GUIDE du code

1. data_preprocessi1. data_preprocessing.py

Objectif :

Préparer les données brutes pour l'analyse en nettoyant et en enrichissant les logs d'activité.

Étapes détaillées :

1.Chargement des données :

```
df = pd.read_csv(file_path)
```

- Lit le fichier CSV contenant les logs d'activité

2.Conversion des timestamps :

```
df['timestamp'] = pd.to_datetime(df['timestamp'], errors='coerce')
```

- Convertit la colonne de timestamp en format datetime
- errors='coerce' transforme les valeurs invalides en NaT (Not a Time)

3.Extraction des caractéristiques temporelles :

```
df['hour_of_day'] = df['timestamp'].dt.hour

df['day_of_week'] = df['timestamp'].dt.dayofweek

df['is_weekend'] = df['day_of_week'].apply(lambda x: 1 if x >= 5 else 0)

df['is_after_hours'] = df['hour_of_day'].apply(lambda x: 1 if x < 8 or x > 18 else 0)
```

- Extrait l'heure de la journée (0-23)
- Extrait le jour de la semaine (0=lundi, 6=dimanche)
- Crée un indicateur pour le weekend
- Crée un indicateur pour les heures hors bureau (avant 8h ou après 18h)

4.Gestion des valeurs manquantes :

```
null_counts = df.isnull().sum()

df = df.dropna(subset=['user', 'pc', 'activity', 'resource'])
```

- Compte les valeurs manquantes

- Supprime les lignes avec des valeurs manquantes dans les colonnes critiques

Explication :

Ce script transforme les données brutes en un format exploitable en ajoutant des caractéristiques temporelles essentielles pour l'analyse comportementale. La gestion des valeurs manquantes garantit la qualité des données.

2. neo4j_setup.py

Objectif :

Établir la connexion à Neo4j et importer les données prétraitées.

Étapes détaillées :

1. **Connexion à Neo4j :**

```
graph = Graph(uri, auth=(username, password))
```

- Établit une connexion à la base Neo4j avec les identifiants fournis

2.Nettoyage de la base :

```
graph.run("MATCH (n) DETACH DELETE n")
```

- Supprime tous les nœuds et relations existants (pour une importation propre)

3.Création des nœuds :

```
users[user_id] = Node("User", name=user_id, type="employee")

graph.create(users[user_id])
```

- Crée des nœuds pour chaque entité (utilisateurs, systèmes, activités, ressources)
- Utilise un dictionnaire pour éviter les doublons

4.Création des relations :

```
user_activity = Relationship(users[user_id], "PERFORMS", activities[activity_id],
                timestamp=row['timestamp'].isoformat(),
                hour_of_day=int(row['hour_of_day']),
                is_after_hours=int(row['is_after_hours']),
                is_weekend=int(row['is_weekend']))
graph.create(user_activity)
```

- Crée des relations typées entre les nœuds
- Ajoute des propriétés aux relations (timestamp, indicateurs temporels)

5.Création des index :

```
graph.run("CREATE INDEX user_name IF NOT EXISTS FOR (u:User) ON (u.name)")
```

- Améliore les performances des requêtes en créant des index sur les propriétés fréquemment interrogées

Explication :

Ce script construit le graphe de connaissances dans Neo4j en transformant les données tabulaires en une structure de graphe riche avec des nœuds, des relations et des propriétés. Cette représentation est essentielle pour les analyses relationnelles complexes.

3. neo4j_analysis.py

Objectif :

Analyser le graphe pour détecter des comportements suspects et extraire des caractéristiques pour le machine learning.

Étapes détaillées :

1. **Détection des activités après heures :**

```
MATCH (u:User)-[r:PERFORMS]->(a:Activity)

WHERE r.is_after_hours = 1

WITH u, count(r) AS after_hours_activity_count

RETURN u.name AS user, after_hours_activity_count
```

- Compte les activités effectuées en dehors des heures normales par utilisateur

2.Calcul d'entropie des activités :

```
entropy = 0

for act in activities:

  p = act['freq'] / total

  entropy -= p * math.log2(p)
```

- Calcule l'entropie de Shannon pour mesurer la régularité des activités
- Une entropie élevée indique un comportement plus aléatoire/diversifié

3.Extraction des caractéristiques ML :

```
MATCH (u:User)-[r:PERFORMS]->()

WHERE r.is_weekend = 1

RETURN count(r) AS weekend_activities
```

- Extrait diverses métriques par utilisateur :
  - Comptes d'activités (totales, weekend, après heures)
  - Nombre de systèmes et ressources uniques
  - Ratios d'activités spéciales
  - Mesures d'entropie

4.Centralité de degré :

```
MATCH (u:User)
CALL {
  WITH u
  MATCH (u)-[r]->()
  RETURN count(r) AS out_degree
}
CALL {
  WITH u
  MATCH (u)<-[r]-()
  RETURN count(r) AS in_degree
}
```

- Calcule le degré entrant et sortant pour chaque utilisateur
- Identifie les utilisateurs très connectés dans le graphe

Explication :

Ce script exploite la puissance des requêtes de graphe pour identifier des patterns comportementaux intéressants. Les mesures comme l'entropie et la centralité fournissent des indicateurs quantitatifs pour détecter des anomalies.

4. Comparaison_ML.py

Objectif :

Comparer différentes approches de détection d'anomalies (supervisées et non supervisées).

Étapes détaillées :

1. Imports & Configuration

```
import os

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns
```

- os : manipuler le système de fichiers (ex : créer des dossiers pour les visualisations)
- pandas : manipulation de tables de données (DataFrames)
- numpy : manipulation de tableaux numériques
- matplotlib et seaborn : pour les visualisations

```
from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.cluster import KMeans, DBSCAN

from sklearn.ensemble import IsolationForest, RandomForestClassifier

from sklearn.svm import OneClassSVM, SVC

from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV

from sklearn.metrics import confusion_matrix, classification_report, silhouette_score, roc_curve, auc

from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score

from sklearn.manifold import TSNE
```

Prétraitement : StandardScaler pour normaliser les données

Réduction de dimension : PCA (ACP)

Clustering / Anomalies :

- KMeans, DBSCAN
- IsolationForest, OneClassSVM

Classification : SVC, RandomForestClassifier

Évaluation : courbes ROC, AUC, matrices de confusion, etc.

Split & CV : pour division en train/test, et cross-validation

t-SNE : projection non-linéaire pour visualisation

```
import xgboost as xgb

import math

from scipy.stats import zscore
```

xgboost : bibliothèque optimisée pour le **boosting** (arbre de décision)

zscore : standardisation basée sur la moyenne et l'écart-type

```
import tensorflow as tf

from tensorflow.keras.layers import Dense, Input, Dropout

from tensorflow.keras.models import Model

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import EarlyStopping
```

Bibliothèque Keras/Tensorflow pour construire un Autoencoder :

- Input, Dense, Dropout : couches du réseau
- Model : définition complète
- Adam : optimiseur
- EarlyStopping : arrête l'entraînement si la validation stagne

```
import matplotlib.gridspec as gridspec

import warnings

warnings.filterwarnings('ignore')
```

Suppression des **warnings** pour une sortie console propre

gridspec : agencement avancé des sous-graphiques

Explication :

Ce script permet une comparaison systématique de différentes approches de détection d'anomalies. Les méthodes non supervisées sont utiles quand on n'a pas d'étiquettes, tandis que les méthodes supervisées offrent généralement de meilleures performances quand des données étiquetées sont disponibles.

2. Fonction create_autoencoder:

```
def create_autoencoder(input_dim, encoding_dim=3, dropout_rate=0.2):




    Args:

        input_dim (int): Dimension des données d'entrée

        encoding_dim (int): Dimension de la couche latente

        dropout_rate (float): Taux de dropout pour la régularisation


    Returns:

        tuple: (autoencoder, encoder_model) - modèle complet et encodeur seul
```

Cette fonction retourne :

- un autoencoder complet (encodeur + décodeur)
- un modèle encoder seul (pour projeter dans l'espace latent)

```
    input_layer = Input(shape=(input_dim,))
```

Déclaration de l'entrée du réseau de neurones. input_dim est le nombre de features (colonnes numériques).

```
    encoder = Dense(int(input_dim * 0.8), activation='relu', kernel_initializer='he_normal')(input_layer)
    encoder = Dropout(dropout_rate)(encoder)
```

1ère couche dense (80% des neurones d'entrée) + Dropout

- Dropout coupe aléatoirement 20% des neurones pour éviter l'overfitting
- he_normal est une initialisation performante pour les activations ReLU

```
        encoder = Dense(int(input_dim * 0.6), activation='relu', kernel_initializer='he_normal')(encoder)
    encoder = Dropout(dropout_rate)(encoder)
```

2ème couche, plus étroite (60%)

```
        encoder = Dense(int(input_dim * 0.4), activation='relu', kernel_initializer='he_normal')(encoder)
    encoder = Dropout(dropout_rate)(encoder)
```

3ème couche, plus étroite encore (40%)

```
    encoder = Dense(encoding_dim, activation='relu', kernel_initializer='he_normal')(encoder)
```

Dernière couche d'encodage = goulot d'étranglement : réduit à encoding_dim (par défaut 3)

```
    bottleneck = encoder
```

On garde ce vecteur comme représentation latente

```
    decoder = Dense(int(input_dim * 0.4), activation='relu', kernel_initializer='he_normal')(bottleneck)
    decoder = Dropout(dropout_rate)(decoder)
```

Couche miroir de la dernière couche de l’encodeur

```
decoder = Dense(int(input_dim * 0.6), activation='relu', kernel_initializer='he_normal')(decoder)
decoder = Dropout(dropout_rate)(decoder)

decoder = Dense(int(input_dim * 0.8), activation='relu', kernel_initializer='he_normal')(decoder)
decoder = Dropout(dropout_rate)(decoder)
```

Montée progressive jusqu’à la taille d’entrée

```
decoder = Dense(input_dim, activation='sigmoid', kernel_initializer='glorot_normal')(decoder)
```

Sortie finale, dimension = entrée. On utilise sigmoid pour forcer les sorties entre 0 et 1.

```
autoencoder = Model(inputs=input_layer, outputs=decoder)
optimizer = Adam(learning_rate=0.001)
autoencoder.compile(optimizer=optimizer, loss='mean_squared_error')
```

perte MSE (plus l’erreur de reconstruction est grande, plus on suspecte une anomalie)

Adam : optimiseur courant

```
encoder_model = Model(inputs=input_layer, outputs=bottleneck)
```

Création d’un modèle uniquement pour l’encodeur (pour projeter dans l’espace latent)

```
    return autoencoder, encoder_model
```

3. Fonction create_synthetic_data(...)

```
def create_synthetic_data(n_samples=200, n_features=8):
```

Cette fonction génère un DataFrame synthétique avec :

- n_samples lignes (par défaut 200 utilisateurs)
- n_features colonnes (caractéristiques)

🔹 Génération des données

```
    np.random.seed(42)
```

Fixe une **graine aléatoire** pour des résultats reproductibles.

```
    X_normal = np.random.randn(n_samples - 20, n_features)
```

Données "normales" :

- Tirées d'une distribution normale standard (moyenne 0, écart-type 1)
- n_samples - 20 = 180 points environ

```
X_anomaly = np.random.randn(20, n_features) * 2 + 3
```

Données "anormales" :

- Distribution **plus dispersée** (* 2) et **centrée autour de 3**

- Donne des comportements très différents

```
X = np.vstack([X_normal, X_anomaly])
```

Fusion des données normales et anormales en une seule matrice X

- Création des étiquettes

```
y = np.zeros(n_samples)

y[n_samples - 20:] = 1
```

Création d’un tableau y d’étiquettes :

- 0 = normal
- 1 = anomalie (derniers 20)

- Création du DataFrame

```
feature_names = ['total_activities', 'after_hours_activities', 'weekend_activities',

                 'unique_systems', 'unique_resources', 'activity_types',

                 'activity_entropy', 'temporal_entropy']
```

Noms des colonnes = caractéristiques métier réalistes.

```
df = pd.DataFrame(X, columns=feature_names)
```

Construction du DataFrame à partir des données X et des noms de colonnes

```
df['user'] = [f'user_{i}' for i in range(n_samples)]
```

Ajout d’une colonne user avec des identifiants user_0, user_1, ...

```
df['is_anomaly'] = y
```

Ajout de la colonne **cible** (0 ou 1) pour identifier les anomalies

◆ Ajout de colonnes dérivées

```
df['after_hours_ratio'] = df['after_hours_activities'] / (df['total_activities'].abs() + 1)

df['weekend_ratio'] = df['weekend_activities'] / (df['total_activities'].abs() + 1)
```

On calcule deux **ratios supplémentaires** :

- Activités hors heures ouvrables / total
- Activités en weekend / total

On ajoute +1 au dénominateur pour éviter les divisions par 0.

Retour

```
return df
```

```
    return df
```

Retour du DataFrame complet avec 200 utilisateurs et une dizaine de colonnes.

4. Fonction unsupervised_analysis(df, contamination=0.1)

But

Appliquer plusieurs **modèles non supervisés** pour détecter les anomalies dans le DataFrame df.

Préparation des données

```
output_dir = os.path.join(os.path.dirname(__file__), 'visualisations')

os.makedirs(output_dir, exist_ok=True)
```

Création du dossier de sortie visualisations/ pour sauvegarder les graphiques.

```
plt.style.use('default')

colors = ['#FF9999', '#66B2FF', '#99FF99', '#FFCC99', '#FF99CC']

plt.rcParams['figure.figsize'] = (12, 8)

plt.rcParams['font.size'] = 10
```

Configuration globale de matplotlib pour l’apparence des graphiques.

```
numeric_columns = df.select_dtypes(include=['int64', 'float64']).columns.drop(['is_anomaly'], errors='ignore')

X = df[numeric_columns].values

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)
```

Prétraitement :

- Sélection des colonnes **numériques**
- Suppression de is_anomaly (car non utilisée pour l'entraînement)
- Standardisation (StandardScaler) = moyenne 0, variance 1

```
results = {}
```

Dictionnaire pour stocker les résultats de chaque modèle.

4.1. Isolation Forest

```
print("Exécution de l'Isolation Forest...")
```

Affiche dans la console le modèle en cours

```
iso_forest = IsolationForest(
    contamination=contamination,       # proportion estimée d'anomalies
    n_estimators=200,                  # nombre d'arbres
    max_samples='auto',                # nombre de points utilisés par arbre
    max_features=1.0,                  # 100% des colonnes utilisées
    bootstrap=True,                    # échantillonnage avec remplacement
    n_jobs=-1,                         # parallélise sur tous les cœurs
    random_state=42                    # reproductibilité
```

Configuration **optimisée** d'un IsolationForest, un algorithme basé sur des arbres qui isole facilement les points rares (anomalies).

```
y_pred_iso = iso_forest.fit_predict(X_scaled)
```

Entraîne le modèle et prédit pour chaque point :

- -1 si anomalie
- 1 si normal

```
dfscores_iso = -iso_forest.decision_function(X_scaled)
```

Donne un **score** pour chaque point (plus il est élevé, plus c'est une anomalie). On prend l'opposé (-) car plus **proche de zéro = normal**.

```
results['isolation_forest'] = {
    'predictions': np.where(y_pred_iso == -1, 1, 0),  # Convertit en 0/1
    'scores': scores_iso,
    'model': iso_forest
}
```

Sauvegarde des résultats :

- predictions : tableau binaire (1 = anomalie)
- scores : mesure de confiance
- model : l'objet IsolationForest lui-même

4.2 One-Class SVM (ocsvm)

```
print("Exécution de One-Class SVM...")
```

print("Exécution de One-Class SVM...")

Affiche le modèle en cours d'exécution.

```
rocsvm = OneClassSVM(
    kernel="rbf",               # noyau gaussien, le plus courant
    nu=contamination,           # proportion estimée d'anomalies
    gamma='scale',              # auto-ajustement de gamma selon la variance
    cache_size=500,             # mémoire cache pour l'optimisation
    max_iter=-1,                # pas de limite d'itérations
    shrinking=True,             # active une heuristique d'optimisation
    tol=1e-4                    # tolérance pour la convergence
```

Création d'un modèle OneClassSVM :

- Apprend les données **normales** et rejette les anomalies
- Le **noyau RBF** permet de modéliser des frontières complexes

```
y_pred_ocsvm = ocsvm.fit_predict(X_scaled)
scores_ocsvm = -ocsvm.decision_function(X_scaled)
```

fit_predict donne 1 pour normal, -1 pour anomalie
decision_function donne un score (on inverse pour avoir "plus grand = plus suspect")

```
results['ocsvm'] = {
    'predictions': np.where(y_pred_ocsvm == -1, 1, 0),
    'scores': scores_ocsvm,
    'model': ocsvm
}
```

4.3 K-Means

```
print("Exécution de K-Means...")
```

Affiche le modèle KMeans en cours.

```
kmeans = KMeans(
    n_clusters=2,
    init='k-means++',
    n_init=20,
    max_iter=500,
    tol=1e-6,
    algorithm='elkan',
    random_state=42
)
```

KMeans va tenter de séparer les données en **2 groupes** :

- Normal et anomalie
- Le cluster minoritaire est supposé représenter les anomalies

```
y_pred_kmeans = kmeans.fit_predict(X_scaled)
counts = np.bincount(y_pred_kmeans)
anomaly_cluster = np.argmin(counts)
```

Trouve le **cluster avec le moins de points** = considéré comme anomalie

```
distances = np.min(kmeans.transform(X_scaled), axis=1)
distances = (distances - np.min(distances)) / (np.max(distances) - np.min(distances))
```

Calcule la **distance au centroïde le plus proche**

Normalise les distances entre 0 et 1

```
results['kmeans'] = {
    'predictions': np.where(y_pred_kmeans == anomaly_cluster, 1, 0),
    'scores': distances,
    'model': kmeans
}
```

4.4 DBSCAN

```
print("Exécution de DBSCAN...")
```

Algorithme basé sur la **densité des points** :

- Points isolés (hors des clusters) = anomalies

```
dbscan = DBSCAN(
    eps=0.5,
    min_samples=max(5, int(len(X_scaled) * 0.01)),
    metric='euclidean',
    algorithm='auto',
    leaf_size=30,
    n_jobs=-1
)
```

eps = rayon de voisinage
min_samples = nb minimum de voisins pour former un cluster
Points assignés -1 = bruit → anomalies

```
y_pred_dbscan = dbscan.fit_predict(X_scaled)
core_samples_mask = np.zeros_like(y_pred_dbscan, dtype=bool)
core_samples_mask[dbscan.core_sample_indices_] = True
```

Masque les points "cœur" du cluster

```
from sklearn.metrics.pairwise import euclidean_distances
distances = np.zeros(len(X_scaled))
for i in range(len(X_scaled)):
    if y_pred_dbscan[i] == -1:
        min_dist = np.min(euclidean_distances([X_scaled[i]], X_scaled[core_samples_mask]))
        distances[i] = min_dist
    else:
        cluster_points = X_scaled[y_pred_dbscan == y_pred_dbscan[i]]
        if len(cluster_points) > 0:
            distances[i] = np.mean(euclidean_distances([X_scaled[i]], cluster_points))
```

Score = distance au cluster ou à un point central

```
distances = (distances - np.min(distances)) / (np.max(distances) - np.min(distances) + 1e-10)
```

Normalisation min-max des scores

```
rresults['dbscan'] = {
    'predictions': np.where(y_pred_dbscan == -1, 1, 0),
    'scores': distances,
    'model': dbscan
}
```

4.5 Autoencoder

```
print("Exécution de l'Autoencoder...")

input_dim = X_scaled.shape[1]

autoencoder, encoder = create_autoencoder(input_dim)
```

Création du réseau neuronal avec la fonction vue précédemment

```
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=0.0001)
```

Ajout de callbacks pour :

- **Arrêter tôt** si la perte de validation stagne
- **Réduire le learning rate** si nécessaire

```
X_train, X_val = train_test_split(X_scaled, test_size=0.2, random_state=42)
```

Séparation en jeu d'entraînement et validation

```
history = autoencoder.fit(
    X_train, X_train,
    epochs=100,
    batch_size=32,
    validation_data=(X_val, X_val),
    callbacks=[early_stopping, reduce_lr],
    verbose=0
)
```

Entraînement : on apprend à reconstruire les données normales

```
reconstructed = autoencoder.predict(X_scaled)

mse = np.mean(np.power(X_scaled - reconstructed, 2), axis=1)

threshold = np.percentile(mse, (1 - contamination) * 100)
```

Calcul de l'**erreur de reconstruction** (MSE)
Définition d'un **seuil** basé sur le percentile (ici 90% = normal)

```
results['autoencoder'] = {
    'predictions': np.where(mse > threshold, 1, 0),
    'scores': mse,
    'model': autoencoder,
    'encoder': encoder,
    'threshold': threshold
}
```

4.6 Vote majoritaire (Ensemble)

```
print("Application du système de vote majoritaire...")
```

Combine les prédictions de tous les modèles

```
all_predictions = np.column_stack([
    results['isolation_forest']['predictions'],
    results['ocsvm']['predictions'],
    results['kmeans']['predictions'],
    results['dbscan']['predictions'],
    results['autoencoder']['predictions']
])
```

Matrice de taille (n_samples, 5) contenant les prédictions

```
ensemble_predictions = np.sum(all_predictions, axis=1)
ensemble_predictions = np.where(ensemble_predictions >= 3, 1, 0)
```

Si au moins **3 modèles sur 5 détectent une anomalie**, on considère que c'est une vraie anomalie

```
ensemble_scores = np.zeros(len(X_scaled))
for i in range(len(X_scaled)):
    weights = [0.25, 0.2, 0.15, 0.2, 0.2]
    normalized_scores = [
        results['isolation_forest']['scores'][i],
        results['ocsvm']['scores'][i],
        results['kmeans']['scores'][i],
        results['dbscan']['scores'][i],
        results['autoencoder']['scores'][i]
    ]
    ensemble_scores[i] = np.average(normalized_scores, weights=weights)
```

```
results['ensemble'] = {
    'predictions': ensemble_predictions,
    'scores': ensemble_scores
}
```

Ensuite, la fonction continue avec la **visualisation** :

- **ACP (PCA)** pour réduire les dimensions à 2
- Création de multiples graphiques :
  - Courbes ROC
  - Distributions de scores
  - Matrices de confusion
  - Heatmap de corrélation
  - Sauvegarde des résultats dans un DataFrame final

5. Fonction supervised_analysis(df)

Vérification et préparation

```
if 'is_anomaly' not in df.columns:
    raise ValueError("Pas d'étiquettes disponibles pour l'apprentissage supervisé")
```

Si la colonne is_anomaly (les étiquettes) n'existe pas, on arrête : pas d'apprentissage possible.

```
numeric_columns = df.select_dtypes(include=['int64', 'float64']).columns.drop(['is_anomaly'], errors='ignore')
X = df[numeric_columns].values
y = df['is_anomaly'].values
```

Séparation des features (X) et de la cible (y)

```
scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)
```

Normalisation des données

Rééquilibrage des classes avec SMOTE + sous-échantillonnage

```
from imblearn.over_sampling import SMOTE

from imblearn.under_sampling import RandomUnderSampler

from imblearn.pipeline import Pipeline
```

Importation d’outils pour gérer le déséquilibre de classes

```
sampling_pipeline = Pipeline([

    ('over', SMOTE(sampling_strategy=0.7, random_state=42)),

    ('under', RandomUnderSampler(sampling_strategy=0.8, random_state=42))

])
```

Pipeline :

- SMOTE : crée des exemples synthétiques d’anomalies
- Sous-échantillonnage : réduit le nombre de normaux

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42, stratify=y)

X_train_resampled, y_train_resampled = sampling_pipeline.fit_resample(X_train, y_train)
```

Split en train/test puis application du pipeline de rééchantillonnage

```
results = {}
```

Dictionnaire pour stocker les résultats

5.1 Random Forest

```
print("Entraînement de Random Forest...")
```

```
rf_params = {
    'n_estimators': [200],
    'max_depth': [10],
    'min_samples_split': [4],
    'min_samples_leaf': [2],
    'max_features': ['sqrt'],
    'class_weight': ['balanced']
}
```

Paramètres optimisés pour GridSearchCV

```
rf = RandomForestClassifier(random_state=42, n_jobs=-1)
rf_cv = GridSearchCV(rf, rf_params, cv=3, scoring='f1', n_jobs=-1)
rf_cv.fit(X_train_resampled, y_train_resampled)
rf = rf_cv.best_estimator_
```

GridSearch sur 3 plis de validation croisée. Récupération du meilleur modèle.

```
y_pred_rf = rf.predict(X_test)

y_prob_rf = rf.predict_proba(X_test)[:, 1]
```

Prédiction + probabilités (pour courbes ROC)

```
results['random_forest'] = {
    'model': rf,
    'y_true': y_test,
    'y_pred': y_pred_rf,
    'y_prob': y_prob_rf,
    'cv_scores': cross_val_score(rf, X_scaled, y, cv=3, scoring='roc_auc')
}
```

5.2 SVM

```
print("Entraînement de SVM...")
```

```
svm_params = {
    'kernel': ['rbf'],
    'C': [10.0],
    'gamma': ['scale'],
    'class_weight': ['balanced'],
    'probability': [True]
}
```

SVM avec noyau RBF + calcul de probabilité

```
svm = SVC(random_state=42)
svm_cv = GridSearchCV(svm, svm_params, cv=3, scoring='f1', n_jobs=-1)
svm_cv.fit(X_train_resampled, y_train_resampled)
svm = svm_cv.best_estimator_
```

Recherche des meilleurs hyperparamètres

```
y_pred_svm = svm.predict(X_test)
y_prob_svm = svm.predict_proba(X_test)[:, 1]
```

Prédictions + proba

```
results['svm'] = {
    'model': svm,
    'y_true': y_test,
    'y_pred': y_pred_svm,
    'y_prob': y_prob_svm,
    'cv_scores': cross_val_score(svm, X_scaled, y, cv=3, scoring='roc_auc')
}
```

5.3 XGBoost

```
print("Entraînement de XGBoost...")
```

```
xgb_params = {
    'learning_rate': [0.01],
    'n_estimators': [300],
    'max_depth': [6],
    'min_child_weight': [2],
    'subsample': [0.8],
    'colsample_bytree': [0.8],
    'gamma': [0.1],
    'reg_alpha': [0.1],
    'reg_lambda': [1.0]
}
```

Paramètres classiques pour un modèle XGBoost équilibré

```
xgb_model = xgb.XGBClassifier(
    objective='binary:logistic',
    scale_pos_weight=1,
    random_state=42,
    eval_metric='logloss'
)
```

XGBoost est très performant pour des problèmes déséquilibrés

```
eval_set = [(X_train_resampled, y_train_resampled), (X_test, y_test)]
xgb_model.fit(X_train_resampled, y_train_resampled, eval_set=eval_set, verbose=False)
```

Entraînement sur le jeu d'entraînement + test en suivi

```
y_pred_xgb = xgb_model.predict(X_test)
y_prob_xgb = xgb_model.predict_proba(X_test)[:, 1]
```

```
results['xgboost'] = {
    'model': xgb_model,
    'y_true': y_test,
    'y_pred': y_pred_xgb,
    'y_prob': y_prob_xgb,
    'cv_scores': cross_val_score(xgb_model, X_scaled, y, cv=3, scoring='roc_auc')
}
```

Construction du DataFrame final

```
eval_df = pd.DataFrame({
    'real_anomaly': y_test,
    'RF_pred': y_pred_rf,
    'RF_prob': y_prob_rf,
    'SVM_pred': y_pred_svm,
    'SVM_prob': y_prob_svm,
    'XGB_pred': y_pred_xgb,
    'XGB_prob': y_prob_xgb
})
```

Contient les vraies valeurs + prédictions de tous les modèles

```
return results, eval_df
```

6. Fonction create_visualizations(...)

Objectif

Créer **toutes les visualisations** comparant les modèles :

- Non supervisés (unsupervised_results)
- Supervisés (supervised_results)
- Avec réduction de dimension (X_pca)
- En se basant sur les DataFrames d'analyse (df, unsup_df, sup_df)

Organisation

La fonction est très riche, donc on va la diviser en **4 blocs principaux** :

Bloc 1 : **Comparaison des performances supervisées / non supervisées**

```
plt.figure(figsize=(20, 12))

gs = gridspec.GridSpec(2, 2)
```

Crée une figure avec 2x2 sous-graphiques.

```
# Performance non supervisée

plt.subplot(gs[0, 0])
```

Barplot des scores (accuracy, precision, recall, f1) pour :

- Isolation Forest, OCSVM, KMeans, DBSCAN, Autoencoder

```
# Performance supervisée

plt.subplot(gs[0, 1])
```

Même chose pour :

- Random Forest, SVM, XGBoost

```
# Courbes ROC supervisées

plt.subplot(gs[1, 0])
```

Tracé des courbes ROC (supervisés)

```
# Distributions des scores non supervisés

plt.subplot(gs[1, 1])
```

Courbes de densité (KDE) des scores d’anomalie

Bloc 2 : **Visualisation des scores par méthode non supervisée**

```
plt.figure(figsize=(20, 16))

gs = gridspec.GridSpec(3, 3)
```

5 graphiques KDE :

- Score d’anomalie pour chaque modèle
- Colorié selon l’étiquette prédite (anomalie ou normal)

```
# Visualisation de l’espace latent de l’autoencoder

plt.subplot(gs[1, 2])
```

Affiche les données dans l’espace latent de l’autoencoder (si possible)

```
# Heatmap F1/precision/recall par modèle

plt.subplot(gs[2, :])
```

Comparaison par heatmap des performances non supervisées (vs vérité terrain)

Bloc 3 : **ACP et t-SNE**

```
plt.figure(figsize=(20, 12))

gs = gridspec.GridSpec(2, 3)
```

3 graphiques :

1. Projection **PCA** avec score d'anomalie
2. Projection **t-SNE** avec score d'anomalie
3. Moyenne des **matrices de confusion** des modèles non supervisés

```
# Courbes ROC améliorées

plt.subplot(gs[1, :])
```

Tracé des courbes ROC des modèles non supervisés

Bloc 4 : **Analyse des caractéristiques**

```
plt.figure(figsize=(20, 10))
```

Deux graphiques :

1. **Corrélation entre les features**
2. **Importance des features** via roc_auc_score
3. Bloc 5 (optionnel) : **Comparaison supervisé vs non supervisé**

Si sup_df est fourni :

- Matrices de confusion des modèles supervisés
- Courbes ROC supervisées

7. Fonction main()

But : tout exécuter de A à Z automatiquement

Étapes clés

```
from neo4j_setup import connect_to_neo4j
```

Si tu as le module neo4j_setup, on tente de se connecter avec :

uri = "bolt://localhost:7689"

username = "neo4j"

password = "2004@2004"

➤ Si Neo4j est dispo :

- On récupère les données via extract_features_for_ml()

➤ Sinon :

```
df = create_synthetic_data(n_samples=200, n_features=8)
```

On crée des données factices

➤ Si pas d'étiquettes :

```
solationForest → crée des étiquettes synthétiques
```

IsolationForest → crée des étiquettes synthétiques

Lancement des analyses

```
unsupervised_results, unsup_df, X_pca = unsupervised_analysis(df)

supervised_results, sup_df = supervised_analysis(df)
```

Lance les deux analyses

```
create_visualizations(...)
```

create_visualizations(...)

Génére les visualisations

Résumé final

Affiche dans la console :

- Nombre d'anomalies détectées par modèle
- Scores des modèles supervisés

5. neo4j_visualization.py

Objectif :

Créer des visualisations pour communiquer les résultats des analyses.

Étapes détaillées :

1. **Visualisation des activités après heures** :

```
sns.barplot(x='user', y='after_hours_activity_count', color='darkred')
```

- Barplot montrant le nombre d'activités après heures par utilisateur

2.Visualisation de l'entropie :

```
fig, ax1 = plt.subplots()

ax1.bar(df['user'], df['entropy'], color='blue')

ax2 = ax1.twinx()

ax2.plot(df['user'], df['activity_count'], color='red')
```

- Combinaison barres (entropie) et ligne (nombre d'activités)
- Permet de voir la relation entre régularité et volume d'activité

3.Matrice de corrélation :

```
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
```

- Montre les corrélations entre différentes caractéristiques
- Aide à identifier les variables redondantes ou complémentaires

4. Tableau de bord :

```
fig, axes = plt.subplots(3, 2, figsize=(18, 16))

axes[0,0].bar(df['user'], df['count'], color='darkred')

axes[1,1].bar(df['user'], df['entropy'], color='purple')
```

- Combine plusieurs visualisations en une seule image
- Donne une vue d'ensemble des résultats d'analyse

Explication :

Ces visualisations transforment les résultats analytiques en insights compréhensibles. Le choix des graphiques est adapté aux différents types de données et métriques analysées.