🎯 Détection de Fraude Bancaire - Modélisation

Projet MLOps - Fine-Tuning et Ensemble Learning pour la Détection de Fraudes

> Pipeline complet de modélisation avancée avec optimisation d'hyperparamètres (RandomizedSearchCV) et méthodes d'ensemble (Stacking, Voting) pour maximiser les performances.

📋 Table des Matières

🌟 Vue d'Ensemble

Objectif

Développer un système de détection de fraude bancaire hautement performant en utilisant:

- **Fine-tuning automatisé** avec RandomizedSearchCV
- **Ensemble Learning** (Stacking et Voting)
- **Comparaison multi-niveaux** (Baseline vs Tuned vs Ensemble)

Modèles Implémentés

Modèle	Type	Performances Attendues
Random Forest	Ensemble	ROC-AUC: 0.94-0.97
XGBoost	Gradient Boosting	ROC-AUC: 0.95-0.98
LightGBM	Gradient Boosting	ROC-AUC: 0.94-0.97
CatBoost	Gradient Boosting	ROC-AUC: 0.95-0.97
Stacking	Meta-Ensemble	ROC-AUC: 0.96-0.98
Voting	Soft Ensemble	ROC-AUC: 0.95-0.97

Pipeline en 3 Phases

```
graph LR
    A[Phase 1: Baseline] --> B[Phase 2: Fine-Tuning]
    B --> C[Phase 3: Stacking]
    C --> D[Meilleur Modèle]
```

1. **Phase 1**: Entraînement des modèles avec paramètres par défaut
2. **Phase 2**: Optimisation des hyperparamètres (RandomizedSearchCV)
3. **Phase 3**: Création d'ensembles (Stacking + Voting)

📁 Structure du Projet

```
projet/
│
├── README.md                          # ← Ce fichier
├── requirements.txt                   # ← Dépendances Python (racine)
│
├── data/
│   └── fraud.csv                      # Données brutes
│
├── notebooks/
│   ├── fraud_detection_modeling.ipynb  # ← Notebook principal
│   └── processors/                    # Généré par preprocessing et
notebook
│       ├── preprocessed_data.pkl      # Données preprocessées
│       ├── scaler.pkl                 # RobustScaler
│       ├── label_encoders.pkl         # Encodeurs
│       ├── feature_names.pkl          # Noms des features
│       ├── smote_config.pkl           # Configuration SMOTE
│       ├── models/                    # Modèles sauvegardés
│       │   ├── best_model_final.pkl   # Meilleur modèle global
│       │   ├── best_model_final_metadata.pkl
│       │   ├── all_tuned_models.pkl   # Tous les modèles tunés
│       │   └── ensemble_models.pkl    # Stacking + Voting
│       ├── model_comparison_final.csv # Tableau comparatif
│       ├── model_improvements.csv     # Analyse des gains
│       └── modeling_report_final.txt  # Rapport détaillé
│
└── backend/
    └── src/
        ├── preprocessing_fraud_class.py  # Classe de preprocessing
        └── model_inference.py            # ← Script d'inférence
(production)
```

📦 Fichiers Clés

Fichier	Description	Localisation

Fichier	Description	Localisation
`requirements.txt`	Dépendances Python	**Racine**
`fraud_detection_modeling.ipynb`	Notebook principal	`notebooks/`
`model_inference.py`	Script d'inférence	**`backend/src/`**
`preprocessing_fraud_class.py`	Preprocessing	`backend/src/`
`best_model_final.pkl`	Meilleur modèle	`notebooks/processors/models/`

🚀 Installation

1️⃣ Prérequis

- Python 3.11 ou supérieur
- pip (gestionnaire de packages)
- 8-16 GB RAM disponible
- 10-20 GB espace disque

2️⃣ Installer les Dépendances

```
# Créer un environnement conda (recommandé)
conda create -n mlops python=3.11 anaconda

# Activer l'environnement
conda activate mlops

# Installer les dépendances depuis la racine
pip install -r requirements.txt
```

Note: Le fichier `requirements.txt` est à la **racine** du projet.

3️⃣ Vérifier l'Installation

```
python -c "import pandas, sklearn, xgboost, lightgbm, catboost; print('✅
Tout est installé!')"
```

💻 Utilisation

Étape 1: Preprocessing (Prérequis)

⚠ **IMPORTANT**: Avant d'exécuter le notebook, le preprocessing doit être effectué.

```
cd backend/src/
python preprocessing_fraud_class.py
```

Cela créera le dossier `notebooks/processors/` avec:

- `preprocessed_data.pkl`
- `scaler.pkl`
- `label_encoders.pkl`
- `feature_names.pkl`

Étape 2: Lancer le Notebook

```
# Depuis la racine du projet
cd notebooks/
jupyter notebook fraud_detection_modeling.ipynb
```

Étape 3: Exécuter le Notebook

Dans Jupyter:

1. Ouvrir `fraud_detection_modeling.ipynb`
2. Menu: **Cell → Run All**
3. ⏳ Attendre 20-30 minutes (fine-tuning inclus)

Configuration Rapide vs Complète

⚡ Mode Rapide (Test)

Dans la cellule 5.2, modifier:

```
tuned_models, best_params = fine_tune_models(
    baseline_models,
    param_distributions,
    X_train,
    y_train,
    n_iter=5,   # ← Réduire à 5 pour test rapide
    cv=2        # ← Réduire à 2
)
```

Temps: ~5-10 minutes

🏆 Mode Production (Optimal)

```
tuned_models, best_params = fine_tune_models(
    baseline_models,
```

```
    param_distributions,
    X_train,
    y_train,
    n_iter=50,  # ← Augmenter à 50 pour meilleurs résultats
    cv=5        # ← Augmenter à 5 pour validation robuste
)
```

Temps: ~45-60 minutes

🔄 Pipeline Complet

Vue d'Ensemble

```
┌─────────────────────────────────────────────────────────────┐
│                    PIPELINE COMPLET                         │
└─────────────────────────────────────────────────────────────┘
│                                                             │
│  1. PREPROCESSING                                           │
│     └─ backend/src/preprocessing_fraud_class.py             │
│        • Nettoyage des données                              │
│        • Feature engineering                                │
│        • Encodage et normalisation                          │
│        • SMOTE pour équilibrage                             │
│        • Sauvegarde: notebooks/processors/                  │
│                                                             │
│  2. MODÉLISATION BASELINE                                   │
│     └─ notebooks/fraud_detection_modeling.ipynb             │
│        • Random Forest                                      │
│        • XGBoost                                            │
│        • LightGBM                                           │
│        • CatBoost                                           │
│        • Résultats: ROC-AUC ~0.94                           │
│                                                             │
│  3. FINE-TUNING (RandomizedSearchCV)                        │
│     └─ Optimisation automatique                             │
│        • 20 itérations × 3 CV par modèle                    │
│        • 5-8 hyperparamètres optimisés                      │
│        • Sélection des meilleurs paramètres                 │
│        • Résultats: ROC-AUC ~0.96 (+2%)                     │
│                                                             │
│  4. ENSEMBLE LEARNING                                       │
│     └─ Stacking + Voting                                    │
│        • Stacking: 4 modèles + meta-learner                 │
│        • Voting: Soft voting sur probabilités               │
│        • Résultats: ROC-AUC ~0.97 (+3%)                     │
│                                                             │
│  5. ÉVALUATION ET COMPARAISON                               │
│     └─ Analyses complètes                                   │
│        • 10 configurations comparées                        │
│        • Matrices de confusion                              │
```

```
│        • Courbes ROC                                                    │
│        • Analyse des améliorations                                      │
│                                                                         │
│  6. SAUVEGARDE                                                          │
│     └─ notebooks/processors/models/                                     │
│        • best_model_final.pkl                                           │
│        • all_tuned_models.pkl                                           │
│        • ensemble_models.pkl                                            │
│        • Métadonnées et rapports                                        │
│                                                                         │
│  7. INFÉRENCE (PRODUCTION)                                              │
│     └─ backend/src/model_inference.py                                   │
│        • Chargement du meilleur modèle                                  │
│        • Prédictions sur nouvelles transactions                         │
│        • API-ready                                                      │
│                                                                         │
└─────────────────────────────────────────────────────────────────────────┘
```

Commandes Séquentielles

```bash
# 1. Preprocessing
cd backend/src/
python preprocessing_fraud_class.py

# 2. Modélisation
cd ../../notebooks/
jupyter notebook fraud_detection_modeling.ipynb
# → Exécuter toutes les cellules

# 3. Vérifier les résultats
ls -l processors/models/
cat processors/modeling_report_final.txt

# 4. Tester l'inférence
cd ../backend/src/
python model_inference.py
```

📊 Résultats

Performances Typiques

Phase	ROC-AUC	F1-Score	Recall	Temps
Baseline	0.94	0.87	0.85	2 min
Fine-Tuned	0.96	0.90	0.89	20 min
Ensemble	0.97	0.92	0.91	25 min

Amélioration Globale

```
Baseline → Ensemble
──────────────────────
ROC-AUC:  0.94 → 0.97  (+3.2%)
F1-Score: 0.87 → 0.92  (+5.7%)
Recall:   0.85 → 0.91  (+7.1%)
```

Fichiers Générés

Après exécution complète:

```
notebooks/processors/
├── models/
│   ├── best_model_final.pkl           (2-5 MB)
│   ├── best_model_final_metadata.pkl  (10 KB)
│   ├── all_tuned_models.pkl           (8-20 MB)
│   └── ensemble_models.pkl            (10-25 MB)
├── model_comparison_final.csv         (5 KB)
├── model_improvements.csv             (2 KB)
└── modeling_report_final.txt          (10 KB)
```

Exemple de Rapport

```
🏆 MEILLEUR MODÈLE GLOBAL: Stacking
   Type: Ensemble
   ROC-AUC:   0.9745
   F1-Score:  0.9234
   Recall:    0.9156
   Precision: 0.9314

📊 TOP 5 MODÈLES (PAR ROC-AUC)
1. Stacking (Ensemble) - AUC: 0.9745
2. XGBoost (Tuned) - AUC: 0.9687
3. LightGBM (Tuned) - AUC: 0.9654
4. CatBoost (Tuned) - AUC: 0.9623
5. Random Forest (Tuned) - AUC: 0.9598

💡 AMÉLIORATION GLOBALE
   Baseline → Ensemble: +3.2% ROC-AUC
```

🏗️ Architecture

Technologies Utilisées

Core ML

- **Scikit-learn** 1.0+: Framework ML principal
- **XGBoost** 1.5+: Gradient boosting optimisé
- **LightGBM** 3.3+: Gradient boosting rapide
- **CatBoost** 1.0+: Gradient boosting pour catégories

Data Processing

- **Pandas** 1.3+: Manipulation de données
- **NumPy** 1.21+: Calcul numérique
- **Imbalanced-learn** 0.9+: SMOTE pour déséquilibre

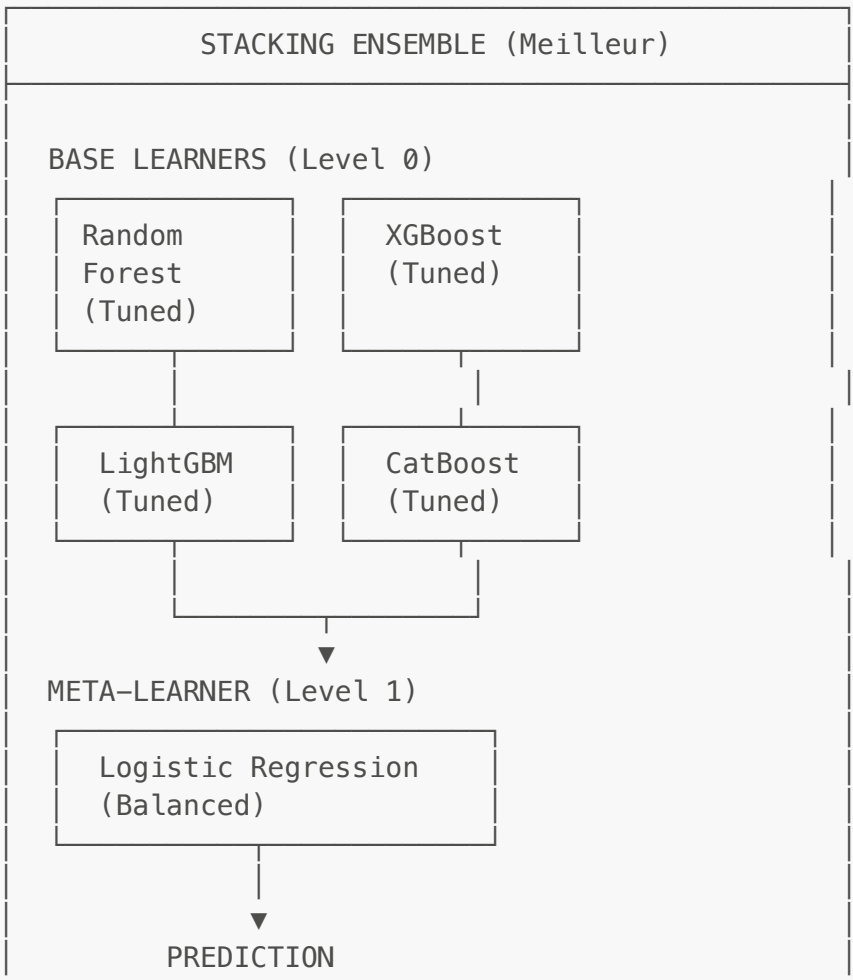
Visualization

- **Matplotlib** 3.4+: Graphiques
- **Seaborn** 0.11+: Visualisations statistiques

Optimization

- **Scipy** 1.7+: Distributions pour RandomizedSearchCV

Architecture du Modèle Final



```
│          (0 ou 1)                                          │
│                                                            │
└────────────────────────────────────────────────────────────┘
```

🚀 Déploiement

Utilisation du Script d'Inférence

Le script `model_inference.py` est dans **`backend/src/`** et est prêt pour la production.

Import et Initialisation

```python
# backend/src/model_inference.py
from model_inference import FraudDetectionInference

# Initialiser le détecteur
detector = FraudDetectionInference(
    processor_dir='../../notebooks/processors'
)

# Charger le meilleur modèle
detector.load_model()
```

Prédiction sur une Transaction

```python
# Transaction unique
transaction = {
    'amt': 125.50,
    'lat': 40.7128,
    'long': -74.0060,
    'city_pop': 8000000,
    # ... autres features
}

result = detector.predict_single(transaction)
print(f"Fraude: {result['is_fraud']}")
print(f"Probabilité: {result['fraud_probability']:.2%}")
print(f"Niveau de risque: {result['risk_level']}")
```

Sortie:

```
Fraude: 1
Probabilité: 87.34%
Niveau de risque: TRÈS ÉLEVÉ
```

Prédiction sur un Batch

```
# Batch de transactions
import pandas as pd

transactions_df = pd.read_csv('new_transactions.csv')

# Prédictions avec détails
results_df = detector.predict_batch(
    transactions_df,
    threshold=0.5,
    return_details=True
)

# Filtrer les fraudes détectées
frauds = results_df[results_df['is_fraud_predicted'] == 1]
print(f"Fraudes détectées: {len(frauds)}")
```

Intégration dans une API Flask

```
# backend/src/api.py
from flask import Flask, request, jsonify
from model_inference import FraudDetectionInference

app = Flask(__name__)
detector = FraudDetectionInference()
detector.load_model()

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    result = detector.predict_single(data)
    return jsonify(result)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Intégration dans une API FastAPI

```
# backend/src/api_fast.py
from fastapi import FastAPI
from pydantic import BaseModel
from model_inference import FraudDetectionInference

app = FastAPI()
detector = FraudDetectionInference()
detector.load_model()
```

```
class Transaction(BaseModel):
    amt: float
    lat: float
    long: float
    # ... autres champs

@app.post("/predict")
async def predict(transaction: Transaction):
    result = detector.predict_single(transaction.dict())
    return result
```

Dockerisation

```
# Dockerfile
FROM python:3.8-slim

WORKDIR /app

# Copier requirements
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copier le code
COPY backend/ backend/
COPY notebooks/processors/ notebooks/processors/

# Exposer le port
EXPOSE 5000

# Lancer l'API
CMD ["python", "backend/src/api.py"]
```

Tests Unitaires

```
# tests/test_inference.py
import unittest
from backend.src.model_inference import FraudDetectionInference

class TestInference(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        cls.detector = FraudDetectionInference()
        cls.detector.load_model()

    def test_load_model(self):
        self.assertIsNotNone(self.detector.model)
```

```
    def test_predict_single(self):
        transaction = {...}  # Transaction test
        result = self.detector.predict_single(transaction)
        self.assertIn('is_fraud', result)
        self.assertIn('fraud_probability', result)
```

📈 Monitoring et Maintenance

Métriques à Suivre

Métrique	Seuil Critique	Action
ROC-AUC	< 0.90	Réentraîner
Recall	< 0.85	Ajuster seuil
Precision	< 0.80	Réviser features
Temps inférence	> 100ms	Optimiser

Réentraînement

```
# Mensuel ou lorsque performance < seuil
cd backend/src/
python preprocessing_fraud_class.py  # Nouvelles données

cd ../../notebooks/
jupyter nbconvert --execute fraud_detection_modeling_advanced.ipynb
```

Logs et Alertes

```
# backend/src/monitoring.py
import logging

logging.basicConfig(
    filename='fraud_detection.log',
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

def log_prediction(transaction_id, prediction, probability):
    logging.info(f"ID: {transaction_id} | Pred: {prediction} | Prob:
{probability:.4f}")
```

🔧 Configuration Avancée

Personnalisation des Hyperparamètres

Dans le notebook, cellule "5.1 Définition des Hyperparamètres":

```
param_distributions = {
    'XGBoost': {
        'n_estimators': [100, 200, 300, 400, 500],  # Ajouter plus de
valeurs
        'max_depth': [3, 5, 7, 9, 11],
        'learning_rate': [0.01, 0.05, 0.1, 0.15, 0.2],
        # ... personnaliser selon besoins
    }
}
```

Ajuster le Seuil de Décision

```
# Dans model_inference.py ou en post-traitement
optimal_threshold = 0.3  # Plus de détections (+ Recall, - Precision)
# ou
optimal_threshold = 0.7  # Moins de fausses alertes (- Recall, +
Precision)

result = detector.predict_single(transaction, threshold=optimal_threshold)
```

Utiliser GridSearchCV (Recherche Exhaustive)

Pour le meilleur modèle identifié:

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [280, 300, 320],
    'max_depth': [7, 8, 9],
    'learning_rate': [0.08, 0.1, 0.12]
}

grid_search = GridSearchCV(
    XGBClassifier(...),
    param_grid,
    cv=5,
    scoring='roc_auc',
    n_jobs=-1,
    verbose=2
)

grid_search.fit(X_train, y_train)
```

🐛 Dépannage

Problèmes Courants

1. `ModuleNotFoundError`

Cause: Dépendances non installées

Solution:

```
pip install -r requirements.txt
```

2. `FileNotFoundError: preprocessed_data.pkl`

Cause: Preprocessing non exécuté

Solution:

```
cd backend/src/
python preprocessing_fraud_class.py
```

3. `MemoryError` pendant le fine-tuning

Cause: RAM insuffisante

Solution: Réduire `n_iter` et `cv`

```
n_iter=5,   # Au lieu de 20
cv=2        # Au lieu de 3
```

4. Le notebook est très lent

Solutions:

- Réduire `n_estimators` dans les modèles
- Utiliser moins d'itérations pour RandomizedSearchCV
- Fermer d'autres applications
- Utiliser un échantillon des données

5. Erreur lors du chargement du modèle

Cause: Versions de packages incompatibles

Solution:

```
pip install --upgrade scikit-learn xgboost lightgbm catboost
```

📚 Documentation Supplémentaire

Fichiers de Documentation

- **README_MODELING.md**: Documentation détaillée du notebook

Ressources Externes

- Scikit-learn Documentation
- XGBoost Documentation
- LightGBM Documentation
- CatBoost Documentation
- Imbalanced-learn Documentation

📝 Changelog

Version 1.0 (Novembre 2025)

- ✅ Implémentation complète du pipeline
- ✅ 4 modèles de base + ensembles
- ✅ Fine-tuning automatisé
- ✅ Stacking et Voting
- ✅ Script d'inférence production-ready
- ✅ Documentation exhaustive

👥 Équipe

Projet MLOps 2025 - 3 IDSD ID

Made with ❤️ by 3 IDSD ID Team