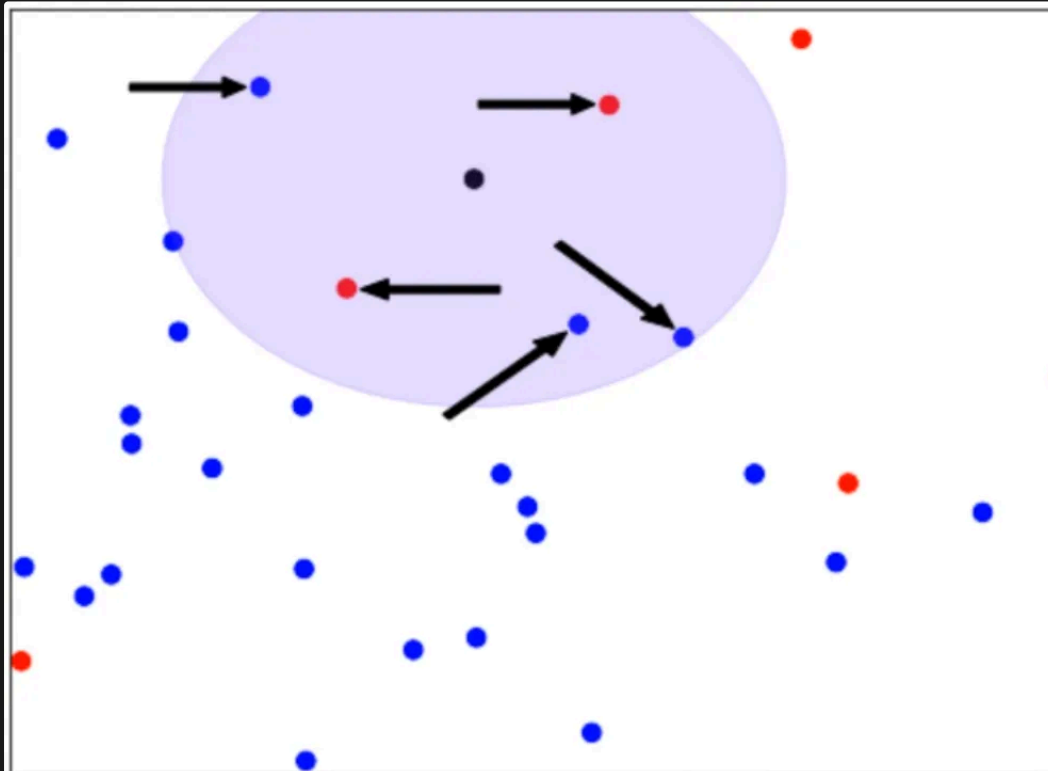
Classificação com $k = 3$

- Se $k = 3$, a observação é classificada como **vermelha**, pois 2 dos 3 vizinhos mais próximos são vermelhos.



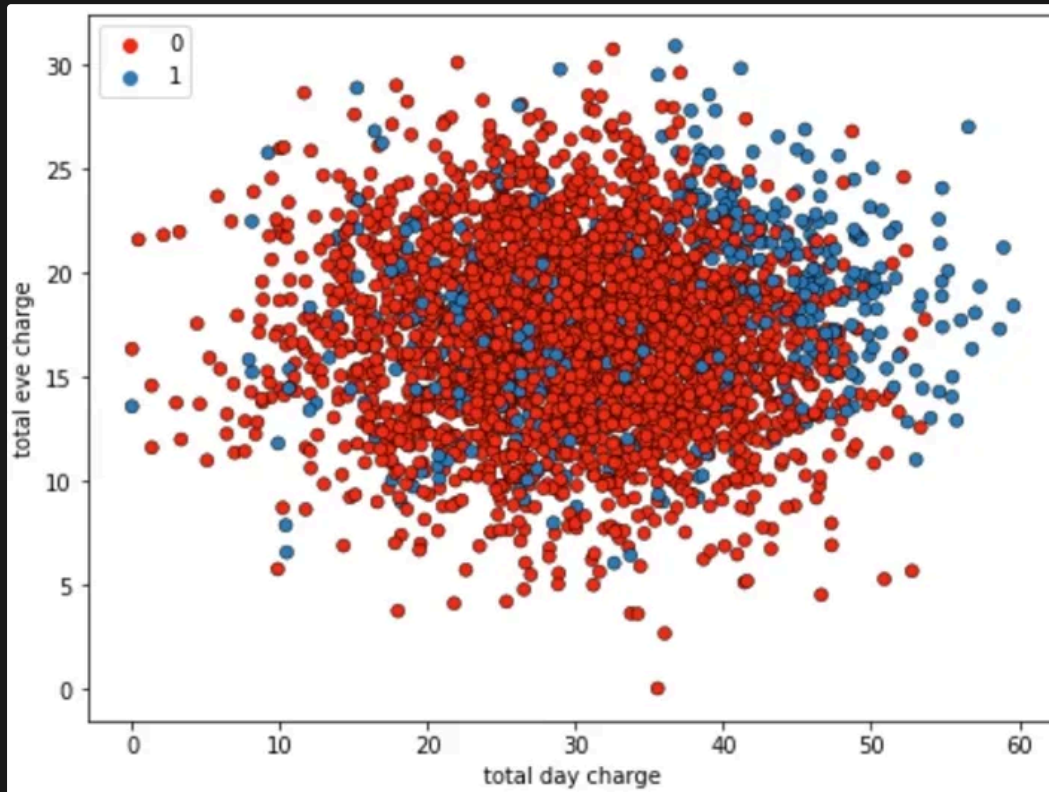
Classificação com $k = 5$

- Se $k = 5$, a observação seria classificada como azul.



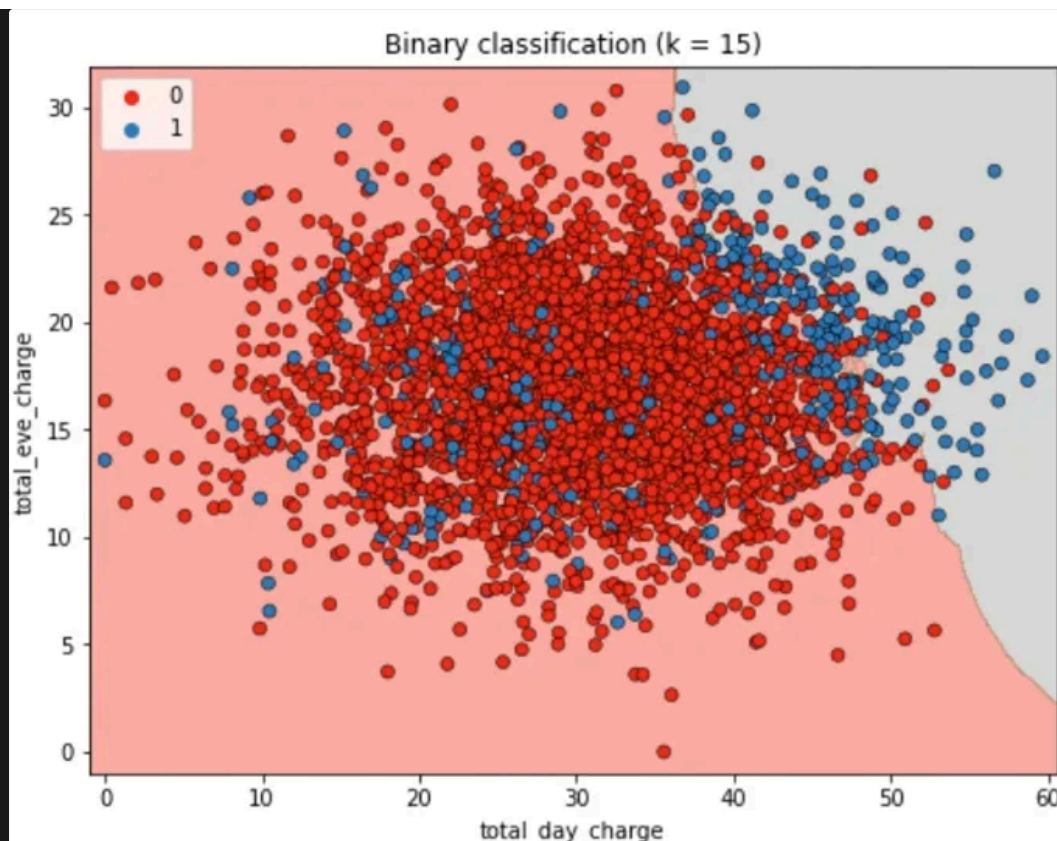
Intuição por trás do KNN

- Gráfico: total de cobranças diurnas vs. cobranças noturnas de clientes de uma empresa de telecomunicações.
- Azul → Clientes que cancelaram o serviço (churn).
- Vermelho → Clientes que não cancelaram.




Como o KNN define padrões

- Aplicação do KNN com $k = 15$.
- O algoritmo cria um **limite de decisão** para prever o cancelamento.
- Áreas **cinza** → Previsão de cancelamento.
- Áreas **vermelhas** → Previsão de não cancelamento.
- O modelo usa esse limite para prever novos dados.



Usando scikit-learn para treinar um classificador

Dataset:

 telecom_churn_clean.csv 258.6KB

- Importamos `KNeighborsClassifier` do scikit-learn.
- Dividimos os dados:
 - `x` : Matriz 2D com as features.
 - `y` : Vetor 1D com os rótulos (churn ou não churn).
- O scikit-learn exige que os features estejam em colunas e os dados em linhas.
- Conversão para NumPy (`.values`).
- Criamos o modelo com `KNeighborsClassifier(n_neighbors=15)` .
- Treinamos o modelo com `.fit(X, y)` .

```
from sklearn.neighbors import KNeighborsClassifier X =  
churn_df[["total_day_charge", "total_eve_charge"]].values y =  
churn_df["churn"].values print(X.shape, y.shape)
```

Saída:

```
(3333, 2), (3333,)
```

```
knn = KNeighborsClassifier(n_neighbors=15) knn.fit(X, y)
```

Fazendo previsões em dados não rotulados

- Temos um novo conjunto de dados `X_new`.
- Ele contém três observações e duas features.
- Utilizamos o método `.predict(X_new)`.
- O modelo retorna previsões:
 - 1 → Churn (cancelamento).
 - 0 → Não churn (não cancelamento).

```
X_new = np.array([[56.8, 17.5], # primeira observação [24.4, 24.1], #  
segunda observação [50.1, 10.9]]) # terceira observação print(X_new.shape)
```

```
predictions = knn.predict(X_new) print('Predictions:  
{ }'.format(predictions))
```

Vamos praticar!

- Agora, vamos construir nosso **próprio modelo KNN** para o conjunto de dados churn!

- Esse modelo será usado ao longo do restante da aula.

Exercício

k-Nearest Neighbors: Ajuste do Modelo

Neste exercício, você construirá seu **primeiro modelo de classificação** usando o conjunto de dados `churn_df`, que já foi pré-carregado para o restante do capítulo.

O alvo, `"churn"`, deve ser uma única coluna contendo o **mesmo número de observações** que os dados de características (*features*). Os dados de *features* já foram convertidos em arrays do NumPy.

As colunas `"account_length"` e `"customer_service_calls"` são tratadas como **features** porque:

- `"account_length"` indica **lealdade do cliente**.
- `"customer_service_calls"` pode indicar **insatisfação**, sendo um bom preditor de cancelamento.

Instruções

1. Importe `KNeighborsClassifier` de `sklearn.neighbors`.
2. Instancie um classificador `KNeighborsClassifier` chamado `knn` com 6 vizinhos.
3. Treine o classificador nos dados usando o método `.fit()`.

```
# Import KNeighborsClassifier from _____._____ import _____ y =
churn_df["churn"].values X = churn_df[["account_length",
"customer_service_calls"]].values # Create a KNN classifier with 6
neighbors knn = _____(_____ = _____) # Fit the classifier to the data
knn._____(_____, _____)
```

Exercício

k-Nearest Neighbors: Predição

Agora que você ajustou um classificador KNN, pode usá-lo para **prever os rótulos de novos pontos de dados**.

Todo o conjunto de dados disponível foi utilizado para o treinamento. No entanto, novas observações estão disponíveis e já foram pré-carregadas como `X_new`.

O modelo `knn`, que você criou e treinou no exercício anterior, já está pré-carregado. Agora, você usará esse classificador para prever os rótulos do seguinte conjunto de novos dados:

```
X_new = np.array([[30.0, 17.5], [107.0, 24.1], [213.0, 10.9]])
```

Instruções

1. Crie `y_pred` prevendo os valores-alvo dos novos dados `X_new` usando o modelo `knn`.
2. Imprima os rótulos previstos para essas novas observações.

```
# Predict the labels for the X_new y_pred = ____ # Print the predictions
print("Predictions: {}".format(____))
```

Medindo o desempenho do modelo

- Agora podemos fazer previsões com um classificador.
- Mas como saber se as previsões estão corretas?
- Precisamos avaliar o desempenho do modelo!

O que é acurácia?

- Na classificação, a acurácia é uma métrica comum.
- Fórmula:

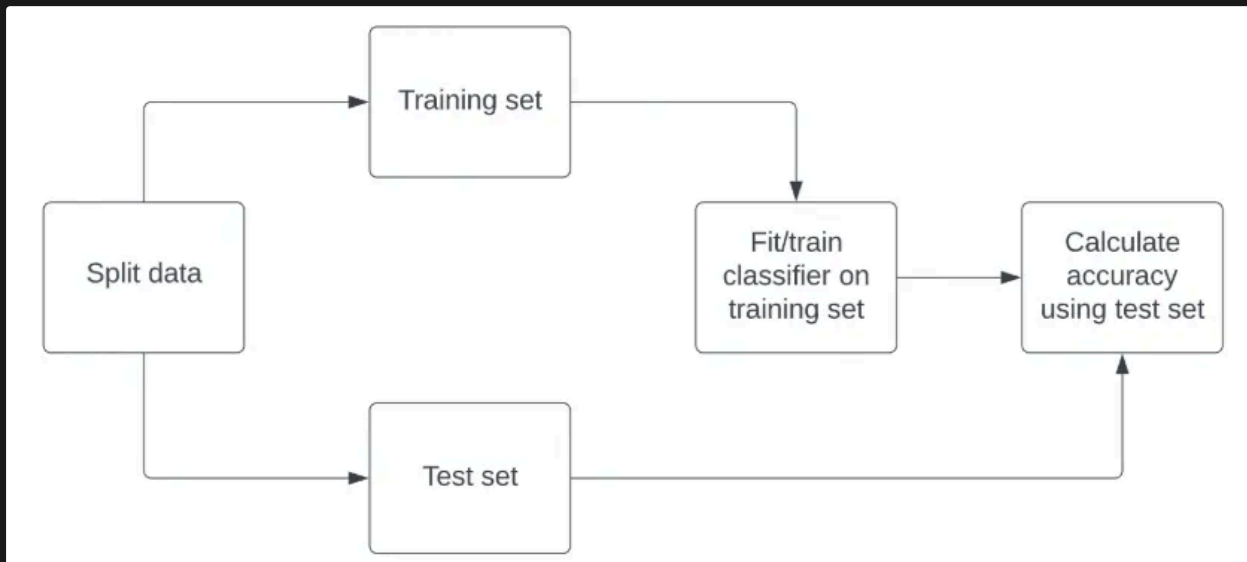
$$\text{Acurácia} = \frac{\text{Número de previsões corretas}}{\text{Total de observações}}$$

Acurácia e generalização

- Podemos calcular a acurácia usando os dados de treino.
- Porém, isso não mede a capacidade do modelo de generalizar.
- O que realmente importa é o desempenho em dados não vistos!

Calculando a acurácia

- O mais comum é dividir os dados em:
 - Conjunto de treinamento
 - Conjunto de teste



Treinando e testando o modelo

1. Treinamos o classificador usando o conjunto de treino.
2. Avaliamos a acurácia comparando as previsões com os rótulos do conjunto de teste.

Divisão treino/teste

- Utilizamos a função `train_test_split` do `sklearn.model_selection`.
- Divisão típica:
 - 70-80% dos dados para **treino**.
 - 20-30% dos dados para **teste**.
- Parâmetros importantes:
 - `test_size=0.3` : 30% dos dados para teste.
 - `random_state` : Garante **reprodutibilidade** dos resultados.
 - `stratify=y` : Mantém a **proporção original das classes** nos conjuntos de treino e teste.

Avaliando o modelo

1. Criamos um modelo KNN e treinamos com `X_train` e `y_train`.
2. Medimos a acurácia no conjunto de teste com `model.score(X_test, y_test)`.
3. O modelo apresentou **88% de acurácia**, mas a distribuição das classes (9:1) pode afetar a interpretação.

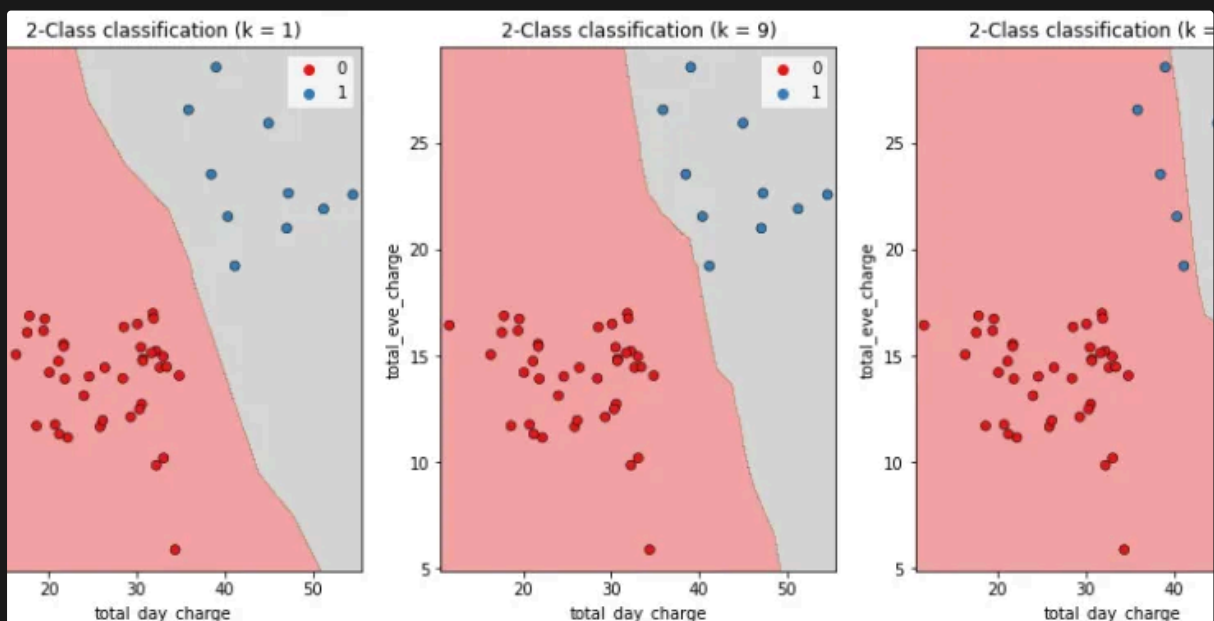
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=21, stratify=y)
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train, y_train)
print(knn.score(X_test, y_test))
```

Saída

```
0.8800599700149925
```

Complexidade do modelo

- O valor de **k** afeta a complexidade do modelo.
- **Limiar de decisão**: À medida que **k** aumenta, a decisão se torna menos sensível a dados individuais.
- Modelos muito simples não capturam padrões corretamente (**underfitting**).
- Modelos muito complexos aprendem ruídos do treino e não generalizam bem (**overfitting**).



- Podemos visualizar a relação entre k e a acurácia.
- **Passo a passo:**
 1. Criamos dicionários para armazenar a acurácia de treino e teste.
 2. Definimos um intervalo de valores para k .
 3. Iteramos sobre esses valores:
 - Criamos um modelo KNN com cada k .
 - Treinamos o modelo e calculamos a acurácia nos conjuntos de treino e teste.

```
train_accuracies = {} test_accuracies = {} neighbors = np.arange(1, 26)
for neighbor in neighbors: knn =
KNeighborsClassifier(n_neighbors=neighbor) knn.fit(X_train, y_train)
train_accuracies[neighbor] = knn.score(X_train, y_train)
test_accuracies[neighbor] = knn.score(X_test, y_test)
```

Plotando os resultados

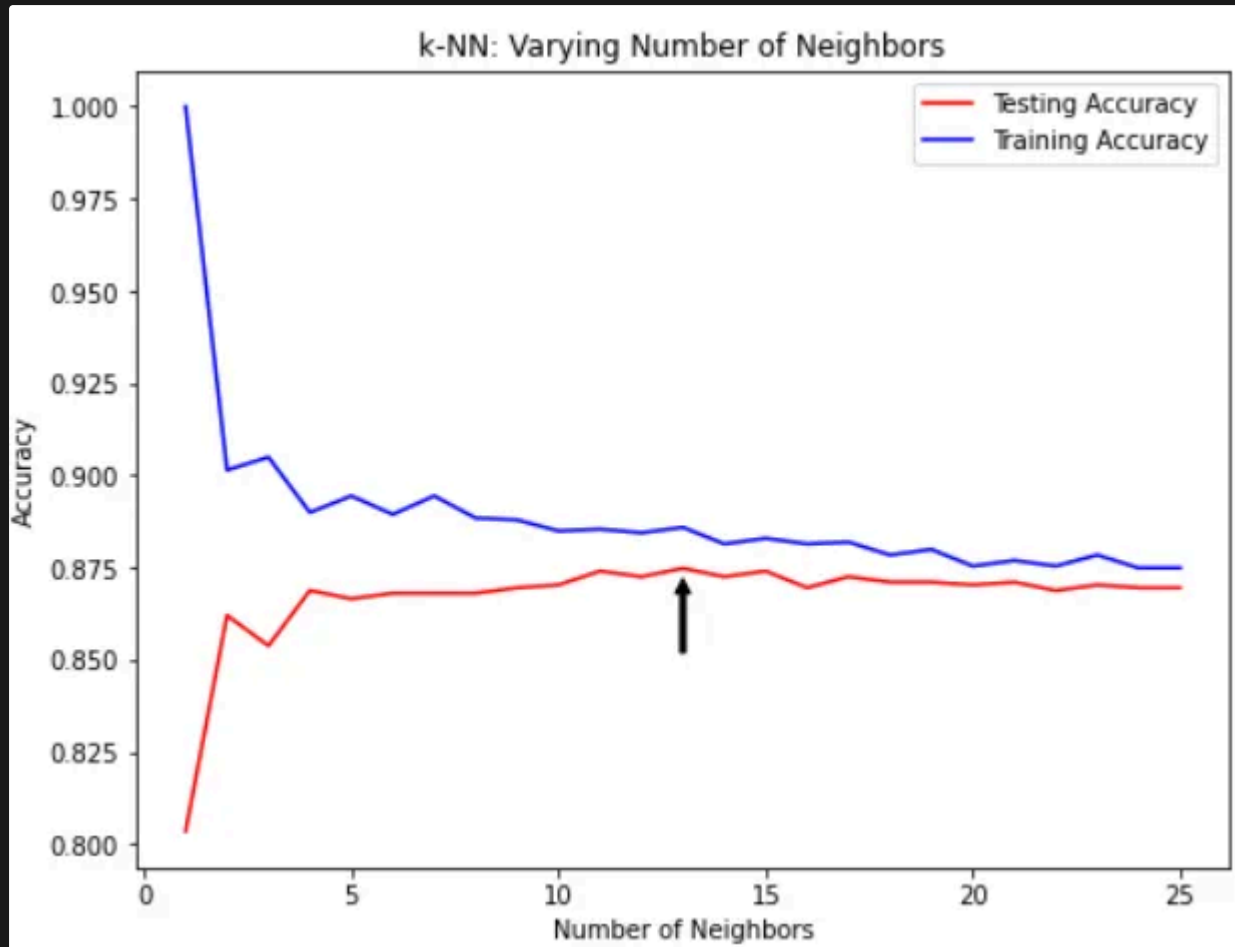
- Após o loop, geramos um gráfico com os valores de acurácia.
- **Incluímos:**
 - Legenda
 - Rótulos nos eixos

```
import matplotlib.pyplot as plt plt.figure(figsize=(8, 6)) plt.title("KNN:
Variação no Número de Vizinhos") plt.plot(neighbors,
train_accuracies.values(), label="Acurácia no Treino") plt.plot(neighbors,
test_accuracies.values(), label="Acurácia no Teste") plt.legend()
plt.xlabel("Número de Vizinhos") plt.ylabel("Acurácia") plt.show()
```

Analisando a curva de complexidade

- Conforme k ultrapassa 15, ocorre **underfitting**, e o desempenho estabiliza.

- A maior acurácia no teste ocorre em $k \approx 13$.



Vamos praticar!

- Agora, vamos praticar a **divisão dos dados**, o cálculo da acurácia e a curva de complexidade do modelo!

Exercício

Divisão Treino/Teste + Cálculo da Acurácia

Agora é hora de praticar a **divisão dos dados** em conjuntos de **treinamento** e **teste** usando o conjunto de dados `churn_df` !

Os arrays do NumPy já foram criados para você:

- `x` → contém as **features**
- `y` → contém a **variável alvo** (churn)

Instruções

✓ Importe `train_test_split` de `sklearn.model_selection`.

✓ Divida `X` e `y` em conjuntos de treino e teste com os seguintes parâmetros:

- `test_size=0.2` (20% dos dados para teste)
- `random_state=42` (para reprodutibilidade)
- `stratify=y` (para manter a proporção original das classes)

✓ Treine o modelo `knn` com os dados de treino usando `.fit()`.

✓ Calcule e imprima a acurácia do modelo nos dados de teste com `.score()`.

```
# Import the module from ____ import ____ X = churn_df.drop("churn",
axis=1).values y = churn_df["churn"].values # Split into training and test
sets X_train, X_test, y_train, y_test = ____(___, ___, test_size=____,
random_state=____, stratify=____) knn =
KNeighborsClassifier(n_neighbors=5) # Fit the classifier to the training
data ____ # Print the accuracy print(knn.score(___, ____))
```

Exercício

Overfitting e Underfitting

Interpretar a complexidade do modelo é uma excelente maneira de avaliar o desempenho em aprendizado supervisionado.

O objetivo é criar um modelo que **aprenda a relação entre as features e a variável-alvo**, e que também consiga **generalizar bem** quando for exposto a **novas observações**.

Os conjuntos de treino e teste já foram criados a partir do dataset `churn_df` e estão pré-carregados como:

- `X_train`, `X_test`, `y_train`, `y_test`

As bibliotecas `KNeighborsClassifier` e `numpy` (como `np`) também já foram importadas para você.

Instruções

✓ Crie `neighbors` como um array NumPy com valores de 1 até 12, inclusive.

✓ Para cada valor `neighbor` no array:

- Instancie um `KNeighborsClassifier` com `n_neighbors=neighbor`
- Treine o modelo com os dados de treino usando `.fit(X_train, y_train)`

- Calcule a acurácia com `.score()` :
 - No conjunto de **treino**, salve em `train_accuracies[neighbor]`
 - No conjunto de **teste**, salve em `test_accuracies[neighbor]`

Use o próprio valor de `neighbor` como índice (chave) nos dicionários `train_accuracies` e `test_accuracies`.

```
# Create neighbors
neighbors = np.arange(1, 21)
train_accuracies = {}
test_accuracies = {}
for neighbor in neighbors:
    # Set up a KNN Classifier
    knn = KNeighborsClassifier(n_neighbors=neighbor)
    # Fit the model
    knn.fit(X_train, y_train)
    # Compute accuracy
    train_accuracies[neighbor] = knn.score(X_train, y_train)
    test_accuracies[neighbor] = knn.score(X_test, y_test)
print(neighbors, '\n',
      train_accuracies, '\n', test_accuracies)
```

Exercício

Visualizando a Complexidade do Modelo

Agora que você já calculou a acurácia do modelo KNN nos conjuntos de **treinamento** e **teste**, utilizando diversos valores de `n_neighbors`, é hora de **visualizar graficamente** como o desempenho muda conforme o modelo se torna **mais ou menos complexo**.

As variáveis `neighbors`, `train_accuracies` e `test_accuracies` — que você criou no exercício anterior — já foram **pré-carregadas** para você.

Você vai **plotar os resultados** para ajudar a encontrar o número ideal de vizinhos (`k`) para seu modelo.

Instruções

- ✓ Adicione o título: `"KNN: Variação no Número de Vizinhos"`
- ✓ Plote a acurácia de treino:
 - No eixo x: `neighbors`
 - No eixo y: `train_accuracies.values()`
 - Rótulo da curva: `"Acurácia no Treino"`
- ✓ Plote a acurácia de teste:
 - No eixo x: `neighbors`