Manuel de Préparation: Séries Temporelles et Prévisions

Introduction

Les séries temporelles sont des données collectées à intervalles réguliers dans le temps. Dans le secteur bancaire, elles sont essentielles pour la prévision des dépôts, la gestion de la liquidité, l'analyse des tendances de crédit, et la planification stratégique.

Partie 1: Concepts Fondamentaux

1.1 Définition

Série temporelle: Séquence d'observations ordonnées dans le temps.

$Y_t = f(t) + \varepsilon_t$

Où:
- Y_t = Valeur observée au temps t
- $f(t)$ = Composante systématique (tendance, saisonnalité)
- ε_t = Bruit aléatoire (résidu)

Exemples bancaires: - Dépôts journaliers/mensuels - Volume de transactions ATM - Taux de défaut trimestriel - Solde moyen par client

---

1.2 Composantes d'une Série Temporelle

$Y_t = T_t + S_t + C_t + I_t$ (Modèle Additif)
$Y_t = T_t \times S_t \times C_t \times I_t$ (Modèle Multiplicatif)

Où:
- T_t = Tendance (Trend)
- S_t = Saisonnalité (Seasonal)
- C_t = Cycle (Cyclical)
- I_t = Irrégulier/Bruit (Irregular)

Mnémotechnique: "TSCI"

```
T - Tendance: Direction long terme (hausse/baisse)
S - Saisonnalité: Pattern répétitif à période fixe
C - Cycle: Fluctuations à long terme (économiques)
I - Irrégulier: Variations aléatoires imprévisibles
```

1.3 Visualisation des Composantes

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose

# Créer une série temporelle bancaire
np.random.seed(42)
```

```
dates = pd.date_range(start='2020-01-01', periods=48, freq='M')

# Composantes
tendance = np.linspace(100, 180, 48)
saisonnalite = 20 * np.sin(2 * np.pi * np.arange(48) / 12)
bruit = np.random.normal(0, 5, 48)

# Dépôts mensuels (en millions HTG)
depots = tendance + saisonnalite + bruit

df = pd.DataFrame({'date': dates, 'depots_millions': depots})
df.set_index('date', inplace=True)

# Décomposition
decomposition = seasonal_decompose(df['depots_millions'], model='additive', period=12)

# Visualisation
fig, axes = plt.subplots(4, 1, figsize=(12, 10))
decomposition.observed.plot(ax=axes[0], title='Observé')
decomposition.trend.plot(ax=axes[1], title='Tendance')
decomposition.seasonal.plot(ax=axes[2], title='Saisonnalité')
decomposition.resid.plot(ax=axes[3], title='Résidus')
plt.tight_layout()
plt.show()
```

Partie 2: Stationnarité

2.1 Définition et Importance

Série stationnaire: Propriétés statistiques constantes dans le temps.

Conditions de stationnarité faible:
1. $E[Y_t] = \mu$ (moyenne constante)
2. $Var(Y_t) = \sigma^2$ (variance constante)
3. $Cov(Y_t, Y_{t+k}) = \gamma_k$ (covariance ne dépend que du lag k)

Importance: - La plupart des modèles (ARIMA) requièrent la stationnarité - Une série non stationnaire donne des régressions fallacieuses - Il faut transformer avant de modéliser

---

2.2 Tests de Stationnarité

Test ADF (Augmented Dickey-Fuller)

```
from statsmodels.tsa.stattools import adfuller

def test_stationnarite(series, nom="Série"):
    """Test ADF de stationnarité"""
    result = adfuller(series.dropna())

    print(f"\n=== Test ADF pour {nom} ===")
    print(f"Statistique ADF: {result[0]:.4f}")
    print(f"P-value: {result[1]:.4f}")
```

```
    print(f"Lags utilisés: {result[2]}")
    print(f"Observations: {result[3]}")
    print("Valeurs critiques:")
    for key, value in result[4].items():
        print(f"  {key}: {value:.4f}")

    if result[1] < 0.05:
        print(f"\n✓ CONCLUSION: {nom} est STATIONNAIRE (p < 0.05)")
        return True
    else:
        print(f"\n⚠ CONCLUSION: {nom} est NON STATIONNAIRE (p ≥ 0.05)")
        return False

# Test
test_stationnarite(df['depots_millions'], "Dépôts")
```

Interprétation: - H_0: La série a une racine unitaire (non stationnaire) - H_1: La série est stationnaire - Si p-value < 0.05 → Rejeter H_0 → Stationnaire

Test KPSS (Kwiatkowski-Phillips-Schmidt-Shin)

```
from statsmodels.tsa.stattools import kpss

def test_kpss(series, nom="Série"):
    """Test KPSS de stationnarité"""
    result = kpss(series.dropna(), regression='c')

    print(f"\n=== Test KPSS pour {nom} ===")
    print(f"Statistique KPSS: {result[0]:.4f}")
    print(f"P-value: {result[1]:.4f}")
    print("Valeurs critiques:")
    for key, value in result[3].items():
        print(f"  {key}: {value:.4f}")

    if result[1] > 0.05:
        print(f"\n✓ CONCLUSION: {nom} est STATIONNAIRE (p > 0.05)")
        return True
    else:
        print(f"\n⚠ CONCLUSION: {nom} est NON STATIONNAIRE (p ≤ 0.05)")
        return False

# Note: KPSS a l'hypothèse inverse de ADF
# H₀: Série stationnaire
# H₁: Série non stationnaire
```

Stratégie combinée:

ADF	KPSS	Conclusion
Stationnaire (p<0.05)	Stationnaire (p>0.05)	✓ Stationnaire
Non stationnaire (p≥0.05)	Non stationnaire (p≤0.05)	✗ Non stationnaire
Mixte	Mixte	Différencier et retester

2.3 Transformation pour Stationnarité

```
# 1. Différenciation (retirer la tendance)
df['depots_diff1'] = df['depots_millions'].diff()

# 2. Différenciation saisonnière (retirer la saisonnalité)
df['depots_diff12'] = df['depots_millions'].diff(12)

# 3. Transformation logarithmique (stabiliser la variance)
df['depots_log'] = np.log(df['depots_millions'])

# 4. Différence du log (rendements)
df['depots_log_diff'] = np.log(df['depots_millions']).diff()

# Vérifier la stationnarité après transformation
test_stationnarite(df['depots_diff1'].dropna(), "Dépôts différenciés")
```

Partie 3: Autocorrélation

3.1 ACF et PACF

```
ACF (Autocorrelation Function):
- Corrélation entre Yₜ et Yₜ₋ₖ pour différents lags k
- Inclut les effets indirects

PACF (Partial Autocorrelation Function):
- Corrélation entre Yₜ et Yₜ₋ₖ APRÈS avoir contrôlé pour les lags intermédiaires
- Effets directs uniquement
```

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

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

# ACF
plot_acf(df['depots_millions'].dropna(), lags=24, ax=axes[0])
axes[0].set_title('ACF - Autocorrélation')

# PACF
plot_pacf(df['depots_millions'].dropna(), lags=24, ax=axes[1])
axes[1].set_title('PACF - Autocorrélation Partielle')

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

3.2 Interprétation ACF/PACF pour ARIMA

Pattern ACF	Pattern PACF	Modèle suggéré
Décroissance exponentielle	Coupure nette au lag p	AR(p)
Coupure nette au lag q	Décroissance exponentielle	MA(q)
Décroissance des deux	Décroissance des deux	ARMA(p,q)

Partie 4: Modèles de Prévision

4.1 Moyennes Mobiles

```
# Moyenne mobile simple
df['MM_3'] = df['depots_millions'].rolling(window=3).mean()
df['MM_12'] = df['depots_millions'].rolling(window=12).mean()

# Moyenne mobile exponentielle
df['EMA_3'] = df['depots_millions'].ewm(span=3, adjust=False).mean()

# Visualisation
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['depots_millions'], label='Observé', alpha=0.7)
plt.plot(df.index, df['MM_12'], label='MM(12)', linewidth=2)
plt.plot(df.index, df['EMA_3'], label='EMA(3)', linewidth=2)
plt.legend()
plt.title('Dépôts et Moyennes Mobiles')
plt.show()
```

4.2 Lissage Exponentiel Simple

```
from statsmodels.tsa.holtwinters import SimpleExpSmoothing

# Modèle
model_ses = SimpleExpSmoothing(df['depots_millions']).fit(smoothing_level=0.3)

# Prévision
forecast_ses = model_ses.forecast(12)
print("Prévisions SES (12 mois):", forecast_ses.values)

# Paramètre alpha:
# - alpha proche de 0: Lissage fort (poids aux anciennes observations)
# - alpha proche de 1: Peu de lissage (poids aux observations récentes)
```

4.3 Holt-Winters (Triple Lissage Exponentiel)

Composantes: - α (alpha): Niveau - β (beta): Tendance - γ (gamma): Saisonnalité

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Modèle Holt-Winters avec saisonnalité additive
model_hw = ExponentialSmoothing(
```

```
    df['depots_millions'],
    trend='add',           # Tendance additive
    seasonal='add',        # Saisonnalité additive
    seasonal_periods=12    # Période de 12 mois
).fit()

# Prévision 12 mois
forecast_hw = model_hw.forecast(12)

# Visualisation
fig, ax = plt.subplots(figsize=(12, 6))
df['depots_millions'].plot(ax=ax, label='Historique')
forecast_hw.plot(ax=ax, label='Prévision Holt-Winters', style='--')
plt.legend()
plt.title('Prévision des Dépôts - Holt-Winters')
plt.show()

print(f"\nPrévisions pour les 12 prochains mois:")
print(forecast_hw)
```

4.4 ARIMA (AutoRegressive Integrated Moving Average)

Notation: ARIMA(p, d, q) - p = Ordre autorégressif (AR) - d = Ordre de différenciation (I) - q = Ordre de moyenne mobile (MA)

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

# Déterminer les ordres avec ACF/PACF ou auto_arima
# Exemple: ARIMA(1, 1, 1)
model_arima = ARIMA(df['depots_millions'], order=(1, 1, 1))
results_arima = model_arima.fit()

print(results_arima.summary())

# Prévision
forecast_arima = results_arima.forecast(steps=12)
print("\nPrévisions ARIMA:")
print(forecast_arima)

# Intervalles de confiance
forecast_ci = results_arima.get_forecast(steps=12)
conf_int = forecast_ci.conf_int()
print("\nIntervalles de confiance à 95%:")
print(conf_int)
```

4.5 SARIMA (Seasonal ARIMA)

Notation: SARIMA(p, d, q)(P, D, Q, s) - (p, d, q): Partie non saisonnière - (P, D, Q, s): Partie saisonnière, s = période

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

```python
# SARIMA avec saisonnalité mensuelle (s=12)
model_sarima = SARIMAX(
    df['depots_millions'],
    order=(1, 1, 1),
    seasonal_order=(1, 1, 1, 12)
)
results_sarima = model_sarima.fit(disp=False)

print(results_sarima.summary())

# Prévision avec intervalles
forecast_sarima = results_sarima.get_forecast(steps=12)
forecast_mean = forecast_sarima.predicted_mean
forecast_ci = forecast_sarima.conf_int()

# Visualisation
fig, ax = plt.subplots(figsize=(12, 6))
df['depots_millions'].plot(ax=ax, label='Historique')
forecast_mean.plot(ax=ax, label='Prévision SARIMA', style='--')
ax.fill_between(forecast_ci.index,
                forecast_ci.iloc[:, 0],
                forecast_ci.iloc[:, 1],
                alpha=0.2)
plt.legend()
plt.title('Prévision des Dépôts - SARIMA')
plt.show()
```

4.6 Sélection Automatique avec Auto-ARIMA

```python
# Installation: pip install pmdarima
from pmdarima import auto_arima

# Recherche automatique des meilleurs paramètres
model_auto = auto_arima(
    df['depots_millions'],
    seasonal=True,
    m=12,  # Période saisonnière
    stepwise=True,
    suppress_warnings=True,
    trace=True  # Afficher la progression
)

print(model_auto.summary())

# Prévision
forecast_auto = model_auto.predict(n_periods=12)
print("\nPrévisions Auto-ARIMA:")
print(forecast_auto)
```

4.7 Prophet (Facebook)

```python
# Installation: pip install prophet
from prophet import Prophet

# Préparer les données au format Prophet
df_prophet = df.reset_index()
df_prophet.columns = ['ds', 'y']

# Modèle Prophet
model_prophet = Prophet(
    yearly_seasonality=True,
    weekly_seasonality=False,
    daily_seasonality=False
)
model_prophet.fit(df_prophet)

# Créer les dates futures
future = model_prophet.make_future_dataframe(periods=12, freq='M')

# Prévision
forecast_prophet = model_prophet.predict(future)

# Visualisation
fig = model_prophet.plot(forecast_prophet)
plt.title('Prévision Prophet')
plt.show()

# Composantes
fig2 = model_prophet.plot_components(forecast_prophet)
plt.show()
```

Partie 5: Évaluation des Modèles

5.1 Métriques d'Évaluation

```python
from sklearn.metrics import mean_absolute_error, mean_squared_error

def evaluer_prevision(y_true, y_pred):
    """Métriques d'évaluation pour séries temporelles"""

    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100

    print("=== MÉTRIQUES D'ÉVALUATION ===")
    print(f"MAE (Mean Absolute Error): {mae:.2f}")
    print(f"MSE (Mean Squared Error): {mse:.2f}")
    print(f"RMSE (Root MSE): {rmse:.2f}")
    print(f"MAPE (Mean Absolute % Error): {mape:.2f}%")

    return {'MAE': mae, 'MSE': mse, 'RMSE': rmse, 'MAPE': mape}
```

Tableau des métriques:

Métrique	Formule	Interprétation
MAE	$\Sigma\|Y_i - \hat{Y}_i\| / n$	Erreur moyenne absolue
MSE	$\Sigma(Y_i - \hat{Y}_i)^2 / n$	Pénalise les grandes erreurs
RMSE	$\sqrt{MSE}$	Même unité que Y
MAPE	$\Sigma\|(Y_i - \hat{Y}_i)/Y_i\| \times 100 / n$	Erreur en pourcentage

---

5.2 Validation Croisée pour Séries Temporelles

```python
from sklearn.model_selection import TimeSeriesSplit

# Important: Ne pas mélanger les données!
# Utiliser TimeSeriesSplit

tscv = TimeSeriesSplit(n_splits=5)

errors = []
for train_idx, test_idx in tscv.split(df['depots_millions']):
    train = df['depots_millions'].iloc[train_idx]
    test = df['depots_millions'].iloc[test_idx]

    # Entraîner le modèle
    model = ExponentialSmoothing(
        train,
        trend='add',
        seasonal='add',
        seasonal_periods=12
    ).fit()

    # Prédire
    pred = model.forecast(len(test))

    # Calculer l'erreur
    mape = np.mean(np.abs((test.values - pred.values) / test.values)) * 100
    errors.append(mape)
    print(f"Fold MAPE: {mape:.2f}%")

print(f"\nMAPE moyen: {np.mean(errors):.2f}%")
```

5.3 Critères de Sélection de Modèle

```
AIC (Akaike Information Criterion):
AIC = 2k - 2ln(L)

BIC (Bayesian Information Criterion):
BIC = k×ln(n) - 2ln(L)

Où:
```

```
- k = nombre de paramètres
- n = nombre d'observations
- L = vraisemblance

Règle: Plus bas = Meilleur
```

```python
# Comparer plusieurs modèles
models_comparison = []

# Modèle 1: ARIMA(1,1,1)
m1 = ARIMA(df['depots_millions'], order=(1,1,1)).fit()
models_comparison.append({
    'Modèle': 'ARIMA(1,1,1)',
    'AIC': m1.aic,
    'BIC': m1.bic
})

# Modèle 2: ARIMA(2,1,1)
m2 = ARIMA(df['depots_millions'], order=(2,1,1)).fit()
models_comparison.append({
    'Modèle': 'ARIMA(2,1,1)',
    'AIC': m2.aic,
    'BIC': m2.bic
})

# Modèle 3: ARIMA(1,1,2)
m3 = ARIMA(df['depots_millions'], order=(1,1,2)).fit()
models_comparison.append({
    'Modèle': 'ARIMA(1,1,2)',
    'AIC': m3.aic,
    'BIC': m3.bic
})

comparison_df = pd.DataFrame(models_comparison).sort_values('AIC')
print(comparison_df)
```

Partie 6: Applications Bancaires

6.1 Cas 1: Prévision de Liquidité

```python
"""
Objectif: Prévoir les retraits nets pour gérer la liquidité
"""
import pandas as pd
import numpy as np
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Données de retraits nets (en millions HTG)
np.random.seed(42)
dates = pd.date_range(start='2022-01-01', periods=36, freq='M')

# Pattern: Pics en fin de mois, augmentation en décembre
retraits = 50 + \
```

```
            10 * np.sin(2 * np.pi * np.arange(36) / 12) + \
            np.linspace(0, 20, 36) + \
            np.random.normal(0, 5, 36)

df_liquidite = pd.DataFrame({
    'date': dates,
    'retraits_nets': retraits
}).set_index('date')

# Modèle Holt-Winters
model_liq = ExponentialSmoothing(
    df_liquidite['retraits_nets'],
    trend='add',
    seasonal='add',
    seasonal_periods=12
).fit()

# Prévision 6 mois
prevision_liq = model_liq.forecast(6)

print("=== PRÉVISION DE LIQUIDITÉ ===")
print("\nRetraits prévus pour les 6 prochains mois:")
for date, val in prevision_liq.items():
    print(f"  {date.strftime('%Y-%m')}: {val:.1f} millions HTG")

# Recommandation
max_retrait = prevision_liq.max()
print(f"\n⚠ Pic prévu: {max_retrait:.1f} millions HTG")
print(f"Recommandation: Maintenir une réserve de {max_retrait * 1.2:.1f} millions HTG")
```

6.2 Cas 2: Prévision des Défauts de Paiement

```
"""
Objectif: Prévoir le taux de défaut trimestriel
"""
from statsmodels.tsa.arima.model import ARIMA

# Données de taux de défaut (%)
np.random.seed(42)
dates = pd.date_range(start='2018-01-01', periods=24, freq='Q')

# Taux de défaut avec cycle économique
taux_defaut = 3.5 + \
              1.5 * np.sin(2 * np.pi * np.arange(24) / 8) + \
              np.random.normal(0, 0.3, 24)

df_defaut = pd.DataFrame({
    'date': dates,
    'taux_defaut': taux_defaut
}).set_index('date')

# Modèle ARIMA
```

```python
model_defaut = ARIMA(df_defaut['taux_defaut'], order=(2, 0, 1)).fit()

# Prévision 4 trimestres
prevision_defaut = model_defaut.get_forecast(4)
mean_forecast = prevision_defaut.predicted_mean
conf_int = prevision_defaut.conf_int()

print("=== PRÉVISION DU TAUX DE DÉFAUT ===")
print("\nTaux prévu (avec IC 95%):")
for i, (idx, val) in enumerate(mean_forecast.items()):
    lower = conf_int.iloc[i, 0]
    upper = conf_int.iloc[i, 1]
    print(f"  {idx.strftime('%Y-Q%q')}: {val:.2f}% [{lower:.2f}% - {upper:.2f}%]")

# Alerte si taux > seuil
seuil_alerte = 5.0
if mean_forecast.max() > seuil_alerte:
    print(f"\n ALERTE: Taux de défaut prévu > {seuil_alerte}%")
    print("   Actions recommandées:")
    print("   - Renforcer les critères d'octroi")
    print("   - Augmenter les provisions")
```

6.3 Cas 3: Prévision du Volume de Transactions

```python
"""
Objectif: Prévoir le volume de transactions pour dimensionner l'infrastructure
"""
from prophet import Prophet

# Données de transactions journalières
np.random.seed(42)
dates = pd.date_range(start='2023-01-01', periods=365, freq='D')

# Pattern: Weekend faible, pic en fin de mois
base = 10000
weekly_pattern = np.array([1.2, 1.1, 1.0, 1.0, 1.3, 0.6, 0.5])  # Lun-Dim
day_of_week = np.array([d.weekday() for d in dates])
weekly_effect = np.array([weekly_pattern[d] for d in day_of_week])

# Effet fin de mois
day_of_month = np.array([d.day for d in dates])
end_month_effect = np.where(day_of_month > 25, 1.4, 1.0)

transactions = base * weekly_effect * end_month_effect + np.random.normal(0, 500, 365)

df_trans = pd.DataFrame({
    'ds': dates,
    'y': transactions
})

# Modèle Prophet
model_trans = Prophet(
```

```
    yearly_seasonality=True,
    weekly_seasonality=True,
    daily_seasonality=False
)
model_trans.add_seasonality(name='monthly', period=30.5, fourier_order=5)
model_trans.fit(df_trans)

# Prévision 30 jours
future = model_trans.make_future_dataframe(periods=30)
forecast_trans = model_trans.predict(future)

# Identifier les pics
peaks = forecast_trans.tail(30).nlargest(5, 'yhat')
print("=== PRÉVISION DU VOLUME DE TRANSACTIONS ===")
print("\n5 jours de pic prévus:")
for _, row in peaks.iterrows():
    print(f"  {row['ds'].strftime('%Y-%m-%d')}: {row['yhat']:.0f} transactions")

print(f"\nCapacité recommandée: {forecast_trans.tail(30)['yhat'].max() * 1.3:.0f} trans/jo
```

Partie 7: Formules Essentielles

7.1 Tableau Récapitulatif

Modèle	Formule/Structure	Usage
SES	$\hat{Y}_{t+1} = \alpha Y_t + (1-\alpha)\hat{Y}_t$	Série sans tendance ni saison
Holt	Niveau + Tendance	Série avec tendance
Holt-Winters	Niveau + Tendance + Saison	Série complète
AR(p)	$Y_t = \varphi_1 Y_{t-1} + ... + \varphi_p Y_{t-p} + \varepsilon_t$	Autocorrélation
MA(q)	$Y_t = \varepsilon_t + \theta_1 \varepsilon_{t-1} + ... + \theta q \varepsilon_{t-q}$	Moyenne mobile
ARIMA(p,d,q)	AR + I + MA	Série non stationnaire
SARIMA	ARIMA + composante saisonnière	Série avec saison

7.2 Mnémotechniques

“SADIM” pour les étapes:

```
S - Stationnarité: Tester (ADF, KPSS)
A - ACF/PACF: Analyser les corrélations
D - Déterminer: Choisir p, d, q
I - Implémenter: Ajuster le modèle
M - Mesurer: Évaluer (AIC, RMSE, MAPE)
```

“ARIMA” pour retenir les composantes:

```
A - AutoRegressive: Passé de Y
R - ... (R de ARIMA)
I - Integrated: Différenciation
M - Moving Average: Passé des erreurs
A - ... (A de ARIMA)
```

Partie 8: Checklist Analyse de Séries Temporelles

```
□ 1. Visualiser la série (tendance, saisonnalité?)
□ 2. Décomposer (additive ou multiplicative?)
□ 3. Tester la stationnarité (ADF, KPSS)
□ 4. Transformer si nécessaire (diff, log)
□ 5. Analyser ACF/PACF
□ 6. Choisir le modèle (Holt-Winters, ARIMA, Prophet)
□ 7. Ajuster et diagnostiquer les résidus
□ 8. Valider (AIC, BIC, cross-validation temporelle)
□ 9. Prévoir avec intervalles de confiance
□ 10. Interpréter pour le business
```

Résumé Express: Questions Probables

1. **“Comment tester si une série est stationnaire?”** → Test ADF: $p < 0.05$ = stationnaire
2. **“Quelle est la différence entre ARIMA et SARIMA?”** → SARIMA inclut une composante saisonnière (P, D, Q, s)
3. **“Comment choisir entre plusieurs modèles?”** → AIC/BIC plus bas = meilleur modèle
4. **“Quelles sont les composantes d’une série temporelle?”** → TSCI: Tendance, Saisonnalité, Cycle, Irrégulier
5. **“Comment prévoir les dépôts du mois prochain?”** → Holt-Winters ou SARIMA avec saisonnalité mensuelle
6. **“Qu’est-ce que le MAPE?”** → Mean Absolute Percentage Error - erreur moyenne en %

Document préparé pour l’examen Data Analyst - UniBank Haiti Séries Temporelles: Maîtriser la dimension temps