TP4 - DÉTECTION DE FRAUDE BANCAIRE

Optimisation ML & Feature Engineering Avancé
Master 1 Data Engineering - Concepts & Techno IA

OBJECTIFS PÉDAGOGIQUES

À l'issue de ce TP, vous serez capable de :
- Maîtriser le Feature Engineering avancé (interactions, agrégations, encodages)
- Gérer des données fortement déséquilibrées (class_weight, SMOTE, métriques adaptées)
- Implémenter une validation croisée robuste (StratifiedKFold, TimeSeriesSplit)
- Optimiser des hyperparamètres avec GridSearchCV et RandomizedSearchCV
- Construire des Pipelines ML complets anti-leakage
- Analyser la performance avec courbes ROC, Learning Curves, Feature Importance, SHAP
- Comparer 6 algorithmes (LR, DT, RF, SVM, KNN, XGBoost)
- Maîtriser l' explicabilité (MDI, Permutation Importance, SHAP Values)
- Déployer un modèle en production (API Flask, tests unitaires, monitoring)

CONTEXTE MÉTIER

Vous êtes Data Scientist dans une néobanque européenne qui traite 2 millions de transactions par jour.

Problématique:

Le système actuel de détection de fraude génère :
- Trop de faux positifs (clients légitimes bloqués → insatisfaction)
- Trop de faux négatifs (fraudes non détectées → pertes financières)

Mission

Développer un modèle ML optimisé qui :
1. Détecte au moins 85% des fraudes réelles (Recall ≥ 0.85)
2. Minimise les faux positifs (Precision maximale)
3. Soit explicable pour les équipes métier (Feature Importance)
4. Soit déployable en production (Pipeline robuste, reproductible)

Contraintes techniques
- Déséquilibre extrême : ~0.17% de fraudes (1 fraude pour 600 transactions)
- Latence : Prédiction en < 100ms (modèle léger)
- Réglementation : Explicabilité requise (RGPD)

DONNÉES

Dataset : Credit Card Fraud Detection

Source : Kaggle - Transactions anonymisées par PCA (284,807 transactions)

Fichier : `data/creditcard.csv`

Structure

```
Time      : Secondes écoulées depuis la 1ère transaction
V1-V28    : Composantes PCA (features anonymisées)
Amount    : Montant de la transaction (€)
Class     : 0 = Légitime, 1 = Fraude
```

Caractéristiques

- 492 fraudes (0.172%) sur 284,807 transactions
- Features déjà normalisées (sauf `Time` et `Amount`)
- Données temporelles (2 jours de transactions)

PARTIE 1 : EXPLORATION & FEATURE ENGINEERING

1.1 Analyse Exploratoire Ciblée

Objectifs :

- Quantifier le déséquilibre
- Analyser les distributions (fraudes vs légitimes)
- Identifier les features discriminantes

Livrables attendus :

```python
À implémenter dans le notebook
- Ratio fraude/légitime
- Statistiques descriptives par classe
- Distribution de 'Amount' et 'Time' (boxplots, histogrammes)
- Matrice de corrélation (top 10 features)
```

Questions métier :

1. Les fraudes ont-elles des montants typiques ?
2. Y a-t-il des patterns temporels (heures, jours) ?
3. Quelles features PCA sont les plus corrélées à la fraude ?

1.2 Analyse Temporelle Approfondie

Objectifs :

- Analyser la distribution des fraudes par heure de la journée
- Calculer le taux de fraude pour chaque heure : `taux_fraude(h) = nb_fraudes(h) / nb_total_transactions(h)`
- Identifier les heures à risque élevé
- Créer une matrice de corrélation entre Time, Amount et les top 5 features PCA

Livrables attendus :

```python
*Graphiques à créer :*
- *Distribution des fraudes par heure (bar plot)*
- *Taux de fraude par heure (line plot)*
- *Heatmap de corrélation Time/Amount/Top5_PCA*
```

1.3 Feature Engineering Avancé

Mission : Créer au minimum 15 nouvelles features pertinentes.

Features Temporelles (5 features minimum)

```python
*Formules mathématiques :*
- *hour = (Time / 3600) % 24*
- *day = (Time / 86400) % 2*
- *hour_sin = sin(2π × hour / 24)*
- *hour_cos = cos(2π × hour / 24)*
- *period = f(hour) avec Nuit:[0-6h], Matin:[6-12h], Après-midi:[12-18h], Soir:[18-24h]*
```

Features sur les Montants (5 features minimum)

```python
*Transformations mathématiques :*
- *amount_log = log(Amount + 1)*
- *amount_sqrt = √Amount*
- *amount_squared = Amount²*
- *amount_cubed = Amount³*
- *is_zero_amount = 1 si Amount = 0, sinon 0*
- *amount_bin = pd.cut(Amount, bins=[0, 10, 50, 100, 500, ∞])*
```

Features d'Interaction (3 features minimum)

```python
*Produits et ratios :*
- *V_i × Amount (pour les 3 features PCA les plus corrélées)*
- *amount_per_hour = Amount / (hour + 1)*
- *time_amount_ratio = Time / (Amount + 1)*
```

```
```
Features d'Agrégation (2 features minimum)
```python
*Statistiques sur les features PCA :*
- *pca_sum_top5 = Σ(V1, V2, V3, V4, V5)*
- *pca_mean_top5 = mean(V1, V2, V3, V4, V5)*
- *pca_std = std(V1, ..., V28)*
```
Features d'Écart (3 features minimum)
```python
*Z-score pour détecter les anomalies :*
- *deviation_Vi = |Vi - μ(Vi)| / σ(Vi)*
*Calculer pour les 3 features PCA les plus corrélées*
```
ATTENTION AU DATA LEAKAGE :

- Les agrégations doivent être calculées uniquement sur le passé
- Utiliser `TimeSeriesSplit` pour valider
- Documenter chaque feature créée

Livrables :
- Fonction `create_features(df)` dans `utils/feature_engineering.py`
- Analyse de l'impact : Corrélation des nouvelles features avec `Class`
- Justification métier de chaque feature

PARTIE 2 : MODÉLISATION BASELINE

2.1 Préparation des Données

Tâches :

1. Split Train/Test :

```python
*Stratégie à justifier :*
- *Train/Test split classique (80/20) ?*
- *OU TimeSeriesSplit (respecter l'ordre temporel) ?*
```

2. Scaling :
```python
Choix du scaler à argumenter :
- StandardScaler ?
- RobustScaler (présence d'outliers) ?
- MinMaxScaler ?
```
3. Gestion du déséquilibre :
```python
Stratégies à comparer :
- class_weight='balanced'
- Undersampling (RandomUnderSampler)
- Oversampling (SMOTE)
- Combinaison (SMOTETomek)
```

2.2 Modèles Baseline

Objectif : Établir des références de performance avec 6 algorithmes.

Modèles à tester :

1. Logistic Regression (avec `class_weight='balanced'`)
2. Decision Tree (profondeur limitée, `max_depth=10`)
3. Random Forest (100 arbres, `class_weight='balanced'`)
4. Support Vector Machine (SVM) (`kernel='rbf'`, `class_weight='balanced'`)
5. K-Nearest Neighbors (KNN) (`n_neighbors=5`, `weights='distance'`)
6. XGBoost (baseline avec `scale_pos_weight` calculé)

Métriques obligatoires :

```python
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    roc_auc_score,
    precision_recall_curve,
    average_precision_score
)
```

À calculer pour chaque modèle :
- *Matrice de confusion*
- *Precision, Recall, F1-Score*

- *ROC-AUC*
- *PR-AUC (Precision-Recall AUC) ← Plus pertinent ici !*

```
Pourquoi PR-AUC > ROC-AUC pour ce cas ?
- Avec 0.17% de fraudes, ROC-AUC est trop optimiste
- PR-AUC se concentre sur la classe minoritaire
Formules clés :
```

```
Precision = TP / (TP + FP)
Recall = TP / (TP + FN)
F1-Score = 2 × (Precision × Recall) / (Precision + Recall)
```

```
Livrables :
- Tableau comparatif des 6 modèles
- Courbes ROC et Precision-Recall
- Analyse : Quel modèle privilégier et pourquoi ?
```

2.3 Comparaison des Stratégies de Rééquilibrage

Objectif : Comparer `class_weight='balanced'` vs SMOTE. SMOTE (Synthetic Minority Over-sampling Technique)

```
Principe mathématique :
```

```
Pour chaque exemple minoritaire x_i :
1. Trouver k plus proches voisins (k=5 par défaut)
2. Choisir aléatoirement un voisin x_neighbor
3. Générer un exemple synthétique :
  x_new = x_i + λ × (x_neighbor - x_i)
  où λ ∈ [0, 1] est un nombre aléatoire
```

```
Tâches :
1. Appliquer SMOTE sur le train set
2. Entraîner le meilleur modèle baseline avec SMOTE
3. Comparer les performances : class_weight vs SMOTE
4. Créer un tableau comparatif : Stratégie | Precision | Recall | F1 | PR-AUC
```

Questions :
- Quelle stratégie donne le meilleur Recall ?
- Y a-t-il un trade-off Precision/Recall ?
- Laquelle recommandez-vous pour la production ?

PARTIE 3 : OPTIMISATION AVANCÉE

3.1 Pipeline ML Complet

Mission : Construire un pipeline Scikit-Learn intégrant :

1. Feature Engineering
2. Scaling
3. Gestion du déséquilibre (optionnel dans le pipeline)
4. Modèle

Template :

```python
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
```

À compléter :

```
pipeline = Pipeline([
    ('feature_eng', ...),    Vos transformations custom
    ('scaler', ...),         Choix du scaler
    ('classifier', ...)      Modèle à optimiser
])
```

Avantages attendus :
- Anti-leakage garanti
- Reproductibilité
- Compatible avec GridSearchCV
- Déployable directement

3.2 Optimisation Hyperparamètres - Random Forest

Objectif : Trouver la meilleure configuration avec `GridSearchCV`.

Grille de recherche suggérée :

```python
param_grid = {
    'classifier__n_estimators': [100, 200, 300],
    'classifier__max_depth': [10, 20, 30, None],
    'classifier__min_samples_split': [2, 5, 10],
    'classifier__min_samples_leaf': [1, 2, 4],
```

```
    'classifier__max_features': ['sqrt', 'log2'],
    'classifier__class_weight': ['balanced', 'balanced_subsample']
}
```

Attention : Cette grille génère 324 combinaisons × 5 folds = 1620 entraînements !

Stratégies d'optimisation :

1. Commencer petit : Réduire la grille pour tester le code
2. RandomizedSearchCV : Tester 50 combinaisons aléatoires (plus rapide)
3. Parallélisation : `n_jobs=-1`
4. Validation : `StratifiedKFold(n_splits=5)`

Livrables :

```*python*
*À afficher :*
- *Meilleurs hyperparamètres trouvés*
- *Score de validation croisée (PR-AUC)*
- *Temps d'exécution*
- *Comparaison avant/après optimisation*
```

3.3 Optimisation avec RandomizedSearchCV

Objectif : Comparer GridSearchCV vs RandomizedSearchCV.

Principe de RandomizedSearchCV

Au lieu de tester toutes les combinaisons, teste n_iter combinaisons aléatoires.

Formule probabiliste :

```
Probabilité de trouver une combinaison dans le top 5% :
P(top 5%) = 1 - (0.95)^n
Exemples :
- n = 20 ⟶ P ≈ 64%
- n = 60 ⟶ P ≈ 95%
- n = 100 ⟶ P ≈ 99.4%
```

Avantages :

- Plus rapide (contrôle du nombre d'itérations)
- Peut explorer un espace plus large
- Bon compromis temps/performance

Tâches :
1. Définir des distributions de paramètres (plus larges que GridSearch)
2. Configurer `RandomizedSearchCV` avec `n_iter=30`
3. Comparer les résultats : GridSearch vs RandomizedSearch
4. Créer un tableau : Méthode | Meilleur Score | Temps (min) | Nb Combinaisons

3.4 Optimisation de XGBoost

Objectif : Optimiser XGBoost avec RandomizedSearchCV.

Paramètres à optimiser :

```python
from scipy.stats import randint, uniform
param_distributions = {
    'classifier__n_estimators': randint(50, 300),
    'classifier__max_depth': [3, 5, 7, 9],
    'classifier__learning_rate': uniform(0.01, 0.29),    [0.01, 0.3]
    'classifier__subsample': uniform(0.6, 0.4),    [0.6, 1.0]
    'classifier__colsample_bytree': uniform(0.6, 0.4),
    'classifier__scale_pos_weight': [calculé automatiquement]
}
```

Formule `scale_pos_weight` :

```
scale_pos_weight = nb_négatifs / nb_positifs
                 = count(Class=0) / count(Class=1)
                 ≈ 284315 / 492 ≈ 578
```

Livrables :
- Meilleurs hyperparamètres pour XGBoost
- Comparaison : Random Forest optimisé vs XGBoost optimisé
- Analyse : Quel modèle atteint l'objectif Recall ≥ 0.85 ?

PARTIE 4 : ANALYSE & DIAGNOSTIC

4.1 Feature Importance (MDI)

Objectifs :

1. Extraire l'importance des features du meilleur modèle (Mean Decrease Impurity)
2. Visualiser le Top 15

Formule MDI (Random Forest) :

```
Importance(feature_i) = Σ (weighted_decrease_in_impurity)
                        sur tous les nœuds utilisant feature_i
```

Questions :

- Les features créées sont-elles utiles ?
- Peut-on simplifier le modèle (réduire les features) ?
- Y a-t-il des surprises (features inattendues importantes) ?

Livrables :

```python
Graphique horizontal des Top 15 features
Analyse écrite : Que révèlent ces importances ?
```

4.2 Permutation Importance

Objectif : Calculer une importance plus robuste que MDI.

Principe mathématique :

```
Pour chaque feature f :
1. Calculer le score baseline : S_baseline
2. Mélanger aléatoirement les valeurs de f (permutation)
3. Calculer le nouveau score : S_permuted
4. Importance(f) = S_baseline - S_permuted
Plus la baisse est importante, plus la feature est cruciale.
```

Avantages vs MDI :

- Non biaisé vers les features à haute cardinalité
- Fonctionne avec n'importe quel modèle
- Plus coûteux en calcul (nécessite n_repeats prédictions)

Tâches :

1. Calculer la Permutation Importance avec `n_repeats=10`
2. Comparer avec MDI : Créer un graphique côte à côte
3. Analyser les différences : Quelles features changent de rang ?

4.3 SHAP Values (Explicabilité Avancée)

Objectif : Expliquer les prédictions individuelles avec la théorie des jeux.

Formule de Shapley :

```
φ_i = Σ [|S|! × (n-|S|-1)! / n!] × [f(S ∪ {i}) - f(S)]
    S ⊆ N\{i}
```

où :

- *φ_i = contribution de la feature i*
- *S = sous-ensemble de features*
- *f(S) = prédiction avec le sous-ensemble S*
- *n = nombre total de features*

Interprétation :

- SHAP value > 0 : la feature pousse vers la classe 1 (Fraude)
- SHAP value < 0 : la feature pousse vers la classe 0 (Légitime)
- |SHAP value| = magnitude de l'impact

Tâches :

1. Calculer les SHAP values avec `TreeExplainer`
2. Créer un summary plot (importance + direction)
3. Expliquer une prédiction individuelle (force plot)
4. Analyser : Quelles features contribuent le plus aux fraudes détectées ?

Livrables :

```python
Summary plot SHAP
Force plot pour 1 fraude correctement détectée
Analyse comparative : MDI vs Permutation vs SHAP
```

4.4 Courbes d'Apprentissage (Learning Curves)

Mission : Diagnostiquer overfitting/underfitting.

```python
from sklearn.model_selection import learning_curve
À tracer :
- Score train vs validation en fonction de la taille du dataset
- Analyse : Le modèle bénéficierait-il de plus de données ?
```

Interprétations attendues :
- Convergence des courbes ?
- Gap important (overfitting) ?
- Scores bas (underfitting) ?
- Recommandations d'amélioration

4.5 Calibration des Probabilités

Objectif : Vérifier et améliorer la calibration du modèle.

Principe :
Un modèle bien calibré : si proba = 0.8, alors 80% des prédictions sont correctes.
Méthode : Courbe de Calibration (Reliability Diagram)
```
1. Diviser les probabilités en bins (ex: [0-0.1], [0.1-0.2], ...)
2. Pour chaque bin, calculer :
  - Probabilité moyenne prédite
  - Fraction réelle de positifs
3. Tracer : x = proba prédite, y = fraction réelle
4. Comparer à la diagonale (calibration parfaite)
```
Interprétation :
- Proche de la diagonale → bien calibré
- Au-dessus → sous-estime les probabilités
- En-dessous → surestime les probabilités
Correction : Platt Scaling
```
Ajuste une régression logistique sur les probabilités :
P_calibrated = 1 / (1 + exp(A × log(p/(1-p)) + B))
```
Tâches :
1. Tracer la courbe de calibration
2. Appliquer `CalibratedClassifierCV` avec `method='sigmoid'`
3. Comparer PR-AUC avant/après calibration
4. Analyser l'impact sur les prédictions

4.6 Optimisation du Seuil de Décision

Contexte : Par défaut, seuil = 0.5, mais ce n'est pas optimal pour données déséquilibrées.
Mission : Trouver le seuil maximisant le F1-Score ou le F2-Score (privilégie le Recall).

```python
from sklearn.metrics import precision_recall_curve
  À implémenter :
1. Calculer precision/recall pour tous les seuils possibles
2. Tracer la courbe Precision-Recall
3. Identifier le seuil optimal selon un critère métier
4. Recalculer les métriques avec ce nouveau seuil
```
Livrables :
- Graphique avec seuil optimal marqué
- Tableau : Métriques avant/après ajustement du seuil
- Recommandation : Quel seuil déployer en production ?

PARTIE 5 : DÉPLOIEMENT & PRODUCTION

5.1 Validation Temporelle (TimeSeriesSplit)

Objectif : Valider que le modèle fonctionne dans le temps.

```python
from sklearn.model_selection import TimeSeriesSplit
  À implémenter :
tscv = TimeSeriesSplit(n_splits=5)
  Évaluer le modèle avec cette stratégie
  Comparer avec StratifiedKFold
```
Questions :
- Les performances sont-elles stables dans le temps ?
- Y a-t-il une dégradation (concept drift) ?
- Faut-il réentraîner régulièrement ?

5.2 Sérialisation & Reproductibilité

Mission : Préparer le modèle pour la production.
Tâches :
1. *Sauvegarder le pipeline complet :*
   ```python
   import joblib
   joblib.dump(best_pipeline, 'models/fraud_detector_v1.joblib')
   ```

2. Créer un script de prédiction (`utils/predict.py`) :

```python
def predict_fraud(transaction_data):
    """
    Prédit si une transaction est frauduleuse.

    Args:
        transaction_data: dict ou DataFrame
    Returns:
        dict: {'is_fraud': bool, 'probability': float, 'risk_level': str}
    """
     À compléter
```

3. Documenter :

- Version des bibliothèques (requirements.txt)
- Seed aléatoire utilisé
- Seuil de décision choisi
- Métriques de performance

Livrables :

- *`models/fraud_detector_v1.joblib`*
- *`models/metadata_v1.joblib` (métadonnées du modèle)*
- *`utils/predict.py` fonctionnel*
- *`DEPLOYMENT_README.md` avec instructions*

5.3 Création d'une API de Prédiction (Flask)

Objectif : Créer une API REST pour servir le modèle en production.

Architecture :

```
API Flask
├── Endpoint POST /predict
│   Input: JSON avec features d'une transaction
│   Output: {"is_fraud": bool, "probability": float, "risk_level": str}
├── Endpoint GET /health
│   Output: {"status": "ok", "model_version": "1.0"}


```

Niveaux de risque :

```python
if proba < 0.3:
    risk_level = 'low'
elif proba < 0.6:
    risk_level = 'medium'
elif proba < 0.85:
    risk_level = 'high'
else:
    risk_level = 'critical'
```

Tâches :

1. Créer le fichier `api_fraud_detection.py`
2. Implémenter les 2 endpoints
3. Charger le modèle au démarrage (éviter de recharger à chaque requête)
4. Gérer les erreurs (try/except)
5. Tester avec curl ou Postman

Exemple de requête :

```bash
curl -X POST http://localhost:5000/predict \
  -H 'Content-Type: application/json' \
  -d '{"Time": 12345, "V1": -1.5, ..., "Amount": 149.62}'
```

5.4 Tests Unitaires du Modèle

Objectif : Garantir la qualité et la robustesse du modèle.

Tests à implémenter (avec `unittest`) :

1. Test de chargement :

```python
def test_model_loading():
    model = joblib.load('models/fraud_detector_v1.joblib')
    assert model is not None
```

2. Test de shape :

```python
def test_prediction_shape():
    predictions = model.predict(X_test_sample)
    assert len(predictions) == len(X_test_sample)
```

```
```

3. Test de valeurs :
```python
def test_prediction_values():
    probas = model.predict_proba(X_test_sample)[:, 1]
    assert all(0 <= p <= 1 for p in probas)
```

4. Test de performance :
```python
def test_prediction_time():
    start = time.time()
    model.predict(X_test_sample)
    elapsed = (time.time() - start) * 1000
    assert elapsed < 100    < 100ms pour 10 transactions
```

Livrables :

- Classe `TestFraudDetector` avec au moins 4 tests
- Rapport d'exécution des tests
- Analyse : Tous les tests passent-ils ?

5.5 Monitoring & Alertes (Réflexion)

Questions ouvertes (pas de code requis) :

1. Métriques à monitorer en production :
   - Quels KPIs suivre quotidiennement ?
   - Comment détecter une dégradation du modèle (concept drift) ?
   - Métriques techniques : latence, throughput, taux d'erreur
2. Réentraînement :
   - À quelle fréquence réentraîner (hebdomadaire, mensuel) ?
   - Quels triggers (seuils d'alerte) ?
   - Comment gérer les nouvelles fraudes (feedback loop) ?
3. Explicabilité :
   - Comment expliquer une prédiction à un client ?
   - Outils à utiliser (SHAP, LIME) ?
   - Conformité RGPD (droit à l'explication)
4. Biais & Équité :
   - Le modèle peut-il être biaisé (montants, heures) ?
   - Comment l'auditer ?
   - Tests de fairness à implémenter

Livrable : Document de quelques lignes avec vos réflexions et recommandations.

LIVRABLES FINAUX

Fichiers à rendre :

```
TD/
├── notebooks/
│   └── TP4_Detection_Fraude_[VOTRE_NOM].ipynb ← Notebook complété
├── utils/
│   ├── feature_engineering.py          ← Fonctions de features
│   └── predict.py                      ← Script de prédiction
├── models/
│   └── fraud_detector_v1.joblib        ← Modèle sauvegardé
└── DEPLOYMENT_README.md                ← Documentation déploiement
```

Critères de qualité :

Code :

- Propre, commenté, PEP8
- Fonctions réutilisables
- Gestion des erreurs

Analyse :

- Justifications des choix techniques
- Interprétations métier
- Visualisations claires

Performance :

- Recall ≥ 0.85 sur la classe Fraude
- PR-AUC > 0.75
- Modèle explicable

Reproductibilité :

- `random_state` fixé partout
- Pipeline complet
- Documentation

BONUS (Optionnel)

Niveau Expert :

1. Stacking / Voting Classifier :

```python
from sklearn.ensemble import VotingClassifier
  Combiner Logistic + RF + XGBoost
  Comparer avec le meilleur modèle seul
```

2. Détection d'Anomalies :

```python
from sklearn.ensemble import IsolationForest
  Approche non supervisée
  Comparer avec l'approche supervisée
```

3. Analyse de Sensibilité :
   - Tester la robustesse aux perturbations
   - Ajouter du bruit aux features et mesurer l'impact
4. Dashboard de Monitoring (Streamlit) :
   - Visualiser les prédictions en temps réel
   - Afficher les métriques de performance
   - Interface pour tester des transactions
5. Optimisation Bayésienne :

```python
from skopt import BayesSearchCV
  Plus efficace que RandomizedSearchCV
  Utilise un modèle probabiliste pour guider la recherche
```

RESSOURCES

Documentation officielle :

- [Scikit-Learn Pipelines](https://scikit-learn.org/stable/modules/compose.html)
- [Imbalanced-Learn](https://imbalanced-learn.org/)
- [XGBoost](https://xgboost.readthedocs.io/)

Articles recommandés :

- "Learning from Imbalanced Data" (He & Garcia, 2009)
- "SMOTE: Synthetic Minority Over-sampling Technique" (Chawla et al., 2002)

Dataset :

- [Kaggle - Credit Card Fraud Detection](https://www.kaggle.com/mlg-ulb/creditcardfraud)

CONSEILS & PIÈGES À ÉVITER

Erreurs fréquentes :

1. Data Leakage :
   - Calculer des stats sur train+test avant le split
   - Utiliser des infos du futur dans les features temporelles
2. Mauvaise métrique :
   - Se fier uniquement à l'Accuracy (99.83% en prédisant tout "légitime" !)
   - Ignorer le coût métier (FP vs FN)
3. Overfitting :
   - Trop de features sans validation
   - Hyperparamètres trop complexes
4. Oublier la reproductibilité :
   - Pas de `random_state`
   - Versions de bibliothèques non fixées

Bonnes pratiques :

- Commencer simple : Baseline avant optimisation
- Valider souvent : Cross-validation systématique
- Documenter : Chaque choix technique
- Penser production : Code propre, réutilisable
- Interpréter : Ne pas être une "boîte noire"

BON COURAGE !

N'oubliez pas : L'objectif n'est pas d'avoir le meilleur score, mais de comprendre et maîtriser le processus complet d'optimisation ML en conditions réelles.