UFCD 10810 – Fundamentos do Desenvolvimento de Modelos Analíticos em Python

1. Introdução à Aprendizagem Automática

A aprendizagem automática (machine learning) é um ramo da inteligência artificial (IA) que se dedica ao desenvolvimento de algoritmos capazes de aprender padrões a partir de dados. Diferentemente de sistemas programados com regras fixas, os modelos de machine learning ajustam-se automaticamente com base em exemplos fornecidos, permitindo previsões, classificações e tomadas de decisão adaptativas.

A IA moderna encontra-se em rápida expansão, sendo utilizada em setores como medicina, segurança, serviços financeiros, retalho, educação e administração pública. A sua base são os dados, que através de métodos estatísticos e computacionais, são transformados em conhecimento e capacidade preditiva.

Bibliotecas como NumPy, Pandas, Matplotlib, scikit-learn, seaborn, XGBoost, TensorFlow e PyTorch integram-se em Python, tornando-o a linguagem mais usada para desenvolvimento de soluções analíticas.

2. Pré-processamento de Dados

O pré-processamento de dados é essencial para garantir a qualidade da informação a ser ingerida pelos modelos. Dados incompletos, ruidosos ou não padronizados prejudicam severamente o desempenho dos algoritmos.

2.1. Limpeza e Normalização

- **Tratamento de valores em falta**: substituição por moda, média, mediana ou utilização de modelos de imputacção.
- **Normalização e escalonamento**: StandardScaler para normalizar variáveis com distribuição normal, MinMaxScaler para distribuições não normalizadas.

StandardScaler – Normalização com média e desvio padrão

O StandardScaler transforma as variáveis numéricas de modo a que cada uma tenha:

- **Média igual a 0**
- **Desvio padrão igual a 1**

Fórmula:

$$z = \frac{x - \mu}{\sigma}$$

Onde:

- x: valor original
- μ: média da variável
- σ: desvio padrão da variável

Objetivo:

Torna as variáveis comparáveis em escala, especialmente útil para algoritmos **baseados em distâncias** (KNN, SVM, PCA, DBSCAN) ou que assumem distribuição normal (Regressão Logística, Linear).

Exemplo:

Se tivermos a variável idade com média 40 anos e desvio padrão 10, um valor de 50 anos será transformado em:

$$z = \frac{50 - 40}{10} = 1.0$$

MinMaxScaler – Escalonamento entre dois limites (normalmente [0, 1])

Este método reescala os valores para um intervalo definido, **sem alterar a forma da distribuição**.

Fórmula:

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

Resultado:

- O valor mínimo da variável torna-se 0
- O valor máximo torna-se 1

Ideal para:

- **Redes Neuronais**, onde os dados devem estar num intervalo pequeno e uniforme
- Visualizações gráficas padronizadas
- Quando os dados **não seguem uma distribuição normal**

2.2. Codificação de Variáveis Categóricas

- **Label Encoding**: para variáveis ordinais.
- **OneHotEncoding**: para variáveis nominais, com uso de ColumnTransformer.

Modelos de aprendizagem automática **exigem dados numéricos**. Variáveis categóricas (ex: sexo, profissão, cor, etc.) precisam ser **transformadas numericamente** para poderem ser processadas por algoritmos como regressão, árvores, KNN, SVM, etc.

LabelEncoder: Codificação Ordinal Simples

✓ **Conceito**

O LabelEncoder converte **cada categoria** de uma variável em um **inteiro único**. A ordem é atribuída automaticamente (ou pode ser definida), mas **não tem significado semântico** salvo em variáveis ordinais.

✓ **Quando Usar?**

- Quando a **ordem entre categorias faz sentido** (ex: baixo, médio, alto)
- Para algoritmos **que não são sensíveis à escala absoluta** (ex: árvores de decisão)

✓ **Problema:**

Em variáveis **nominais**, o LabelEncoder introduz **ordem artificial**, o que pode **violar pressupostos estatísticos** e induzir erros em algoritmos como Regressão Logística e KNN.

Exemplo:

```
from sklearn.preprocessing import LabelEncoder

cidades = ['Lisboa', 'Porto', 'Braga', 'Lisboa', 'Braga']
le = LabelEncoder()
valores_codificados = le.fit_transform(cidades)

print(le.classes_)  # ['Braga', 'Lisboa', 'Porto']
print(valores_codificados)  # [1, 2, 0, 1, 0]
```

OneHotEncoder: Codificação Binária Explicita

✓ **Conceito**

O OneHotEncoder converte cada categoria em uma **coluna binária distinta**, onde cada linha tem **1 apenas na categoria correspondente**, e **0 nas demais**.

✓ **Quando Usar?**

- Para **variáveis nominais** (ex: sexo, cor, profissão)
- Para algoritmos sensíveis à escala, como **Regressão Logística, KNN, SVM**

✓ **Considerações:**

- Pode gerar **alta dimensionalidade** com muitas categorias
- Requer cuidados com **multicolinearidade** (usa-se drop='first')

Exemplo:

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

df = pd.DataFrame({
    'profissao': ['admin', 'blue-collar', 'technician', 'admin'],
    'idade': [25, 45, 35, 33]
})

preprocessador = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(drop='first'), ['profissao'])
    ],
    remainder='passthrough'
)

dados_transformados = preprocessador.fit_transform(df)
print(preprocessador.get_feature_names_out())
print(dados_transformados.toarray())
```

✓ **Comparação entre LabelEncoder e OneHotEncoder**

Critério	LabelEncoder	OneHotEncoder
Tipo de variável	Ordinal (com ordem)	Nominal (sem ordem)
Forma da saída	Vetor unidimensional de inteiros	Matriz binária
Introduz ordem artificial?	Sim	Não
Aumenta a dimensionalidade?	Não	Sim (uma coluna por categoria)
Interpretabilidade	Média (difícil de justificar ordem)	Alta (preserva significado claro)
Exemplo típico	Grau académico: Primário, Secundário, Superior	Sexo, Cidade, Profissão

Boas Práticas

- Usa **LabelEncoder** apenas para variáveis ordinais com ordem intrínseca (ex: avaliação de risco: baixo, médio, alto).
- Prefere OneHotEncoder para todas as variáveis nominais.
- Combina **ColumnTransformer** com **Pipeline** para automação do pré-processamento (**ColumnTransformer p**ermite aplicar **transformações diferentes a colunas específicas** do DataFrame como, por exemplo, aplicar StandardScaler às colunas numéricas e OneHotEncoder às categóricas e **Pipeline** permite **encadear uma sequência de etapas** (ex: pré-processamento + modelo), funcionando como um modelo único e coeso).

  Exemplo prático:

  Suponhamos um DataFrame com variáveis mistas:

```
import pandas as pd

df = pd.DataFrame({
    'idade': [25, 45, 35],
    'salario': [20000, 40000, 30000],
    'cidade': ['Lisboa', 'Porto', 'Braga']
})
y = [0, 1, 0]  # variável alvo
```

  Comecemos por criar o **ColumnTransformer:**

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

transformador = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), ['idade', 'salario']),
        ('cat', OneHotEncoder(), ['cidade'])
    ]
)
```

Vamos integrar com um modelo usando Pipeline:

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline(steps=[
    ('preprocessamento', transformador),
    ('modelo', LogisticRegression())
])
```

Vamos treinar e avaliar:

```
pipeline.fit(df, y)
predicoes = pipeline.predict(df)
print(predicoes)
```

Combinar **ColumnTransformer** com **Pipeline** significa estruturar o código para que cada tipo de variável seja tratado corretamente, e que o modelo final receba dados preparados de forma segura, automática e replicável.

- Usa drop='first' em OneHotEncoder para evitar multicolinearidade (útil em regressões).

O parâmetro `drop='first'` usado no **`OneHotEncoder`** da biblioteca `scikit-learn` significa literalmente: "Eliminar a primeira coluna codificada de cada variável categórica." Isto serve para **evitar a multicolinearidade** — um problema estatístico onde as variáveis preditoras estão fortemente correlacionadas entre si, o que pode afetar negativamente modelos lineares (como Regressão Logística, Linear, etc.).

Quando usar drop='first'?

Situação	**Usar drop='first'?**
Regressão Linear ou Logística	**Sim** (evita multicolinearidade)
Modelos lineares sensíveis a redundância	**Sim**
Árvores de decisão, Random Forest, etc.	**Não é necessário** (insensíveis a colinearidade)
Visualizações simplificadas	**Sim** (menos colunas)

- `drop='first'` melhora a **estabilidade estatística** dos modelos lineares.
- É uma forma de definir uma **categoria de referência implícita**, como se fosse o "grupo base" numa análise estatística tradicional.
- Muito útil em **análise preditiva com regressão**.

2.3. Detecção e Tratamento de Outliers

- Método do IQR e Z-score
- Avaliação visual com boxplots e histogramas

O que são Outliers?

Outliers são observações que se distanciam significativamente dos restantes dados. Podem ocorrer por:

- Erros de medição ou entrada
- Variabilidade natural extrema
- Situações excecionais que merecem análise separada

Outliers podem:

- Prejudicar a média e o desvio padrão
- Afetar a performance de modelos (especialmente regressões, SVMs, KNN, PCA)

1. Método do IQR (Interquartile Range)

Conceito:

O IQR é a diferença entre o 3.º quartil (Q3) e o 1.º quartil (Q1):

$$\text{IQR} = Q3 - Q1$$

Outliers são definidos como valores que estão **fora dos limites**:

$$\text{Limite Inferior} = Q1 - 1.5 \times \text{IQR}$$

$$\text{Limite Superior} = Q3 + 1.5 \times \text{IQR}$$

Vantagens:

- Não assume distribuição normal dos dados
- Resistente a outliers (usa mediana)

Exemplo com Pandas:

```
import pandas as pd

df = pd.DataFrame({'idade': [22, 23, 23, 24, 24, 25, 100]})  # 100 é outlier
Q1 = df['idade'].quantile(0.25)
Q3 = df['idade'].quantile(0.75)
IQR = Q3 - Q1

limite_inf = Q1 - 1.5 * IQR
limite_sup = Q3 + 1.5 * IQR

outliers = df[(df['idade'] < limite_inf) | (df['idade'] > limite_sup)]
print(outliers)
```

2. Z-score

O Z-score mede quantos desvios padrão um valor está acima ou abaixo da média:

$$Z = \frac{X - \mu}{\sigma}$$

Vantagens:

- Baseado em estatísticas paramétricas
- Fácil de implementar e interpreter

Limitação:

- Supõe que os dados têm distribuição **normal**
- Sensível a valores extremos (média e σ são afetados)

Exemplo com scipy.stats:

```
from scipy.stats import zscore
import numpy as np

dados = np.array([22, 23, 23, 24, 24, 25, 100])
z = np.abs(zscore(dados))

outliers = dados[z > 3]
print(outliers)  # resultado: [100]
```

Aplicação em Machine Learning

- Aplicar **IQR** no início do pré-processamento para remover extremos evidentes
- Aplicar **Z-score** após normalização, em datasets que seguem (ou aproximam) uma distribuição normal
- Nunca aplicar cegamente: confirmar com **visualização (boxplot, histograma, scatter)**

3. Aprendizagem Supervisionada – Fundamentos Teóricos e Aplicações

3.1. Conceito

A aprendizagem supervisionada parte de um conjunto de dados rotulados, ou seja, onde cada exemplo possui uma entrada (features) e uma saída esperada (label). O objetivo é construir um modelo que generalize a relação entre entradas e saídas para prever respostas futuras.

3.2. Classificação vs Regressão

- **Classificação**: saída discreta (ex: "spam" ou "não spam").
- **Regressão**: saída contínua (ex: previsão de preço).

3.3. Algoritmos Clássicos

- **Regressão Logística**: utiliza a função sigmóide para prever probabilidades. Estimada por máxima verosimilhança, pode ser regularizada com L1/L2.
- **K-Nearest Neighbors (KNN)**: classifica pela maioria dos vizinhos mais próximos. Sensível à normalização.
- **Árvores de Decisão e Random Forest**: algoritmo baseado em divisões sucessivas dos dados. Random Forest cria ensembles com votação.
- **SVM**: busca um hiperplano com maior margem entre classes. Pode ser linear ou com kernel.
- **Redes Neuronais**: aplicação de múltiplas camadas de neurónios artificiais com ativação não linear (sigmoid, ReLU).

3.4. Avaliação de Modelos

- **Matriz de confusão, Acurácia, Precisão, Revocação, F1-score**
- **Validação cruzada**, hold-out, estratificação
- **GridSearchCV** para otimização de hiperparâmetros

O que é?

A matriz de confusão é uma tabela 2x2 (ou maior, para múltiplas classes) que resume os resultados de um modelo de classificação binária.

	Previsto Positivo	**Previsto Negativo**
Real Positivo	Verdadeiro Positivo (TP)	Falso Negativo (FN)
Real Negativo	Falso Positivo (FP)	Verdadeiro Negativo (TN)

1. **Matriz de Confusão e Métricas Derivadas**

Acurácia – Proporção de classificações corretas

$$\text{Acurácia} = \frac{TP + TN}{TP + TN + FP + FN}$$

Boa para datasets equilibrados. Pode enganar em classes desbalanceadas.

Precisão – Fiabilidade das previsões positivas

$$\text{Precisão} = \frac{TP}{TP + FP}$$

Útil quando **falsos positivos** são críticos (ex: diagnóstico falso de cancro).

Revocação (Recall / Sensibilidade) – Capacidade de identificar todos os positivos reais

$$\text{Recall} = \frac{TP}{TP + FN}$$

Essencial quando **falsos negativos** são graves (ex: deteção de fraudes ou doenças).

F1-score – Média harmónica entre precisão e recall

$$F1 = 2 \times \frac{\text{Precisão} \times \text{Recall}}{\text{Precisão} + \text{Recall}}$$

Ideal para classes desequilibradas. Penaliza valores extremos entre precisão e recall.

Exemplo com scikit-learn:

```python
from sklearn.metrics import confusion_matrix, classification_report

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred, digits=3))
```

2. Validação Cruzada, Hold-out e Estratificação

Hold-out

Divisão simples dos dados em treino e teste (ex: 80%/20%).
Problema: resultados podem variar muito dependendo da partição.

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Validação Cruzada (K-Fold Cross Validation)

Divide o conjunto em **K subconjuntos**. O modelo é treinado K vezes, cada vez com um subset diferente como teste.

$$\text{Acurácia média} = \frac{1}{K} \sum_{i=1}^{K} \text{acurácia}_i$$

```python
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, X, y, cv=5)
print(scores.mean())
```

StratifiedKFold

Garante que a **distribuição das classes** se mantém igual em cada partição.

```python
from sklearn.model_selection import StratifiedKFold
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

Essencial para datasets com classes **desequilibradas**!

3. **GridSearchCV – Otimização de Hiperparâmetros**

Conceito

Procura **combinatórias ótimas de hiperparâmetros** para um modelo, testando cada combinação com validação cruzada.

Exemplo com Random Forest:

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [5, 10, None],
    'min_samples_split': [2, 5]
}

grid = GridSearchCV(RandomForestClassifier(), param_grid, cv=5)
grid.fit(X, y)
print(grid.best_params_)
print(grid.best_score_)
```

Como funciona:

- **Exaustivo**: testa todas as combinações possíveis
- Usa o scoring (ex: accuracy, f1, roc_auc) para escolher o melhor
- Pode ser **computacionalmente pesado**

Dica:
Para problemas grandes, usar **RandomizedSearchCV** ou **HalvingGridSearchCV** como alternativa.

Método	Utilidade Principal
Matriz de Confusão	Visualização detalhada da performance
Acurácia	Visão geral, útil em datasets equilibrados
Precisão & Revocação	Diagnóstico detalhado em problemas críticos
F1-score	Métrica equilibrada em classes desbalanceadas
Hold-out	Simples, mas instável
Cross-validation	Avaliação robusta, evita overfitting
Stratification	Essencial para classes desbalanceadas
GridSearchCV	Otimiza o modelo com base em hiperparâmetros

4. Técnicas de Clustering (Aprendizagem Não Supervisionada)

4.1. K-Means

- Algoritmo que minimiza a soma das distâncias intra-cluster.
- Necessita definir k previamente.

Modelo do Cotovelo

Utiliza o gráfico da soma dos erros quadráticos (inertia) em função de k. O ponto de inflexão ("cotovelo") indica o número ótimo de clusters. Implementação:

```
inertias = []
for k in range(1, 11):
    model = KMeans(n_clusters=k).fit(data)
    inertias.append(model.inertia_)
plt.plot(range(1, 11), inertias)
```

O algoritmo **K-Means** exige que especifiquemos **à priori** o número de clusters k. No entanto, este valor não é conhecido em muitos contextos reais.

O **Modelo do Cotovelo** é uma técnica heurística para **estimar o valor ótimo de k**, observando como a variabilidade total dentro dos clusters diminui com o aumento de k.

Fundamento Estatístico:

Função de Custo: Soma dos Erros Quadráticos Intra-cluster (SSE ou "Inertia")
A função objetivo do K-Means minimiza a **distância euclidiana ao centroide** de cada grupo. Essa soma de distâncias quadradas é chamada:

$$SSE(k) = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2$$

Onde:

- x: ponto de dados
- μ_i: centroide do cluster i
- C_i: conjunto de pontos no cluster i

Lógica do Cotovelo

À medida que aumentamos k:

- A inércia (SSE) diminui (clusters menores → menos erro).
- Mas o ganho diminui progressivamente — cada novo cluster explica menos variância adicional.

O "cotovelo" é o ponto onde a taxa de melhoria sofre uma inflexão — ou seja, adicionar mais clusters traz ganhos marginais insignificantes.

Implementação em Python

```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

inertias = []
K = range(1, 11)

for k in K:
    model = KMeans(n_clusters=k, random_state=42)
    model.fit(X)
    inertias.append(model.inertia_)

plt.figure(figsize=(8, 4))
plt.plot(K, inertias, marker='o')
plt.xlabel("Número de Clusters (k)")
plt.ylabel("Inércia (SSE)")
plt.title("Método do Cotovelo")
plt.grid(True)
plt.show()
```

Interpretação Visual

- **Queda acentuada inicial**: indica que aumentar k está a reduzir significativamente a inércia.
- **Ponto de inflexão** (cotovelo): é onde **deixar de ganhar muito** ao aumentar k.
- **Após o cotovelo**: o modelo começa a **ajustar-se demais aos dados (overfitting)**.

Limitações e Alternativas

Limitação	Descrição
Subjetividade	A interpretação visual do cotovelo pode variar
Nem sempre há "cotovelo" claro	Alguns datasets não apresentam inflexão definida
Usa apenas inércia	Ignora qualidade dos clusters (cohesão/separação)

Alternativas Complementares

- Silhouette Score: mede separação e coesão dos clusters
- Gap Statistic
- Davies-Bouldin Index
- BIC/AIC (para GMM)

Quando usar o Modelo do Cotovelo?

- Como primeira abordagem exploratória
- Em datasets de dimensão reduzida a média
- Quando se pretende um modelo interpretável com poucos clusters
- Em conjunto com outras métricas (silhouette, DB index)

4.2. DBSCAN

- Algoritmo baseado em densidade. Define-se eps (raio) e min_samples.
- Identifica pontos centrais, bordas e ruídos.
- Pode descobrir clusters com formas arbitrárias.

4.3. Gaussian Mixture Models (GMM)

- Assume distribuição normal dos dados e calcula probabilidade de pertença a cada cluster.
- Usa Expectation-Maximization (EM).

4.4. Clustering Hierárquico

- Usa ligação entre grupos (ward, average, complete).
- Gera dendrogramas.

5. Visualização e Redução de Dimensionalidade com PCA

5.1. Teoria

PCA (Principal Component Analysis) transforma variáveis originais em componentes principais ortogonais, ordenados pela variância explicada. Baseia-se na decomposição da matriz de covariância via autovalores e autovetores.

5.2. Aplicação

Usado para:

- Visualização em 2D/3D de dados multidimensionais
- Redução de ruído
- Compressão de dados

Exemplo:

```
pca = PCA(n_components=2)
data_2d = pca.fit_transform(X)
```

A Análise de Componentes Principais (PCA) é uma **técnica de redução de dimensionalidade** que transforma um conjunto de variáveis possivelmente correlacionadas num conjunto menor de **componentes não correlacionados** (ortogonais), chamados **componentes principais**, mantendo a **maior parte da variabilidade dos dados**.

Fundamento Matemático

Dado um conjunto de dados com p variáveis $X_1, X_2, \ldots, X_p$, o PCA visa construir novos eixos (componentes) $Z_1, Z_2, \ldots, Z_p$, que são:

- **Combinações lineares** das variáveis originais
- **Ortogonais entre si**
- Ordenados de forma que:
  - Z_1 explica a **maior variância possível**
  - Z_2 explica a maior variância restante, e assim por diante

Etapas do PCA

1. **Normalização** dos dados (obrigatório!)

$$x_{ij}^{norm} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

2. **Construção da matriz de covariância** dos dados normalizados
3. **Cálculo de autovalores e autovetores**
   - Autovalores: medem **quanta variância** é explicada
   - Autovetores: definem a **direção dos novos eixos**
4. **Ordenação** dos autovalores e escolha dos **componentes principais**
5. **Projeção dos dados** no novo espaço vetorial reduzido

```python
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import pandas as pd

# Carregar dataset e normalizar
X = pd.read_csv("winequality-red.csv").drop(columns=['quality'])
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# PCA com 2 componentes
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

print("Variância explicada:", pca.explained_variance_ratio_)
```

Interpretação da Variância Explicada

- explained_variance_ratio_ mostra **quanto da variância total** está retida em cada componente.
- Soma de variâncias acumuladas ajuda a decidir quantos componentes usar.

Exemplo: Se os 2 primeiros componentes explicam 85% da variância, pode-se reduzir de 11 dimensões para 2 com pouca perda de informação.

Visualização com PCA

```python
import matplotlib.pyplot as plt

plt.scatter(X_pca[:, 0], X_pca[:, 1], alpha=0.6)
plt.xlabel("Componente Principal 1")
plt.ylabel("Componente Principal 2")
plt.title("Visualização com PCA")
plt.grid(True)
plt.show()
```

Benefícios do PCA

Benefício	Impacto na Análise
Reduz complexidade computacional	Menos variáveis, modelos mais leves

Elimina redundância	Remove correlações entre features
Melhora visualização	Permite ver dados em 2D/3D
Pode melhorar modelos	Reduz overfitting e ruído

Limitações

Limitação	Descrição
Linearidade	PCA assume que as componentes são combinações lineares
Perda de interpretabilidade	Componentes não correspondem a variáveis reais (são combinações)
Sensível à escala	Deve-se sempre normalizar os dados antes
Não preserva variância local	Outras técnicas (ex: t-SNE, UMAP) são melhores para clusters não-lineares

Quando Usar PCA?

- Quando o número de features é elevado
- Quando há multicolinearidade
- Quando se quer reduzir o ruído
- Para visualizações em 2D/3D
- Como etapa antes de clustering ou regressão

Integração com Modelos

O PCA pode ser facilmente integrado com pipelines de Machine Learning:

```
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier

pipeline = Pipeline(steps=[
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=5)),
    ('model', RandomForestClassifier())
])
```

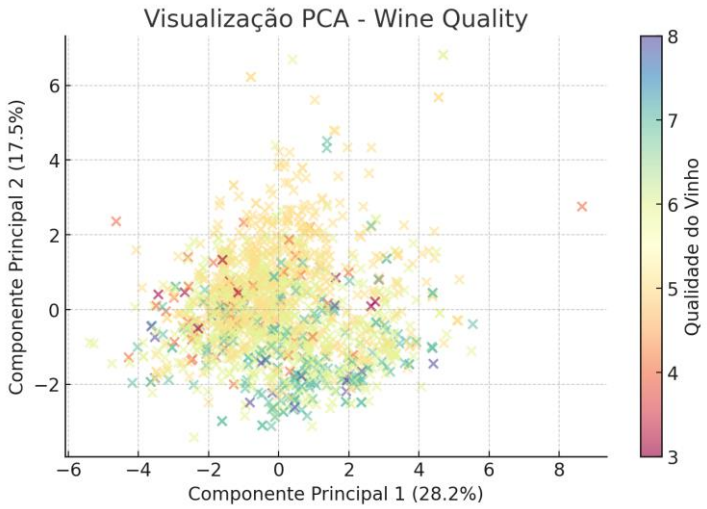
PCA é uma ferramenta matemática poderosa que atua como **lupa estatística**, revelando padrões ocultos e simplificando modelos sem comprometer (muito) a informação original. Apesar de abstrato, é prático, rápido e essencial em qualquer caixa de ferramentas de Data Scientist.

Temos aqui a visualização dos dados do vinho (Wine Quality) (utilizou-se o csv já usado nas aulas) após aplicação de **PCA com 2 componentes principais**:

- Cada ponto representa um vinho;
- A posição no gráfico reflete as **características químicas principais** (reduzidas via PCA);
- A cor representa a **qualidade atribuída** (de 3 a 8).

Observa-se:

- **Agrupamentos e transições suaves entre qualidades próximas**;
- **Alguma sobreposição entre classes médias (ex: qualidade 5, 6)** — esperado, pois os critérios de qualidade são subjetivos e discretos.



6. Avaliação de Clustering

6.1. Coeficiente de Silhueta

Mede a separação e coesão dos clusters. Valor entre -1 e 1.

```
from sklearn.metrics import silhouette_score
silhouette_score(X, labels)
```

O **Coeficiente de Silhueta (Silhouette Coefficient)** mede, para cada ponto de dados, **o quão bem ele está atribuído ao seu cluster** em comparação com outros clusters.
É uma forma de quantificar **coesão interna** (similaridade com o próprio grupo) e **separação externa** (diferença em relação aos outros grupos).

Cálculo Matemático

Para um ponto iii:

- a(i)a(i)a(i) = distância média entre o ponto iii e todos os outros pontos no **mesmo cluster** → **coesão**
- b(i)b(i)b(i) = distância média entre o ponto iii e todos os pontos do **cluster mais próximo diferente** → **separação**

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

Interpretação

Valor de $s(i)$	Interpretação
$s(i) \approx 1$	Ponto bem ajustado ao cluster, distante dos outros
$s(i) \approx 0$	Ponto na fronteira entre clusters
$s(i) < 0$	Ponto pode estar mal atribuído (mais próximo de outro cluster que do seu)

O coeficiente médio de silhueta para todos os pontos fornece uma métrica global de qualidade de clustering.

Exemplo com Scikit-Learn:

```
from sklearn.metrics import silhouette_score
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris
import pandas as pd

# Usar dados do Iris como exemplo
iris = load_iris()
X = iris.data

# Clustering com K-Means
kmeans = KMeans(n_clusters=3, random_state=42)
labels = kmeans.fit_predict(X)

# Avaliação
sil_score = silhouette_score(X, labels)
print(f"Coeficiente de Silhueta médio: {sil_score:.3f}")
```

Visualização da Silhueta:

Também é possível visualizar a silhueta de cada ponto:

```
from sklearn.metrics import silho
import matplotlib.pyplot as plt
import numpy as np

sample_silhouette_values = silhou
plt.hist(sample_silhouette_values
plt.title("Distribuição do Coefic
plt.xlabel("Valor da Silhueta")
plt.ylabel("Número de Pontos")
plt.grid(True)
plt.show()
```

Vantagens

Vantagem	Explicação
Métrica intuitiva e padronizada	Intervalo entre -1 e 1, fácil de interpretar
Compatível com qualquer algoritmo	K-Means, DBSCAN, GMM, Hierárquico, etc.
Usada para comparar diferentes `k`	Ajuda a determinar o número ideal de clusters

Limitações

Limitação	Descrição
Custo computacional elevado	Requer cálculo de distâncias ponto-a-ponto para todos os clusters
Pode ser enganoso com clusters de formas irregulares	Silhueta favorece formas esféricas (como as do K-Means)
Requer o número de clusters já definido	Não é uma técnica de clustering, apenas de avaliação

Como usar na prática?

- Em **K-Means**, corre o algoritmo para vários valores de k, e escolhe o k com maior silhueta média;
- Em **DBSCAN**, testa diferentes combinações de eps e min_samples e compara silhuetas;
- Em relatórios de clustering, apresenta **gráficos de silhueta** por cluster.

O Coeficiente de Silhueta é uma métrica poderosa para validar a estrutura de agrupamentos. Quando usado com consciência das suas limitações, fornece **informações ricas sobre a qualidade e a consistência dos clusters**.

6.2. Adjusted Rand Index (ARI)

Compara agrupamentos com rótulos reais. Corrige aleatoriedade.

O **Adjusted Rand Index (ARI)** é uma medida de **similaridade entre duas partições** de um mesmo conjunto de dados:

- A **partição verdadeira** (ex: rótulos reais, como "tipo de vinho", "espécie da flor")
- A **partição prevista** (ex: resultado do algoritmo de clustering)

Objetivo:
Avaliar **quantas decisões de pares de elementos** foram iguais em ambas as partições, **corrigindo o valor esperado por acaso**.

Interpretação dos Valores do ARI:

Valor do ARI	Interpretação
1.0	Agrupamentos idênticos
0.0	Semelhança igual ao acaso
< 0	Pior do que o acaso (desagrupamento sistemático)

Exemplo com Scikit-learn:

```
from sklearn.metrics import adjusted_rand_score

# Rótulos reais e previstos
y_true = [0, 0, 1, 1, 2, 2]
y_pred = [1, 1, 0, 0, 2, 2]

ari = adjusted_rand_score(y_true, y_pred)
print(f"Adjusted Rand Index: {ari:.3f}")
```

6.3 ARI vs Silhouette Score

Métrica	Necessita rótulos verdadeiros?	Tipo de avaliação
ARI	Sim	**Avaliação externa**
Silhouette Score	Não	**Avaliação interna**

- ARI é usado quando tens acesso à verdade (ex: para validar um algoritmo com ground truth).
- Silhouette é usado quando não tens rótulos verdadeiros.

Quando usar o ARI?

- Em estudos comparativos entre algoritmos de clustering
- Para medir quão bem um método não supervisionado se aproxima de um agrupamento conhecido
- Em benchmarks de performance (ex: Iris, MNIST, Wine Quality)

Cuidados a ter

Atenção a:	Porque é relevante
Classes muito desbalanceadas	O ARI pode ser dominado pela classe maior
Agrupamentos com ruído	O ARI considera todos os pares, incluindo ruído
Sensível à relabeling	Labels diferentes mas mesma estrutura → ARI = 1

O **ARI é uma métrica de comparação de agrupamentos precisa, corrigida para o acaso**. É essencial quando há dados com **rótulos conhecidos**, permitindo avaliar com rigor a fidelidade de algoritmos de clustering.

7. Projeto Integrado: Modelo Não-Supervisionado Análise do Dataset Wine Quality

- **Clusterização com K-Means, DBSCAN, GMM**
- **Visualização com PCA**
- **Previsão de novo registo de vinho com modelos treinados**
- **Interpretação estatística e recomendação de modelo adequado**

Este projeto consolida competências analíticas através de uma experiência prática, integrando pré-processamento, clustering, visualização, previsão e comunicação de resultados.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans, DBSCAN
from sklearn.mixture import GaussianMixture
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score, adjusted_rand_score
import matplotlib.pyplot as plt

# 1. Carregar e preparar os dados
df = pd.read_csv("winequality-red.csv")  # Certifica-te de que o ficheiro está no mesmo diretório
X = df.drop(columns=['quality']).values  # Usamos todas as features exceto a variável de qualidade

# 2. Normalização
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 3. PCA para visualização
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# 4. K-Means
kmeans = KMeans(n_clusters=3, random_state=42)
labels_kmeans = kmeans.fit_predict(X_scaled)
print("Silhouette (K-Means):", silhouette_score(X_scaled, labels_kmeans))

# Visualizar K-Means
plt.figure(figsize=(8, 4))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=labels_kmeans, cmap='viridis', alpha=0.6)
plt.title("Clusters com K-Means")
plt.xlabel("PC1")
plt.ylabel("PC2")
```

```
plt.grid(True)
plt.show()

# 5. DBSCAN
dbscan = DBSCAN(eps=1.5, min_samples=5)
labels_dbscan = dbscan.fit_predict(X_scaled)
print("Silhouette (DBSCAN):", silhouette_score(X_scaled, labels_dbscan))

# Visualizar DBSCAN
plt.figure(figsize=(8, 4))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=labels_dbscan, cmap='Set1', alpha=0.6)
plt.title("Clusters com DBSCAN")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True)
plt.show()

# 6. Gaussian Mixture Model (GMM)
gmm = GaussianMixture(n_components=3, random_state=42)
labels_gmm = gmm.fit_predict(X_scaled)
print("Silhouette (GMM):", silhouette_score(X_scaled, labels_gmm))

# Visualizar GMM
plt.figure(figsize=(8, 4))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=labels_gmm, cmap='coolwarm', alpha=0.6)
plt.title("Clusters com Gaussian Mixture")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True)
plt.show()

# 7. Previsão de novo vinho
novo_vinho = [[7.4, 0.70, 0.00, 1.9, 0.076, 11.0, 34.0, 0.9978, 3.51, 0.56, 9.4]]
novo_vinho_scaled = scaler.transform(novo_vinho)

cluster_kmeans = kmeans.predict(novo_vinho_scaled)
cluster_dbscan = dbscan.fit_predict(X_scaled)
cluster_gmm = gmm.predict(novo_vinho_scaled)
probs_gmm = gmm.predict_proba(novo_vinho_scaled)

print(f"Novo vinho pertence ao cluster (K-Means): {cluster_kmeans[0]}")
print(f"Novo vinho pertence ao cluster (GMM): {cluster_gmm[0]}")
print("Probabilidades GMM:", probs_gmm[0])
```

8. Projeto Integrado: Modelo Supervisionado para Previsão de Doença Cardíaca

- **Dataset**: heart.csv com variáveis clínicas de pacientes

- **Objetivo**: Prever presença de doença cardíaca (target binário: 0 = não, 1 = sim)
- **Algoritmos usados**: Regressão Logística, Random Forest, SVM

Este projeto permite compreender os conceitos de classificação binária, validação cruzada, ajuste de hiperparâmetros e avaliação de modelos preditivos num cenário médico.

```
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix

# Carregar dados
df = pd.read_csv("heart.csv")
X = df.drop(columns=["target"])
y = df["target"]

# Normalização
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Divisão treino/teste
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
random_state=42, stratify=y)

# Modelo 1: Regressão Logística
log_model = LogisticRegression(max_iter=1000)
log_model.fit(X_train, y_train)
print("\n--- Regressão Logística ---")
print(classification_report(y_test, log_model.predict(X_test)))

# Modelo 2: Random Forest
rf = RandomForestClassifier(random_state=42)
rf.fit(X_train, y_train)
print("\n--- Random Forest ---")
print(classification_report(y_test, rf.predict(X_test)))

# Modelo 3: SVM com GridSearch
param_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf']}
svc = SVC(probability=True)
grid = GridSearchCV(svc, param_grid, cv=5)
grid.fit(X_train, y_train)
print("\n--- SVM (melhor modelo) ---")
print(classification_report(y_test, grid.predict(X_test)))

# Previsão para um novo paciente
novo_paciente = [[63, 1, 3, 145, 233, 1, 0, 150, 0, 2.3, 0, 0, 1]]
```

```
# exemplo real do dataset
novo_paciente_scaled = scaler.transform(novo_paciente)
pred_log = log_model.predict(novo_paciente_scaled)[0]
pred_rf = rf.predict(novo_paciente_scaled)[0]
pred_svm = grid.predict(novo_paciente_scaled)[0]

print("\nPrevisão para novo paciente:")
print(f"Regressão Logística: {pred_log}")
print(f"Random Forest: {pred_rf}")
print(f"SVM: {pred_svm}")
```

Este projeto ilustra na prática a aplicação de **aprendizagem supervisionada** em contexto médico, com comparação entre três modelos, avaliação com métricas clássicas e previsão personalizada para novos dados.