Manuel de Préparation: Types de Modèles en Data Science

Introduction

Un modèle en data science est une représentation mathématique ou statistique d'un phénomène réel. Choisir le bon type de modèle est crucial pour résoudre efficacement un problème d'analyse. Ce manuel couvre les différents types de modèles, leurs caractéristiques et leurs applications spécifiques au secteur bancaire.

Partie 1: Classification Générale des Modèles

1.1 Vue d'Ensemble

```
MODÈLES
│
├── PAR OBJECTIF
│   ├── Descriptifs (comprendre)
│   ├── Prédictifs (anticiper)
│   └── Prescriptifs (recommander)
│
├── PAR APPRENTISSAGE
│   ├── Supervisé (avec étiquettes)
│   ├── Non supervisé (sans étiquettes)
│   └── Semi-supervisé (mixte)
│
├── PAR APPROCHE
│   ├── Statistiques (hypothèses, inférence)
│   └── Machine Learning (patterns, performance)
│
└── PAR SORTIE
    ├── Classification (catégories)
    ├── Régression (valeurs continues)
    └── Clustering (groupes)
```

1.2 Définitions Fondamentales

Catégorie	Définition	Usage Principal
Descriptif	Résume et explique les données historiques	Comprendre ce qui s'est passé
Prédictif	Estime des valeurs futures ou inconnues	Anticiper ce qui va se passer
Prescriptif	Recommande des actions optimales	Décider quoi faire
Supervisé	Apprend à partir d'exemples étiquetés	Classification, Régression
Non supervisé	Découvre des structures dans les données	Clustering, Réduction de dimension

---

Partie 2: Modèles Descriptifs

2.1 Statistiques Descriptives

Objectif Résumer les caractéristiques principales d'un jeu de données.

Composants

```
- Mesures de tendance centrale: Moyenne, Médiane, Mode
- Mesures de dispersion: Variance, Écart-type, IQR
- Mesures de forme: Asymétrie, Kurtosis
- Tableaux de fréquences
- Statistiques bivariées: Corrélation, Covariance
```

Application Bancaire: Profil Client

```python
def profil_client_descriptif(df):
    """Statistiques descriptives du portefeuille client"""

    profil = {
        'nb_clients': len(df),
        'age_moyen': df['age'].mean(),
        'age_median': df['age'].median(),
        'solde_moyen': df['solde'].mean(),
        'solde_median': df['solde'].median(),
        'distribution_segments': df['segment'].value_counts(normalize=True),
        'correlation_age_solde': df['age'].corr(df['solde'])
    }
    return profil
```

2.2 Analyse Exploratoire (EDA)

Objectif Découvrir des patterns, anomalies et relations dans les données.

Techniques

- Visualisations univariées (histogrammes, box plots)
- Visualisations bivariées (scatter plots, heatmaps)
- Détection d'outliers
- Analyse des valeurs manquantes

Application Bancaire: Analyse du Portefeuille de Crédit

```python
def eda_portefeuille_credit(df_prets):
    """EDA complète du portefeuille de crédit"""

    # Distribution des montants
    print("Distribution des montants:")
    print(df_prets['montant'].describe())

    # Répartition par type
    print("\nRépartition par type de prêt:")
    print(df_prets['type_pret'].value_counts())
```

```python
    # Taux de défaut par segment
    print("\nTaux de défaut par segment:")
    print(df_prets.groupby('segment')['defaut'].mean())

    # Corrélations
    print("\nCorrélations numériques:")
    print(df_prets[['montant', 'taux', 'duree', 'defaut']].corr())
```

2.3 Modèles de Segmentation Descriptive

Objectif Identifier des groupes homogènes dans les données.

Techniques

- Analyse RFM (Recency, Frequency, Monetary)
- Segmentation par règles métier
- Clustering

Application Bancaire: Segmentation RFM

```python
def segmentation_rfm(df_transactions, client_col, date_col, montant_col):
    """Segmentation RFM des clients"""

    today = pd.Timestamp.now()

    rfm = df_transactions.groupby(client_col).agg({
        date_col: lambda x: (today - x.max()).days,  # Recency
        client_col: 'count',                          # Frequency
        montant_col: 'sum'                            # Monetary
    }).rename(columns={
        date_col: 'Recency',
        client_col: 'Frequency',
        montant_col: 'Monetary'
    })

    # Scoring par quintile
    rfm['R_Score'] = pd.qcut(rfm['Recency'], 5, labels=[5,4,3,2,1])
    rfm['F_Score'] = pd.qcut(rfm['Frequency'].rank(method='first'), 5, labels=[1,2,3,4,5])
    rfm['M_Score'] = pd.qcut(rfm['Monetary'].rank(method='first'), 5, labels=[1,2,3,4,5])

    rfm['RFM_Segment'] = rfm['R_Score'].astype(str) + rfm['F_Score'].astype(str) + rfm['M_

    return rfm
```

Partie 3: Modèles Prédictifs

3.1 Classification Supervisée

Définition Prédire une variable catégorielle (classe) à partir de variables explicatives.

Types de Problèmes

```
- Classification binaire: 2 classes (défaut: oui/non)
- Classification multiclasse: 3+ classes (rating: A/B/C/D)
- Classification multilabel: Plusieurs labels possibles
```

Modèles Principaux Régression Logistique:

```python
from sklearn.linear_model import LogisticRegression

# Prédiction de défaut de paiement
model = LogisticRegression()
model.fit(X_train, y_train)

# Probabilités de défaut
probas = model.predict_proba(X_test)[:, 1]

# Interprétation des coefficients
for feature, coef in zip(features, model.coef_[0]):
    print(f"{feature}: odds ratio = {np.exp(coef):.2f}")
```

Arbre de Décision:

```python
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier(max_depth=5, min_samples_leaf=100)
model.fit(X_train, y_train)

# Importance des variables
for feature, importance in zip(features, model.feature_importances_):
    print(f"{feature}: {importance:.3f}")
```

Random Forest:

```python
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100, max_depth=10)
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

Gradient Boosting (XGBoost, LightGBM):

```python
import xgboost as xgb

model = xgb.XGBClassifier(n_estimators=100, max_depth=5, learning_rate=0.1)
model.fit(X_train, y_train)
```

Application Bancaire: Scoring de Crédit

```python
def build_credit_scoring_model(df):
    """Construction d'un modèle de scoring de crédit"""

    from sklearn.model_selection import train_test_split
    from sklearn.ensemble import GradientBoostingClassifier
    from sklearn.metrics import roc_auc_score, classification_report

    # Features
```

```
    features = ['revenu', 'anciennete_emploi', 'dette_existante',
                'nb_credits', 'age', 'montant_demande']
    X = df[features]
    y = df['defaut']

    # Split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=

    # Modèle
    model = GradientBoostingClassifier(n_estimators=100, max_depth=4)
    model.fit(X_train, y_train)

    # Évaluation
    y_pred_proba = model.predict_proba(X_test)[:, 1]
    auc = roc_auc_score(y_test, y_pred_proba)
    print(f"AUC-ROC: {auc:.3f}")

    return model
```

Métriques d'Évaluation

```
- Accuracy: (TP + TN) / Total
- Precision: TP / (TP + FP)
- Recall (Sensibilité): TP / (TP + FN)
- F1-Score: 2 × (Precision × Recall) / (Precision + Recall)
- AUC-ROC: Aire sous la courbe ROC
- Matrice de confusion
```

3.2 Régression Supervisée

Définition Prédire une variable continue à partir de variables explicatives.

Modèles Principaux Régression Linéaire:

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)

# Coefficients
print(f"Intercept: {model.intercept_:.2f}")
for feature, coef in zip(features, model.coef_):
    print(f"{feature}: {coef:.4f}")
```

Régression Ridge (L2) et Lasso (L1):

```
from sklearn.linear_model import Ridge, Lasso

# Ridge: pénalise les coefficients élevés
ridge = Ridge(alpha=1.0)
ridge.fit(X_train, y_train)

# Lasso: sélection de variables (coefficients peuvent être 0)
```

```
lasso = Lasso(alpha=0.1)
lasso.fit(X_train, y_train)
```

Régression Polynomiale:

```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)

model = LinearRegression()
model.fit(X_poly, y)
```

Random Forest Regressor:

```
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators=100)
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

Application Bancaire: Prédiction de LTV (Lifetime Value)

```
def predict_customer_ltv(df):
    """Prédiction de la valeur vie client"""

    from sklearn.ensemble import GradientBoostingRegressor
    from sklearn.metrics import mean_absolute_error, r2_score

    features = ['anciennete', 'nb_produits', 'solde_moyen',
                'nb_transactions', 'age', 'segment_encoded']

    X = df[features]
    y = df['ltv_actuel']  # LTV calculé historiquement

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

    model = GradientBoostingRegressor(n_estimators=100)
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)

    print(f"MAE: {mean_absolute_error(y_test, y_pred):,.0f} HTG")
    print(f"R²: {r2_score(y_test, y_pred):.3f}")

    return model
```

Métriques d'Évaluation

```
- MAE (Mean Absolute Error): Erreur moyenne absolue
- MSE (Mean Squared Error): Erreur quadratique moyenne
- RMSE: Racine de MSE
- R² (Coefficient de détermination): Variance expliquée
- MAPE: Erreur absolue moyenne en pourcentage
```

3.3 Modèles de Séries Temporelles

Définition Modèles spécialisés pour données séquentielles dans le temps.

Modèles Principaux Moyennes Mobiles:

```
# Moyenne mobile simple
df['MA_7'] = df['valeur'].rolling(window=7).mean()

# Moyenne mobile exponentielle
df['EMA_7'] = df['valeur'].ewm(span=7).mean()
```

ARIMA (AutoRegressive Integrated Moving Average):

```
from statsmodels.tsa.arima.model import ARIMA

# Ordre (p, d, q)
model = ARIMA(series, order=(1, 1, 1))
results = model.fit()

# Prévisions
forecast = results.forecast(steps=30)
```

Prophet (Facebook):

```
from prophet import Prophet

df_prophet = df[['date', 'valeur']].rename(
    columns={'date': 'ds', 'valeur': 'y'}
)

model = Prophet(yearly_seasonality=True, weekly_seasonality=True)
model.fit(df_prophet)

future = model.make_future_dataframe(periods=30)
forecast = model.predict(future)
```

Application Bancaire: Prévision des Dépôts

```
def prevoir_depots(df_depots, horizon=30):
    """Prévision des dépôts totaux"""

    from prophet import Prophet

    df_prep = df_depots[['date', 'total_depots']].rename(
        columns={'date': 'ds', 'total_depots': 'y'}
    )

    model = Prophet(
        yearly_seasonality=True,
        weekly_seasonality=True,
        changepoint_prior_scale=0.05
    )
    model.fit(df_prep)

    future = model.make_future_dataframe(periods=horizon)
```

```
    forecast = model.predict(future)

    return forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']]
```

Partie 4: Modèles Non Supervisés

4.1 Clustering (Regroupement)

Définition Identifier des groupes naturels dans les données sans étiquettes préalables.

Modèles Principaux K-Means:

```
from sklearn.cluster import KMeans

# Déterminer le nombre optimal de clusters
inertias = []
for k in range(2, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X)
    inertias.append(kmeans.inertia_)

# Méthode du coude
plt.plot(range(2, 11), inertias, 'bo-')
plt.xlabel('Nombre de clusters')
plt.ylabel('Inertie')
plt.show()

# Modèle final
kmeans = KMeans(n_clusters=4, random_state=42)
df['cluster'] = kmeans.fit_predict(X)
```

Clustering Hiérarchique:

```
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage

# Dendrogramme
Z = linkage(X, method='ward')
dendrogram(Z)
plt.show()

# Clustering
hc = AgglomerativeClustering(n_clusters=4)
df['cluster'] = hc.fit_predict(X)
```

DBSCAN (Density-Based):

```
from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=0.5, min_samples=5)
df['cluster'] = dbscan.fit_predict(X)

# Identifie aussi les outliers (cluster = -1)
outliers = df[df['cluster'] == -1]
```

Application Bancaire: Segmentation Clients par Comportement

```python
def segmenter_clients_comportement(df):
    """Segmentation comportementale des clients"""

    from sklearn.preprocessing import StandardScaler
    from sklearn.cluster import KMeans

    # Features comportementales
    features = ['nb_transactions_mois', 'montant_moyen_tx',
                'nb_canaux_utilises', 'solde_moyen',
                'ratio_credit_debit', 'nb_produits']

    X = df[features].copy()

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

    # K-Means avec 5 clusters
    kmeans = KMeans(n_clusters=5, random_state=42)
    df['segment_comportement'] = kmeans.fit_predict(X_scaled)

    # Profil de chaque segment
    profil = df.groupby('segment_comportement')[features].mean()

    return df, profil
```

4.2 Réduction de Dimensionnalité

Définition Réduire le nombre de variables tout en préservant l'information essentielle.

Modèles Principaux PCA (Principal Component Analysis):

```python
from sklearn.decomposition import PCA

# Réduire à 2 dimensions pour visualisation
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Variance expliquée
print(f"Variance expliquée: {pca.explained_variance_ratio_.sum():.1%}")

# Pour choisir le nombre de composantes
pca_full = PCA()
pca_full.fit(X_scaled)
cumsum = np.cumsum(pca_full.explained_variance_ratio_)
n_components = np.argmax(cumsum >= 0.95) + 1
print(f"Composantes pour 95% de variance: {n_components}")
```

t-SNE (Visualisation):

```python
from sklearn.manifold import TSNE
```

```
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_scaled)

plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=df['segment'])
plt.show()
```

4.3 Détection d'Anomalies

Définition Identifier les observations qui dévient significativement du comportement normal.

Modèles Principaux Isolation Forest:

```
from sklearn.ensemble import IsolationForest

iso_forest = IsolationForest(contamination=0.01, random_state=42)
df['anomaly'] = iso_forest.fit_predict(X)
# -1 = anomalie, 1 = normal

anomalies = df[df['anomaly'] == -1]
```

One-Class SVM:

```
from sklearn.svm import OneClassSVM

oc_svm = OneClassSVM(nu=0.01)
df['anomaly'] = oc_svm.fit_predict(X_scaled)
```

Local Outlier Factor:

```
from sklearn.neighbors import LocalOutlierFactor

lof = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
df['anomaly'] = lof.fit_predict(X_scaled)
```

Application Bancaire: Détection de Fraude

```
def detecter_transactions_suspectes(df_transactions):
    """Détection de transactions frauduleuses"""

    from sklearn.ensemble import IsolationForest

    features = ['montant', 'heure', 'jour_semaine',
                'distance_localisation_habituelle',
                'nb_tx_jour', 'montant_vs_moyenne']

    X = df_transactions[features]

    # Modèle
    model = IsolationForest(contamination=0.005, random_state=42)
    df_transactions['suspect'] = model.fit_predict(X)

    suspects = df_transactions[df_transactions['suspect'] == -1]
```

```
    print(f"Transactions suspectes: {len(suspects)} ({len(suspects)/len(df_transactions):.

    return suspects
```

Partie 5: Modèles Spécifiques au Secteur Bancaire

5.1 Modèles de Risque de Crédit

PD (Probability of Default)

```
def model_pd(df):
    """Modèle de probabilité de défaut"""

    from sklearn.linear_model import LogisticRegression

    features = ['score_bureau', 'ratio_dette_revenu', 'anciennete_emploi',
                'montant_pret', 'taux_interet', 'ltv']

    X = df[features]
    y = df['defaut_12m']  # Défaut dans les 12 mois

    model = LogisticRegression()
    model.fit(X, y)

    # PD pour chaque client
    df['PD'] = model.predict_proba(X)[:, 1]

    return model, df
```

LGD (Loss Given Default)

```
def model_lgd(df_defaults):
    """Modèle de perte en cas de défaut"""

    # Uniquement sur les cas de défaut

    features = ['type_garantie', 'ltv', 'anciennete_pret', 'montant_original']

    X = df_defaults[features]
    y = df_defaults['perte_reelle'] / df_defaults['exposition']  # LGD observé

    from sklearn.ensemble import GradientBoostingRegressor
    model = GradientBoostingRegressor()
    model.fit(X, y)

    return model
```

EAD (Exposure at Default)

```
def calculer_ead(df):
    """Exposition en cas de défaut"""

    # Pour prêts amortissables: solde restant
```

```
    df['EAD'] = df['solde_restant']

    # Pour lignes de crédit: solde + partie non utilisée
    ccf = 0.75  # Credit Conversion Factor
    df.loc[df['type'] == 'ligne_credit', 'EAD'] = (
        df['solde_utilise'] + ccf * (df['limite'] - df['solde_utilise'])
    )

    return df
```

Expected Loss (Perte Attendue)

```
def calculer_expected_loss(df):
    """Calcul de la perte attendue"""

    df['EL'] = df['PD'] * df['LGD'] * df['EAD']

    el_total = df['EL'].sum()
    el_portefeuille = el_total / df['EAD'].sum() * 100

    print(f"Perte attendue totale: {el_total:,.0f} HTG")
    print(f"EL ratio: {el_portefeuille:.2f}%")

    return df
```

5.2 Modèles de Scoring

Scorecard (Grille de Score)

```
def construire_scorecard(df, model, features):
    """Construction d'une scorecard à partir d'un modèle logistique"""

    # Points de base
    base_score = 600
    pdo = 20  # Points to Double the Odds

    # Coefficients du modèle
    intercept = model.intercept_[0]
    coefficients = model.coef_[0]

    # Facteur de scaling
    factor = pdo / np.log(2)
    offset = base_score - factor * intercept

    # Points par variable
    scorecard = {}
    for feature, coef in zip(features, coefficients):
        points = -coef * factor
        scorecard[feature] = points

    return scorecard, factor, offset
```

5.3 Modèles de Valeur Client

CLV/LTV Prediction

```python
def model_ltv(df):
    """Prédiction de la valeur vie client"""

    # Approche simplifiée: CLV = Marge × Durée × (1 - Churn)

    features = ['anciennete', 'nb_produits', 'solde_moyen',
                'marge_mensuelle', 'nb_transactions']

    from sklearn.ensemble import GradientBoostingRegressor

    X = df[features]
    y = df['ltv_observe']  # LTV historique

    model = GradientBoostingRegressor()
    model.fit(X, y)

    df['ltv_predit'] = model.predict(X)

    return model, df
```

Churn Prediction

```python
def model_churn(df):
    """Prédiction d'attrition client"""

    features = ['nb_transactions_3m', 'variation_solde',
                'nb_reclamations', 'anciennete', 'age',
                'nb_produits', 'derniere_activite_jours']

    X = df[features]
    y = df['churned']  # 1 si parti dans les 6 mois

    from sklearn.ensemble import RandomForestClassifier

    model = RandomForestClassifier(n_estimators=100)
    model.fit(X, y)

    df['proba_churn'] = model.predict_proba(X)[:, 1]

    # Clients à risque
    at_risk = df[df['proba_churn'] > 0.3]

    return model, at_risk
```

5.4 Modèles de Pricing

Tarification Basée sur le Risque

```python
def calculer_taux_ajuste_risque(df, cout_fonds, marge_cible):
    """Calcul du taux ajusté au risque"""
```

```python
    # Composantes du taux
    df['prime_risque'] = df['PD'] * df['LGD']
    df['cout_capital'] = 0.12 * df['rwa_poids']  # Coût du capital réglementaire
    df['frais_gestion'] = 0.01  # Frais fixes

    df['taux_minimum'] = (
        cout_fonds +
        df['prime_risque'] +
        df['cout_capital'] +
        df['frais_gestion'] +
        marge_cible
    )

    return df
```

Partie 6: Comparaison et Choix des Modèles

6.1 Tableau de Comparaison

Modèle	Avantages	Inconvénients	Usage Bancaire
Régression Logistique	Interprétable, rapide	Linéaire	Scoring crédit
Arbre de Décision	Très interprétable	Overfitting	Règles métier
Random Forest	Robuste, précis	Moins interprétable	Scoring, fraude
XGBoost	Très précis	Complexe	Compétitions, scoring
K-Means	Simple	Sensible initialisation	Segmentation
ARIMA	Bon pour séries	Stationnarité requise	Prévisions

6.2 Critères de Sélection

```
1. INTERPRÉTABILITÉ
   - Réglementaire: Obligation d'expliquer les décisions
   - Banque: Privilégier modèles interprétables (logistique, arbres)

2. PERFORMANCE
   - Classification: AUC, Gini, KS
   - Régression: RMSE, MAE, R²

3. ROBUSTESSE
   - Stabilité dans le temps
   - Performance sur données nouvelles

4. COÛT COMPUTATIONNEL
   - Temps d'entraînement
   - Temps de prédiction (scoring en temps réel)

5. CONTRAINTES RÉGLEMENTAIRES
   - Modèles validés par le régulateur
```

- Documentation requise

6.3 Workflow de Sélection

```python
def selectionner_modele(X_train, X_test, y_train, y_test):
    """Comparaison de plusieurs modèles"""

    from sklearn.linear_model import LogisticRegression
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
    from sklearn.metrics import roc_auc_score

    models = {
        'Logistic': LogisticRegression(),
        'DecisionTree': DecisionTreeClassifier(max_depth=5),
        'RandomForest': RandomForestClassifier(n_estimators=100),
        'GradientBoosting': GradientBoostingClassifier(n_estimators=100)
    }

    results = {}
    for name, model in models.items():
        model.fit(X_train, y_train)
        y_pred_proba = model.predict_proba(X_test)[:, 1]
        auc = roc_auc_score(y_test, y_pred_proba)
        results[name] = auc
        print(f"{name}: AUC = {auc:.3f}")

    best_model = max(results, key=results.get)
    print(f"\nMeilleur modèle: {best_model}")

    return results
```

Partie 7: Validation des Modèles

7.1 Techniques de Validation

Holdout (Train/Test Split):

```python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
```

Cross-Validation:

```python
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, X, y, cv=5, scoring='roc_auc')
print(f"AUC: {scores.mean():.3f} (+/- {scores.std():.3f})")
```

Validation Temporelle (Walk-Forward):

```python
from sklearn.model_selection import TimeSeriesSplit
```

```python
tscv = TimeSeriesSplit(n_splits=5)
for train_idx, test_idx in tscv.split(X):
    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]
    # Entraîner et évaluer
```

7.2 Métriques Spécifiques Bancaires

```python
def metriques_bancaires(y_true, y_pred_proba, y_pred):
    """Métriques spécifiques au secteur bancaire"""

    from sklearn.metrics import roc_auc_score, confusion_matrix

    # Gini = 2 × AUC - 1
    auc = roc_auc_score(y_true, y_pred_proba)
    gini = 2 * auc - 1

    # KS (Kolmogorov-Smirnov)
    from scipy.stats import ks_2samp
    good = y_pred_proba[y_true == 0]
    bad = y_pred_proba[y_true == 1]
    ks_stat, _ = ks_2samp(good, bad)

    # Taux de mauvaise classification
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()

    print(f"AUC: {auc:.3f}")
    print(f"Gini: {gini:.3f}")
    print(f"KS: {ks_stat:.3f}")
    print(f"Type I Error (FP rate): {fp/(fp+tn):.2%}")
    print(f"Type II Error (FN rate): {fn/(fn+tp):.2%}")
```

Questions d'Entretien

1. **Quelle est la différence entre classification et régression?** → Classification: prédire une catégorie; Régression: prédire une valeur continue
2. **Quand utiliser K-Means vs clustering hiérarchique?** → K-Means: grands datasets, clusters sphériques; Hiérarchique: petits datasets, exploration
3. **Pourquoi la régression logistique est-elle préférée en scoring bancaire?** → Interprétable (coefficients = odds ratios), exigence réglementaire
4. **Comment choisir le nombre de clusters en K-Means?** → Méthode du coude (inertie), silhouette score, métier
5. **Qu'est-ce que le Gini coefficient en scoring?** → Gini = 2 × AUC - 1, mesure le pouvoir discriminant du modèle
6. **Différence entre PD, LGD, EAD?** → PD: probabilité de défaut; LGD: perte si défaut; EAD: exposition au moment du défaut

---

Checklist Modélisation

```
□ Définir clairement l'objectif business
□ Choisir le type de modèle approprié
□ Préparer et nettoyer les données
□ Séparer train/test correctement (temporel si nécessaire)
□ Entraîner plusieurs modèles candidats
□ Évaluer avec les métriques appropriées
□ Valider par cross-validation
□ Interpréter les résultats
□ Documenter le modèle
□ Planifier le monitoring en production
```

Rappel final: Le choix du modèle doit équilibrer performance prédictive, interprétabilité et contraintes opérationnelles/réglementaires. En banque, un modèle moins performant mais explicable est souvent préféré.