


Valeurs Manquantes (Missing Values)

Trois approches

1/ Une option simple : supprimer les colonnes avec des valeurs manquantes.

Bed	Bath
1.0	1.0
2.0	1.0
3.0	2.0
NaN	2.0




Bath
1.0
1.0
2.0
2.0

À moins que la plupart des valeurs dans les colonnes supprimées ne soient manquantes, le modèle perd l'accès à beaucoup d'informations avec cette approche.

2/ Une meilleure option : l'imputation

L'imputation remplit les valeurs manquantes avec un certain nombre. Par exemple, nous pouvons remplir la valeur moyenne de chaque colonne.

Bed	Bath
1.0	1.0
2.0	1.0
3.0	2.0
NaN	2.0




Bed	Bath
1.0	1.0
2.0	1.0
3.0	2.0
2.0	2.0

3/ Une extension à l'imputation

L'imputation est l'approche standard et elle fonctionne généralement bien.

Bed	Bath	
1.0	1.0	
2.0	1.0	
3.0	2.0	
NaN	2.0	



Bed	Bath	Bed_was_missing
1.0	1.0	FALSE
2.0	1.0	FALSE
3.0	2.0	FALSE
2.0	2.0	TRUE

Dans cette approche, nous imputons les valeurs manquantes et ajoutons une nouvelle colonne pour indiquer l'emplacement des valeurs imputées dans les colonnes d'origine.

Exemple

In [1]:

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Load the data
data = pd.read_csv('./immodata.csv')

# Select target
y = data.Price

# To keep things simple, we'll use only numerical predictors
melb_predictors = data.drop(['Price'], axis=1)
X = melb_predictors.select_dtypes(exclude=['object'])

# Divide data into training and validation subsets
X_train, X_valid, y_train, y_valid = train_test_split(X, y, train_size=0.8, test_size=0.2,
                                                    random_state=0)
```

Définir une fonction pour mesurer la qualité de chaque approche

Nous définissons une fonction `score_dataset()` pour comparer différentes approches de gestion des valeurs manquantes. Cette fonction rapporte l'erreur absolue moyenne (MAE) d'un modèle de forêt aléatoire.

In [2]:

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error

# Function for comparing different approaches
def score_dataset(X_train, X_valid, y_train, y_valid):
    model = RandomForestRegressor(n_estimators=10, random_state=0)
    model.fit(X_train, y_train)
    preds = model.predict(X_valid)
    return mean_absolute_error(y_valid, preds)
```

Score de l'approche 1 (Supprimer les colonnes avec des valeurs manquantes)

Étant donné que nous travaillons à la fois avec les ensembles d'entraînement et de validation, nous veillons à supprimer les mêmes colonnes

dans les deux DataFrames.

```
In [3]: # Get names of columns with missing values
cols_with_missing = [col for col in X_train.columns
                      if X_train[col].isnull().any()]

# Drop columns in training and validation data
reduced_X_train = X_train.drop(cols_with_missing, axis=1)
reduced_X_valid = X_valid.drop(cols_with_missing, axis=1)

print("MAE from Approach 1 (Drop columns with missing values):")
print(score_dataset(reduced_X_train, reduced_X_valid, y_train, y_valid))

MAE from Approach 1 (Drop columns with missing values):
183550.22137772635
```

Score de l'approche 2 (Imputation)

Nous utilisons SimpleImputer pour remplacer les valeurs manquantes par la moyenne de chaque colonne. Bien que cette méthode soit simple et généralement efficace, les approches plus complexes, comme l'imputation par régression, n'offrent souvent pas d'avantages supplémentaires avec des modèles d'apprentissage automatique avancés.

```
In [4]: from sklearn.impute import SimpleImputer

# Imputation
my_imputer = SimpleImputer()
imputed_X_train = pd.DataFrame(my_imputer.fit_transform(X_train))
imputed_X_valid = pd.DataFrame(my_imputer.transform(X_valid))

# Imputation removed column names; put them back
imputed_X_train.columns = X_train.columns
imputed_X_valid.columns = X_valid.columns

print("MAE from Approach 2 (Imputation):")
print(score_dataset(imputed_X_train, imputed_X_valid, y_train, y_valid))

MAE from Approach 2 (Imputation):
178166.46269899711
```

Nous constatons que l'approche 2 a une MAE inférieure à celle de l'approche 1, donc l'approche 2 a mieux performé sur cet ensemble de données que l'approche car sa MAE est plus faible.

Score de l'approche 3 (Une extension à l'imputation)

Ensuite, nous imputons les valeurs manquantes tout en gardant une trace des valeurs qui ont été imputées.

```
In [5]: # Make copy to avoid changing original data (when imputing)
X_train_plus = X_train.copy()
X_valid_plus = X_valid.copy()

# Make new columns indicating what will be imputed
for col in cols_with_missing:
    X_train_plus[col + '_was_missing'] = X_train_plus[col].isnull()
    X_valid_plus[col + '_was_missing'] = X_valid_plus[col].isnull()

# Imputation
my_imputer = SimpleImputer()
imputed_X_train_plus = pd.DataFrame(my_imputer.fit_transform(X_train_plus))
imputed_X_valid_plus = pd.DataFrame(my_imputer.transform(X_valid_plus))

# Imputation removed column names; put them back
imputed_X_train_plus.columns = X_train_plus.columns
imputed_X_valid_plus.columns = X_valid_plus.columns

print("MAE from Approach 3 (An Extension to Imputation):")
print(score_dataset(imputed_X_train_plus, imputed_X_valid_plus, y_train, y_valid))
```

```
MAE from Approach 3 (An Extension to Imputation):
178927.503183954
```

Variables Catégorielles (Categorical Variables)

Une variable catégorique se limite à un nombre restreint de valeurs, comme des réponses à une enquête. Pour traiter ces variables dans l'apprentissage automatique, trois approches sont proposées :

1/ Supprimer les Variables Catégorielles

2/ Encodage Ordinal

Assigner des entiers à des valeurs uniques selon un ordre, utile pour les variables ordinales qui ont un classement évident (ex. : "Jamais" < "Rarement").

Breakfast	Breakfast
Every day	3
Never	0
Rarely	1
Most days	2
Never	0

3/ Encodage One-Hot

Créer des colonnes indiquant la présence de chaque valeur sans supposer d'ordre. Cela convient mieux aux variables nominales, mais est moins efficace pour celles avec plus de 15 valeurs différentes.

Color	Red	Yellow	Green
Red	1	0	0
Red	1	0	0
Yellow	0	1	0
Green	0	0	1
Yellow	0	1	0

yellow

U

I

U

In [6]:

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Read the data
data = pd.read_csv('./immodata.csv')

# Separate target from predictors
y = data.Price
X = data.drop(['Price'], axis=1)

# Divide data into training and validation subsets
X_train_full, X_valid_full, y_train, y_valid = train_test_split(X, y, train_size=0.8, test_size=0.2,
                                                                random_state=0)

# Drop columns with missing values (simplest approach)
cols_with_missing = [col for col in X_train_full.columns if X_train_full[col].isnull().any()]
X_train_full.drop(cols_with_missing, axis=1, inplace=True)
X_valid_full.drop(cols_with_missing, axis=1, inplace=True)

# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality (convenient but arbitrary)
low_cardinality_cols = [cname for cname in X_train_full.columns if X_train_full[cname].nunique() < 10 and
                        X_train_full[cname].dtype == "object"]

# Select numerical columns
numerical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].dtype in ['int64', 'float64']]

# Keep selected columns only
my_cols = low_cardinality_cols + numerical_cols
X_train = X_train_full[my_cols].copy()
X_valid = X_valid_full[my_cols].copy()
```

```
In [7]: X_train.head()
```

Out[7]:

	Type	Method	Regionname	Rooms	Distance	Postcode	Bedroom2	Bathroom	Landsize	Latitude	Longitude	Propertycount
12167	u	S	Southern Metropolitan	1	5.0	3182.0	1.0	1.0	0.0	-37.85984	144.9867	13240.0
6524	h	SA	Western Metropolitan	2	8.0	3016.0	2.0	2.0	193.0	-37.85800	144.9005	6380.0
8413	h	S	Western Metropolitan	3	12.6	3020.0	3.0	1.0	555.0	-37.79880	144.8220	3755.0
2919	u	SP	Northern Metropolitan	3	13.0	3046.0	3.0	1.0	265.0	-37.70830	144.9158	8870.0
6043	h	S	Western Metropolitan	3	13.3	3020.0	3.0	1.0	673.0	-37.76230	144.8272	4217.0

Nous obtenons une liste de toutes les variables catégoriques dans les données d'entraînement. Nous le faisons en vérifiant le type de données (ou dtype) de chaque colonne. Le type de données « object » indique qu'une colonne contient du texte (il pourrait théoriquement s'agir d'autres types, mais cela n'est pas important pour nos objectifs). Pour cet ensemble de données, les colonnes contenant du texte indiquent des variables catégoriques.

```
In [8]: # Get list of categorical variables
s = (X_train.dtypes == 'object')
object_cols = list(s[s].index)

print("Categorical variables:")
print(object_cols)
```

```
Categorical variables:
['Type', 'Method', 'Regionname']
```

Nous définissons la fonction `score_dataset()` pour comparer les trois approches de traitement des variables catégoriques. Elle mesure l'erreur absolue moyenne (MAE) d'un modèle de forêt aléatoire, avec l'objectif d'obtenir une MAE aussi faible que possible.

In [9]:

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error

# Function for comparing different approaches
def score_dataset(X_train, X_valid, y_train, y_valid):
    model = RandomForestRegressor(n_estimators=100, random_state=0)
    model.fit(X_train, y_train)
    preds = model.predict(X_valid)
    return mean_absolute_error(y_valid, preds)
```

Score de l'Approche 1 (Supprimer les Variables Catégoriques)

Nous supprimons les colonnes de type « object » avec la méthode `select_dtypes()`.

In [10]:

```
drop_X_train = X_train.select_dtypes(exclude=['object'])
drop_X_valid = X_valid.select_dtypes(exclude=['object'])

print("MAE from Approach 1 (Drop categorical variables):")
print(score_dataset(drop_X_train, drop_X_valid, y_train, y_valid))
```

```
MAE from Approach 1 (Drop categorical variables):
175703.48185157913
```

Score de l'Approche 2 (Encodage Ordinal)

Scikit-learn dispose d'une classe `OrdinalEncoder` qui peut être utilisée pour obtenir des encodages ordinaux. Nous parcourons les variables catégoriques et appliquons l'encodeur ordinal séparément à chaque colonne.

```
In [11]: from sklearn.preprocessing import OrdinalEncoder

# Make copy to avoid changing original data
label_X_train = X_train.copy()
label_X_valid = X_valid.copy()

# Apply ordinal encoder to each column with categorical data
ordinal_encoder = OrdinalEncoder()
label_X_train[object_cols] = ordinal_encoder.fit_transform(X_train[object_cols])
label_X_valid[object_cols] = ordinal_encoder.transform(X_valid[object_cols])

print("MAE from Approach 2 (Ordinal Encoding):")
print(score_dataset(label_X_train, label_X_valid, y_train, y_valid))
```

```
MAE from Approach 2 (Ordinal Encoding):
165936.40548390493
```

Ci-dessus Nous assignons aléatoirement un entier différent à chaque valeur unique pour chaque colonne, une approche simple qui pourrait être améliorée avec des étiquettes mieux informées pour les variables ordinales.

Score de l'Approche 3 (Encodage One-Hot)

Nous utilisons la classe OneHotEncoder de scikit-learn pour créer des encodages one-hot, avec des paramètres tels que `handle_unknown='ignore'` pour gérer les classes inconnues et `sparse=False` pour obtenir un tableau numpy. Pour encoder les données d'entraînement, nous fournissons uniquement les colonnes catégoriques à encoder, en utilisant `X_train[object_cols]`.

```
In [12]: from sklearn.preprocessing import OneHotEncoder

# Apply one-hot encoder to each column with categorical data
OH_encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
OH_cols_train = pd.DataFrame(OH_encoder.fit_transform(X_train[object_cols]))
OH_cols_valid = pd.DataFrame(OH_encoder.transform(X_valid[object_cols]))

# One-hot encoding removed index; put it back
OH_cols_train.index = X_train.index
OH_cols_valid.index = X_valid.index

# Remove categorical columns (will replace with one-hot encoding)
num_X_train = X_train.drop(object_cols, axis=1)
num_X_valid = X_valid.drop(object_cols, axis=1)

# Add one-hot encoded columns to numerical features
OH_X_train = pd.concat([num_X_train, OH_cols_train], axis=1)
OH_X_valid = pd.concat([num_X_valid, OH_cols_valid], axis=1)

# Ensure all columns have string type
OH_X_train.columns = OH_X_train.columns.astype(str)
OH_X_valid.columns = OH_X_valid.columns.astype(str)

print("MAE from Approach 3 (One-Hot Encoding):")
print(score_dataset(OH_X_train, OH_X_valid, y_train, y_valid))
```

MAE from Approach 3 (One-Hot Encoding):

```
/home/pnl/anaconda3/lib/python3.10/site-packages/sklearn/preprocessing/_encoders.py:828: FutureWarning: `
sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `sparse_output` is ignor
ed unless you leave `sparse` to its default value.
  warnings.warn(
```

166089.4893009678

Supprimer les colonnes catégoriques (Approche 1) a donné les pires résultats avec le score MAE le plus élevé. Les scores MAE des deux autres approches sont proches, sans avantage significatif l'une par rapport à l'autre. En général, l'encodage one-hot (Approche 3) performe le mieux, tandis que la suppression des colonnes donne généralement les pires résultats, mais cela varie selon le cas. Savoir utiliser les données catégoriques est essentiel pour être un data scientist efficace.

Pipelines

Les pipelines sont un moyen efficace d'organiser le prétraitement et la modélisation des données en les regroupant en une seule étape. Bien que certains data scientists n'utilisent pas de pipelines, ceux-ci offrent des avantages tels que :

- Code plus propre : Simplifie le suivi des données à chaque étape.
- Moins de bogues : Réduit les erreurs potentielles dans les étapes de prétraitement.
- Facilite la mise en production : Aide à déployer des modèles à grande échelle.
- Options de validation améliorées : Permet des méthodes comme la validation croisée

In [13]:

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Read the data
data = pd.read_csv('./immodata.csv')

# Separate target from predictors
y = data.Price
X = data.drop(['Price'], axis=1)

# Divide data into training and validation subsets
X_train_full, X_valid_full, y_train, y_valid = train_test_split(X, y, train_size=0.8, test_size=0.2,
                                                                random_state=0)

# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality (convenient but arbitrary)
categorical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].nunique() < 10 and
                    X_train_full[cname].dtype == "object"]

# Select numerical columns
numerical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].dtype in ['int64', 'float64']]

# Keep selected columns only
my_cols = categorical_cols + numerical_cols
X_train = X_train_full[my_cols].copy()
X_valid = X_valid_full[my_cols].copy()
```

```
In [14]: X_train.head()
```

```
Out[14]:
```

	Type	Method	Regionname	Rooms	Distance	Postcode	Bedroom2	Bathroom	Car	Landsize	BuildingArea	YearBuilt	Latitude	Longitude
12167	u	S	Southern Metropolitan	1	5.0	3182.0	1.0	1.0	1.0	0.0	NaN	1940.0	-37.85984	144.98
6524	h	SA	Western Metropolitan	2	8.0	3016.0	2.0	2.0	1.0	193.0	NaN	NaN	-37.85800	144.90
8413	h	S	Western Metropolitan	3	12.6	3020.0	3.0	1.0	1.0	555.0	NaN	NaN	-37.79880	144.85
2919	u	SP	Northern Metropolitan	3	13.0	3046.0	3.0	1.0	1.0	265.0	NaN	1995.0	-37.70830	144.91
6043	h	S	Western Metropolitan	3	13.3	3020.0	3.0	1.0	2.0	673.0	673.0	1970.0	-37.76230	144.85

Nous construisons le pipeline complet en trois étapes.

Étape 1 : Définir les Étapes de Prétraitement

Nous utilisons la classe `ColumnTransformer` pour regrouper différentes étapes de prétraitement, notamment l'imputation des valeurs manquantes dans les données numériques et l'application d'un encodage one-hot aux données catégoriques.

```
In [15]: from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# Preprocessing for numerical data
numerical_transformer = SimpleImputer(strategy='constant')

# Preprocessing for categorical data
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Bundle preprocessing for numerical and categorical data
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ]
)
```

Étape 2 : Définir le Modèle

Nous définissons un modèle de forêt aléatoire avec la classe RandomForestRegressor.

```
In [16]: from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators=100, random_state=0)
```

Étape 3 : Créer et Évaluer le Pipeline

Nous utilisons la classe Pipeline pour regrouper les étapes de prétraitement et de modélisation. Les avantages incluent :

- Simplicité : Le prétraitement et l'ajustement du modèle se font en une seule ligne de code, contrairement à une approche manuelle qui peut devenir confuse, surtout avec des variables numériques et catégoriques.
- Automatisation : Le pipeline prétraite automatiquement les données de validation lors de l'utilisation de la commande predict(), éliminant le

besoin de prétraitement séparé.

```
In [17]: from sklearn.metrics import mean_absolute_error

# Bundle preprocessing and modeling code in a pipeline
my_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('model', model)
                              ])

# Preprocessing of training data, fit model
my_pipeline.fit(X_train, y_train)

# Preprocessing of validation data, get predictions
preds = my_pipeline.predict(X_valid)

# Evaluate the model
score = mean_absolute_error(y_valid, preds)
print('MAE:', score)
```

MAE: 160679.18917034855

Validation Croisée (cross validation)

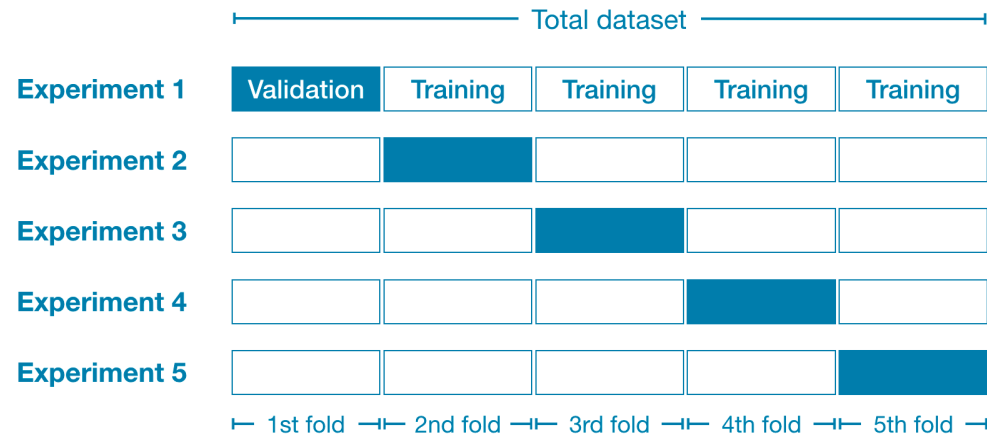
Introduction

L'apprentissage automatique est un processus itératif impliquant des choix sur les variables prédictives, les types de modèles et leurs arguments. Jusqu'à présent, ces choix ont été basés sur les données, en mesurant la qualité du modèle à l'aide d'un ensemble de validation.

Cependant, cette méthode présente des inconvénients, notamment le risque de chance dans l'évaluation des performances. Par exemple, si vous avez un ensemble de 10000 lignes et que vous en gardez 2000 pour la validation, les scores peuvent varier selon les sous-ensembles choisis. Dans des cas extrêmes, un seul point de données peut fausser les comparaisons entre modèles. En général, un ensemble de validation plus grand réduit le bruit dans les mesures de qualité, mais cela nécessite de retirer des données d'entraînement, ce qui peut dégrader les performances des modèles.

Définition

La validation croisée consiste à exécuter le processus de modélisation sur différents sous-ensembles de données pour obtenir plusieurs mesures de la qualité du modèle. Par exemple, en divisant les données en 5 "pliages" de 20 %, nous pouvons utiliser chaque pliage une fois comme ensemble de test, permettant d'utiliser 100 % des données comme ensemble de validation à un moment donné.



Quand utiliser la validation croisée ?

Elle fournit une mesure plus précise de la qualité du modèle, surtout pour les nombreuses décisions de modélisation. Toutefois, elle nécessite plus de temps, car elle entraîne plusieurs modèles. Pour les petits ensembles de données, utilisez la validation croisée, tandis que pour les grands ensembles, un seul ensemble de validation est suffisant. Si le modèle s'exécute rapidement, envisagez la validation croisée, mais vérifiez que les résultats sont cohérents entre les expériences.

Exemple

Nous allons travailler avec les mêmes données que dans le tutoriel précédent.

In [18]:

```
import pandas as pd

# Read the data
data = pd.read_csv('./immodata.csv')

# Select subset of predictors
cols_to_use = ['Rooms', 'Distance', 'Landsize', 'BuildingArea', 'YearBuilt']
X = data[cols_to_use]

# Select target
y = data.Price
```

Nous créons un pipeline qui remplit les valeurs manquantes avec un imputeur et utilise un modèle de forêt aléatoire pour les prédictions. L'utilisation d'un pipeline simplifie considérablement le processus de validation croisée.

In [19]:

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer

my_pipeline = Pipeline(steps=[('preprocessor', SimpleImputer()),
                               ('model', RandomForestRegressor(n_estimators=50,
                                                                random_state=0))
                              ])
```

Nous obtenons les scores de validation croisée avec la fonction `cross_val_score()`. Nous définissons le nombre de folds avec le paramètre `cv`.

```
In [20]: from sklearn.model_selection import cross_val_score

# Multiply by -1 since sklearn calculates *negative* MAE
scores = -1 * cross_val_score(my_pipeline, X, y,
                              cv=5,
                              scoring='neg_mean_absolute_error')

print("MAE scores:\n", scores)
```

```
MAE scores:
[301628.7893587  303164.4782723  287298.331666   236061.84754543
 260383.45111427]
```

Le paramètre de scoring détermine la mesure de qualité du modèle à rapporter, ici l'erreur absolue moyenne (MAE) négative.

```
In [21]: print("Average MAE score (across experiments):")
print(scores.mean())
```

```
Average MAE score (across experiments):
277707.3795913405
```

La validation croisée offre une mesure de la qualité du modèle plus précise et simplifie le code, car il n'est plus nécessaire de gérer des ensembles d'entraînement et de validation séparés. C'est particulièrement bénéfique pour les petits ensembles de données.

Conclusion :

1. Gestion des valeurs manquantes

Approche 1 : Supprimer les colonnes avec des valeurs manquantes

Simple mais inefficace, car cela peut entraîner une perte d'informations importante.

Approche 2 : Imputation

Remplacer les valeurs manquantes par des statistiques (comme la moyenne).
Améliore les performances, mais peut introduire un biais.

Approche 3 : Extension de l'imputation

Impute les valeurs tout en ajoutant une colonne pour marquer les emplacements imputés.
C'est généralement la meilleure option, car elle conserve une trace des données manquantes.

2. Gestion des variables catégoriques

Approche 1 : Supprimer les variables catégoriques

Cette méthode simple entraîne généralement une perte d'informations.

Approche 2 : Encodage ordinal

Utile quand il y a un ordre naturel dans les catégories,
mais peut causer des erreurs si l'ordre est arbitraire.

Approche 3 : Encodage One-Hot

Méthode courante qui transforme les catégories en colonnes binaires,
bien que moins efficace avec de nombreuses catégories.

3. Pipelines

Les pipelines structurent le prétraitement et la modélisation,
rendant le processus plus propre et adapté à la mise en production.

4. Validation croisée

La validation croisée donne une estimation plus fiable de la qualité du modèle,
surtout avec des petits ensembles de données,
en réduisant les biais dus à un seul ensemble de validation.

Conclusion globale :

Ces techniques de gestion des valeurs manquantes et des variables catégoriques sont essentielles pour préparer les données efficacement avant la modélisation. L'utilisation de pipelines et de validation croisée renforce la rigueur et l'automatisation du processus.