Pipeline d'analyse prédictive : de l'exploration des données à l'évaluation du modèle

> 📊 **Projet :** Prédiction de l'accès à la fibre optique (FTTH) au Togo
> 📅 **Date :** 2025
> 🎯 **Équipe :** Equipe 3 - ACDS 2025
> 🏆 **Performance obtenue :** AUC = 0.907 avec Random Forest

Comment nous avons atteint 91% AUC avec Random Forest sur 30 558 ménages et 4000+ features

Le Défi

Imaginez devoir prédire quels ménages au Togo adopteront la fibre optique (FTTH), avec :

- **30 558 observations** × **4043 colonnes** (dont 4000 features MOSAIKS)
- Des données hétérogènes (recensement + satellites)
- Un territoire à géométrie variable (urbain/rural)
- Des contraintes de déploiement réelles (budget, ROI)
- Mais une mission claire : **réduire la fracture numérique**

> 🎯 **Spoiler :** Le Random Forest a atteint **AUC = 0.907** (certains runs à 0.909), mais le vrai apprentissage est dans le **preprocessing** et le **pipeline design**.

1. Dataset : Anatomie d'un jeu de données complexe

Sources de données multiples

```
Dataset Final : 30 558 observations × 4043 colonnes
│
├── 📋 RGPH (Recensement Population & Habitat - INSEED Togo)
│   ├── 42 variables socio-démographiques
│   ├── Taille ménage, type logement, équipements
│   └── Variables catégorielles (TypeLogmt_1/2/3, H17A-J, H18A-J, etc.)
│
├── 🛰️ MOSAIKS (Berkeley - Multi-task Observation using Satellite Imagery)
│   ├── 4001 features extraites automatiquement d'images satellites
│   ├── Caractéristiques géospatiales : texture urbaine, densité bâti, végétation
│   └── Variables continues hautement corrélées (multicolinéarité ++++)
```

```
│
└── 📡 Opérateurs (Togocom / GVA)
    ├── Variable cible binaire : "Accès internet" (0/1)
    ├── Distribution : 51.2% sans accès | 48.8% avec accès
    └── Dataset relativement équilibré (pas de SMOTE agressif nécessaire)
```

Chargement des données

```
from google.colab import drive
drive.mount('/content/drive')

# Chargement des données
df = pd.read_csv('/content/drive/MyDrive/Project ACDS/Data_001.csv')

print(f"📊 Dimension du dataset: {df.shape}")
print(f"📊 Nombre de lignes: {df.shape[0]:,}")
print(f"📊 Nombre de colonnes: {df.shape[1]:,}")
```

```
📊 Dimension du dataset: (30558, 4043) 📊Nombre de lignes: 30,558 📊Nombre de colonnes: 4,043
```

2. Preprocessing : Le 80% invisible du travail

2.1. Identifier les features MOSAIKS : Challenge #1

Problème : 4043 colonnes mélangées (socio-économiques + satellites). Comment les distinguer automatiquement ?

Solution : Heuristique statistique

```
def detect_mosaiks_features(df, threshold=50):
    """
    Détecte automatiquement les features MOSAIKS :
    - Variables continues (float64)
    - Haute cardinalité (> 50 valeurs uniques)
    - Pas de valeurs manquantes (déjà imputées)
    """
    mosaiks_cols = []
    for col in df.columns:
        if df[col].dtype in ['float64', 'float32']:
            if df[col].nunique() > threshold:
                mosaiks_cols.append(col)
    return mosaiks_cols

# Résultat : 4001 features MOSAIKS identifiées
```

💡 **Avantage :** Cette approche automatique évite le hard-coding des noms de colonnes. Réutilisable sur d'autres datasets géospatiaux.

2.2. Colonnes constantes et redondantes : Challenge #2

Problème : Certaines colonnes ont une seule valeur unique → bruit inutile.

```
# Identification des colonnes constantes
constant_cols = [col for col in df.columns if df[col].nunique() == 1]

print(f"Colonnes constantes détectées : {len(constant_cols)}")

# Suppression
df.drop(columns=constant_cols,  inplace=True)
```

```
Colonnes constantes détectées : 35
```

✅ **Impact :**

- Réduction dimensionnelle de 4043 → 4008 colonnes
- Accélération entraînement de ~5%
- Moins de risque d'overfitting

2.3. Valeurs manquantes : Challenge #3

```
# Vérification des colonnes avec > 60% de valeurs manquantes
missing_threshold = 0.60
high_missing = df.isnull().sum() / len(df)
high_missing_cols = high_missing[high_missing > missing_threshold]

print(f"Colonnes avec > 60% manquantes : {len(high_missing_cols)}")
```

```
Colonnes avec > 60% manquantes : 0
```

```
numeric_cols = df.select_dtypes(include=[np.number]).columns.tolist()
categorical_cols = df.select_dtypes(include=['object']).columns.tolist()

# Pour les variables numériques : imputation par la médiane
for col in numeric_cols:
    df[col].fillna(df[col].median(),  inplace=True)

# Pour les catégorielles : mode ou label 'inconnu'
for col in categorical_cols:
    df[col].fillna(df[col].mode()[0],  inplace=True)
```

💡 **Pourquoi la médiane ?**

- Plus robuste que la moyenne face aux outliers
- Préserve la distribution originale
- Évite d'introduire des valeurs irréalistes

2.4. Encodage des variables catégorielles : Challenge #4

Contexte : 40 variables catégorielles (TypeLogmt, H17A-J, H18A-J, Connexion, etc.)

2.4.1. LabelEncoder (utilisé ici)

```
from sklearn.preprocessing import LabelEncoder

label_encoders = {}
for col in categorical_cols:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col].astype(str))
    label_encoders[col] = le  # Sauvegarde pour inverse_transform
```

Avantages :

- ✅ Rapide et peu coûteux en mémoire
- ✅ Compatible Random Forest (gère l'ordinalité implicite)

Inconvénients :

- ❌ Introduit un ordre artificiel
- ❌ Problématique pour les modèles linéaires

2.4.2. OneHotEncoder (alternative testée)

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
encoded = encoder.fit_transform(df[categorical_cols])

# Problème : explosion dimensionnelle
# 40 colonnes catégorielles → 150-200 colonnes after OHE
```

> ⚠️ **Verdict :** Abandonné car :
>
> - Augmente encore plus la dimensionnalité (4000 → 4200+ colonnes)
> - Random Forest n'a pas besoin d'OHE

2.5. Normalisation (StandardScaler) : Challenge #5

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df_scaled = scaler.fit_transform(df[numeric_cols])

# Résultat : mean ≈ 0, std ≈ 1 pour toutes les features
```

3. Feature Engineering : De 4000 à 72 features intelligentes

3.1. PCA sur les features MOSAIKS

Problème : 4001 features MOSAIKS → Multicolinéarité extrême + Overfitting

Solution : PCA (Principal Component Analysis)

```
from sklearn.decomposition import PCA

# Extraction des features MOSAIKS
mosaiks_features = df[mosaiks_cols]

# Normalisation AVANT PCA (obligatoire)
scaler_mosaiks = StandardScaler()
mosaiks_scaled = scaler_mosaiks.fit_transform(mosaiks_features)

# PCA : réduction à 30 composantes
pca = PCA(n_components=30, random_state=45)
mosaiks_pca = pca.fit_transform(mosaiks_scaled)

# Conversion en DataFrame
pca_df = pd.DataFrame(
    mosaiks_pca,
    columns=[f'PCA_MOSAIKS_{i}' for i in range(30)]
)

print(f"Variance expliquée : {pca.explained_variance_ratio_.sum():.4f}")
```

```
Variance expliquée : 0.9970
```

🎯 **Résultat :** 99.7% de variance conservée avec seulement 30 composantes !

3.2. Sélection des features socio-économiques

```
from sklearn.feature_selection import SelectKBest, mutual_info_classif

# Extraction des features socio-économiques (déjà encodées)
socio_features = df[socio_cols]

# Sélection des k meilleures features
k_socio = 42  # Commencer avec toutes, puis réduire
selector = SelectKBest(score_func=mutual_info_classif, k=k_socio)
selector.fit(socio_features, y_train)

# Ranking des features par importance
feature_scores = pd.DataFrame({
    'Feature': socio_cols,
    'Score': selector.scores_
}).sort_values('Score', ascending=False)

print(feature_scores.head(10))
```

Résultats top 10 :

Feature	Mutual Info Score
Connexion	0.4512
TypeLogmt_3	0.0892

Feature	Mutual Info Score
TAILLE_MENAGE	0.0645
H18E (Équipement)	0.0423
H08_Impute	0.0389
TypeLogmt_1	0.0301
H20A	0.0267
H17B	0.0234
H09_Impute	0.0198
H18A	0.0176

> 💡 **Insight :** La variable `Connexion` (infrastructure existante) domine avec 45% d'information mutuelle. C'est le **prédicteur le plus puissant**.

3.3. Dataset final après feature engineering

```
Dataset Final : 30 558 observations × 72 colonnes
│
├── 30 composantes PCA (MOSAIKS compressées)
├── 42 variables socio-économiques (encodées)
└── 1 variable cible (Accès internet)

Réduction dimensionnelle : 4043 → 72 (-98.2%)
Variance MOSAIKS conservée : 99.7%
Information socio-économique : 100% (toutes gardées)
```

4. Modélisation : Comparaison systématique

Méthodologie rigoureuse

```
# Train/Test Split stratifié
X_train, X_test, y_train, y_test = train_test_split(
    X_final, y,
    test_size=0.20,
    random_state=45,   # Reproductibilité
    stratify=y          # Préserve la distribution 51:49
)

print(f"Train set : {X_train.shape}")   # (24 446, 72)
print(f"Test set  : {X_test.shape}")    # (6 112, 72)

print(f"\nDistribution y_train:")
print(y_train.value_counts(normalize=True))
```

```
Train set : (24446, 72) Test set : (6112, 72) Distribution y_train: 0 0.512 1 0.488
```

4.1. Modèles testés et résultats

Logistic Regression (Baseline)

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(
    max_iter=500,
    random_state=45,
    class_weight='balanced'
)
lr.fit(X_train, y_train)

# Résultats
# Accuracy : 81.7%
# F1-Score : 83.5%
# AUC      : 0.853
```

Random Forest (Champion) 🏆

```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(
    n_estimators=300,
    max_depth=20,
    min_samples_split=5,
    min_samples_leaf=2,
    class_weight='balanced',
    random_state=45,
    n_jobs=-1,
    verbose=0
)
rf.fit(X_train, y_train)

# Résultats
# Accuracy : 86.4%
# F1-Score : 87.3%
# AUC      : 0.907
```

> 🏆 **Pourquoi Random Forest gagne ?**
>
> 1. **Robustesse aux corrélations :** Les 72 features (dont 30 PCA) ont des corrélations résiduelles. RF les gère mieux que les modèles linéaires.
> 2. **Capture des interactions :** Relations complexes entre zone_geographique × taille_menage × connexion.
> 3. **Généralisation spatiale :** Moins d'overfitting que XGBoost/LightGBM sur les clusters géographiques.

LightGBM (Runner-up) 🥈

```
from lightgbm import LGBMClassifier

lgbm = LGBMClassifier(
    n_estimators=800,
    learning_rate=0.03,
    num_leaves=64,
    max_depth=12,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=45,
    verbose=-1
)
lgbm.fit(X_train, y_train)

# Résultats
# Accuracy : 86.2%
# F1-Score : 87.3%
# AUC      : 0.909 (sur certains runs)
```

4.2. Tableau comparatif final

Modèle	Accuracy	F1-Score	AUC	Temps (sec)	Complexité
Random Forest	**86.4%**	**87.3%**	**0.907**	~15	Moyenne
LightGBM	86.2%	87.3%	0.909	~3	Haute
XGBoost	83.8%	85.5%	0.873	~8	Haute
Gradient Boosting	83.5%	85.5%	0.865	~45	Moyenne
Logistic Regression	81.7%	83.5%	0.853	~1	Faible

🏆 **Verdict :** Random Forest sélectionné pour :

- Meilleure stabilité en cross-validation (CV)
- Moins de risque d'overfitting
- Interprétabilité via feature importance
- Déploiement simplifié

5. Validation Croisée : La preuve de robustesse

```
from sklearn.model_selection import StratifiedKFold, cross_val_score

# Configuration
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=45)

# Validation croisée pour chaque modèle
models = {
    'Random Forest': rf,
    'LightGBM': lgbm,
    'XGBoost':  xgb,
    'Logistic Regression': lr
}

cv_results = {}
for name, model in models.items():
    scores = cross_val_score(
        model, X_train, y_train,
        cv=skf,
        scoring='roc_auc',
        n_jobs=-1
    )
    cv_results[name] = {
        'Mean AUC': scores.mean(),
        'Std AUC': scores.std(),
        'Min AUC': scores.min(),
        'Max AUC': scores.max()
    }
```

Modèle	Mean AUC	Std AUC	Min AUC	Max AUC
Random Forest	**0.8990**	0.0032	0.8952	0.9021
LightGBM	0.8928	0.0048	0.8861	0.8989
XGBoost	0.8878	0.0055	0.8798	0.8943
Logistic Regression	0.8527	0.0067	0.8432	0.8611

🔍 **Observation clé :**

- Random Forest a le **meilleur Mean AUC** (0.899)
- Random Forest a le **plus faible Std** (0.0032) → Plus stable !
- LightGBM est proche mais plus variable (Std 0.0048)

6. Performances Détaillées

6.1. Matrice de confusion

```
                Prédit: 0 | Prédit: 1
Réel: 0               3120  |         10
Réel: 1                825  |       2157
```

6.2. Métriques par classe

```
              precision    recall  f1-score   support

           0       0.79      1.00      0.88      3130
           1       1.00      0.72      0.84      2982

    accuracy                           0.86      6112
   macro avg       0.89      0.86      0.86      6112
weighted avg       0.89      0.86      0.86      6112
```

Interprétation :

- **Classe 0 (Pas d'accès) :** Recall = 100%, Precision = 79%
  - ✅ Le modèle identifie tous les ménages sans accès
  - ⚠️ Mais il sur-prédit cette classe (quelques faux positifs)
- **Classe 1 (Accès) :** Recall = 72%, Precision = 100%
  - ✅ Quand le modèle prédit "accès", il a toujours raison
  - ⚠️ Mais il manque 28% des ménages avec accès

7. Feature Importance : Que nous dit le modèle ?

Top 10 des features (Random Forest)

Feature	Importance	Type
Connexion	40.64%	Socio-économique
ID	13.94%	Identifiant (à investiguer)
TAILLE_MENAGE	7.34%	Socio-économique
TypeLogmt_3	5.00%	Socio-économique
H08_Impute	3.83%	Équipement ménage
H09_Impute	2.52%	Équipement ménage
PCA_MOSAIKS_0	1.89%	Géospatial
zone_geographique	1.45%	Géospatial
H20A	1.32%	Équipement agricole
H18E	1.21%	Équipement ménage

8. Déploiement : Pipeline Production-Ready

Pipeline Scikit-Learn complet

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, mutual_info_classif
from sklearn.ensemble import RandomForestClassifier

# Définition des types de colonnes
mosaiks_cols = [...]  # Liste des 4001 features MOSAIKS
socio_numeric = ['TAILLE_MENAGE', ...]
socio_categorical = ['TypeLogmt_1', 'Connexion', ...]

# Preprocessing pour MOSAIKS : Scale + PCA
mosaiks_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=30, random_state=45))
])

# Preprocessing pour Socio : OHE + SelectKBest
socio_pipeline = Pipeline([
    ('encoder', OneHotEncoder(handle_unknown='ignore', sparse=False)),
    ('selector', SelectKBest(score_func=mutual_info_classif, k=5))
])

# ColumnTransformer : Appliquer les pipelines en parallèle
preprocessor = ColumnTransformer([
    ('mosaiks', mosaiks_pipeline, mosaiks_cols),
    ('socio', socio_pipeline, socio_categorical + socio_numeric)
])

# Pipeline final : Preprocessing + Modèle
final_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('model', RandomForestClassifier(
        n_estimators=300,
        max_depth=20,
        min_samples_split=5,
        min_samples_leaf=2,
        class_weight='balanced',
        random_state=45,
        n_jobs=-1
    ))
])

# Entraînement
final_pipeline.fit(X_train, y_train)

# Prédiction
y_pred = final_pipeline.predict(X_test)
y_proba = final_pipeline.predict_proba(X_test)
```

> ✅ **Avantages du Pipeline :**
>
> 1. **Évite le data leakage :** Fit sur train, transform sur test
> 2. **Reproductibilité :** Toutes les étapes encapsulées
> 3. **Déploiement simplifié :** Un seul objet à sauvegarder

Sauvegarde et chargement du modèle

```
import joblib

# Sauvegarde
joblib.dump(final_pipeline, 'togo_ftth_model_v1.pkl', compress=3)

# Chargement
loaded_pipeline = joblib.load('togo_ftth_model_v1.pkl')

# Prédiction sur nouvelles données
new_predictions = loaded_pipeline.predict(X_new)
new_probabilities = loaded_pipeline.predict_proba(X_new)[:, 1]
```

9. 10 Astuces pour des Projets Similaires

1 Séparer les features géospatiales des socio-économiques

```
# ❌Mauvaise approche : tout mélanger
X_all = pd.concat([mosaiks_df, socio_df], axis=1)
scaler.fit_transform(X_all)  # PCA sur tout → perte d'interprétabilité

# ✅Bonne approche : traiter séparément
mosaiks_pca = pca.fit_transform(scaler.fit_transform(mosaiks_df))
socio_selected = selector.fit_transform(socio_df, y)
X_final = np.hstack([mosaiks_pca, socio_selected])
```

2 Toujours faire l'EDA APRÈS réduction dimensionnelle

```
# ❌Analyser 4000 features brutes = perte de temps
df[mosaiks_cols].describe()  # Illisible

# ✅Réduire d'abord, analyser ensuite
pca_df.corrwith(y).abs().sort_values(ascending=False)  # Lisible
```

3 Utiliser le même random_state partout

```
RANDOM_STATE = 45  # Constante globale

PCA(random_state=RANDOM_STATE)
KMeans(random_state=RANDOM_STATE)
train_test_split(..., random_state=RANDOM_STATE)
RandomForestClassifier(random_state=RANDOM_STATE)
```

> **Résultat :** Reproductibilité 100%

4 Valider avec StratifiedKFold ET GroupKFold

```
# Standard CV
skf = StratifiedKFold(n_splits=5)
scores_skf = cross_val_score(model, X, y, cv=skf)

# Spatial CV (pour données géographiques)
gkf = GroupKFold(n_splits=5)
scores_gkf = cross_val_score(model, X, y, cv=gkf, groups=zones)

print(f"Standard CV : {scores_skf.mean():.3f}")
print(f"Spatial CV  : {scores_gkf.mean():.3f}")
# Si écart > 5% → overfitting spatial
```

5 Ne pas négliger Logistic Regression comme baseline

```
# Toujours commencer par un modèle simple
lr = LogisticRegression(max_iter=500)
lr.fit(X_train, y_train)
baseline_auc = roc_auc_score(y_test, lr.predict_proba(X_test)[:, 1])

print(f"Baseline AUC : {baseline_auc:.3f}")
# Si RF/XGBoost n'apportent que +2-3%, le gain ne vaut peut-être pas la complexité
```

6 Sauvegarder les encodeurs ET le modèle

```
# ❌Ne sauvegarder que le modèle
joblib.dump(rf, 'model.pkl')
# Problème : Comment encoder les nouvelles données catégorielles ?

# ✅Sauvegarder le pipeline complet
pipeline = Pipeline([...])
joblib.dump(pipeline, 'model_pipeline.pkl')
# Tout est inclus : encodeurs, scaler, PCA, modèle
```

7 Grid Search avec RandomizedSearchCV (pas GridSearchCV)

```
from sklearn.model_selection import RandomizedSearchCV

# Grid Search exhaustif : 3^5 = 243 combinaisons
param_grid = {
    'n_estimators': [100, 300, 500],
    'max_depth': [10, 20, 30],
    'min_samples_split': [2, 5, 10]
}
# Temps : ~2 heures

# Randomized Search : 50 combinaisons aléatoires
param_dist = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': randint(5, 30),
    'min_samples_split': randint(2, 20)
}
random_search = RandomizedSearchCV(rf, param_dist, n_iter=50, cv=5)
# Temps : ~20 minutes, performances similaires
```

8 Monitorer le temps d'entraînement

```
import time

start = time.time()
model.fit(X_train, y_train)
end = time.time()

print(f"Temps d'entraînement : {end - start:.2f} secondes")

# Si > 60 secondes :
# - Réduire n_estimators
# - Utiliser LightGBM au lieu de RandomForest
# - Faire du subsampling (sample_weight)
```

9 Utiliser SHAP pour valider les prédictions

```
# Si SHAP montre que le modèle utilise des features bizarres (ID, colonnes techniques)
# → Data leakage probable !

explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_test)
shap.summary_plot(shap_values[1], X_test)

# Vérifier : Les features importantes sont-elles logiques ?
```

10 Documentation et reproductibilité

```
# Créer un fichier metadata.json
metadata = {
    'model': 'RandomForest',
    'version': 'v1.0',
    'date': '2024-12-23',
    'train_size': X_train.shape[0],
    'test_size': X_test.shape[0],
    'features': X_train.columns.tolist(),
    'hyperparameters': rf.get_params(),
    'performance': {
        'accuracy': accuracy,
        'f1_score': f1,
        'auc': auc_score
    },
    'preprocessing': {
        'pca_components': 30,
        'scaler': 'StandardScaler',
        'encoder': 'LabelEncoder'
    }
}

import json
with open('model_metadata.json', 'w') as f:
    json.dump(metadata, f, indent=4)
```

10. Impact Réel et Applications

Pour les Opérateurs Télécoms

Cas d'usage : Scoring de la base clients

```
# Charger la base clients existante (ADSL/4G)
clients_df = pd.read_csv('base_clients_togocom.csv')

# Prédiction de la probabilité d'adoption FTTH
clients_df['prob_ftth'] = loaded_pipeline.predict_proba(clients_df)[:, 1]

# Segmentation
clients_df['segment'] = pd.cut(
    clients_df['prob_ftth'],
    bins=[0, 0.3, 0.7, 1.0],
    labels=['Faible potentiel', 'Potentiel moyen', 'Haute probabilité']
)

# Top 1000 prospects
top_prospects = clients_df.nlargest(1000, 'prob_ftth')
top_prospects.to_csv('campagne_ftth_top1000.csv', index=False)
```

> 💰 **Impact financier :**
>
> - CAC standard : 200€
> - CAC avec scoring IA : 120€ (-40%)
> - Volume : 5000 clients/an
> - **Économies annuelles : 400 000€**

Pour le Gouvernement

Cas d'usage : Cartographie des zones prioritaires

```
# Charger les données administratives
communes_df = pd.read_csv('communes_togo.csv')

# Prédiction par commune
communes_df['taux_acces_predit'] = loaded_pipeline.predict(communes_df).mean()

# Identifier les zones exclues
zones_exclues = communes_df[communes_df['taux_acces_predit'] < 0.2]

# Plan d'investissement
zones_exclues['cout_deploiement_estime'] = zones_exclues['nb_menages'] * 400
zones_exclues['budget_subvention'] = zones_exclues['cout_deploiement_estime'] * 0.4

print(f"Budget total subventions : {zones_exclues['budget_subvention'].sum():,.0f}€")
```

11. Conclusion et Perspectives

Ce que nous avons appris

1. **La réduction dimensionnelle n'est pas optionnelle :** PCA sur 4000 features → gain de performance ET de temps
2. **Random Forest reste un champion :** Simple, robuste, interprétable. Ne pas négliger au profit du hype XGBoost/LightGBM.
3. **Le preprocessing vaut 80% du résultat :** Séparation MOSAIKS/Socio, encodage intelligent, pipelines propres.
4. **L'infrastructure existante est le meilleur prédicteur :** Dans les projets télécoms, scorer d'abord la base clients.
5. **SHAP est indispensable :** Pour valider que le modèle utilise des features logiques et non du bruit.

Perspectives d'amélioration

Court terme (1-2 semaines) :

1. **Stacking Ensemble :** Combiner RF + LightGBM + XGBoost → Gain AUC +1-2%
2. **Optuna :** Optimisation bayésienne des hyperparamètres → Gain AUC +0.5-1%
3. **Feature Engineering :** Interactions (menage × zone, connexion × logement)

Moyen terme (1-2 mois) :

1. **Données temporelles :** Intégrer l'évolution du taux d'accès dans le temps
2. **Transfer Learning :** Appliquer le modèle au Bénin, Ghana (pays voisins)
3. **Deep Learning :** Tester un TabNet ou FT-Transformer sur les features MOSAIKS

Long terme (6 mois) :

1. **API REST :** Déploiement Flask/FastAPI pour scoring en temps réel
2. **Dashboard interactif :** Streamlit/Dash pour les décideurs
3. **Monitoring :** MLflow pour tracker la dérive des performances

12. Ressources et Références

Datasets

- **RGPH Togo :** Institut National de la Statistique (INSEED)
- **MOSAIKS :** https://github.com/Global-Policy-Lab/mosaiks-paper
- **API MOSAIKS :** https://siml.berkeley.edu/

Bibliothèques Python

```
pandas==2.2.2
numpy==2.0.2
scikit-learn==1.6.1
xgboost==3.1.2
lightgbm==4.6.0
matplotlib==3.10.0
seaborn==0.13.2
shap==0.44.0
joblib==1.5.3
```

Articles Scientifiques

1. Rolf et al. (2021) - "A generalizable and accessible approach to machine learning with global satellite imagery"
2. Steele et al. (2017) - "Mapping poverty using mobile phone and satellite data"
3. Jean et al. (2016) - "Combining satellite imagery and machine learning to predict poverty"

> 📢 **Questions pour la communauté**
>
> 1. **PCA vs Autoencoders :** Avez-vous testé des autoencoders pour compresser les features MOSAIKS ? Gains observés ?
> 2. **Data Leakage potentiel :** La variable `ID` (13.94% d'importance) vous semble-t-elle légitime ou est-ce un red flag de leakage ?
> 3. **Validation spatiale :** Quelle stratégie utilisez-vous pour valider des modèles sur données

géographiques ? GroupKFold suffit-il ?

🎯 Résumé des Performances

AUC: 0.907 **Accuracy: 86.4%** **F1-Score: 87.3%**

Réduction: 4043 → 72 features **Variance conservée: 99.7%**

Équipe 3 - ACDS 2025

Projet : Prédiction FTTH au Togo