Valeurs Manquantes: Guide Complet pour Data Analyst Bancaire

1. Introduction et Enjeux

1.1 Qu'est-ce qu'une valeur manquante?

Une valeur manquante (NA, NaN, NULL) représente l'absence d'information pour une observation donnée. En Python/Pandas, elles sont représentées par `np.nan`, `None`, ou `pd.NA`.

1.2 Pourquoi c'est critique en banque?

Impact	Conséquence	Exemple
Décisions erronées	Scoring biaisé	Refuser un bon client
Biais statistique	Moyennes fausses	Sous-estimer le risque
Modèles défaillants	Prédictions incorrectes	Manquer des fraudes
Non-conformité	Sanctions régulateur	Audit BRH/ACPR

---

2. Types de Valeurs Manquantes

2.1 Classification Théorique

```
MCAR (Missing Completely At Random)
│
│   La probabilité de manquant ne dépend de RIEN
│   Exemple: Erreur de saisie aléatoire
│   Impact: Réduit la taille mais pas de biais
│
├── MAR (Missing At Random)
│
│   La probabilité de manquant dépend d'AUTRES variables observées
│   Exemple: Les jeunes déclarent moins leur patrimoine
│   Impact: Biais corrigible si on contrôle les autres variables
│
└── MNAR (Missing Not At Random)

    La probabilité de manquant dépend de la VALEUR elle-même
    Exemple: Les hauts revenus cachent leur revenu
    Impact: Biais non corrigible, estimation impossible
```

2.2 Diagnostic du Type de Manquant

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats

def diagnostiquer_manquants(df):
    """Analyse complète des valeurs manquantes"""
```

```
    # 1. Vue d'ensemble
    print("=" * 60)
    print("DIAGNOSTIC DES VALEURS MANQUANTES")
    print("=" * 60)

    missing = df.isnull().sum()
    missing_pct = (missing / len(df)) * 100

    summary = pd.DataFrame({
        'Nb_Manquants': missing,
        'Pourcentage': missing_pct.round(2),
        'Type': df.dtypes
    })
    summary = summary[summary['Nb_Manquants'] > 0].sort_values(
        'Pourcentage', ascending=False
    )

    print("\nVariables avec valeurs manquantes:")
    print(summary)
    print(f"\nTotal observations: {len(df)}")
    print(f"Observations complètes: {df.dropna().shape[0]}")

    return summary

# Application bancaire
df_clients = pd.DataFrame({
    'client_id': range(1000),
    'age': np.random.choice([np.nan] + list(range(18, 80)), 1000, p=[0.02] + [0.98/62]*62)
    'revenu': np.where(np.random.random(1000) < 0.15, np.nan, np.random.lognormal(10, 1, 1
    'ancienneté': np.random.choice([np.nan] + list(range(0, 40)), 1000, p=[0.05] + [0.95/4
    'score_credit': np.where(np.random.random(1000) < 0.08, np.nan, np.random.randint(300,
    'type_emploi': np.random.choice(['CDI', 'CDD', 'Indépendant', np.nan], 1000, p=[0.5, 0
})

summary = diagnostiquer_manquants(df_clients)
```

2.3 Test de Little pour MCAR

```
def test_little_mcar(df):
    """
    Test de Little pour MCAR
    H0: Les données sont MCAR
    p-value < 0.05 => Rejeter MCAR
    """
    # Version simplifiée (le test complet nécessite des packages spécialisés)

    # Créer une variable indicatrice de manquant
    df_test = df.copy()

    for col in df.select_dtypes(include=[np.number]).columns:
        if df[col].isnull().any():
            # Comparer les autres variables selon que col est manquant ou non
            mask_missing = df[col].isnull()
```

```
            for other_col in df.select_dtypes(include=[np.number]).columns:
                if other_col != col and not df[other_col].isnull().all():
                    group_missing = df.loc[mask_missing, other_col].dropna()
                    group_present = df.loc[~mask_missing, other_col].dropna()

                    if len(group_missing) > 1 and len(group_present) > 1:
                        stat, p_value = stats.mannwhitneyu(
                            group_missing, group_present, alternative='two-sided'
                        )
                        if p_value < 0.05:
                            print(f"Possible MAR: {col} manquant lié à {other_col} (p={p_v

test_little_mcar(df_clients)
```

---

3. Visualisation des Manquants

3.1 Matrice de Manquants

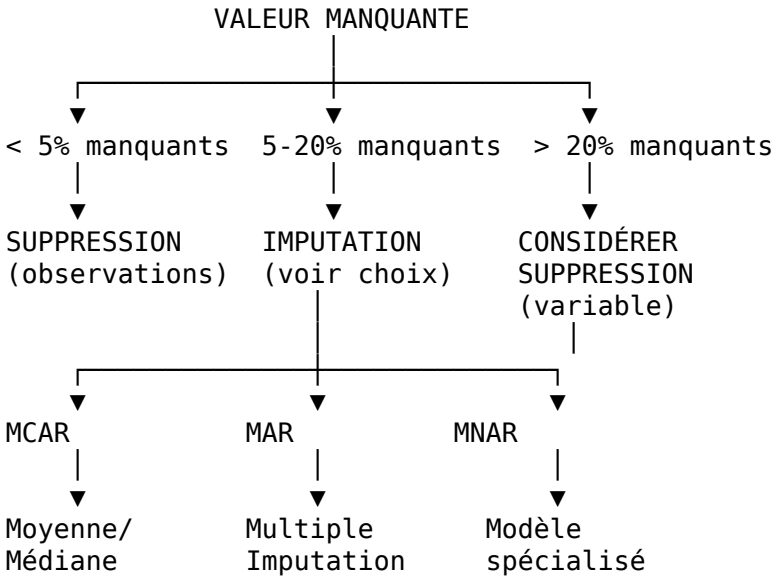
```
import missingno as msno

def visualiser_manquants(df, titre="Analyse des Valeurs Manquantes"):
    """Visualisation complète des patterns de manquants"""

    fig, axes = plt.subplots(2, 2, figsize=(15, 12))
    fig.suptitle(titre, fontsize=14, fontweight='bold')

    # 1. Bar chart des manquants
    ax1 = axes[0, 0]
    missing_pct = (df.isnull().sum() / len(df) * 100).sort_values(ascending=True)
    missing_pct.plot(kind='barh', ax=ax1, color='coral')
    ax1.set_xlabel('Pourcentage de manquants')
    ax1.set_title('Taux de valeurs manquantes par variable')
    ax1.axvline(x=5, color='red', linestyle='--', label='Seuil 5%')
    ax1.legend()

    # 2. Heatmap des manquants
    ax2 = axes[0, 1]
    sns.heatmap(df.isnull(), cbar=True, yticklabels=False, ax=ax2, cmap='viridis')
    ax2.set_title('Pattern des valeurs manquantes')

    # 3. Corrélation des manquants
    ax3 = axes[1, 0]
    missing_corr = df.isnull().corr()
    mask = np.triu(np.ones_like(missing_corr, dtype=bool))
    sns.heatmap(missing_corr, mask=mask, annot=True, fmt='.2f', ax=ax3,
                cmap='coolwarm', center=0)
    ax3.set_title('Corrélation entre manquants')

    # 4. Dendrogram
    ax4 = axes[1, 1]
    # Clustering des colonnes par pattern de manquants
```

```
    from scipy.cluster import hierarchy
    missing_matrix = df.isnull().astype(int)
    if missing_matrix.sum().sum() > 0:  # S'il y a des manquants
        linkage = hierarchy.linkage(missing_matrix.T, method='ward')
        hierarchy.dendrogram(linkage, labels=df.columns, ax=ax4, leaf_rotation=90)
    ax4.set_title('Clustering des patterns de manquants')

    plt.tight_layout()
    plt.savefig('analyse_manquants.png', dpi=300, bbox_inches='tight')
    plt.show()

visualiser_manquants(df_clients)
```

---

4. Méthodes de Traitement

4.1 Arbre de Décision pour le Traitement



4.2 Suppression

```
def traiter_par_suppression(df, seuil_col=0.5, seuil_row=0.3):
    """
    Suppression des lignes et colonnes avec trop de manquants

    Paramètres:
    - seuil_col: Supprimer colonnes avec plus de X% de manquants
    - seuil_row: Supprimer lignes avec plus de X% de manquants
    """
    print("TRAITEMENT PAR SUPPRESSION")
    print("-" * 40)

    n_initial = len(df)
    cols_initial = len(df.columns)
```

```
    # 1. Supprimer les colonnes avec trop de manquants
    missing_pct_col = df.isnull().sum() / len(df)
    cols_to_drop = missing_pct_col[missing_pct_col > seuil_col].index.tolist()

    if cols_to_drop:
        print(f"Colonnes supprimées (>{seuil_col*100}% manquants): {cols_to_drop}")
        df = df.drop(columns=cols_to_drop)

    # 2. Supprimer les lignes avec trop de manquants
    missing_pct_row = df.isnull().sum(axis=1) / len(df.columns)
    rows_to_drop = missing_pct_row[missing_pct_row > seuil_row].index

    if len(rows_to_drop) > 0:
        print(f"Lignes supprimées (>{seuil_row*100}% manquants): {len(rows_to_drop)}")
        df = df.drop(index=rows_to_drop)

    print(f"\nRésultat: {len(df)}/{n_initial} observations, {len(df.columns)}/{cols_initia

    return df

# Application
df_clean = traiter_par_suppression(df_clients.copy())
```

4.3 Imputation Simple

```
def imputer_simple(df, strategie='median'):
    """
    Imputation simple pour les variables numériques et catégorielles

    Stratégies:
    - 'mean': Moyenne (sensible aux outliers)
    - 'median': Médiane (robuste)
    - 'mode': Mode (pour catégorielles)
    - 'constant': Valeur fixe
    """
    from sklearn.impute import SimpleImputer

    df_imputed = df.copy()

    # Variables numériques
    num_cols = df.select_dtypes(include=[np.number]).columns
    if len(num_cols) > 0:
        imputer_num = SimpleImputer(strategy=strategie)
        df_imputed[num_cols] = imputer_num.fit_transform(df[num_cols])
        print(f"Variables numériques imputées avec {strategie}: {list(num_cols)}")

    # Variables catégorielles
    cat_cols = df.select_dtypes(include=['object', 'category']).columns
    if len(cat_cols) > 0:
        imputer_cat = SimpleImputer(strategy='most_frequent')
        df_imputed[cat_cols] = imputer_cat.fit_transform(df[cat_cols])
        print(f"Variables catégorielles imputées avec mode: {list(cat_cols)}")

    return df_imputed
```

```
# Application
df_imputed_simple = imputer_simple(df_clients.copy(), strategie='median')
```

4.4 Imputation par Groupe (Contexte Bancaire)

```
def imputer_par_groupe(df, col_to_impute, group_cols, strategie='median'):
    """
    Imputation par groupe pour plus de précision

    Exemple bancaire:
    - Imputer le revenu par type d'emploi et tranche d'âge
    """
    df_imputed = df.copy()

    if strategie == 'median':
        func = 'median'
    elif strategie == 'mean':
        func = 'mean'
    else:
        raise ValueError("Stratégie non supportée")

    # Calculer la valeur par groupe
    group_values = df.groupby(group_cols)[col_to_impute].transform(func)

    # Imputer seulement les manquants
    mask = df_imputed[col_to_impute].isnull()
    df_imputed.loc[mask, col_to_impute] = group_values[mask]

    # Pour les groupes sans valeur, utiliser la médiane globale
    still_missing = df_imputed[col_to_impute].isnull()
    if still_missing.any():
        global_value = df[col_to_impute].agg(func)
        df_imputed.loc[still_missing, col_to_impute] = global_value

    print(f"Imputé {mask.sum()} valeurs de '{col_to_impute}' par groupe {group_cols}")

    return df_imputed

# Application bancaire: Imputer revenu par type d'emploi
df_grouped = imputer_par_groupe(
    df_clients.copy(),
    col_to_impute='revenu',
    group_cols=['type_emploi'],
    strategie='median'
)
```

4.5 Imputation par KNN

```
def imputer_knn(df, n_neighbors=5):
    """
    Imputation par K plus proches voisins
    Utilise la similarité entre observations
    """
```

```
    from sklearn.impute import KNNImputer
    from sklearn.preprocessing import LabelEncoder

    df_imputed = df.copy()

    # Encoder les catégorielles pour KNN
    le_dict = {}
    cat_cols = df.select_dtypes(include=['object', 'category']).columns

    for col in cat_cols:
        le = LabelEncoder()
        # Gérer les NA
        df_imputed[col] = df_imputed[col].fillna('__MISSING__')
        df_imputed[col] = le.fit_transform(df_imputed[col])
        le_dict[col] = le

    # Appliquer KNN
    imputer = KNNImputer(n_neighbors=n_neighbors, weights='distance')
    df_imputed_values = imputer.fit_transform(df_imputed)
    df_imputed = pd.DataFrame(df_imputed_values, columns=df.columns)

    # Décoder les catégorielles
    for col in cat_cols:
        df_imputed[col] = df_imputed[col].round().astype(int)
        df_imputed[col] = le_dict[col].inverse_transform(df_imputed[col])
        df_imputed.loc[df_imputed[col] == '__MISSING__', col] = np.nan

    print(f"Imputation KNN avec k={n_neighbors} terminée")

    return df_imputed

# Application
df_knn = imputer_knn(df_clients.copy(), n_neighbors=5)
```

4.6 Imputation Multiple (MICE)

```
def imputer_mice(df, n_imputations=5, random_state=42):
    """
    Multiple Imputation by Chained Equations (MICE)

    Avantages:
    - Tient compte de l'incertitude
    - Génère plusieurs jeux de données
    - Combine les résultats pour inférence
    """
    from sklearn.experimental import enable_iterative_imputer
    from sklearn.impute import IterativeImputer
    from sklearn.linear_model import BayesianRidge

    # Sélectionner les colonnes numériques
    num_cols = df.select_dtypes(include=[np.number]).columns
    df_num = df[num_cols].copy()

    # Créer n imputations
```

```
    imputations = []

    for i in range(n_imputations):
        imputer = IterativeImputer(
            estimator=BayesianRidge(),
            max_iter=10,
            random_state=random_state + i,
            initial_strategy='median'
        )

        imputed = imputer.fit_transform(df_num)
        imputations.append(pd.DataFrame(imputed, columns=num_cols))

    # Combiner les imputations (moyenne)
    df_final = pd.concat(imputations).groupby(level=0).mean()

    # Remettre les colonnes non-numériques
    for col in df.columns:
        if col not in num_cols:
            df_final[col] = df[col].values

    print(f"Imputation MICE avec {n_imputations} imputations terminée")

    return df_final, imputations

# Application
df_mice, all_imputations = imputer_mice(df_clients.copy())
```

---

5. Cas Pratique Bancaire: Scoring de Crédit

5.1 Le Problème

```
"""
CONTEXTE: Développement d'un modèle de scoring de crédit
PROBLÈME: 15% de revenus manquants, 8% de scores manquants

Impact potentiel:
- Biais de sélection (les hauts revenus cachent leur revenu?)
- Modèle entraîné sur données incomplètes
- Prédictions erronées
"""

# Simuler un dataset de scoring avec manquants réalistes
np.random.seed(42)
n = 5000

df_scoring = pd.DataFrame({
    'client_id': range(n),
    'age': np.random.randint(18, 75, n),
    'revenu_mensuel': np.random.lognormal(10, 0.8, n),
    'anciennete_emploi': np.random.exponential(5, n),
    'nb_credits_actifs': np.random.poisson(2, n),
```

```
    'montant_demande': np.random.lognormal(11, 0.5, n),
    'type_emploi': np.random.choice(['CDI', 'CDD', 'Independant', 'Fonctionnaire'], n,
                                    p=[0.5, 0.2, 0.2, 0.1]),
    'proprietaire': np.random.choice([0, 1], n, p=[0.6, 0.4]),
    'defaut': np.random.choice([0, 1], n, p=[0.95, 0.05])
})

# Introduire des manquants MNAR pour revenu (hauts revenus cachent plus)
revenu_percentile = df_scoring['revenu_mensuel'].rank(pct=True)
proba_missing = 0.05 + 0.15 * revenu_percentile  # 5-20% selon revenu
df_scoring.loc[np.random.random(n) < proba_missing, 'revenu_mensuel'] = np.nan

# MAR pour ancienneté (CDD ont plus de manquants)
mask_cdd = df_scoring['type_emploi'] == 'CDD'
df_scoring.loc[mask_cdd & (np.random.random(n) < 0.25), 'anciennete_emploi'] = np.nan

print("Dataset de scoring créé:")
diagnostiquer_manquants(df_scoring)
```

5.2 Solution Complète

```
class ImputerScoringBancaire:
    """
    Pipeline d'imputation spécialisé pour le scoring bancaire
    """

    def __init__(self):
        self.imputers = {}
        self.statistics = {}

    def fit(self, df, target_col='defaut'):
        """
        Apprendre les paramètres d'imputation sur les données d'entraînement
        """
        print("Apprentissage des paramètres d'imputation...")

        # 1. Sauvegarder les statistiques par groupe
        self.statistics['revenu_par_emploi'] = df.groupby('type_emploi')['revenu_mensuel']
        self.statistics['anciennete_par_emploi'] = df.groupby('type_emploi')['anciennete_e

        # 2. Médiane globale comme fallback
        self.statistics['revenu_global'] = df['revenu_mensuel'].median()
        self.statistics['anciennete_global'] = df['anciennete_emploi'].median()

        # 3. Créer indicateurs de manquant (pour le modèle)
        self.cols_with_missing = df.columns[df.isnull().any()].tolist()

        print(f"Colonnes avec manquants: {self.cols_with_missing}")

        return self

    def transform(self, df):
        """
        Appliquer l'imputation
```

```
        """
        df_out = df.copy()

        # 1. Créer les indicateurs de manquant AVANT imputation
        for col in self.cols_with_missing:
            df_out[f'{col}_manquant'] = df_out[col].isnull().astype(int)

        # 2. Imputer revenu par type d'emploi
        for emploi, median_revenu in self.statistics['revenu_par_emploi'].items():
            mask = (df_out['type_emploi'] == emploi) & (df_out['revenu_mensuel'].isnull())
            df_out.loc[mask, 'revenu_mensuel'] = median_revenu

        # Fallback pour types d'emploi non vus
        df_out['revenu_mensuel'].fillna(self.statistics['revenu_global'], inplace=True)

        # 3. Imputer ancienneté par type d'emploi
        for emploi, median_anc in self.statistics['anciennete_par_emploi'].items():
            mask = (df_out['type_emploi'] == emploi) & (df_out['anciennete_emploi'].isnull
            df_out.loc[mask, 'anciennete_emploi'] = median_anc

        df_out['anciennete_emploi'].fillna(self.statistics['anciennete_global'], inplace=T

        print(f"Imputation terminée. Manquants restants: {df_out.isnull().sum().sum()}")

        return df_out

    def fit_transform(self, df, target_col='defaut'):
        return self.fit(df, target_col).transform(df)

# Application
imputer = ImputerScoringBancaire()
df_scoring_imputed = imputer.fit_transform(df_scoring)

# Vérification
print("\nDistribution avant/après imputation:")
print("Revenu - Avant:", df_scoring['revenu_mensuel'].describe()['mean'])
print("Revenu - Après:", df_scoring_imputed['revenu_mensuel'].describe()['mean'])
```

5.3 Validation de l'Imputation

```
def valider_imputation(df_original, df_imputed, col):
    """
    Valider que l'imputation n'a pas trop biaisé les données
    """
    fig, axes = plt.subplots(1, 3, figsize=(15, 4))

    # 1. Distribution avant/après
    ax1 = axes[0]
    df_original[col].hist(ax=ax1, alpha=0.5, label='Original (sans NA)', bins=30)
    df_imputed[col].hist(ax=ax1, alpha=0.5, label='Après imputation', bins=30)
    ax1.legend()
    ax1.set_title(f'Distribution de {col}')

    # 2. Comparer les statistiques
```

```
    ax2 = axes[1]
    stats_original = df_original[col].describe()
    stats_imputed = df_imputed[col].describe()

    x = np.arange(len(stats_original))
    width = 0.35
    ax2.bar(x - width/2, stats_original.values, width, label='Original')
    ax2.bar(x + width/2, stats_imputed.values, width, label='Imputé')
    ax2.set_xticks(x)
    ax2.set_xticklabels(stats_original.index, rotation=45)
    ax2.legend()
    ax2.set_title('Statistiques comparées')

    # 3. QQ-Plot
    ax3 = axes[2]
    from scipy import stats as scipy_stats
    scipy_stats.probplot(df_imputed[col].dropna(), dist="norm", plot=ax3)
    ax3.set_title('QQ-Plot après imputation')

    plt.tight_layout()
    plt.show()

    # Test statistique
    ks_stat, p_value = stats.ks_2samp(
        df_original[col].dropna(),
        df_imputed[col]
    )
    print(f"\nTest KS (distribution similaire): stat={ks_stat:.4f}, p-value={p_value:.4f}"
    if p_value > 0.05:
        print("=> Distributions similaires (imputation OK)")
    else:
        print("=> Distributions différentes (attention au biais!)")

valider_imputation(df_scoring, df_scoring_imputed, 'revenu_mensuel')
```

---

6. Impact sur les Modèles

6.1 Comparaison des Approches

```
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import StandardScaler

def comparer_strategies_imputation(df, target='defaut'):
    """
    Comparer l'impact des différentes stratégies d'imputation
    sur la performance du modèle
    """
    resultats = []
```

```python
# Préparer les features
feature_cols = ['age', 'revenu_mensuel', 'anciennete_emploi',
                'nb_credits_actifs', 'montant_demande', 'proprietaire']

# Stratégie 1: Suppression des lignes avec NA
df_dropna = df.dropna(subset=feature_cols)
X_drop = df_dropna[feature_cols]
y_drop = df_dropna[target]

if len(X_drop) > 100:
    scores = cross_val_score(
        LogisticRegression(max_iter=1000), X_drop, y_drop,
        cv=5, scoring='roc_auc'
    )
    resultats.append({
        'Stratégie': 'Suppression NA',
        'N_observations': len(X_drop),
        'AUC_mean': scores.mean(),
        'AUC_std': scores.std()
    })

# Stratégie 2: Imputation médiane
df_median = df.copy()
for col in feature_cols:
    df_median[col].fillna(df_median[col].median(), inplace=True)

X_median = df_median[feature_cols]
y_median = df_median[target]

scores = cross_val_score(
    LogisticRegression(max_iter=1000), X_median, y_median,
    cv=5, scoring='roc_auc'
)
resultats.append({
    'Stratégie': 'Imputation Médiane',
    'N_observations': len(X_median),
    'AUC_mean': scores.mean(),
    'AUC_std': scores.std()
})

# Stratégie 3: Imputation + Indicateur manquant
df_indicator = df.copy()
for col in feature_cols:
    df_indicator[f'{col}_NA'] = df_indicator[col].isnull().astype(int)
    df_indicator[col].fillna(df_indicator[col].median(), inplace=True)

feature_cols_extended = feature_cols + [f'{col}_NA' for col in feature_cols]
X_indicator = df_indicator[feature_cols_extended]
y_indicator = df_indicator[target]

scores = cross_val_score(
    LogisticRegression(max_iter=1000), X_indicator, y_indicator,
    cv=5, scoring='roc_auc'
)
```

```
    resultats.append({
        'Stratégie': 'Médiane + Indicateur NA',
        'N_observations': len(X_indicator),
        'AUC_mean': scores.mean(),
        'AUC_std': scores.std()
    })

    # Afficher les résultats
    df_resultats = pd.DataFrame(resultats)
    print("\nComparaison des stratégies d'imputation:")
    print(df_resultats.to_string(index=False))

    return df_resultats

resultats = comparer_strategies_imputation(df_scoring)
```

---

7. Bonnes Pratiques et Pièges à Éviter

7.1 Checklist

```
AVANT L'IMPUTATION:
□ Documenter le taux de manquants par variable
□ Identifier le type (MCAR, MAR, MNAR)
□ Analyser les patterns (corrélation entre manquants)
□ Décider de la stratégie par variable

PENDANT L'IMPUTATION:
□ NE JAMAIS fit sur les données de test
□ Créer des indicateurs de manquant
□ Utiliser l'imputation par groupe si pertinent
□ Considérer MICE pour analyses statistiques

APRÈS L'IMPUTATION:
□ Valider les distributions
□ Vérifier l'impact sur les corrélations
□ Tester l'impact sur le modèle final
□ Documenter les choix effectués
```

7.2 Erreurs Courantes

Erreur	Conséquence	Solution
Fit sur tout le dataset	Data leakage	Fit sur train uniquement
Ignorer le type de manquant	Biais non détecté	Tester MCAR/MAR/MNAR
Imputer les ID	Données corrompues	Exclure les identifiants
Imputer la cible	Overfitting	Ne jamais imputer Y
Même imputation pour tout	Perte d'information	Adapter par variable

7.3 Code de Production

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

class ImputerBancaireProduction(BaseEstimator, TransformerMixin):
    """
    Imputer prêt pour la production
    Compatible avec sklearn Pipeline
    """

    def __init__(self, strategie_num='median', strategie_cat='most_frequent',
                 creer_indicateurs=True, cols_groupe=None):
        self.strategie_num = strategie_num
        self.strategie_cat = strategie_cat
        self.creer_indicateurs = creer_indicateurs
        self.cols_groupe = cols_groupe or {}

    def fit(self, X, y=None):
        self.num_cols_ = X.select_dtypes(include=[np.number]).columns.tolist()
        self.cat_cols_ = X.select_dtypes(include=['object', 'category']).columns.tolist()

        # Calculer les valeurs d'imputation
        if self.strategie_num == 'median':
            self.num_values_ = X[self.num_cols_].median().to_dict()
        else:
            self.num_values_ = X[self.num_cols_].mean().to_dict()

        self.cat_values_ = X[self.cat_cols_].mode().iloc[0].to_dict()

        # Colonnes avec manquants
        self.missing_cols_ = X.columns[X.isnull().any()].tolist()

        return self

    def transform(self, X):
        X_out = X.copy()

        # Créer indicateurs
        if self.creer_indicateurs:
            for col in self.missing_cols_:
                X_out[f'{col}_NA'] = X_out[col].isnull().astype(int)

        # Imputer numériques
        for col in self.num_cols_:
            if col in self.num_values_:
                X_out[col].fillna(self.num_values_[col], inplace=True)

        # Imputer catégorielles
        for col in self.cat_cols_:
            if col in self.cat_values_:
                X_out[col].fillna(self.cat_values_[col], inplace=True)

        return X_out
```

```python
# Utilisation dans un pipeline
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ('imputer', ImputerBancaireProduction(creer_indicateurs=True)),
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(max_iter=1000))
])
```

---

8. Résumé et Mnémotechniques

Mnémotechnique: “DITS”

- **D**étection: Identifier les manquants et leur type
- **I**mpact: Évaluer les conséquences sur l’analyse
- **T**raitement: Choisir la stratégie appropriée
- **S**uivi: Valider et documenter

Règles d’Or

1. **< 5% manquants** → Suppression acceptable
2. **5-20% manquants** → Imputation nécessaire
3. **> 20% manquants** → Envisager suppression de la variable
4. **MCAR** → Imputation simple OK
5. **MAR** → Imputation par groupe ou MICE
6. **MNAR** → Modèle spécialisé ou variable proxy

Formule de Décision

```
SI manquants_trop_nombreux (>50%):
    SUPPRIMER la variable
SINON SI MCAR:
    IMPUTER avec médiane/mode
SINON SI MAR:
    IMPUTER par groupe ou MICE
SINON (MNAR):
    CRÉER indicateur + IMPUTER + MODÉLISER séparément
```