Prétraitement des données

avec Python

Un cours pratique de 30 heures

Yaé Ulrich Gaba

AIRINA Labs

2026

Table des matières

Préface

Le prétraitement des données est la phase ingrate mais décisive de tout projet data. Les études le montrent régulièrement : les data scientists passent entre 60 et 80% de leur temps à nettoyer et préparer leurs données. Pourtant, la plupart des cours survolent cette phase en un seul chapitre, pressés d'arriver aux parties « intéressantes » : modèles, algorithmes, prédictions.

Ce cours prend le parti inverse. Nous consacrons 30 heures à l'art et la science de transformer des données brutes, désordonnées, réelles, en jeux de données propres et prêts pour l'analyse. Si vous apprenez à bien prétraiter vos données, chaque modèle que vous construirez ensuite sera meilleur.

Nous utilisons Python parce que son écosystème — `pandas`, `scikit-learn`, `missingno`, `nltk` — offre la boîte à outils de prétraitement la plus complète de tous les langages. Chaque exemple s'appuie sur un jeu de données public réel, avec de vrais problèmes : valeurs manquantes, types incohérents, valeurs aberrantes, encodages mixtes, texte désordonné.

Ce que ce cours n'est pas. Ce n'est pas un cours de machine learning (même si nous préparons les données pour le ML). Ce n'est pas un cours de statistiques (même si nous utilisons le raisonnement statistique). C'est un *cours pratique de préparation de données* pour qui veut maîtriser la partie la plus chronophage du pipeline data.

Prérequis. Python de base (variables, listes, boucles, fonctions). Une familiarité avec les DataFrames pandas est utile mais sera rappelée au chapitre 1.

Logiciels. Tout le code tourne dans des notebooks Jupyter, soit en local (Anaconda), soit dans le cloud (Google Colab — gratuit, sans installation).

Yaé Ulrich Gaba
Cotonou, 2026

Chapitre 1

Paysage et pipelines de données

> *« La donnée est le nouveau pétrole — mais comme le pétrole, elle doit être raffinée avant d'être utile. »* — Clive Humby

1.1 Qu'est-ce que le prétraitement des données ?

Le prétraitement des données est l'ensemble des transformations appliquées aux données brutes avant analyse ou modélisation. Il fait le pont entre les données que vous *avez* et celles dont vous *avez besoin*.

Un pipeline typique de prétraitement inclut :

1. **Chargement** — lire les données depuis fichiers, bases de données ou API.
2. **Inspection** — comprendre structure, types et qualité.
3. **Nettoyage** — gérer valeurs manquantes, doublons et erreurs.
4. **Transformation** — encoder, mettre à l'échelle, créer des variables.
5. **Validation** — confirmer que la sortie répond aux besoins en aval.

Astuce données

Selon les enquêtes sectorielles, les praticiens consacrent 60–80% du temps de projet au prétraitement. Maîtriser cette phase est la compétence la plus levier de la science des données.

1.2 Types et structures de données

1.2.1 Données structurées, semi-structurées et non structurées

- **Structurées :** tables avec lignes et colonnes (CSV, bases SQL, Excel).
- **Semi-structurées :** données à schémas souples (JSON, XML, YAML).
- **Non structurées :** texte libre, images, audio, vidéo.

Ce cours se concentre sur les données structurées et semi-structurées, avec un chapitre (Chapitre 7) consacré au texte.

1.2.2 Types de variables

```
import pandas as pd

# Charger un dataset r\'eel pour inspecter les types
url = ("https://raw.githubusercontent.com/datasciencedojo/"
       "datasets/master/titanic.csv")
df = pd.read_csv(url)
print(df.dtypes)
```

Sortie (abrégée) :

```
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Fare           float64
Embarked        object
```

Astuce prétraitement

Le dtype `object` dans pandas indique généralement des chaînes, mais peut aussi cacher des types mélangés. Vérifiez toujours avec `df[col].apply(type).value_counts()`.

1.2.3 Numérique vs catégoriel

```
numerical = df.select_dtypes(include="number").columns.tolist()
categorical = df.select_dtypes(include="object").columns.tolist()

print(f"Numerical columns ({len(numerical)}): {numerical}")
print(f"Categorical columns ({len(categorical)}): {categorical}")
```

1.3 Formats de données courants

Format	Extension	Lecteur pandas	Remarques
CSV	`.csv`	`read_csv`	Le plus courant ; attention au délimiteur et à l'encodage
Excel	`.xlsx`	`read_excel`	Nécessite `openpyxl` ; support multi-feuilles

JSON	`.json`	`read_json`	Les structures imbriquées demandent `json_normalize`
Parquet	`.parquet`	`read_parquet`	Colonnaire, rapide, préserve les types
SQL	—	`read_sql`	Nécessite une connexion `sqlalchemy`

1.4 Premier regard sur un dataset réel

Nous utiliserons le dataset Melbourne Housing dans plusieurs chapitres :

```
# Dataset Melbourne Housing
melb = pd.read_csv("melb_data.csv")

# Shape : lignes x colonnes
print(f"Shape: {melb.shape}")

# Trois premi\`eres lignes
print(melb.head(3))

# Statistiques de r\'esum\'e
print(melb.describe())

# Valeurs manquantes par colonne
print(melb.isnull().sum().sort_values(ascending=False).head(10))
```

```
# Rapport rapide de qualit\'e
def data_quality_report(df):
    """Generate a quick quality report for any DataFrame."""
    report = pd.DataFrame({
        "dtype": df.dtypes,
        "non_null": df.notnull().sum(),
        "null_count": df.isnull().sum(),
        "null_pct": (df.isnull().sum() / len(df) * 100).round(1),
        "n_unique": df.nunique(),
        "sample_value": df.iloc[0]
    })
    return report.sort_values("null_pct", ascending=False)

print(data_quality_report(melb))
```

Astuce données

Faites toujours un rapport de qualité *avant* toute analyse. Les cinq nombres dont vous avez immédiatement besoin : shape, dtypes, comptes de nuls, comptes d'unicité, et statistiques de résumé.

1.5 Le workflow de prétraitement

```python
# Workflow minimal de bout en bout
import pandas as pd
import numpy as np

# 1. Charger
df = pd.read_csv("melb_data.csv")

# 2. Inspecter
print(df.info())

# 3. Nettoyer : doublons, valeurs manquantes
df = df.drop_duplicates()
df["Car"] = df["Car"].fillna(df["Car"].median())
df = df.dropna(subset=["Price"])

# 4. Transformer : encoder cat\'egoriel, mettre \`a l'\'echelle num\'erique
df["Type_encoded"] = df["Type"].map({"h": 0, "u": 1, "t": 2})

# 5. Valider
assert df.isnull().sum().sum() == 0 or True  # relax\'e pour la d\'emo
print(f"Clean shape: {df.shape}")
```

1.6 Révision pandas : opérations essentielles

1.6.1 Sélection et filtrage

```python
# S\'electionner des colonnes
prices = melb[["Suburb", "Price", "Rooms"]]

# Filtrer des lignes
expensive = melb[melb["Price"] > 2_000_000]
south = melb[melb["Regionname"] == "Southern Metropolitan"]

# Combin\'e
big_south = melb[(melb["Regionname"] == "Southern Metropolitan") &
                 (melb["Rooms"] >= 4)]
print(f"Large southern houses: {len(big_south)}")
```

1.6.2 Groupby et agrégation

```python
# Prix moyen par r\'egion
region_stats = (melb.groupby("Regionname")["Price"]
                .agg(["mean", "median", "count"])
                .sort_values("median", ascending=False))
```

```
print(region_stats)
```

Attention

Pandas `groupby` élimine silencieusement les clés `NaN` par défaut. Si votre colonne de groupement a des valeurs manquantes, elles seront exclues du résultat. Utilisez `dropna=False` pour les inclure.

1.7 Aperçu des outils et bibliothèques

```
# Pile de pr\'etraitement de base
import pandas as pd                # manipulation de donn\'ees
import numpy as np                  # op\'erations num\'eriques
import matplotlib.pyplot as plt     # visualisation
import seaborn as sns               # graphiques statistiques
import missingno as msno            # visualisation des manquants
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
```

1.8 Exercices

Exercice

1. Téléchargez le dataset Melbourne Housing. Écrivez un rapport de qualité montrant : nombre de lignes, colonnes, dtype de chaque colonne, pourcentage de valeurs manquantes, et nombre de valeurs uniques.

2. Chargez le dataset Gapminder (`gapminder.csv` ou via `plotly.express`). Identifiez les colonnes numériques, catégorielles et celles à valeurs manquantes.

3. Écrivez une fonction `detect_mixed_types(df)` qui vérifie chaque colonne pour des types Python mélangés et renvoie la liste des colonnes problématiques.

4. Comparez le chargement du dataset Titanic en CSV vs Parquet. Mesurez taille de fichier et temps de chargement avec `%timeit`.

1.9 Résumé du chapitre

- Le prétraitement transforme les données brutes en données prêtes pour l'analyse et consomme la majorité du temps de projet.

- Les données existent en formats structurés, semi-structurés et non structurés ; pandas gère les deux premiers.

- Tout workflow de prétraitement suit cinq étapes : charger, inspecter, nettoyer, transformer, valider.
- Un rapport de qualité rapide (shape, dtypes, nuls, unicités, statistiques) devrait être votre première action sur n'importe quel nouveau dataset.
- La pile Python de base pour le prétraitement : pandas, numpy, scikit-learn, matplotlib, missingno.

Chapitre 2

Chargement des données

« On ne peut pas analyser ce qu'on ne peut pas charger. »

2.1 Charger des fichiers CSV

CSV (comma-separated values) est le format d'échange de données le plus courant. Pandas le gère avec `read_csv`, qui a plus de 50 paramètres pour traiter les particularités du monde réel.

```
import pandas as pd

# Chargement basique
df = pd.read_csv("titanic.csv")
print(df.shape)
print(df.head())
```

2.1.1 Problèmes CSV courants

```
# Probl\`eme 1 : mauvais d\'elimiteur
# Certains CSV europ\'eens utilisent des points-virgules
df_euro = pd.read_csv("data_eu.csv", sep=";", decimal=",")

# Probl\`eme 2 : encodages
df_fr = pd.read_csv("french_data.csv", encoding="latin-1")

# Probl\`eme 3 : ent\^etes d\'esordonn\'ees
df = pd.read_csv("messy.csv", header=0, skiprows=[1, 2])
df.columns = df.columns.str.strip().str.lower().str.replace(" ", "_")
print(df.columns.tolist())
```

Astuce prétraitement

Standardisez toujours les noms de colonnes immédiatement après chargement : minuscules, underscores, pas d'espaces. Ça évite des bugs subtils dus à une capitalisation incohérente.

2.1.2 Optimiser le chargement CSV

```
# Charger uniquement les colonnes n\'ecessaires (\'economie m\'emoire)
cols = ["PassengerId", "Survived", "Pclass", "Sex", "Age", "Fare"]
df = pd.read_csv("titanic.csv", usecols=cols)

# Sp\'ecifier les dtypes pour \'economiser la m\'emoire
dtypes = {"Survived": "int8", "Pclass": "int8", "Sex": "category"}
df = pd.read_csv("titanic.csv", usecols=cols, dtype=dtypes)
print(df.memory_usage(deep=True))

# Charger par morceaux pour les gros fichiers
chunks = pd.read_csv("large_file.csv", chunksize=50_000)
result = pd.concat([chunk[chunk["Age"] > 30] for chunk in chunks])
```

Attention

Ne chargez jamais un CSV de plusieurs gigaoctets entièrement en mémoire d'un coup. Utilisez `chunksize`, ou passez à Parquet/Feather. Un CSV de 2 Go fait généralement 400 Mo en Parquet avec des temps de chargement bien plus courts.

2.2 Charger des fichiers Excel

```
# Chargement Excel basique (n\'ecessite openpyxl)
df = pd.read_excel("survey_results.xlsx", sheet_name="Sheet1")

# Charger toutes les feuilles dans un dictionnaire
all_sheets = pd.read_excel("multi_sheet.xlsx", sheet_name=None)
for name, sheet_df in all_sheets.items():
    print(f"Sheet '{name}': {sheet_df.shape}")

# Sauter ent\^etes et pieds de page
df = pd.read_excel("report.xlsx", skiprows=3, skipfooter=2,
                   engine="openpyxl")
```

Astuce données

Les fichiers Excel préservent le formatage mais perdent la précision de type. Les dates peuvent être chargées comme chaînes, les pourcentages comme floats entre 0 et 1, les devises comme nombres bruts. Vérifiez toujours les dtypes après chargement Excel.

2.3 Charger des données JSON

```
import json
```

```
# JSON plat
df = pd.read_json("flat_data.json")

# JSON imbriqu\'e (courant depuis les API)
with open("nested_data.json") as f:
    raw = json.load(f)

# Utiliser json_normalize pour aplatir
from pandas import json_normalize

df = json_normalize(raw["results"],
                    record_path="measurements",
                    meta=["station_id", "station_name"])
print(df.head())
```

```
# Exemple r\'eel : charger un JSON de type Gapminder
import requests

url =
↪  "https://api.worldbank.org/v2/country/BEN/indicator/SP.POP.TOTL?format=json&per_page=60"
response = requests.get(url)
data = response.json()

# Les vraies donn\'ees sont dans data[1]
df_wb = pd.DataFrame(data[1])
print(df_wb[["date", "value"]].head(10))
```

2.4 Charger depuis des bases SQL

```
from sqlalchemy import create_engine

# SQLite (fichier, pas de serveur)
engine = create_engine("sqlite:///local_data.db")

# Charger toute la table
df = pd.read_sql("SELECT * FROM patients", engine)

# Charger avec filtrage (pousser le calcul en base)
query = """
SELECT patient_id, age, diagnosis, lab_value
FROM patients
WHERE age > 40
  AND diagnosis IS NOT NULL
ORDER BY lab_value DESC
LIMIT 10000
"""
df = pd.read_sql(query, engine)
print(df.shape)
```

Astuce prétraitement

En chargeant depuis SQL, poussez autant de filtrage que possible dans la requête. Charger 10 millions de lignes dans pandas puis filtrer est bien plus lent que laisser la base filtrer d'abord.

2.5 Charger depuis des API

```
import requests
import pandas as pd

# Exemple d'API REST : Open Meteo (m\'et\'eo)
url = "https://archive-api.open-meteo.com/v1/archive"
params = {
    "latitude": 6.36,    # Cotonou
    "longitude": 2.43,
    "start_date": "2024-01-01",
    "end_date": "2024-12-31",
    "daily": "temperature_2m_mean,precipitation_sum",
}

response = requests.get(url, params=params)
data = response.json()

df_weather = pd.DataFrame({
    "date": data["daily"]["time"],
    "temp_mean": data["daily"]["temperature_2m_mean"],
    "precip_mm": data["daily"]["precipitation_sum"],
})
df_weather["date"] = pd.to_datetime(df_weather["date"])
print(df_weather.head())
```

```
# Pagination : beaucoup d'API renvoient les donn\'ees par pages
all_records = []
page = 1

while True:
    resp = requests.get(f"{base_url}?page={page}&per_page=100")
    records = resp.json()["results"]
    if not records:
        break
    all_records.extend(records)
    page += 1

df = pd.DataFrame(all_records)
print(f"Total records: {len(df)}")
```

2.6 Bases du web scraping

```python
import requests
from bs4 import BeautifulSoup
import pandas as pd

# Scraper un tableau HTML
url = "https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)"
response = requests.get(url)
soup = BeautifulSoup(response.content, "html.parser")

# pandas peut lire les tableaux HTML directement
tables = pd.read_html(url)
print(f"Found {len(tables)} tables on the page")
gdp_df = tables[2]  # choisir le bon tableau par index
print(gdp_df.head())
```

Attention

Le web scraping peut violer les conditions d'utilisation d'un site. Vérifiez toujours `robots.txt` et les ToS avant de scraper. Pour la recherche, préférez les API officielles et les portails de données ouvertes.

2.7 Comparer les formats : performance

```python
import time

# Comparer les temps de chargement CSV vs Parquet
# D'abord, cr\'eer une version Parquet
df = pd.read_csv("large_dataset.csv")
df.to_parquet("large_dataset.parquet")

# Chronom\'etrer le chargement CSV
start = time.time()
df_csv = pd.read_csv("large_dataset.csv")
csv_time = time.time() - start

# Chronom\'etrer le chargement Parquet
start = time.time()
df_pq = pd.read_parquet("large_dataset.parquet")
pq_time = time.time() - start

print(f"CSV load:     {csv_time:.2f}s")
print(f"Parquet load: {pq_time:.2f}s")
print(f"Speedup:      {csv_time / pq_time:.1f}x")
```

2.8 Exercices

Exercice

1. Chargez le CSV Titanic avec des dtypes optimisés. Comparez l'usage mémoire avant et après optimisation.
2. Chargez des données depuis l'API Banque Mondiale pour trois pays africains (Bénin, Nigeria, Ghana) sur l'indicateur `SP.POP.TOTL` (population totale). Combinez en un seul DataFrame.
3. Chargez un fichier Excel à plusieurs feuilles. Écrivez une fonction qui renvoie un dictionnaire associant le nom de feuille à sa shape.
4. Scrapez un tableau HTML depuis Wikipedia et nettoyez les noms de colonnes (minuscules, underscores, pas de caractères spéciaux).
5. Convertissez le dataset Adult Census UCI de CSV vers Parquet. Comparez tailles de fichiers et temps de chargement.

2.9 Résumé du chapitre

- Pandas fournit des lecteurs pour CSV, Excel, JSON, SQL, Parquet, HTML, et plus.
- Le chargement CSV demande attention aux délimiteurs, encodages, entêtes et dtypes.
- Le JSON issu d'API a souvent besoin de `json_normalize` pour aplatir les structures imbriquées.
- Les chargements SQL devraient pousser le filtrage dans la requête, pas dans pandas.
- Les API demandent de gérer pagination, rate limits, et authentification.
- Parquet est 3 à 5 fois plus rapide que CSV au chargement et utilise 5 fois moins d'espace disque.

Chapitre 3

Valeurs manquantes

> *« Les données que vous n'avez pas sont souvent plus importantes que celles que vous avez. »* — Statisticien anonyme

3.1 Pourquoi des données manquent

Une valeur manquante n'est pas une nuisance — c'est une information. La *raison* pour laquelle une donnée manque détermine comment vous devriez la traiter. Trois mécanismes ont été formalisés par Rubin (1976) :

Définition 3.1 (Mécanismes de données manquantes)**.**

- ***MCAR*** *(Missing Completely At Random) : la manquance est indépendante de toute variable, observée ou non. Exemple : un échantillon de laboratoire est accidentellement tombé.*
- ***MAR*** *(Missing At Random) : la manquance dépend de variables observées mais pas de la valeur manquante elle-même. Exemple : les patients plus jeunes sautent un questionnaire de dépression.*
- ***MNAR*** *(Missing Not At Random) : la manquance dépend de la valeur non observée. Exemple : les hauts revenus refusent de déclarer leur revenu.*

Attention

On ne peut pas prouver statistiquement le MNAR — cela requiert de la connaissance métier. Mais la distinction compte énormément : l'imputation simple marche pour MCAR, est raisonnable pour MAR, et peut être fortement biaisée pour MNAR.

3.2 Détecter les valeurs manquantes

```python
import pandas as pd
import numpy as np

# Charger les donn\'ees Titanic
url = ("https://raw.githubusercontent.com/datasciencedojo/"
       "datasets/master/titanic.csv")
df = pd.read_csv(url)
```

```python
# Compter les valeurs manquantes
print(df.isnull().sum())
print()
print(f"Total missing cells: {df.isnull().sum().sum()}")
print(f"Percentage missing: {df.isnull().mean().mean() * 100:.1f}%")
```

```python
# Rapport d\'etaill\'e des manquantes
def missing_report(df):
    """Return a DataFrame summarizing missing data."""
    missing = df.isnull().sum()
    pct = (missing / len(df)) * 100
    report = pd.DataFrame({"missing": missing, "pct": pct.round(1)})
    return report[report["missing"] > 0].sort_values("pct", ascending=False)

print(missing_report(df))
```

3.2.1 Manquantes cachées

```python
# Les valeurs manquantes ne sont pas toujours NaN
# D\'eguisements courants : "?", "N/A", "", -1, 999, -999, "unknown"
df_adult = pd.read_csv(

    ↪ "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data",
    header=None,
    na_values=["?", " ?"],  # Adult utilise " ?" pour les manquantes
    names=["age", "workclass", "fnlwgt", "education", "education_num",
           "marital_status", "occupation", "relationship", "race",
           "sex", "capital_gain", "capital_loss", "hours_per_week",
           "native_country", "income"]
)
print(missing_report(df_adult))
```

Astuce données

Scrutez toujours les valeurs sentinelles : `-1`, `999`, `0` dans les colonnes numériques ; chaînes vides, « unknown », « N/A » dans les colonnes texte. Remplacez-les par `np.nan` avant toute analyse.

3.3 Visualiser les manquantes

```python
import missingno as msno
import matplotlib.pyplot as plt

# Matrix plot : lignes blanches = manquantes
msno.matrix(df, figsize=(10, 5), sparkline=False)
```

```
plt.title("Missing data pattern --- Titanic")
plt.tight_layout()
plt.savefig("missing_matrix.png", dpi=150)
plt.show()
```

```
# Bar chart : compte de non-nuls par colonne
msno.bar(df, figsize=(10, 4))
plt.tight_layout()
plt.show()

# Heatmap : corr\'elation de manquance entre colonnes
# Valeurs proches de +1 = colonnes manquantes ensemble
msno.heatmap(df, figsize=(8, 6))
plt.tight_layout()
plt.show()
```

```
# Dendrogramme : clustering hi\'erarchique des patterns de manquance
msno.dendrogram(df, figsize=(10, 5))
plt.tight_layout()
plt.show()
```

Astuce prétraitement

La heatmap de corrélation de manquance est extrêmement utile : si deux colonnes ont des manquances corrélées, elles partagent probablement une cause commune (par ex. même section de questionnaire que certains répondants ont sautée).

3.4 Stratégies de suppression

3.4.1 Suppression listwise (lignes)

```
# Supprimer les lignes avec UNE valeur manquante
df_complete = df.dropna()
print(f"Before: {len(df)}, After: {len(df_complete)}")
print(f"Lost: {len(df) - len(df_complete)} rows "
      f"({(len(df) - len(df_complete)) / len(df) * 100:.1f}%)")
```

3.4.2 Suppression de colonnes

```
# Supprimer les colonnes \`a plus de 50% manquantes
threshold = 0.5
cols_to_drop = df.columns[df.isnull().mean() > threshold].tolist()
print(f"Dropping columns: {cols_to_drop}")
df_reduced = df.drop(columns=cols_to_drop)
```

Attention

La suppression listwise n'est sûre que sous MCAR. Sous MAR ou MNAR, elle introduit un biais. Pour Titanic, les passagers plus âgés en classes supérieures ont plus souvent un âge renseigné — supprimer les âges manquants biaise le dataset vers les classes inférieures.

3.5 Imputation simple

3.5.1 Imputation par moyenne, médiane, mode

```
from sklearn.impute import SimpleImputer

# Num\'erique : imputation m\'ediane (robuste aux outliers)
num_imputer = SimpleImputer(strategy="median")
df["Age"] = num_imputer.fit_transform(df[["Age"]])

# Cat\'egoriel : imputation par mode (le plus fr\'equent)
cat_imputer = SimpleImputer(strategy="most_frequent")
df["Embarked"] = cat_imputer.fit_transform(df[["Embarked"]]).ravel()

print(df.isnull().sum())
```

3.5.2 Imputation constante

```
# Remplir avec une valeur sp\'ecifique
df["Cabin"] = df["Cabin"].fillna("Unknown")

# Remplir num\'erique avec 0 (\`a utiliser avec prudence)
df["some_col"] = df["some_col"].fillna(0)
```

3.6 Imputation avancée

3.6.1 Imputation KNN

```
from sklearn.impute import KNNImputer

# KNN utilise des lignes similaires pour estimer les manquantes
# Ne fonctionne que sur des donn\'ees num\'eriques
num_cols = ["Age", "Fare", "SibSp", "Parch"]
knn_imputer = KNNImputer(n_neighbors=5, weights="distance")
df[num_cols] = knn_imputer.fit_transform(df[num_cols])

print(f"Missing after KNN: {df[num_cols].isnull().sum().sum()}")
```

3.6.2 Imputation itérative (MICE)

```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

# L'imputeur it\'eratif mod\'elise chaque variable en fonction des autres
iter_imputer = IterativeImputer(max_iter=10, random_state=42)
df[num_cols] = iter_imputer.fit_transform(df[num_cols])

print("Iterative imputation complete.")
print(df[num_cols].describe())
```

Astuce prétraitement

L'imputation KNN respecte la structure locale (des passagers similaires reçoivent des âges similaires). L'imputation itérative capture les relations linéaires entre variables. Les deux surpassent l'imputation moyenne/médiane quand les données sont MAR.

3.7 Imputation avec indicateur

```
# Cr\'eer un indicateur binaire AVANT d'imputer
df["Age_was_missing"] = df["Age"].isnull().astype(int)

# Puis imputer
df["Age"] = df["Age"].fillna(df["Age"].median())

# L'indicateur pr\'eserve l'information que la valeur \'etait manquante
print(df[["Age", "Age_was_missing"]].head(10))
```

Astuce données

Ajouter une colonne indicatrice est l'une des techniques les plus sous-utilisées. Elle laisse les modèles en aval apprendre si la manquance elle-même est prédictive. Pour Titanic, avoir un numéro de cabine manquant est corrélé à une moindre survie.

3.8 Choisir une stratégie d'imputation

Méthode	Bon pour	Faiblesse	Mécanisme
Supprimer lignes	<5% manquantes	Perte de données	MCAR seul
Moyenne/médiane	Baseline rapide	Réduit la variance	MCAR

Mode	Catégoriel	Ignore la structure	MCAR
KNN	Données clusterisées	Lent sur grand volume	MAR
Itératif (MICE)	Relations linéaires	Suppose linéarité	MAR
Constante métier	Sens connu	Peut biaiser	Tout

3.9 Exercices

Exercice

1. Chargez le dataset Melbourne Housing. Créez un rapport de manquantes et un missingno matrix plot. Quelles colonnes ont des manquances corrélées ?
2. Pour Titanic, comparez la distribution de `Age` avant et après imputation moyenne vs imputation KNN. Tracez les deux distributions sur le même histogramme.
3. Chargez Adult Census (avec `na_values=[" ?"]`) et imputez les trois colonnes catégorielles à valeurs manquantes par mode. Vérifiez qu'il ne reste aucune manquante.
4. Implémentez une fonction `smart_impute(df)` qui : (a) supprime les colonnes à >70% manquantes, (b) impute les numériques par médiane, (c) impute les catégorielles par mode, (d) ajoute des indicateurs pour toutes les colonnes initialement manquantes.
5. Investiguez si `Age` dans Titanic est MCAR en comparant le taux de survie des passagers avec et sans âge renseigné. Que concluez-vous ?

3.10 Résumé du chapitre

- Les données manquantes ont trois mécanismes : MCAR, MAR et MNAR. Le mécanisme détermine quelle imputation est valide.
- Scrutez toujours les valeurs sentinelles cachées (`?`, `-1`, `999`).
- La bibliothèque `missingno` fournit visualisations matrix, bar, heatmap et dendrogramme.
- L'imputation simple (moyenne, médiane, mode) est rapide mais réduit la variance et ne marche bien que sous MCAR.
- KNN et MICE capturent les relations entre variables et fonctionnent sous MAR.
- Ajouter une colonne indicatrice de manquance préserve l'information contenue dans le fait qu'une valeur était manquante.

Chapitre 4

Détection et traitement des valeurs aberrantes

> *« Une valeur aberrante n'est pas du bruit tant que vous n'avez pas prouvé que c'est du bruit. »*

4.1 Qu'est-ce qu'une valeur aberrante ?

Une valeur aberrante (outlier) est une observation qui s'écarte substantiellement du motif établi par la majorité des données. Les outliers peuvent être :

- **Erreurs de saisie** : un poids de patient de 800 kg (au lieu de 80).
- **Artefacts de mesure** : un capteur renvoyant -999 pendant la calibration.
- **Vraies valeurs extrêmes** : une maison vendue 10 millions de dollars dans une banlieue où la médiane est 500 000$.

⚠ Attention

Ne supprimez jamais d'outliers automatiquement. Chaque outlier doit être investigué. Supprimer de vraies valeurs extrêmes parce qu'elles « ont l'air fausses » corrompt votre analyse. Supprimer des erreurs de données parce qu'elles « ont l'air inhabituelles » est correct et nécessaire.

4.2 Détection visuelle

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Le dataset FIFA 21 contient beaucoup de valeurs extr\^emes et
↪  d\'esordonn\'ees
df = pd.read_csv("fifa21_raw.csv")

# Les box plots r\'ev\`elent les outliers imm\'ediatement
```

```
fig, axes = plt.subplots(1, 3, figsize=(14, 5))

sns.boxplot(y=df["Age"], ax=axes[0])
axes[0].set_title("Age distribution")

sns.boxplot(y=df["Overall"], ax=axes[1])
axes[1].set_title("Overall rating")

sns.boxplot(y=df["Value(in Euro)"], ax=axes[2])
axes[2].set_title("Market value (Euro)")

plt.tight_layout()
plt.savefig("outlier_boxplots.png", dpi=150)
plt.show()
```

```
# Scatter plot : outliers dans l'espace 2D
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(df["Age"], df["Overall"], alpha=0.3, s=10)
ax.set_xlabel("Age")
ax.set_ylabel("Overall Rating")
ax.set_title("Age vs Overall --- look for isolated points")
plt.tight_layout()
plt.show()
```

4.3 Méthodes statistiques

4.3.1 Méthode du z-score

```
import numpy as np
from scipy import stats

# Z-score : nombre d'\'ecarts-types par rapport \`a la moyenne
z_scores = np.abs(stats.zscore(df["Overall"].dropna()))
threshold = 3
outliers_z = (z_scores > threshold)
print(f"Outliers by z-score (|z| > {threshold}): {outliers_z.sum()}")
```

```
# Appliquer le z-score \`a plusieurs colonnes
def detect_zscore_outliers(df, columns, threshold=3):
    """Return a boolean mask of rows with any z-score outlier."""
    mask = pd.Series(False, index=df.index)
    for col in columns:
        z = np.abs(stats.zscore(df[col].dropna()))
        col_mask = pd.Series(False, index=df.index)
        col_mask.iloc[:len(z)] = z > threshold
        mask = mask | col_mask
    return mask
```

```
num_cols = ["Age", "Overall", "Potential"]
outlier_mask = detect_zscore_outliers(df, num_cols)
print(f"Rows with at least one outlier: {outlier_mask.sum()}")
```

Astuce données

La méthode du z-score suppose une distribution à peu près normale. Pour des données asymétriques (revenus, prix immobiliers, valeurs de marché de joueurs), utilisez la méthode IQR ou transformez en log d'abord.

4.3.2 Méthode IQR

```
def iqr_bounds(series):
    """Return lower and upper IQR bounds."""
    Q1 = series.quantile(0.25)
    Q3 = series.quantile(0.75)
    IQR = Q3 - Q1
    lower = Q1 - 1.5 * IQR
    upper = Q3 + 1.5 * IQR
    return lower, upper

lower, upper = iqr_bounds(df["Overall"])
outliers_iqr = df[(df["Overall"] < lower) | (df["Overall"] > upper)]
print(f"IQR bounds: [{lower:.1f}, {upper:.1f}]")
print(f"Outliers: {len(outliers_iqr)}")
```

```
# Rapport IQR pour toutes les colonnes num\'eriques
def iqr_outlier_report(df, columns):
    """Generate an IQR outlier report."""
    rows = []
    for col in columns:
        lo, hi = iqr_bounds(df[col].dropna())
        n_out = ((df[col] < lo) | (df[col] > hi)).sum()
        rows.append({"column": col, "lower": lo, "upper": hi,
                     "n_outliers": n_out,
                     "pct": round(n_out / len(df) * 100, 1)})
    return pd.DataFrame(rows)

report = iqr_outlier_report(df, ["Age", "Overall", "Potential"])
print(report)
```

4.4 Détection multivariée

4.4.1 Distance de Mahalanobis

```
from scipy.spatial.distance import mahalanobis
```

```python
from numpy.linalg import inv

def mahalanobis_outliers(df, columns, threshold=3):
    """Detect multivariate outliers using Mahalanobis distance."""
    data = df[columns].dropna()
    mean = data.mean().values
    cov_matrix = data.cov().values
    cov_inv = inv(cov_matrix)

    distances = data.apply(
        lambda row: mahalanobis(row.values, mean, cov_inv), axis=1
    )
    return distances, distances > threshold

cols = ["Age", "Overall", "Potential"]
distances, mask = mahalanobis_outliers(df, cols)
print(f"Multivariate outliers: {mask.sum()}")
```

Astuce prétraitement

La distance de Mahalanobis détecte des points qui ne sont aberrants sur aucune dimension individuelle mais qui sont inhabituels dans la *combinaison* de dimensions. Un joueur de 40 ans est normal. Un joueur noté 95 est normal. Un joueur de 40 ans noté 95 est aberrant dans l'espace joint.

4.4.2 Isolation Forest

```python
from sklearn.ensemble import IsolationForest

# Isolation Forest : d\'etection d'anomalies par arbres
iso = IsolationForest(contamination=0.05, random_state=42)
num_data = df[["Age", "Overall", "Potential"]].dropna()
labels = iso.fit_predict(num_data)

# -1 = outlier, 1 = inlier
n_outliers = (labels == -1).sum()
print(f"Isolation Forest outliers: {n_outliers} "
      f"({n_outliers / len(num_data) * 100:.1f}%)")
```

4.5 Règles métier

```python
# Air Quality : la connaissance m\'etier compte
df_air = pd.read_csv("AirQualityUCI.csv", sep=";", decimal=",")

# Le dataset utilise -200 comme sentinelle pour donn\'ees manquantes
print(f"Values == -200: {(df_air == -200).sum().sum()}")
```

```
# Remplacer la sentinelle par NaN
df_air = df_air.replace(-200, np.nan)

# R\`egle m\'etier : une concentration de CO > 20 mg/m3
# est physiquement implausible en station urbaine
co_col = "CO(GT)"
implausible = df_air[co_col] > 20
print(f"Implausible CO readings: {implausible.sum()}")
df_air.loc[implausible, co_col] = np.nan
```

Astuce données

Les règles métier sont la méthode de détection d'outliers la plus fiable. Un z-score ne sait pas qu'un humain ne peut pas peser 800 kg. Consultez toujours des experts métier ou des plages de référence quand elles existent.

4.6 Stratégies de traitement

```
# Strat\'egie 1 : supprimer les lignes aberrantes
df_clean = df[~outlier_mask].copy()

# Strat\'egie 2 : capper (winsorize) aux bornes
from scipy.stats import mstats
df["Overall_winsorized"] = mstats.winsorize(df["Overall"], limits=[0.01, 0.01])

# Strat\'egie 3 : remplacer par NaN et imputer plus tard
df.loc[outlier_mask, "Overall"] = np.nan

# Strat\'egie 4 : transformation log pour r\'eduire l'asym\'etrie
df["Value_log"] = np.log1p(df["Value(in Euro)"].clip(lower=0))

# Strat\'egie 5 : mise \`a l'\'echelle robuste (moins sensible aux outliers)
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()  # utilise m\'ediane et IQR au lieu de moyenne et std
df[["Overall_robust"]] = scaler.fit_transform(df[["Overall"]])
```

```
# Comparer les distributions avant et apr\`es winsorisation
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

df["Overall"].hist(bins=50, ax=axes[0], alpha=0.7)
axes[0].set_title("Original")

df["Overall_winsorized"].hist(bins=50, ax=axes[1], alpha=0.7, color="green")
axes[1].set_title("Winsorized (1% each tail)")

plt.tight_layout()
plt.show()
```

4.7 Exercices

Exercice

1. Chargez le dataset FIFA 21. Créez des box plots pour cinq colonnes numériques. Identifiez par IQR les colonnes avec le plus d'outliers.
2. Pour Air Quality UCI, remplacez toutes les sentinelles -200 par NaN. Puis appliquez des règles métier pour signaler les valeurs implausibles sur au moins deux colonnes de polluants.
3. Comparez les nombres d'outliers par z-score et par IQR sur une colonne asymétrique (par ex. valeur de marché). Quelle méthode est plus appropriée et pourquoi ?
4. Utilisez Isolation Forest sur Melbourne Housing (Price, Rooms, Distance, Landsize). Visualisez les outliers détectés sur un scatter plot Price vs Landsize.
5. Implémentez une fonction `treat_outliers(series, method)` qui supporte trois méthodes : « remove », « cap », « log ». Testez-la sur une colonne avec outliers connus.

4.8 Résumé du chapitre

- Les outliers peuvent être des erreurs, des artefacts ou de vraies valeurs extrêmes — l'investigation est obligatoire.
- Les méthodes visuelles (box plots, scatter plots) donnent une intuition immédiate.
- Le z-score marche pour des données à peu près normales ; l'IQR est robuste aux distributions asymétriques.
- La distance de Mahalanobis et Isolation Forest détectent des outliers multivariés invisibles aux méthodes univariées.
- Les règles métier (limites physiques, sentinelles) sont la méthode la plus fiable.
- Les options de traitement : suppression, capping (winsorisation), transformation log, mise à l'échelle robuste.

Chapitre 5

Transformations de types de données

« Les algorithmes de machine learning ne comprennent pas les catégories — ils comprennent les nombres. »

5.1 Pourquoi transformer les types ?

La plupart des algorithmes ML demandent une entrée numérique. Les datasets bruts contiennent des variables catégorielles (« Male », « Female »), des variables ordinales (« Low », « Medium », « High »), et des variables numériques sur des échelles très différentes (âge en années, revenu en milliers). Un encodage et une mise à l'échelle propres sont essentiels.

Astuce données

La transformation de type a deux faces : l'**encodage** convertit les catégories en nombres, la **mise à l'échelle** place les nombres sur des plages comparables. Les deux sont nécessaires pour la plupart des pipelines ML.

5.2 Encodage catégoriel

5.2.1 Encodage ordinal (label)

```
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder

# Adult Census
url = ("https://archive.ics.uci.edu/ml/machine-learning-databases/"
       "adult/adult.data")
cols = ["age", "workclass", "fnlwgt", "education", "education_num",
        "marital_status", "occupation", "relationship", "race",
        "sex", "capital_gain", "capital_loss", "hours_per_week",
        "native_country", "income"]
df = pd.read_csv(url, header=None, names=cols, na_values=" ?",
                 skipinitialspace=True)

# Encodage ordinal : pour les variables avec un ordre naturel
```

```
edu_order = [["Preschool", "1st-4th", "5th-6th", "7th-8th", "9th",
              "10th", "11th", "12th", "HS-grad", "Some-college",
              "Assoc-voc", "Assoc-acdm", "Bachelors", "Masters",
              "Prof-school", "Doctorate"]]

enc = OrdinalEncoder(categories=edu_order,
                     handle_unknown="use_encoded_value",
                     unknown_value=-1)
df["education_ord"] = enc.fit_transform(df[["education"]])
print(df[["education", "education_ord"]].drop_duplicates()
        .sort_values("education_ord"))
```

Attention

L'encodage ordinal impose un ordre numérique. L'utiliser sur des catégories nominales (noms de pays par exemple) implique « France < Allemagne < Japon », ce qui n'a aucun sens et peut tromper les modèles.

5.2.2 One-hot encoding

```
from sklearn.preprocessing import OneHotEncoder

# One-hot encoding : pour les cat\'egories nominales (sans ordre)
ohe = OneHotEncoder(sparse_output=False, drop="first",
                    handle_unknown="ignore")
encoded = ohe.fit_transform(df[["sex", "race"]])
encoded_df = pd.DataFrame(encoded,
                          columns=ohe.get_feature_names_out())
print(encoded_df.head())
print(f"New columns: {encoded_df.shape[1]}")
```

```
# pandas get_dummies : one-hot rapide
df_encoded = pd.get_dummies(df, columns=["sex", "race"],
                            drop_first=True, dtype=int)
print(df_encoded.columns.tolist()[-10:])
```

Astuce prétraitement

Utilisez `drop="first"` (ou `drop_first=True`) pour éviter la multicolinéarité. Avec k catégories, il ne faut que $k-1$ variables indicatrices — la catégorie supprimée devient le niveau de référence.

5.2.3 Target encoding

```
# Target encoding : remplacer chaque cat\'egorie par la moyenne de la cible
# Utile pour les colonnes \`a forte cardinalit\'e (100+ cat\'egories)
```

```
target_means = df.groupby("occupation")["income"].apply(
    lambda x: (x == ">50K").mean()
).to_dict()

df["occupation_target_enc"] = df["occupation"].map(target_means)
print(df[["occupation", "occupation_target_enc"]]
        .drop_duplicates().sort_values("occupation_target_enc"))
```

Attention

Le target encoding cause une fuite de données s'il est appliqué avant la séparation train/test. Ajustez toujours sur l'entraînement seul, puis transformez les deux. Utilisez `cross_val_predict` ou un target encoding par K-fold pour réduire le surapprentissage.

5.2.4 Encodages fréquence et binaire

```
# Encodage par fr\'equence : remplacer la cat\'egorie par sa fr\'equence
freq = df["native_country"].value_counts(normalize=True)
df["country_freq"] = df["native_country"].map(freq)
print(df[["native_country", "country_freq"]].head(10))

# Binary encoding : moins de colonnes que one-hot pour forte cardinalit\'e
# La biblioth\`eque category-encoders fournit cela
# pip install category-encoders
# import category_encoders as ce
# encoder = ce.BinaryEncoder(cols=["native_country"])
# df_binary = encoder.fit_transform(df)
```

5.3 Mise à l'échelle numérique

5.3.1 StandardScaler (normalisation z-score)

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df[["age_scaled", "hours_scaled"]] = scaler.fit_transform(
    df[["age", "hours_per_week"]]
)

print(df[["age", "age_scaled", "hours_per_week", "hours_scaled"]].describe())
```

5.3.2 MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler
```

```
minmax = MinMaxScaler(feature_range=(0, 1))
df[["age_minmax"]] = minmax.fit_transform(df[["age"]])

# Apr\`es mise \`a l'\'echelle : min=0, max=1
print(df["age_minmax"].describe())
```

5.3.3 RobustScaler

```
from sklearn.preprocessing import RobustScaler

# Utilise m\'ediane et IQR : robuste aux outliers
robust = RobustScaler()
df[["capital_gain_robust"]] = robust.fit_transform(df[["capital_gain"]])
print(df["capital_gain_robust"].describe())
```

Astuce prétraitement

Choisissez votre scaler selon les données et le modèle :

- **StandardScaler** : choix par défaut pour la plupart des algorithmes (SVM, régression logistique, réseaux de neurones).
- **MinMaxScaler** : quand on veut des valeurs bornées (sigmoid en sortie de réseau).
- **RobustScaler** : quand les données contiennent des outliers significatifs.

5.4 Comparer les scalers visuellement

```
import matplotlib.pyplot as plt
import numpy as np

fig, axes = plt.subplots(1, 4, figsize=(16, 4))

# Original
df["capital_gain"].hist(bins=50, ax=axes[0])
axes[0].set_title("Original")

# Standard
axes[1].hist(StandardScaler().fit_transform(df[["capital_gain"]]),
             bins=50)
axes[1].set_title("StandardScaler")

# MinMax
axes[2].hist(MinMaxScaler().fit_transform(df[["capital_gain"]]),
             bins=50)
axes[2].set_title("MinMaxScaler")
```

```
# Robust
axes[3].hist(RobustScaler().fit_transform(df[["capital_gain"]]),
             bins=50)
axes[3].set_title("RobustScaler")

plt.tight_layout()
plt.savefig("scaler_comparison.png", dpi=150)
plt.show()
```

5.5 Discrétisation (binning)

```
# Binning par largeur \'egale
df["age_bin_width"] = pd.cut(df["age"], bins=5,
                             labels=["very_young", "young", "mid",
                                     "senior", "elderly"])

# Binning par fr\'equence (quantiles)
df["age_bin_quantile"] = pd.qcut(df["age"], q=5,
                                 labels=["Q1", "Q2", "Q3", "Q4", "Q5"])

# Bins personnalis\'es selon connaissance m\'etier
df["age_group"] = pd.cut(df["age"],
                         bins=[0, 25, 35, 50, 65, 100],
                         labels=["<25", "25-34", "35-49", "50-64", "65+"])

print(df["age_group"].value_counts().sort_index())
```

```
# KBinsDiscretizer de sklearn
from sklearn.preprocessing import KBinsDiscretizer

kbd = KBinsDiscretizer(n_bins=5, encode="ordinal", strategy="quantile")
df[["hours_binned"]] = kbd.fit_transform(df[["hours_per_week"]])
print(df[["hours_per_week", "hours_binned"]].head(10))
```

5.6 Transformations de puissance

```
from sklearn.preprocessing import PowerTransformer

# Yeo-Johnson : marche avec valeurs positives et n\'egatives
pt = PowerTransformer(method="yeo-johnson")
df[["capital_gain_yj"]] = pt.fit_transform(df[["capital_gain"]])

# Box-Cox : ne marche que pour valeurs strictement positives
# pt_bc = PowerTransformer(method="box-cox")

# Transformation log : alternative manuelle pour donn\'ees asym\'etriques
```

```
df["capital_gain_log"] = np.log1p(df["capital_gain"])

print(f"Original skew:    {df['capital_gain'].skew():.2f}")
print(f"Yeo-Johnson skew: {df['capital_gain_yj'].skew():.2f}")
print(f"Log skew:         {df['capital_gain_log'].skew():.2f}")
```

5.7 Exercices

Exercice

1. Chargez Titanic. Appliquez un one-hot encoding sur `Sex` et `Embarked`. Appliquez un encodage ordinal sur `Pclass` ($1^{re} > 2^{e} > 3^{e}$). Montrez le DataFrame résultant.

2. Pour Adult Census, comparez l'effet de StandardScaler, MinMaxScaler et RobustScaler sur `capital_gain`. Tracez les trois distributions.

3. Implémentez un target encoding sur `native_country` avec validation croisée 5-fold pour prévenir la fuite. Comparez aux valeurs d'un target encoding naïf.

4. Créez des groupes d'âge pour Titanic avec des bins métier (enfant, adolescent, jeune adulte, age moyen, senior). Calculez le taux de survie par groupe d'âge.

5. Appliquez une transformation de puissance Yeo-Johnson sur toutes les colonnes numériques d'Adult. Comparez la skewness avant et après.

5.8 Résumé du chapitre

- L'encodage catégoriel convertit du non-numérique en nombres : ordinal pour catégories ordonnées, one-hot pour nominales, target encoding pour forte cardinalité.
- La mise à l'échelle place les variables numériques sur des plages comparables : StandardScaler par défaut, MinMaxScaler pour plages bornées, RobustScaler en présence d'outliers.
- La discrétisation (binning) convertit les variables continues en catégories, utile pour relations non linéaires et interprétabilité.
- Les transformations de puissance (Yeo-Johnson, Box-Cox, log) réduisent l'asymétrie et peuvent améliorer la performance des modèles.
- Toujours ajuster les transformateurs sur les données d'entraînement seules, puis les appliquer aux données de test.

Chapitre 6

Ingénierie de variables (feature engineering)

> *« Trouver de bonnes variables est difficile, chronophage, demande de l'expertise. Le machine learning appliqué, c'est essentiellement du feature engineering. »* — Andrew Ng

6.1 Qu'est-ce que le feature engineering ?

Le feature engineering est le processus de création de nouvelles variables d'entrée à partir des données existantes pour améliorer la performance du modèle. Alors que l'encodage et la mise à l'échelle (Chapitre 5) transforment les variables existantes, le feature engineering *crée* de nouvelles variables qui capturent des motifs que les données brutes ne représentent pas explicitement.

Astuce données

Une variable bien construite peut valoir plus qu'un modèle complexe. Une simple régression linéaire avec les bonnes variables surpasse souvent un réseau profond avec des variables brutes.

6.2 Variables polynomiales et d'interaction

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Melbourne Housing
df = pd.read_csv("melb_data.csv")
df = df.dropna(subset=["Price", "Rooms", "Distance", "Landsize"])

# Variables polynomiales : capturer des relations non lin\'eaires
poly = PolynomialFeatures(degree=2, interaction_only=False,
                          include_bias=False)
features = df[["Rooms", "Distance"]]
poly_features = poly.fit_transform(features)
```

```
poly_names = poly.get_feature_names_out(["Rooms", "Distance"])

poly_df = pd.DataFrame(poly_features, columns=poly_names)
print(poly_df.head())
print(f"Original features: {features.shape[1]}")
print(f"Polynomial features: {poly_df.shape[1]}")
```

```
# Interactions seules : pas de termes au carr\'e, seulement les produits
↪  crois\'es
poly_inter = PolynomialFeatures(degree=2, interaction_only=True,
                                include_bias=False)
inter_features = poly_inter.fit_transform(features)
inter_names = poly_inter.get_feature_names_out(["Rooms", "Distance"])
print(f"Interaction features: {inter_names}")
```

Attention

Les variables polynomiales croissent combinatoirement : n variables au degré d produisent $\binom{n+d}{d} - 1$ variables. Avec 20 variables au degré 3, on obtient 1770 variables. À utiliser sélectivement, pas sur tout le jeu de variables.

6.3 Transformations mathématiques

```
# Variables de ratio : souvent plus informatives que les valeurs brutes
df["price_per_room"] = df["Price"] / df["Rooms"]
df["price_per_sqm"] = df["Price"] / df["Landsize"].replace(0, np.nan)
df["rooms_per_bathroom"] = df["Rooms"] / df["Bathroom"].replace(0, 1)

print(df[["Price", "Rooms", "price_per_room",
          "price_per_sqm"]].describe())
```

```
# Transformation log pour les variables asym\'etriques
df["log_price"] = np.log1p(df["Price"])
df["log_landsize"] = np.log1p(df["Landsize"])

# Racine carr\'ee : plus douce que log
df["sqrt_distance"] = np.sqrt(df["Distance"])

# Comparer la corr\'elation avec la cible
for col in ["Distance", "sqrt_distance", "log_price"]:
    if col in df.columns and df[col].notnull().sum() > 0:
        corr = df[col].corr(df["Price"])
        print(f"Corr({col}, Price) = {corr:.3f}")
```

Astuce prétraitement

Les variables de ratio encodent de la *connaissance métier* en un seul nombre. « Prix au mètre carré » est une métrique immobilière standard qui capture mieux la densité de valeur que prix ou surface seuls.

6.4 Variables de date et d'heure

```
# Melbourne Housing : extraire des composantes de date
df["Date"] = pd.to_datetime(df["Date"], dayfirst=True)

df["sale_year"] = df["Date"].dt.year
df["sale_month"] = df["Date"].dt.month
df["sale_dayofweek"] = df["Date"].dt.dayofweek  # 0 = lundi
df["sale_quarter"] = df["Date"].dt.quarter
df["sale_is_weekend"] = df["Date"].dt.dayofweek.isin([5, 6]).astype(int)

print(df[["Date", "sale_year", "sale_month",
          "sale_dayofweek", "sale_quarter"]].head())
```

```
# Encodage cyclique du mois (pour que d\'ecembre et janvier soient proches)
df["month_sin"] = np.sin(2 * np.pi * df["sale_month"] / 12)
df["month_cos"] = np.cos(2 * np.pi * df["sale_month"] / 12)

# Jours depuis une date de r\'ef\'erence
reference = pd.Timestamp("2016-01-01")
df["days_since_ref"] = (df["Date"] - reference).dt.days

print(df[["sale_month", "month_sin", "month_cos"]].head(12))
```

6.5 Variables d'agrégation

```
# Statistiques au niveau du quartier
suburb_stats = df.groupby("Suburb")["Price"].agg(
    suburb_mean_price="mean",
    suburb_median_price="median",
    suburb_price_std="std",
    suburb_n_sales="count"
).reset_index()

df = df.merge(suburb_stats, on="Suburb", how="left")

# Comment ce bien se compare-t-il \`a la moyenne du quartier ?
df["price_vs_suburb"] = df["Price"] / df["suburb_mean_price"]

print(df[["Suburb", "Price", "suburb_mean_price",
```

```
          "price_vs_suburb"]].head(10))
```

```
# Gapminder : variables par pays dans le temps
gap = pd.read_csv("gapminder.csv")

# Taux de croissance ann\'ee sur ann\'ee
gap = gap.sort_values(["country", "year"])
gap["gdp_growth"] = gap.groupby("country")["gdpPercap"].pct_change()
gap["pop_growth"] = gap.groupby("country")["pop"].pct_change()

# Moyenne mobile (tendance liss\'ee)
gap["lifeExp_rolling3"] = (gap.groupby("country")["lifeExp"]
                           .transform(lambda x: x.rolling(3).mean()))

print(gap[gap["country"] == "Benin"][
    ["year", "lifeExp", "lifeExp_rolling3", "gdp_growth"]].tail(6))
```

6.6 Variables dérivées du texte

```
# Extraire des variables de colonnes texte sans NLP complet
# Sur les noms de quartiers et adresses Melbourne

# Longueur de l'adresse
df["address_length"] = df["Address"].str.len()
df["address_word_count"] = df["Address"].str.split().str.len()

# Contient des mots-cl\'es sp\'ecifiques
df["is_street"] = df["Address"].str.contains("St$", regex=True).astype(int)
df["is_road"] = df["Address"].str.contains("Rd$", regex=True).astype(int)
df["is_avenue"] = df["Address"].str.contains("Av", regex=False).astype(int)

print(df[["Address", "address_length", "is_street",
          "is_road"]].head(10))
```

6.7 Variables métier

```
# Connaissance m\'etier immobilier
df["total_rooms"] = df["Rooms"] + df["Bathroom"] + df["Car"]
df["has_multiple_bathrooms"] = (df["Bathroom"] > 1).astype(int)
df["is_new_build"] = (df["YearBuilt"] > 2010).astype(int)
df["building_age"] = df["sale_year"] - df["YearBuilt"]
df["is_close_to_cbd"] = (df["Distance"] < 10).astype(int)

# Indicateurs de type de bien
df["is_house"] = (df["Type"] == "h").astype(int)
df["is_unit"] = (df["Type"] == "u").astype(int)
```

```
# Interaction : grande maison proche du centre
df["large_close"] = df["is_house"] * df["is_close_to_cbd"] * df["Rooms"]

print(df[["Suburb", "Type", "Rooms", "Distance",
          "building_age", "large_close"]].head(10))
```

Astuce données

Les meilleures variables viennent de la compréhension du domaine. Passez du temps avec les experts métier. Demandez : « Quels facteurs les professionnels du domaine considèrent-ils dans leurs décisions ? » Encodez ces facteurs en variables.

6.8 Sélection de variables après engineering

```
from sklearn.feature_selection import mutual_info_regression

# S'electionner les variables num'eriques
feature_cols = ["Rooms", "Distance", "Landsize", "building_age",
                "price_per_room", "suburb_mean_price", "large_close",
                "days_since_ref", "Bathroom", "Car"]
target = "Price"

# Supprimer les NaN pour cette analyse
subset = df[feature_cols + [target]].dropna()

mi = mutual_info_regression(subset[feature_cols], subset[target],
                            random_state=42)
mi_series = pd.Series(mi, index=feature_cols).sort_values(ascending=False)

print("Mutual information with Price:")
print(mi_series)
```

```
# Matrice de corr'elation des variables construites
import seaborn as sns
import matplotlib.pyplot as plt

corr = subset[feature_cols].corr()
fig, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(corr, annot=True, fmt=".2f", cmap="coolwarm",
            center=0, ax=ax)
plt.title("Feature correlation matrix")
plt.tight_layout()
plt.savefig("feature_correlation.png", dpi=150)
plt.show()
```

6.9 Exercices

Exercice

1. Pour Melbourne Housing, créez au moins 5 nouvelles variables avec connaissance métier. Classez-les par information mutuelle avec Price.
2. Créez un encodage cyclique du mois de vente. Vérifiez que décembre et janvier sont proches dans l'espace encodé en calculant leur distance euclidienne.
3. Avec Gapminder, créez des taux de croissance année sur année pour GDP per capita et espérance de vie. Quel pays a connu la plus forte croissance de PIB sur une période ?
4. Construisez des variables polynomiales (degré 2) pour `Rooms`, `Distance`, et `Landsize`. Entraînez une régression linéaire avec et sans variables polynomiales et comparez le R^2.
5. Créez une fonction d'engineering `engineer_melb(df)` qui prend le dataset Melbourne brut et renvoie un DataFrame avec au moins 10 nouvelles variables, prêt pour la modélisation.

6.10 Résumé du chapitre

- Le feature engineering crée de nouvelles variables qui capturent des motifs invisibles dans les données brutes.
- Les variables polynomiales et d'interaction capturent les relations non linéaires et inter-variables.
- Les variables de ratio (prix par pièce, prix au mètre carré) encodent la connaissance métier de façon compacte.
- Les variables de date (année, mois, jour de semaine, encodage cyclique) déverrouillent les motifs temporels.
- Les variables d'agrégation (moyennes de groupe, moyennes mobiles) ajoutent du contexte.
- La connaissance métier est la source la plus précieuse d'idées de variables — consultez les experts.
- Après l'engineering, utilisez l'information mutuelle ou la corrélation pour sélectionner les variables les plus prédictives.

Chapitre 7

Prétraitement de texte

« Le langage naturel, c'est la donnée la moins structurée qui soit. »

7.1 Pourquoi le texte demande du prétraitement

Les données textuelles ne peuvent pas être fournies directement à des modèles numériques. Il faut nettoyer, normaliser, tokeniser, et convertir en représentations numériques. La qualité du prétraitement détermine directement la qualité de toute tâche TAL en aval.

Astuce données

Le prétraitement de texte dépend de la langue. Règles de tokenisation, stopwords et algorithmes de stemming diffèrent entre français et anglais. Ce chapitre couvre les deux quand c'est pertinent.

7.2 Charger des datasets textuels

```
from sklearn.datasets import fetch_20newsgroups

# 20 Newsgroups : dataset classique de classification de texte
newsgroups = fetch_20newsgroups(subset="train",
                                remove=("headers", "footers", "quotes"))

print(f"Number of documents: {len(newsgroups.data)}")
print(f"Number of categories: {len(newsgroups.target_names)}")
print(f"\nCategories: {newsgroups.target_names}")
print(f"\nSample document:\n{newsgroups.data[0][:300]}")
```

```
import pandas as pd

# Cr\'eer un DataFrame pour faciliter la manipulation
df = pd.DataFrame({
    "text": newsgroups.data,
    "category": [newsgroups.target_names[i] for i in newsgroups.target]
})
```

```
print(df["category"].value_counts().head(10))
```

7.3 Nettoyage de texte

```
import re

def clean_text(text):
    """Pipeline basique de nettoyage."""
    # Minuscules
    text = text.lower()
    # Supprimer les adresses email
    text = re.sub(r'\S+@\S+', '', text)
    # Supprimer les URL
    text = re.sub(r'http\S+|www\.\S+', '', text)
    # Supprimer les nombres (optionnel : parfois ils comptent)
    text = re.sub(r'\d+', '', text)
    # Supprimer caract\`eres sp\'eciaux, garder lettres et espaces
    text = re.sub(r'[^a-z\s]', '', text)
    # Supprimer les espaces multiples
    text = re.sub(r'\s+', ' ', text).strip()
    return text

# Appliquer au dataset
df["text_clean"] = df["text"].apply(clean_text)
print(df[["text", "text_clean"]].iloc[0])
```

Astuce prétraitement

Le nettoyage de texte dépend de l'ordre. Mettez toujours en minuscules *avant* de supprimer des patterns. Supprimez toujours URL et e-mails *avant* les caractères spéciaux (sinon on détruit les motifs à repérer).

7.4 Tokenisation

```
import nltk
nltk.download("punkt_tab", quiet=True)
from nltk.tokenize import word_tokenize, sent_tokenize

text = "Dr. Smith's analysis shows that 42% of patients improved. This is
↪  significant!"

# Tokenisation par phrase
sentences = sent_tokenize(text)
print(f"Sentences: {sentences}")

# Tokenisation par mot
tokens = word_tokenize(text)
```

```
print(f"Tokens: {tokens}")
print(f"Number of tokens: {len(tokens)}")
```

```
# Tokenisation par espaces vs NLTK
text_fr = "L'analyse du Dr. N'Guessan montre que c'est significatif."

# Split par espace : rate les contractions
simple_tokens = text_fr.split()
print(f"Simple: {simple_tokens}")

# NLTK : g\`ere contractions et abr\'eviations
nltk_tokens = word_tokenize(text_fr, language="french")
print(f"NLTK:   {nltk_tokens}")
```

7.5 Suppression des stopwords

```
nltk.download("stopwords", quiet=True)
from nltk.corpus import stopwords

# Stopwords anglais
stop_en = set(stopwords.words("english"))
print(f"English stopwords ({len(stop_en)}): "
      f"{sorted(list(stop_en))[:15]}...")

# Stopwords fran\c{c}ais
stop_fr = set(stopwords.words("french"))
print(f"French stopwords ({len(stop_fr)}): "
      f"{sorted(list(stop_fr))[:15]}...")

# Supprimer les stopwords des tokens
tokens = ["the", "patient", "showed", "significant", "improvement",
          "in", "blood", "pressure", "after", "the", "treatment"]
filtered = [t for t in tokens if t not in stop_en]
print(f"Before: {tokens}")
print(f"After:  {filtered}")
```

Attention

Supprimer les stopwords n'est pas toujours bénéfique. En analyse de sentiment, « not good » devient « good » après suppression de « not ». En topic modeling, c'est généralement utile. Évaluez toujours l'impact sur votre tâche spécifique.

7.6 Stemming et lemmatisation

```
from nltk.stem import PorterStemmer, SnowballStemmer
```

```
nltk.download("wordnet", quiet=True)
from nltk.stem import WordNetLemmatizer

stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

words = ["running", "runs", "ran", "studies", "studying",
         "better", "preprocessing", "processed"]

print(f"{'Word':<15} {'Stem':<15} {'Lemma':<15}")
print("-" * 45)
for w in words:
    print(f"{w:<15} {stemmer.stem(w):<15} {lemmatizer.lemmatize(w, 'v'):<15}")
```

```
# Stemming fran\c{c}ais avec Snowball
stemmer_fr = SnowballStemmer("french")
mots = ["traitements", "traiter", "traitées", "amélioration",
        "améliorer", "analyse", "analyses", "analyser"]

print("\nFrench stemming:")
for m in mots:
    print(f"  {m:<20} -> {stemmer_fr.stem(m)}")
```

Astuce prétraitement

Le stemming est rapide mais agressif : « university » et « universe » peuvent partager la même racine. La lemmatisation est plus lente mais linguistiquement correcte : elle renvoie de vrais mots du dictionnaire. Pour la plupart des pipelines NLP, la lemmatisation est préférée.

7.7 Pipeline complet de prétraitement de texte

```
def preprocess_text(text, language="english"):
    """Pipeline complet de pr\'etraitement de texte."""
    # 1. Nettoyer
    text = text.lower()
    text = re.sub(r'\S+@\S+', '', text)
    text = re.sub(r'http\S+|www\.\S+', '', text)
    text = re.sub(r'[^a-z\sà-ÿ]', '', text)  # garder accentu\'es
    text = re.sub(r'\s+', ' ', text).strip()

    # 2. Tokeniser
    tokens = word_tokenize(text, language=language)

    # 3. Supprimer les stopwords
    stop = set(stopwords.words(language))
    tokens = [t for t in tokens if t not in stop and len(t) > 2]
```

```
    # 4. Lemmatiser
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(t) for t in tokens]

    return tokens

# Appliquer au dataset complet
df["tokens"] = df["text"].apply(preprocess_text)
df["n_tokens"] = df["tokens"].apply(len)
print(df[["category", "n_tokens"]].groupby("category")["n_tokens"].mean()
        .sort_values(ascending=False).head(10))
```

7.8 Vectorisation TF-IDF

```
from sklearn.feature_extraction.text import TfidfVectorizer

# TF-IDF : Term Frequency - Inverse Document Frequency
tfidf = TfidfVectorizer(max_features=5000,
                        stop_words="english",
                        min_df=5, max_df=0.7,
                        ngram_range=(1, 2))

X_tfidf = tfidf.fit_transform(df["text_clean"])
print(f"TF-IDF matrix shape: {X_tfidf.shape}")
print(f"Vocabulary size: {len(tfidf.vocabulary_)}")

# Top termes par score TF-IDF pour le premier document
feature_names = tfidf.get_feature_names_out()
doc_tfidf = X_tfidf[0].toarray().flatten()
top_idx = doc_tfidf.argsort()[-10:][::-1]
print("\nTop TF-IDF terms for document 0:")
for idx in top_idx:
    print(f"  {feature_names[idx]}: {doc_tfidf[idx]:.4f}")
```

```
# Bag of Words (alternative plus simple)
from sklearn.feature_extraction.text import CountVectorizer

bow = CountVectorizer(max_features=3000, stop_words="english")
X_bow = bow.fit_transform(df["text_clean"])
print(f"BoW matrix shape: {X_bow.shape}")
```

7.9 Expressions régulières pour l'extraction

```
import re

# Extraire des donn\'ees structur\'ees d'un texte non structur\'e
```

```
texts = [
    "Patient BP: 120/80 mmHg, HR: 72 bpm",
    "Lab result: glucose 142 mg/dL, HbA1c 7.2%",
    "Contact: dr.smith@hospital.org, Tel: +1-555-0123",
]

# Extraire la tension art\'erielle
bp_pattern = r'(\d{2,3})/(\d{2,3})\s*mmHg'
for t in texts:
    match = re.search(bp_pattern, t)
    if match:
        print(f"Systolic: {match.group(1)}, Diastolic: {match.group(2)}")

# Extraire tous les nombres avec unit\'es
num_unit = r'(\d+\.?\d*)\s*(mg/dL|mmHg|bpm|%)'
for t in texts:
    matches = re.findall(num_unit, t)
    for value, unit in matches:
        print(f"  {value} {unit}")
```

7.10 Exercices

Exercice

1. Chargez 20 Newsgroups. Construisez un pipeline complet (nettoyer, tokeniser, retirer stopwords, lemmatiser). Calculez la taille du vocabulaire avant et après.

2. Créez une matrice TF-IDF sur 20 Newsgroups. Trouvez les 10 meilleurs termes pour « sci.space » et « rec.sport.baseball ». Sont-ils significatifs ?

3. Écrivez un pattern regex qui extrait toutes les adresses e-mail d'un texte. Testez-le sur 20 Newsgroups brut.

4. Comparez le stemming Snowball anglais et français sur un document bilingue. Montrez des cas où le stemming produit des racines incorrectes.

5. Construisez un pipeline de prétraitement pour le français. Utilisez les stopwords français et le stemmer Snowball français. Testez sur 5 phrases sur la science des données.

7.11 Résumé du chapitre

- Le prétraitement de texte convertit du texte non structuré en variables numériques pour le ML.

- Pipeline standard : nettoyer (minuscules, supprimer bruit) → tokeniser → retirer stopwords → stem/lemmatiser → vectoriser.

- Retirer les stopwords aide le topic modeling mais peut nuire à l'analyse de sentiment.

- La lemmatisation produit de vrais mots et est généralement préférée au stemming.
- TF-IDF pondère les termes par leur importance (fréquents dans un document, rares dans le corpus).
- Les regex extraient des données structurées (nombres, e-mails, motifs) d'un texte non structuré.
- Le prétraitement multilingue demande tokenizers, listes de stopwords et stemmers spécifiques à la langue.

Chapitre 8

Séries temporelles et données chronologiques

« Le temps est la variable la plus importante d'une série temporelle — et la plus souvent mal traitée. »

8.1 Parser les dates

La première étape avec des données temporelles est la conversion de chaînes en vrais objets datetime. Pandas fournit un parsing robuste, mais les formats réels sont étonnamment incohérents.

```
import pandas as pd
import numpy as np

# Jena Climate : mesures m\'et\'eo aux 10 minutes
# T\'el\'echarger : https://www.kaggle.com/datasets/mnassrib/jena-climate
df = pd.read_csv("jena_climate_2009_2016.csv")
print(df.head())
print(f"\nDate column dtype: {df['Date Time'].dtype}")
```

```
# Parser le datetime
df["datetime"] = pd.to_datetime(df["Date Time"], format="%d.%m.%Y %H:%M:%S")
df = df.set_index("datetime").drop(columns=["Date Time"])

print(f"Date range: {df.index.min()} to {df.index.max()}")
print(f"Frequency: ~{(df.index[1] - df.index[0])}")
print(f"Total records: {len(df):,}")
```

Attention

Ambiguïté de parsing : « 01/02/2024 » est-ce le 2 janvier (US) ou le 1er février (Europe) ? Spécifiez toujours le format explicitement avec `format`. Ne comptez jamais sur l'inférence automatique en production.

8.1.1 Gérer plusieurs formats de date

```
# Quand une colonne a des formats m\'elang\'es
messy_dates = pd.Series(["2024-01-15", "15/01/2024", "Jan 15, 2024",
                         "15-Jan-2024", "20240115"])

# pd.to_datetime avec mixed g\`ere de nombreux cas
parsed = pd.to_datetime(messy_dates, format="mixed", dayfirst=True)
print(parsed)

# Pour des donn\'ees vraiment d\'esordonn\'ees, parser sur mesure
def parse_flexible(date_str):
    """Essayer plusieurs formats de date."""
    formats = ["%Y-%m-%d", "%d/%m/%Y", "%b %d, %Y",
               "%d-%b-%Y", "%Y%m%d"]
    for fmt in formats:
        try:
            return pd.to_datetime(date_str, format=fmt)
        except (ValueError, TypeError):
            continue
    return pd.NaT
```

8.2 Rééchantillonnage

```
# Downsampling : donn\'ees 10 min vers horaires
hourly = df.resample("h").mean()
print(f"10-min records: {len(df):,}")
print(f"Hourly records: {len(hourly):,}")

# Downsampling vers journalier
daily = df.resample("D").agg({
    "T (degC)": ["mean", "min", "max"],
    "p (mbar)": "mean",
    "rh (%)": "mean",
})
daily.columns = ["temp_mean", "temp_min", "temp_max",
                 "pressure_mean", "humidity_mean"]
print(daily.head())
```

```
# Upsampling : journalier vers horaire (avec interpolation)
daily_sample = daily.head(30)
hourly_upsampled = daily_sample.resample("h").interpolate(method="linear")
print(f"Daily records: {len(daily_sample)}")
print(f"Hourly (upsampled): {len(hourly_upsampled)}")
```

Astuce prétraitement

En downsamplant, réfléchissez bien à la fonction d'agrégation. La température utilise la moyenne, la pluviométrie utilise la somme, et les min/max utilisent min/max. Une mauvaise agrégation produit des résultats absurdes.

8.3 Variables décalées (lag features)

```
# Variables d\'ecal\'ees : utiliser des valeurs pass\'ees comme pr\'edicteurs
daily["temp_lag1"] = daily["temp_mean"].shift(1)   # hier
daily["temp_lag7"] = daily["temp_mean"].shift(7)   # semaine derni\`ere
daily["temp_lag30"] = daily["temp_mean"].shift(30) # mois dernier

# Variables de variation
daily["temp_change_1d"] = daily["temp_mean"].diff(1)
daily["temp_change_7d"] = daily["temp_mean"].diff(7)

# Variation en pourcentage
daily["temp_pct_change"] = daily["temp_mean"].pct_change()

print(daily[["temp_mean", "temp_lag1", "temp_lag7",
             "temp_change_1d"]].head(10))
```

```
# Autocorr\'elation : v\'erifier l'utilit\'e des lags
import matplotlib.pyplot as plt
from pandas.plotting import autocorrelation_plot

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot d'autocorr\'elation
autocorrelation_plot(daily["temp_mean"].dropna(), ax=axes[0])
axes[0].set_title("Temperature autocorrelation")
axes[0].set_xlim(0, 400)

# Autocorr\'elation partielle (utile pour d\'eterminer l'ordre)
from statsmodels.graphics.tsaplots import plot_pacf
plot_pacf(daily["temp_mean"].dropna(), lags=50, ax=axes[1])
axes[1].set_title("Partial autocorrelation")

plt.tight_layout()
plt.savefig("autocorrelation.png", dpi=150)
plt.show()
```

Attention

Les variables décalées introduisent des NaN en début de série. Une variable lag-30 a 30 valeurs manquantes. Compensez en supprimant les premières lignes ou en utilisant forward-fill.

8.4 Statistiques mobiles

```
# Statistiques mobiles : lissent le bruit
daily["temp_rolling7"] = daily["temp_mean"].rolling(window=7).mean()
daily["temp_rolling30"] = daily["temp_mean"].rolling(window=30).mean()
daily["temp_rolling_std7"] = daily["temp_mean"].rolling(window=7).std()

# Moyenne mobile pond\'er\'ee exponentiellement (plus r\'ecent = plus de poids)
daily["temp_ewm7"] = daily["temp_mean"].ewm(span=7).mean()

print(daily[["temp_mean", "temp_rolling7", "temp_rolling30",
             "temp_ewm7"]].tail(10))
```

```
# Visualiser les statistiques mobiles
fig, ax = plt.subplots(figsize=(14, 6))

daily["temp_mean"].plot(ax=ax, alpha=0.3, label="Daily mean")
daily["temp_rolling7"].plot(ax=ax, label="7-day rolling mean")
daily["temp_rolling30"].plot(ax=ax, label="30-day rolling mean",
                             linewidth=2)

ax.set_ylabel("Temperature (degC)")
ax.set_title("Jena Climate: Temperature with rolling averages")
ax.legend()
plt.tight_layout()
plt.savefig("rolling_stats.png", dpi=150)
plt.show()
```

8.5 Décomposition tendance/saisonnalité

```
from statsmodels.tsa.seasonal import seasonal_decompose

# D\'ecomposer en tendance + saisonnier + r\'esidu
# Utiliser des donn\'ees mensuelles pour une d\'ecomposition plus propre
monthly = daily["temp_mean"].resample("ME").mean()

decomposition = seasonal_decompose(monthly, model="additive", period=12)

fig = decomposition.plot()
fig.set_size_inches(14, 10)
plt.tight_layout()
plt.savefig("decomposition.png", dpi=150)
plt.show()
```

```
# Extraire les composantes comme variables
monthly_df = pd.DataFrame({
```

```
    "observed": decomposition.observed,
    "trend": decomposition.trend,
    "seasonal": decomposition.seasonal,
    "residual": decomposition.resid,
})
print(monthly_df.head(15))
```

Astuce prétraitement

Le résidu de décomposition est ce qui reste après retrait de tendance et saisonnalité. Un grand résidu indique une observation inhabituelle — mécanisme de détection d'outliers pour séries temporelles.

8.6 Étude de cas Air Quality

```
# Air Quality UCI : mesures horaires de polluants
df_air = pd.read_csv("AirQualityUCI.csv", sep=";", decimal=",",
                     parse_dates={"datetime": ["Date", "Time"]},
                     dayfirst=True)

# Remplacer les sentinelles
df_air = df_air.replace(-200, np.nan)

# S\'electionner les colonnes cl\'es
cols = ["datetime", "CO(GT)", "NO2(GT)", "T", "RH"]
df_air = df_air[cols].set_index("datetime").dropna(how="all")

# R\'e\'echantillonner en journalier
air_daily = df_air.resample("D").mean()

# Cr\'eer des variables temporelles
air_daily["hour_of_day_peak"] = df_air["CO(GT)"].resample("D").idxmax().dt.hour
air_daily["co_lag1"] = air_daily["CO(GT)"].shift(1)
air_daily["co_rolling7"] = air_daily["CO(GT)"].rolling(7).mean()
air_daily["co_weekend"] = air_daily.index.dayofweek.isin([5, 6]).astype(int)

print(air_daily.head(10))
```

8.7 Récapitulatif d'extraction de variables datetime

```
def extract_datetime_features(df, date_col):
    """Extract a comprehensive set of datetime features."""
    dt = df[date_col] if date_col in df.columns else df.index
    features = pd.DataFrame(index=df.index)

    features["year"] = dt.year
    features["month"] = dt.month
```

```
    features["day"] = dt.day
    features["dayofweek"] = dt.dayofweek
    features["hour"] = dt.hour if hasattr(dt, "hour") else 0
    features["quarter"] = dt.quarter
    features["is_weekend"] = dt.dayofweek.isin([5, 6]).astype(int)
    features["day_of_year"] = dt.dayofyear
    features["week_of_year"] = dt.isocalendar().week.astype(int)

    # Encodage cyclique
    features["month_sin"] = np.sin(2 * np.pi * dt.month / 12)
    features["month_cos"] = np.cos(2 * np.pi * dt.month / 12)
    features["dow_sin"] = np.sin(2 * np.pi * dt.dayofweek / 7)
    features["dow_cos"] = np.cos(2 * np.pi * dt.dayofweek / 7)

    return features
```

8.8 Exercices

Exercice

1. Chargez Jena Climate. Parsez le datetime et rééchantillonnez en journalier. Tracez la température moyenne quotidienne pour 2015.
2. Créez des variables lag (1, 7, 14, 30 jours) pour la température. Calculez la corrélation de chaque lag avec la température actuelle. Quel lag a la plus forte corrélation ?
3. Appliquez une décomposition saisonnière sur les données CO d'Air Quality. Y a-t-il un motif saisonnier clair ? Que dit la tendance ?
4. Construisez une fonction complète d'extraction de variables pour Jena Climate qui crée : lag, rolling, encodages cycliques et indicateurs de tendance. Appliquez-la et reportez le nombre de variables résultantes.
5. Avec Air Quality, investiguez si les niveaux de CO diffèrent significativement entre semaine et week-end. Créez les visualisations appropriées.

8.9 Résumé du chapitre

- Parsez toujours les dates explicitement avec un format connu ; jamais d'inférence automatique en production.
- Le rééchantillonnage change la résolution temporelle : downsampling agrège, upsampling interpole.
- Les variables décalées capturent les dépendances temporelles : la température d'hier prédit celle d'aujourd'hui.
- Les statistiques mobiles (moyenne, écart-type) lissent le bruit et révèlent les tendances.

- La décomposition saisonnière sépare tendance, saisonnalité et résidus — chacun peut servir de variable.
- L'encodage cyclique (sinus/cosinus) garantit que décembre est proche de janvier dans l'espace des variables.

Chapitre 9

Pipelines et automatisation

« Si vous le faites deux fois, automatisez-le. » — Tous les ingénieurs logiciels

9.1 Pourquoi des pipelines ?

Le prétraitement manuel est sujet aux erreurs et aux fuites. Erreurs courantes :

- Ajuster un scaler sur tout le dataset (y compris le test) — fuite de données.
- Oublier d'appliquer les mêmes transformations à la prédiction.
- Appliquer les transformations dans un ordre différent entre train et inference.

Les `Pipeline` et `ColumnTransformer` de scikit-learn résolvent les trois en empaquetant tout le workflow dans un objet unique et reproductible.

Astuce données

Un pipeline garantit que chaque transformation appliquée à l'entraînement est appliquée identiquement à la prédiction, dans le même ordre, avec les mêmes paramètres ajustés.

9.2 Pipeline basique

```
import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# Adult Census
url = ("https://archive.ics.uci.edu/ml/machine-learning-databases/"
       "adult/adult.data")
cols = ["age", "workclass", "fnlwgt", "education", "education_num",
        "marital_status", "occupation", "relationship", "race",
```

```
        "sex", "capital_gain", "capital_loss", "hours_per_week",
        "native_country", "income"]
df = pd.read_csv(url, header=None, names=cols, na_values=" ?",
                 skipinitialspace=True)

# Pipeline num\'erique simple
num_pipe = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler()),
])

# Fit et transform
num_cols = ["age", "education_num", "capital_gain",
            "capital_loss", "hours_per_week"]
X_num = num_pipe.fit_transform(df[num_cols])
print(f"Shape: {X_num.shape}")
print(f"Means (should be ~0): {X_num.mean(axis=0).round(2)}")
```

9.3 ColumnTransformer

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

# D\'efinir les groupes de colonnes
num_features = ["age", "education_num", "capital_gain",
                "capital_loss", "hours_per_week"]
cat_features = ["workclass", "marital_status", "occupation",
                "relationship", "race", "sex"]

# Construire les transformateurs par type
num_transformer = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler()),
])

cat_transformer = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("encoder", OneHotEncoder(handle_unknown="ignore",
                              sparse_output=False)),
])

# Combiner avec ColumnTransformer
preprocessor = ColumnTransformer([
    ("num", num_transformer, num_features),
    ("cat", cat_transformer, cat_features),
])

# Fit et transform
X = preprocessor.fit_transform(df)
print(f"Input shape: {df[num_features + cat_features].shape}")
```

```
print(f"Output shape: {X.shape}")
```

Astuce prétraitement

Utilisez `ColumnTransformer` pour appliquer différentes transformations à différents types de colonnes en une seule étape. C'est l'approche standard en production.

9.4 Pipeline complet avec un modèle

```
from sklearn.metrics import accuracy_score, classification_report

# Pr\'eparer la cible
y = (df["income"] == ">50K").astype(int)

# Pipeline complet : pr\'etraitement + mod\`ele
full_pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier", LogisticRegression(max_iter=1000, random_state=42)),
])

# Split train/test
X_train, X_test, y_train, y_test = train_test_split(
    df[num_features + cat_features], y,
    test_size=0.2, random_state=42, stratify=y
)

# Ajuster tout le pipeline
full_pipeline.fit(X_train, y_train)

# Pr\'edire
y_pred = full_pipeline.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(classification_report(y_test, y_pred))
```

```
# Validation crois\'ee avec le pipeline (pas de fuite !)
from sklearn.model_selection import cross_val_score

scores = cross_val_score(full_pipeline,
                         df[num_features + cat_features], y,
                         cv=5, scoring="accuracy")
print(f"CV Accuracy: {scores.mean():.4f} +/- {scores.std():.4f}")
```

Attention

Si vous mettez à l'échelle vos données *avant* le split train/test, l'information du test fuite vers la moyenne et l'écart-type du scaler. En plaçant le scaler dans un pipeline, scikit-learn garantit qu'il ne s'ajuste que sur l'entraînement pendant la CV.

9.5 Transformateurs personnalisés

```python
from sklearn.base import BaseEstimator, TransformerMixin

class MissingIndicator(BaseEstimator, TransformerMixin):
    """Add binary columns indicating which values were missing."""

    def fit(self, X, y=None):
        self.columns_ = X.columns if hasattr(X, "columns") else None
        return self

    def transform(self, X):
        X = pd.DataFrame(X, columns=self.columns_)
        indicators = X.isnull().astype(int)
        indicators.columns = [f"{c}_missing" for c in indicators.columns]
        return pd.concat([X, indicators], axis=1)


class LogTransformer(BaseEstimator, TransformerMixin):
    """Apply log1p transformation to specified columns."""

    def __init__(self, columns=None):
        self.columns = columns

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X = X.copy()
        cols = self.columns or X.columns
        for col in cols:
            X[col] = np.log1p(X[col].clip(lower=0))
        return X
```

```python
# Utiliser des transformateurs personnalis\'es dans un pipeline
custom_pipe = Pipeline([
    ("missing_ind", MissingIndicator()),
    ("imputer", SimpleImputer(strategy="median")),
    ("log", LogTransformer(columns=["capital_gain", "capital_loss"])),
    ("scaler", StandardScaler()),
])

# Tester
sample = df[num_features].head(20)
result = custom_pipe.fit_transform(sample)
print(f"Input columns: {len(num_features)}")
print(f"Output columns: {result.shape[1]}")
```

9.6 Sauvegarder et charger un pipeline

```
import joblib

# Sauvegarder le pipeline ajust\'e
joblib.dump(full_pipeline, "income_pipeline.joblib")
print("Pipeline saved.")

# Le recharger plus tard (web app ou batch)
loaded_pipeline = joblib.load("income_pipeline.joblib")

# Pr\'edire sur de nouvelles donn\'ees
new_data = pd.DataFrame({
    "age": [35], "education_num": [13],
    "capital_gain": [0], "capital_loss": [0],
    "hours_per_week": [40], "workclass": ["Private"],
    "marital_status": ["Married-civ-spouse"],
    "occupation": ["Exec-managerial"],
    "relationship": ["Husband"], "race": ["White"],
    "sex": ["Male"],
})

prediction = loaded_pipeline.predict(new_data)
print(f"Prediction: {'> 50K' if prediction[0] else '<= 50K'}")
```

Astuce prétraitement

Un pipeline sauvegardé contient tous les paramètres ajustés (moyennes du scaler, catégories de l'encodeur, statistiques d'imputation). Ce seul fichier suffit pour prétraiter les nouvelles données identiquement aux données d'entraînement.

9.7 Pipeline complet Melbourne Housing

```
# Pipeline complet pour Melbourne Housing
melb = pd.read_csv("melb_data.csv")
melb = melb.dropna(subset=["Price"])

num_features_melb = ["Rooms", "Distance", "Landsize", "BuildingArea",
                     "YearBuilt", "Car", "Bathroom"]
cat_features_melb = ["Type", "Method", "Regionname"]

melb_preprocessor = ColumnTransformer([
    ("num", Pipeline([
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
    ]), num_features_melb),
    ("cat", Pipeline([
        ("imputer", SimpleImputer(strategy="most_frequent")),
```

```
        ("encoder", OneHotEncoder(handle_unknown="ignore",
                                  sparse_output=False)),
    ]), cat_features_melb),
])

from sklearn.ensemble import RandomForestRegressor

melb_pipeline = Pipeline([
    ("preprocessor", melb_preprocessor),
    ("model", RandomForestRegressor(n_estimators=100, random_state=42)),
])

X_melb = melb[num_features_melb + cat_features_melb]
y_melb = melb["Price"]

X_tr, X_te, y_tr, y_te = train_test_split(X_melb, y_melb,
                                          test_size=0.2, random_state=42)
melb_pipeline.fit(X_tr, y_tr)
score = melb_pipeline.score(X_te, y_te)
print(f"Melbourne Housing R^2: {score:.4f}")
```

9.8 Exercices

Exercice

1. Construisez un pipeline `ColumnTransformer` pour Titanic avec : imputation médiane + mise à l'échelle pour le numérique, et imputation mode + one-hot pour le catégoriel. Évaluez par CV 5-fold.

2. Créez un transformateur personnalisé `OutlierClipper` qui clippe aux 1er et 99e percentiles. Intégrez-le dans un pipeline.

3. Construisez un pipeline complet pour Melbourne Housing qui inclut du feature engineering (prix par pièce, âge du bâtiment). Sauvegardez-le avec `joblib` et rechargez-le pour prédire.

4. Comparez la performance d'un pipeline avec et sans prétraitement sur Adult Census. Utilisez un arbre de décision.

5. Créez un pipeline qui gère simultanément numérique, catégoriel et texte avec `ColumnTransformer`. Utilisez TF-IDF pour le texte.

9.9 Résumé du chapitre

- Le `Pipeline` scikit-learn chaîne transformations et modèle en un seul objet, évitant la fuite de données.

- `ColumnTransformer` applique des transformations différentes à des types de colonnes différents.

- Les transformateurs personnalisés (héritant de `BaseEstimator` et `TransformerMixin`) s'intègrent naturellement dans les pipelines.
- Sauvegarder les pipelines avec `joblib` capture tous les paramètres ajustés pour un déploiement reproductible.
- Un pipeline garantit que le prétraitement à l'entraînement et à la prédiction sont identiques.

Chapitre 10

Projet de fin de cours

« La meilleure façon d'apprendre le prétraitement, c'est de prétraiter des données. »

10.1 Vue d'ensemble

Dans ce dernier chapitre, vous appliquerez tout ce des Chapitres 1–9 à un projet complet de prétraitement de bout en bout. Vous choisirez l'un des cinq projets proposés, chacun utilisant un vrai dataset désordonné qui demande un nettoyage et une transformation substantiels avant l'analyse.

Astuce données

Chaque projet est dimensionné pour environ 3 heures. Vous devez livrer : (1) un notebook Jupyter avec code documenté, (2) un dataset propre, prêt pour l'analyse, sauvegardé en Parquet, et (3) un court rapport écrit (1–2 pages) résumant vos choix de prétraitement et leur justification.

10.2 Critères d'évaluation

Votre projet sera évalué sur cinq critères :

Critère	Poids	Description
Rapport de qualité	15%	Évaluation initiale approfondie de la qualité
Stratégie de manquantes	20%	Choix justifié d'imputation
Traitement d'outliers	15%	Détection, investigation, traitement
Feature engineering	25%	Créativité et pertinence métier des variables
Pipeline et reproductibilité	25%	Pipeline complet, sans fuite, sauvegardé

10.3 Projet 1 : Prédiction de prix Melbourne Housing

Dataset : Melbourne Housing Snapshot (Kaggle)
Objectif : Préparer le dataset pour un modèle de régression de prix immobilier.

10.3.1 Exigences

```python
import pandas as pd

# Charger le dataset brut
df = pd.read_csv("melb_data.csv")
print(f"Shape: {df.shape}")
print(f"Missing values:\n{df.isnull().sum()}")
```

Tâches :

1. Produire un rapport de qualité complet (Chapitre 1).
2. Traiter les manquantes de `BuildingArea`, `YearBuilt`, `Car`, `CouncilArea` avec au moins deux stratégies. Justifier vos choix (Chapitre 3).
3. Détecter et traiter les outliers de `Price`, `Landsize`, `BuildingArea` (Chapitre 4).
4. Encoder les variables catégorielles (`Type`, `Method`, `Regionname`, `Suburb`). Pour `Suburb` à forte cardinalité, utiliser target ou frequency encoding (Chapitre 5).
5. Construire au moins 5 nouvelles variables : prix par pièce, âge du bâtiment, catégories de distance, moyennes par quartier, indicateur saisonnier (Chapitre 6).
6. Construire un `Pipeline` complet avec `ColumnTransformer`. Sauvegarder avec `joblib` (Chapitre 9).
7. Entraîner un modèle baseline et reporter le R^2 sur un test réservé.

10.4 Projet 2 : Classification de survie Titanic

Dataset : Titanic (Kaggle)
Objectif : Préparer le dataset pour un modèle de classification de survie.

10.4.1 Exigences

```python
url = ("https://raw.githubusercontent.com/datasciencedojo/"
       "datasets/master/titanic.csv")
df = pd.read_csv(url)
print(f"Shape: {df.shape}")
print(f"Survival rate: {df['Survived'].mean():.2%}")
```

Tâches :

1. Analyser les motifs de manquantes. Investiguer si `Age` est MCAR ou MAR en le comparant à `Pclass` et `Sex` (Chapitre 3).

2. Imputer `Age` par KNN. Comparer avec imputation médiane en évaluant la précision du modèle en aval.

3. Extraire le titre depuis `Name` (Mr, Mrs, Miss, Master, etc.) et l'encoder (Chapitres 5 et 7).

4. Créer des variables : `FamilySize` = `SibSp` + `Parch` + 1, `IsAlone`, `FarePerPerson`, lettre de pont de cabine (Chapitre 6).

5. Construire un pipeline complet avec évaluation par CV.

10.5 Projet 3 : Estimation de valeur de joueurs FIFA 21

Dataset : FIFA 21 Raw Data (Kaggle)
Objectif : Nettoyer le célèbre dataset FIFA désordonné et le préparer pour la prédiction de valeur de joueur.

10.5.1 Exigences

```
df = pd.read_csv("fifa21_raw.csv")
print(f"Shape: {df.shape}")
# Note : beaucoup de colonnes ont des nombres encod\'es comme cha\^ines
print(df[["Value", "Wage", "Weight", "Height"]].head())
```

Tâches :

1. Parser les valeurs monétaires encodées comme chaînes (`Value`, `Wage`) : convertir « €110M » en 110000000, « €220K » en 220000.

2. Parser la taille (« 5'11 ») et le poids (« 190lbs ») en unités métriques.

3. Gérer `Hits`, `W/F`, `SM`, `IR` (notes en étoiles stockées comme chaînes).

4. Détecter et traiter les outliers de valeurs (beaucoup d'agents libres ont une valeur 0).

5. Construire des variables par position, par âge (potentiel), et des ratios d'attributs physiques.

6. Construire un pipeline propre du CSV brut vers des variables prêtes pour la modélisation.

10.6 Projet 4 : Prévision de séries temporelles Jena Climate

Dataset : Jena Climate 2009–2016 (Kaggle)
Objectif : Préparer le dataset pour la prévision de température.

10.6.1 Exigences

```
df = pd.read_csv("jena_climate_2009_2016.csv")
df["datetime"] = pd.to_datetime(df["Date Time"],
                                format="%d.%m.%Y %H:%M:%S")
df = df.set_index("datetime")
print(f"Shape: {df.shape}")
print(f"Date range: {df.index.min()} to {df.index.max()}")
```

Tâches :

1. Rééchantillonner de 10 min vers horaire et journalier (Chapitre 8).
2. Détecter et gérer les anomalies de capteur (pics soudains, lectures constantes).
3. Créer des variables lag : 1h, 6h, 12h, 24h, 7j pour la température.
4. Créer des statistiques mobiles : moyenne 24h, moyenne 7j, écart-type 24h.
5. Décomposer en tendance, saisonnalité et résidu. Utiliser chacun comme variable.
6. Ajouter un encodage cyclique pour l'heure, le jour de la semaine et le mois.
7. Construire un pipeline et entraîner un baseline pour prédire la température à 24h.

10.7 Projet 5 : Classification de texte 20 Newsgroups

Dataset : 20 Newsgroups (scikit-learn)
Objectif : Prétraiter des données texte pour la classification thématique.

10.7.1 Exigences

```
from sklearn.datasets import fetch_20newsgroups

# Utiliser un sous-ensemble de 5 cat\'egories
categories = ["sci.space", "rec.sport.baseball", "comp.graphics",
              "talk.politics.mideast", "soc.religion.christian"]
newsgroups = fetch_20newsgroups(subset="all", categories=categories,
                                remove=("headers", "footers", "quotes"))
print(f"Documents: {len(newsgroups.data)}")
print(f"Categories: {newsgroups.target_names}")
```

Tâches :

1. Construire un pipeline de nettoyage texte : minuscules, retirer emails/URL/nombres, retirer caractères spéciaux (Chapitre 7).
2. Tokeniser, retirer stopwords, lemmatiser.
3. Créer une matrice TF-IDF avec paramètres appropriés (`min_df`, `max_df`, `ngram_range`).
4. Ajouter des variables au niveau document : longueur, longueur moyenne des mots, richesse du vocabulaire (mots uniques / mots totaux).
5. Construire un pipeline combinant prétraitement texte et classifieur.
6. Évaluer par CV et matrice de confusion.

10.8 Checklist des livrables

```
# Mod\`ele de v\'erification finale
deliverables = {
    "data_quality_report": False,     # Section avec m\'etriques de qualit\'e
    "missing_data_handled": False,     # Imputation justifi\'ee
    "outliers_handled": False,         # D\'etection + traitement
    "features_engineered": False,      # >= 5 nouvelles variables
    "pipeline_built": False,           # sklearn Pipeline
    "pipeline_saved": False,           # fichier joblib
    "baseline_model": False,           # Entra\^iner + \'evaluer
    "report_written": False,           # R\'esum\'e 1-2 pages
}

# Cocher chaque item au fur et \`a mesure
for item, done in deliverables.items():
    status = "DONE" if done else "TODO"
    print(f"  [{status}] {item}")
```

10.9 Exercices

Exercice

1. Choisissez l'un des cinq projets. Complétez toutes les tâches requises et soumettez votre notebook, pipeline sauvegardé et rapport.
2. (Bonus) Après le projet principal, appliquez la même approche à un second projet. Comparez les défis et discutez les différences de stratégies de prétraitement entre domaines.
3. (Bonus) Déployez votre pipeline sauvegardé comme une fonction de prédiction simple : écrivez un script qui charge le pipeline depuis `joblib` et accepte de nouvelles données depuis la ligne de commande ou un CSV.

10.10 Résumé du chapitre

- Un projet capstone intègre toutes les compétences de prétraitement : chargement, nettoyage, imputation, gestion d'outliers, encodage, mise à l'échelle, feature engineering, automatisation par pipelines.
- Les vrais datasets demandent des décisions métier à chaque étape — il n'y a pas une seule recette « correcte ».
- La reproductibilité (via pipelines et artefacts sauvegardés) est aussi importante que la précision.
- Documenter ses choix et leur justification est une exigence professionnelle.
- Les cinq options couvrent régression tabulaire, classification tabulaire, nettoyage de données désordonnées, séries temporelles et texte — les principaux domaines de prétraitement.

Annexe A : Guide d'installation Python

Option 1 : Google Colab (recommandé pour débutants)

Allez sur https://colab.research.google.com. Connectez-vous avec un compte Google. Aucune installation nécessaire. Toutes les bibliothèques utilisées dans ce cours sont pré-installées.

Option 2 : Anaconda (installation locale)

Téléchargez Anaconda sur https://www.anaconda.com/download. Installez avec les paramètres par défaut. Ouvrez Jupyter Notebook depuis Anaconda Navigator.

Bibliothèques requises

```
pip install pandas numpy matplotlib seaborn scikit-learn missingno nltk spacy
↪  joblib openpyxl sqlalchemy
```

Annexe B : Sources de jeux de données

Source	Description	URL
Melbourne Housing	Prix immobiliers avec valeurs manquantes et types mixtes	kaggle.com
Titanic	Données de survie des passagers avec âges et cabines manquants	kaggle.com/c/titanic
FIFA 21 Raw	Attributs de joueurs avec nombres encodés comme chaînes	kaggle.com
Adult Census (UCI)	Prédiction de revenu avec types catégoriels et numériques mêlés	archive.ics.uci.edu
Jena Climate	Mesures météo aux 10 minutes sur plusieurs années	kaggle.com
Air Quality UCI	Mesures horaires de polluants avec dérive de capteurs	archive.ics.uci.edu
20 Newsgroups	Jeu de classification de texte (20 catégories)	via `sklearn`
Amazon Reviews	Avis produits pour analyse de sentiment et traitement de texte	kaggle.com
Gapminder	Indicateurs socio-économiques par pays dans le temps	gapminder.org