Manuel de Préparation: Machine Learning pour Data Analyst Bancaire

Introduction

Le Machine Learning (ML) est un sous-domaine de l'intelligence artificielle qui permet aux systèmes d'apprendre à partir des données sans être explicitement programmés. Pour un Data Analyst en banque, comprendre les fondamentaux du ML est essentiel pour développer des modèles prédictifs, détecter les fraudes, segmenter les clients et optimiser les processus.

Partie 1: Fondamentaux du Machine Learning

1.1 Définitions Clés

Machine Learning:

```
Ensemble de techniques permettant à un algorithme d'améliorer
ses performances sur une tâche donnée grâce à l'expérience (données).
```

Modèle:

```
Représentation mathématique apprise à partir des données,
capable de faire des prédictions sur de nouvelles observations.
```

Feature (Variable explicative):

```
Caractéristique utilisée comme input du modèle.
Exemples: âge, revenu, nombre de transactions
```

Target (Variable cible):

```
Ce qu'on cherche à prédire.
Exemples: défaut (oui/non), montant de fraude
```

1.2 Types d'Apprentissage

```
MACHINE LEARNING
│
├── SUPERVISÉ
│   ├── Classification (catégories)
│   │   └── Défaut, Fraude, Segment
│   └── Régression (valeurs continues)
│       └── Montant, Score, Durée
│
├── NON SUPERVISÉ
│   ├── Clustering (groupes)
│   │   └── Segmentation clients
│   ├── Réduction de dimension
│   │   └── PCA, t-SNE
│   └── Détection d'anomalies
│       └── Fraude, Outliers
│
├── SEMI-SUPERVISÉ
│   └── Peu de labels + beaucoup de données non labelées
│
```

```
└── RENFORCEMENT
    └── Agent apprend par essai/erreur (moins courant en banque)
```

1.3 Pipeline ML Standard

```
1. DÉFINITION DU PROBLÈME
   └── Objectif business, métriques de succès

2. COLLECTE DES DONNÉES
   └── Sources, extraction, qualité

3. PRÉPARATION DES DONNÉES
   ├── Nettoyage
   ├── Feature engineering
   └── Train/Test split

4. MODÉLISATION
   ├── Sélection d'algorithmes
   ├── Entraînement
   └── Tuning hyperparamètres

5. ÉVALUATION
   ├── Métriques
   └── Validation croisée

6. DÉPLOIEMENT
   ├── Mise en production
   └── Monitoring
```

Partie 2: Préparation des Données

2.1 Nettoyage des Données

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

def nettoyer_donnees(df):
    """Pipeline de nettoyage standard"""

    # 1. Valeurs manquantes
    print("Valeurs manquantes par colonne:")
    print(df.isnull().sum())

    # Stratégies de gestion
    # - Numériques: médiane (robuste aux outliers)
    for col in df.select_dtypes(include=[np.number]).columns:
        df[col].fillna(df[col].median(), inplace=True)

    # - Catégorielles: mode ou catégorie "Unknown"
    for col in df.select_dtypes(include=['object']).columns:
        df[col].fillna(df[col].mode()[0], inplace=True)
```

```
    # 2. Doublons
    duplicates = df.duplicated().sum()
    print(f"\nDoublons: {duplicates}")
    df.drop_duplicates(inplace=True)

    # 3. Types de données
    # Convertir les dates
    if 'date' in df.columns:
        df['date'] = pd.to_datetime(df['date'])

    return df
```

2.2 Feature Engineering

```
def creer_features_bancaires(df):
    """Création de features pour modèles bancaires"""

    # Ratios financiers
    df['ratio_dette_revenu'] = df['dette_totale'] / df['revenu_mensuel']
    df['ratio_epargne'] = df['solde_epargne'] / df['revenu_annuel']

    # Features temporelles
    df['anciennete_jours'] = (pd.Timestamp.now() - df['date_ouverture']).dt.days
    df['anciennete_annees'] = df['anciennete_jours'] / 365

    # Features d'activité
    df['tx_par_mois'] = df['nb_transactions'] / df['anciennete_mois']
    df['montant_moyen_tx'] = df['volume_total'] / df['nb_transactions']

    # Features de comportement
    df['ratio_credit_debit'] = df['total_credits'] / (df['total_debits'] + 1)
    df['variation_solde'] = df['solde_actuel'] / (df['solde_6m_avant'] + 1) - 1

    # Binning (discrétisation)
    df['tranche_age'] = pd.cut(df['age'],
                               bins=[0, 25, 35, 50, 65, 100],
                               labels=['Jeune', 'Adulte', 'Mature', 'Senior', 'Retraité'])

    # Indicateurs binaires
    df['est_proprietaire'] = (df['type_logement'] == 'Propriétaire').astype(int)
    df['a_credit_actif'] = (df['nb_credits'] > 0).astype(int)

    return df
```

2.3 Encodage des Variables

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, OrdinalEncoder

def encoder_variables(df, target_col=None):
    """Encodage des variables catégorielles"""

    df_encoded = df.copy()

    # Variables nominales: One-Hot Encoding
```

```
    nominal_cols = ['type_compte', 'region', 'canal_acquisition']
    df_encoded = pd.get_dummies(df_encoded, columns=nominal_cols, drop_first=True)

    # Variables ordinales: Ordinal Encoding
    ordinal_mappings = {
        'niveau_risque': ['Faible', 'Moyen', 'Élevé', 'Très élevé'],
        'education': ['Primaire', 'Secondaire', 'Universitaire', 'Post-grad']
    }

    for col, order in ordinal_mappings.items():
        if col in df_encoded.columns:
            mapping = {val: idx for idx, val in enumerate(order)}
            df_encoded[col] = df_encoded[col].map(mapping)

    return df_encoded
```

2.4 Normalisation et Standardisation

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler

def normaliser_features(X_train, X_test, method='standard'):
    """Normalisation des features numériques"""

    scalers = {
        'standard': StandardScaler(),      # z-score: (x - μ) / σ
        'minmax': MinMaxScaler(),          # [0, 1]
        'robust': RobustScaler()           # Robuste aux outliers
    }

    scaler = scalers[method]

    # IMPORTANT: Fit uniquement sur train
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    return X_train_scaled, X_test_scaled, scaler
```

2.5 Split Train/Test

```
from sklearn.model_selection import train_test_split, TimeSeriesSplit

def split_donnees(df, target_col, test_size=0.3, temporal=False):
    """Séparation train/test"""

    X = df.drop(columns=[target_col])
    y = df[target_col]

    if temporal:
        # Pour données temporelles: split chronologique
        split_idx = int(len(df) * (1 - test_size))
        X_train, X_test = X[:split_idx], X[split_idx:]
        y_train, y_test = y[:split_idx], y[split_idx:]
    else:
        # Split aléatoire stratifié
```

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

    print(f"Train: {len(X_train)} échantillons")
    print(f"Test: {len(X_test)} échantillons")
    print(f"Ratio positifs train: {y_train.mean():.2%}")
    print(f"Ratio positifs test: {y_test.mean():.2%}")

    return X_train, X_test, y_train, y_test
```

Partie 3: Algorithmes de Classification

3.1 Régression Logistique

Concept:

Modèle linéaire qui prédit la probabilité d'appartenance à une classe.
Utilise la fonction sigmoïde pour transformer une combinaison linéaire en probabilité.

P(Y=1|X) = 1 / (1 + e^(-z))
où $z = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n$

Implémentation:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, roc_auc_score

def modele_logistique(X_train, X_test, y_train, y_test):
    """Régression logistique avec interprétation"""

    model = LogisticRegression(random_state=42, max_iter=1000)
    model.fit(X_train, y_train)

    # Prédictions
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1]

    # Évaluation
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
    print(f"\nAUC-ROC: {roc_auc_score(y_test, y_proba):.3f}")

    # Interprétation des coefficients
    print("\nCoefficients (Odds Ratios):")
    for feature, coef in zip(X_train.columns, model.coef_[0]):
        odds_ratio = np.exp(coef)
        print(f"  {feature}: OR = {odds_ratio:.3f}")

    return model
```

Avantages: - Très interprétable (coefficients = odds ratios) - Rapide à entraîner - Fournit des probabilités calibrées - Exigé par les régulateurs bancaires

Inconvénients: - Suppose une relation linéaire - Sensible aux outliers - Moins performant sur relations complexes

---

3.2 Arbres de Décision

Concept:

```
Modèle qui apprend des règles de décision hiérarchiques.
Chaque nœud représente un test sur une feature.
Chaque feuille représente une classe/prédiction.
```

Implémentation:

```python
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt

def modele_arbre(X_train, X_test, y_train, y_test):
    """Arbre de décision avec visualisation"""

    model = DecisionTreeClassifier(
        max_depth=5,
        min_samples_leaf=50,
        random_state=42
    )
    model.fit(X_train, y_train)

    # Prédictions
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1]

    # Visualisation
    plt.figure(figsize=(20, 10))
    plot_tree(model, feature_names=X_train.columns,
              class_names=['Non-défaut', 'Défaut'],
              filled=True, rounded=True)
    plt.savefig('arbre_decision.png', dpi=150, bbox_inches='tight')
    plt.show()

    # Importance des features
    importance = pd.DataFrame({
        'feature': X_train.columns,
        'importance': model.feature_importances_
    }).sort_values('importance', ascending=False)

    print("\nImportance des features:")
    print(importance.head(10))

    return model
```

Avantages: - Très interprétable (règles explicites) - Gère variables mixtes - Pas de normalisation requise

Inconvénients: - Tendance à l'overfitting - Instable (petites variations = arbres différents) - Moins performant que les ensembles

---

3.3 Random Forest

Concept:

```
Ensemble d'arbres de décision entraînés sur des sous-échantillons.
Chaque arbre vote, la classe majoritaire gagne.
Réduit l'overfitting par bagging (bootstrap aggregating).
```

Implémentation:

```python
from sklearn.ensemble import RandomForestClassifier

def modele_random_forest(X_train, X_test, y_train, y_test):
    """Random Forest avec optimisation"""

    model = RandomForestClassifier(
        n_estimators=100,      # Nombre d'arbres
        max_depth=10,          # Profondeur max
        min_samples_leaf=20,   # Échantillons min par feuille
        random_state=42,
        n_jobs=-1              # Parallélisation
    )
    model.fit(X_train, y_train)

    # Prédictions
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1]

    # Importance des features
    importance = pd.DataFrame({
        'feature': X_train.columns,
        'importance': model.feature_importances_
    }).sort_values('importance', ascending=False)

    print("Top 10 features:")
    print(importance.head(10))

    return model
```

3.4 Gradient Boosting (XGBoost, LightGBM)

Concept:

```
Ensemble d'arbres construits séquentiellement.
Chaque nouvel arbre corrige les erreurs des précédents.
Optimise une fonction de perte par descente de gradient.
```

Implémentation XGBoost:

```python
import xgboost as xgb

def modele_xgboost(X_train, X_test, y_train, y_test):
    """XGBoost avec early stopping"""
```

```
    model = xgb.XGBClassifier(
        n_estimators=500,
        max_depth=5,
        learning_rate=0.1,
        subsample=0.8,
        colsample_bytree=0.8,
        eval_metric='auc',
        early_stopping_rounds=50,
        random_state=42
    )

    model.fit(
        X_train, y_train,
        eval_set=[(X_test, y_test)],
        verbose=False
    )

    y_proba = model.predict_proba(X_test)[:, 1]
    print(f"AUC: {roc_auc_score(y_test, y_proba):.3f}")

    # Importance
    xgb.plot_importance(model, max_num_features=15)
    plt.show()

    return model
```

Comparaison des méthodes d'ensemble:

Aspect	Random Forest	XGBoost
Construction	Parallèle (bagging)	Séquentielle (boosting)
Vitesse	Plus rapide	Plus lent
Performance	Bonne	Souvent meilleure
Overfitting	Résistant	Plus sensible
Tuning	Moins sensible	Crucial

Partie 4: Algorithmes de Régression

4.1 Régression Linéaire

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso

def modele_regression(X_train, X_test, y_train, y_test):
    """Régression linéaire avec régularisation"""

    models = {
        'Linear': LinearRegression(),
        'Ridge': Ridge(alpha=1.0),   # L2 regularization
        'Lasso': Lasso(alpha=0.1)    # L1 regularization
    }
```

```
    results = {}
    for name, model in models.items():
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

        rmse = np.sqrt(mean_squared_error(y_test, y_pred))
        r2 = r2_score(y_test, y_pred)

        results[name] = {'RMSE': rmse, 'R²': r2}
        print(f"{name}: RMSE = {rmse:.2f}, R² = {r2:.3f}")

    return results
```

4.2 Régression avec Arbres

```
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

def modele_regression_ensemble(X_train, X_test, y_train, y_test):
    """Régression avec méthodes d'ensemble"""

    models = {
        'RandomForest': RandomForestRegressor(n_estimators=100, max_depth=10),
        'GradientBoosting': GradientBoostingRegressor(n_estimators=100, max_depth=5)
    }

    for name, model in models.items():
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

        mae = mean_absolute_error(y_test, y_pred)
        print(f"{name}: MAE = {mae:,.0f}")
```

Partie 5: Algorithmes Non Supervisés

5.1 K-Means Clustering

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

def clustering_clients(df, features, n_clusters=5):
    """Segmentation clients par K-Means"""

    X = df[features]

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

    # Déterminer le nombre optimal de clusters
    inertias = []
    silhouettes = []
```

```
    for k in range(2, 11):
        kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
        kmeans.fit(X_scaled)
        inertias.append(kmeans.inertia_)
        silhouettes.append(silhouette_score(X_scaled, kmeans.labels_))

    # Visualisation méthode du coude
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    axes[0].plot(range(2, 11), inertias, 'bo-')
    axes[0].set_xlabel('Nombre de clusters')
    axes[0].set_ylabel('Inertie')
    axes[0].set_title('Méthode du Coude')

    axes[1].plot(range(2, 11), silhouettes, 'ro-')
    axes[1].set_xlabel('Nombre de clusters')
    axes[1].set_ylabel('Silhouette Score')
    axes[1].set_title('Score Silhouette')

    plt.show()

    # Modèle final
    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
    df['cluster'] = kmeans.fit_predict(X_scaled)

    # Profil des clusters
    profil = df.groupby('cluster')[features].mean()
    print("\nProfil moyen par cluster:")
    print(profil)

    return df, kmeans
```

5.2 Détection d'Anomalies

```
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor

def detecter_anomalies(df, features, contamination=0.01):
    """Détection d'anomalies multi-méthodes"""

    X = df[features]
    X_scaled = StandardScaler().fit_transform(X)

    # Isolation Forest
    iso = IsolationForest(contamination=contamination, random_state=42)
    df['anomalie_iso'] = iso.fit_predict(X_scaled)

    # Local Outlier Factor
    lof = LocalOutlierFactor(n_neighbors=20, contamination=contamination)
    df['anomalie_lof'] = lof.fit_predict(X_scaled)

    # Consensus: anomalie si détectée par les deux
    df['anomalie'] = ((df['anomalie_iso'] == -1) &
                      (df['anomalie_lof'] == -1)).astype(int)
```

```
    n_anomalies = df['anomalie'].sum()
    print(f"Anomalies détectées: {n_anomalies} ({n_anomalies/len(df):.2%})")

    return df[df['anomalie'] == 1]
```

Partie 6: Évaluation des Modèles

6.1 Métriques de Classification

```
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, confusion_matrix, classification_report,
    precision_recall_curve, roc_curve
)

def evaluer_classification(y_true, y_pred, y_proba):
    """Évaluation complète d'un modèle de classification"""

    print("=== MÉTRIQUES DE CLASSIFICATION ===\n")

    # Métriques de base
    print(f"Accuracy:  {accuracy_score(y_true, y_pred):.3f}")
    print(f"Precision: {precision_score(y_true, y_pred):.3f}")
    print(f"Recall:    {recall_score(y_true, y_pred):.3f}")
    print(f"F1-Score:  {f1_score(y_true, y_pred):.3f}")
    print(f"AUC-ROC:   {roc_auc_score(y_true, y_proba):.3f}")

    # Gini (utilisé en banque)
    gini = 2 * roc_auc_score(y_true, y_proba) - 1
    print(f"Gini:      {gini:.3f}")

    # Matrice de confusion
    print("\nMatrice de confusion:")
    cm = confusion_matrix(y_true, y_pred)
    print(pd.DataFrame(cm,
                       index=['Réel: 0', 'Réel: 1'],
                       columns=['Prédit: 0', 'Prédit: 1']))

    # Courbes ROC et Precision-Recall
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    # ROC
    fpr, tpr, _ = roc_curve(y_true, y_proba)
    axes[0].plot(fpr, tpr, 'b-', label=f'AUC = {roc_auc_score(y_true, y_proba):.3f}')
    axes[0].plot([0, 1], [0, 1], 'k--')
    axes[0].set_xlabel('Taux de Faux Positifs')
    axes[0].set_ylabel('Taux de Vrais Positifs')
    axes[0].set_title('Courbe ROC')
    axes[0].legend()

    # Precision-Recall
```

```
    precision, recall, _ = precision_recall_curve(y_true, y_proba)
    axes[1].plot(recall, precision, 'r-')
    axes[1].set_xlabel('Recall')
    axes[1].set_ylabel('Precision')
    axes[1].set_title('Courbe Precision-Recall')

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

6.2 Métriques de Régression

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

def evaluer_regression(y_true, y_pred):
    """Évaluation d'un modèle de régression"""

    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_true, y_pred)
    mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100

    print("=== MÉTRIQUES DE RÉGRESSION ===\n")
    print(f"MAE:  {mae:,.2f}")
    print(f"RMSE: {rmse:,.2f}")
    print(f"R²:   {r2:.3f}")
    print(f"MAPE: {mape:.1f}%")

    # Graphique résidus
    residus = y_true - y_pred

    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    axes[0].scatter(y_pred, residus, alpha=0.5)
    axes[0].axhline(0, color='red', linestyle='--')
    axes[0].set_xlabel('Prédictions')
    axes[0].set_ylabel('Résidus')
    axes[0].set_title('Résidus vs Prédictions')

    axes[1].hist(residus, bins=50, edgecolor='black')
    axes[1].set_xlabel('Résidus')
    axes[1].set_title('Distribution des Résidus')

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

6.3 Validation Croisée

```
from sklearn.model_selection import cross_val_score, StratifiedKFold

def validation_croisee(model, X, y, cv=5):
    """Validation croisée avec intervalles de confiance"""

    # Stratified pour classification
```

```
    cv_strategy = StratifiedKFold(n_splits=cv, shuffle=True, random_state=42)

    # Scores sur plusieurs métriques
    metrics = ['accuracy', 'precision', 'recall', 'f1', 'roc_auc']

    print("=== VALIDATION CROISÉE ===\n")
    for metric in metrics:
        scores = cross_val_score(model, X, y, cv=cv_strategy, scoring=metric)
        print(f"{metric:12s}: {scores.mean():.3f} (+/- {scores.std()*2:.3f})")
```

Partie 7: Applications Bancaires du ML

7.1 Scoring de Crédit

```
def scoring_credit_complet(df):
    """Pipeline complet de scoring de crédit"""

    # 1. Préparation des données
    features = [
        'age', 'revenu', 'anciennete_emploi', 'ratio_dette_revenu',
        'nb_credits_actifs', 'nb_retards_12m', 'montant_demande',
        'type_emploi_encoded', 'proprietaire'
    ]

    X = df[features]
    y = df['defaut']

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

    # 2. Modèle (Logistique pour interprétabilité réglementaire)
    from sklearn.linear_model import LogisticRegression

    model = LogisticRegression(random_state=42, max_iter=1000)
    model.fit(X_train, y_train)

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

    # 4. Conversion en score (0-1000)
    def proba_to_score(proba, base=600, pdo=20):
        """Convertit probabilité en score"""
        odds = proba / (1 - proba)
        score = base - pdo * np.log(odds) / np.log(2)
        return np.clip(score, 300, 850)

    scores = proba_to_score(y_proba)

    # 5. Évaluation
    auc = roc_auc_score(y_test, y_proba)
    gini = 2 * auc - 1
```

```
    print(f"AUC: {auc:.3f}, Gini: {gini:.3f}")

    return model, scores
```

7.2 Détection de Fraude

```
def detecter_fraude(df_transactions):
    """Modèle de détection de fraude"""

    # Features de fraude
    features = [
        'montant', 'heure', 'jour_semaine',
        'distance_location_habituelle', 'nb_tx_jour',
        'ratio_montant_moyenne', 'est_weekend',
        'canal_risque_encoded', 'pays_risque'
    ]

    X = df_transactions[features]
    y = df_transactions['fraude']

    # Gestion du déséquilibre (fraude = rare)
    from imblearn.over_sampling import SMOTE
    smote = SMOTE(random_state=42)
    X_resampled, y_resampled = smote.fit_resample(X, y)

    X_train, X_test, y_train, y_test = train_test_split(
        X_resampled, y_resampled, test_size=0.3, random_state=42
    )

    # Modèle optimisé pour recall (détecter toutes les fraudes)
    from sklearn.ensemble import RandomForestClassifier

    model = RandomForestClassifier(
        n_estimators=100,
        class_weight='balanced',
        random_state=42
    )
    model.fit(X_train, y_train)

    # Seuil ajusté pour maximiser recall
    y_proba = model.predict_proba(X_test)[:, 1]

    # Chercher le seuil optimal
    from sklearn.metrics import precision_recall_curve
    precision, recall, thresholds = precision_recall_curve(y_test, y_proba)

    # Seuil pour recall > 90%
    idx = np.argmax(recall >= 0.90)
    optimal_threshold = thresholds[idx]

    y_pred = (y_proba >= optimal_threshold).astype(int)

    print(f"Seuil optimal: {optimal_threshold:.3f}")
    print(f"Recall: {recall_score(y_test, y_pred):.2%}")
```

```
    print(f"Precision: {precision_score(y_test, y_pred):.2%}")

    return model, optimal_threshold
```

7.3 Prédiction de Churn

```
def predire_churn(df_clients):
    """Prédiction d'attrition client"""

    features = [
        'anciennete_mois', 'nb_produits', 'solde_moyen',
        'nb_transactions_3m', 'variation_solde_3m',
        'nb_contacts_service_client', 'nb_reclamations',
        'derniere_activite_jours', 'age', 'segment_encoded'
    ]

    X = df_clients[features]
    y = df_clients['churned_6m']

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

    # Gradient Boosting
    from sklearn.ensemble import GradientBoostingClassifier

    model = GradientBoostingClassifier(
        n_estimators=100,
        max_depth=4,
        learning_rate=0.1,
        random_state=42
    )
    model.fit(X_train, y_train)

    # Prédictions
    y_proba = model.predict_proba(X_test)[:, 1]

    # Identifier clients à risque
    df_clients['proba_churn'] = model.predict_proba(X)[:, 1]

    clients_risque = df_clients[df_clients['proba_churn'] > 0.5].sort_values(
        'proba_churn', ascending=False
    )

    print(f"Clients à risque élevé: {len(clients_risque)}")
    print(f"AUC: {roc_auc_score(y_test, y_proba):.3f}")

    return model, clients_risque
```

7.4 Segmentation Clients ML

```
def segmentation_ml(df_clients):
    """Segmentation avancée avec ML"""
```

```
    features = [
        'age', 'anciennete', 'nb_produits', 'solde_total',
        'nb_transactions', 'volume_transactions',
        'ratio_epargne_credit', 'canal_principal_encoded'
    ]

    X = df_clients[features]
    X_scaled = StandardScaler().fit_transform(X)

    # K-Means
    kmeans = KMeans(n_clusters=5, random_state=42, n_init=10)
    df_clients['segment_ml'] = kmeans.fit_predict(X_scaled)

    # Nommer les segments
    profil = df_clients.groupby('segment_ml')[features].mean()

    segment_names = {
        0: 'Mass Market',
        1: 'Affluent',
        2: 'Jeunes Actifs',
        3: 'Seniors Fidèles',
        4: 'Digital First'
    }

    # Analyser chaque segment
    print("\n=== PROFIL DES SEGMENTS ===\n")
    for seg in sorted(df_clients['segment_ml'].unique()):
        subset = df_clients[df_clients['segment_ml'] == seg]
        print(f"\nSegment {seg} - {segment_names.get(seg, 'Unknown')}:")
        print(f"  Taille: {len(subset)} ({len(subset)/len(df_clients):.1%})")
        print(f"  Âge moyen: {subset['age'].mean():.0f}")
        print(f"  Solde moyen: {subset['solde_total'].mean():,.0f} HTG")
        print(f"  Produits moyens: {subset['nb_produits'].mean():.1f}")

    return df_clients
```

Partie 8: Bonnes Pratiques

8.1 Éviter le Data Leakage

```
# MAUVAIS: Fit scaler sur toutes les données
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # FUITE!
X_train, X_test = train_test_split(X_scaled, ...)

# BON: Fit uniquement sur train
X_train, X_test = train_test_split(X, ...)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)  # Utilise stats du train
```

8.2 Gestion du Déséquilibre de Classes

```python
# Techniques pour classes déséquilibrées (ex: fraude = 1%)

# 1. Poids des classes
model = LogisticRegression(class_weight='balanced')

# 2. SMOTE (suréchantillonnage)
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

# 3. Sous-échantillonnage
from imblearn.under_sampling import RandomUnderSampler
rus = RandomUnderSampler(random_state=42)
X_resampled, y_resampled = rus.fit_resample(X_train, y_train)

# 4. Ajuster le seuil de décision
y_proba = model.predict_proba(X_test)[:, 1]
y_pred = (y_proba >= 0.1).astype(int)  # Seuil bas pour classe rare
```

8.3 Interprétabilité des Modèles

```python
import shap

def expliquer_predictions(model, X_train, X_test):
    """Explication des prédictions avec SHAP"""

    # Créer explainer
    explainer = shap.TreeExplainer(model)
    shap_values = explainer.shap_values(X_test)

    # Importance globale
    shap.summary_plot(shap_values, X_test)

    # Explication d'une prédiction individuelle
    idx = 0  # Premier exemple
    shap.waterfall_plot(shap.Explanation(
        values=shap_values[idx],
        base_values=explainer.expected_value,
        data=X_test.iloc[idx],
        feature_names=X_test.columns.tolist()
    ))
```

Questions d'Entretien

1. **Différence entre overfitting et underfitting?** → Overfitting: modèle trop complexe, mémorise le train; Underfitting: trop simple, ne capture pas les patterns
2. **Comment gérer une variable cible déséquilibrée?** → SMOTE, class_weight, sous-échantillonnage, ajuster le seuil

3. **Pourquoi normaliser les données?** → Nécessaire pour algorithmes sensibles à l'échelle (régression, SVM, k-NN)
4. **Quand utiliser régression logistique vs Random Forest?** → Logistique: interprétabilité requise; RF: performance prime
5. **Qu'est-ce que le feature engineering?** → Création de nouvelles variables à partir des données brutes pour améliorer les modèles
6. **Comment valider un modèle de scoring en banque?** → Back-testing, validation out-of-time, métriques Gini/KS, tests de stabilité

Checklist ML

```
□ Définir clairement l'objectif business
□ Explorer et nettoyer les données (EDA)
□ Créer des features pertinentes
□ Séparer train/test correctement
□ Gérer le déséquilibre de classes si nécessaire
□ Essayer plusieurs algorithmes
□ Optimiser les hyperparamètres
□ Évaluer avec métriques appropriées
□ Valider par cross-validation
□ Interpréter le modèle
□ Documenter le processus
□ Planifier le monitoring
```

Rappel final: Le ML en banque doit équilibrer performance prédictive, interprétabilité et conformité réglementaire. La documentation et la traçabilité sont essentielles pour les audits.