COURS DE SCIENCE DES DONNÉES ## École Nationale de Commerce et de Gestion (ENCG) - 4ème Année --- # PARTIE 2 : APPRENTISSAGE AUTOMATIQUE ET APPLICATIONS AVANCÉES --- ## MODULE 3 : APPRENTISSAGE AUTOMATIQUE (MACHINE LEARNING) ### 3.5.2 Clustering Hiérarchique Le clustering hiérarchique crée une hiérarchie de clusters représentée par un dendrogramme. **Avantages :** - Pas besoin de spécifier k à l'avance - Visualisation intuitive - Plusieurs méthodes de liaison (linkage) ```python from scipy.cluster.hierarchy import dendrogram, linkage from sklearn.cluster import AgglomerativeClustering print("=" * 80) print("CLUSTERING HIÉRARCHIQUE") print("=" * 80) # Utiliser un échantillon pour la visualisation sample_size = 50 sample_indices = np.random.choice(len(X_scaled), sample_size, replace=False) X_sample = X_scaled[sample_indices] # Calculer la matrice de liaison Z = linkage(X_sample, method='ward') # Visualiser le dendrogramme plt.figure(figsize=(14, 6)) dendrogram(Z, labels=sample_indices) plt.xlabel('Index des observations') plt.ylabel('Distance') plt.title('Dendrogramme - Clustering Hiérarchique (Ward)') plt.axhline(y=10, color='r', linestyle='--', label='Seuil de coupure') plt.legend() plt.grid(True, alpha=0.3, axis='y') plt.show() # Appliquer le clustering hiérarchique hierarchical = AgglomerativeClustering(n_clusters=3, linkage='ward') hier_clusters = hierarchical.fit_predict(X_scaled) df_iris['Hier_Cluster'] = hier_clusters print("\nComparaison K-Means vs Hiérarchique:") comparison = pd.crosstab(df_iris['Cluster'], df_iris['Hier_Cluster'], rownames=['K-Means'], colnames=['Hiérarchique']) print(comparison) ``` ### 3.5.3 Réduction de Dimensionnalité : Analyse en Composantes Principales (ACP/PCA) L'ACP transforme les variables corrélées en composantes principales non corrélées. **Objectifs :** - Réduire la dimensionnalité - Visualiser des données complexes - Éliminer le bruit - Accélérer les algorithmes **Formule mathématique :** Les composantes principales sont les vecteurs propres de la matrice de covariance des données standardisées. ```python from sklearn.decomposition import PCA print("=" * 80) print("ANALYSE EN COMPOSANTES PRINCIPALES (PCA)") print("=" * 80) # Appliquer PCA pca = PCA() X_pca = pca.fit_transform(X_scaled) # Variance expliquée variance_expliquee = pca.explained_variance_ratio_ variance_cumulee = np.cumsum(variance_expliquee) print("Variance expliquée par composante:") for i, var in enumerate(variance_expliquee): print(f"PC{i+1}: {var:.4f} ({var*100:.2f}%)") print(f"\nVariance cumulée avec 2 composantes: {variance_cumulee[1]:.4f} ({variance_cumulee[1]*100:.2f}%)") # Visualisation fig, axes = plt.subplots(1, 3, figsize=(18, 5)) # 1. Variance expliquée axes[0].bar(range(1, len(variance_expliquee)+1), variance_expliquee, alpha=0.7) axes[0].plot(range(1, len(variance_cumulee)+1), variance_cumulee, 'ro-', linewidth=2, markersize=8, label='Cumulée') axes[0].set_xlabel('Composante Principale') axes[0].set_ylabel('Variance Expliquée') axes[0].set_title('Variance Expliquée par Composante') axes[0].legend() axes[0].grid(True, alpha=0.3) # 2. Projection sur PC1 et PC2 (par espèce réelle) species_map = {species: i for i, species in enumerate(df_iris['species'].unique())} colors = df_iris['species'].map(species_map) scatter = axes[1].scatter(X_pca[:, 0], X_pca[:, 1], c=colors, cmap='viridis', alpha=0.6, s=50) axes[1].set_xlabel(f'PC1 ({variance_expliquee[0]*100:.1f}%)') axes[1].set_ylabel(f'PC2 ({variance_expliquee[1]*100:.1f}%)') axes[1].set_title('Projection PCA - Espèces Réelles') axes[1].grid(True, alpha=0.3) plt.colorbar(scatter, ax=axes[1]) # 3. Projection sur PC1 et PC2 (par cluster) scatter2 = axes[2].scatter(X_pca[:, 0], X_pca[:, 1], c=clusters, cmap='plasma', alpha=0.6, s=50) axes[2].set_xlabel(f'PC1 ({variance_expliquee[0]*100:.1f}%)') axes[2].set_ylabel(f'PC2 ({variance_expliquee[1]*100:.1f}%)') axes[2].set_title('Projection PCA - Clusters K-Means') axes[2].grid(True, alpha=0.3) plt.colorbar(scatter2, ax=axes[2]) plt.tight_layout() plt.show() # Biplot (contribution des variables) print("\nContribution des variables aux composantes principales:") loadings = pca.components_.T * np.sqrt(pca.explained_variance_) loading_df = pd.DataFrame(loadings[:, :2], columns=['PC1', 'PC2'], index=X.columns) print(loading_df) # Visualiser le biplot plt.figure(figsize=(10, 8)) plt.scatter(X_pca[:, 0], X_pca[:, 1], alpha=0.3, s=30) for i, feature in enumerate(X.columns): plt.arrow(0, 0, loadings[i, 0]*3, loadings[i, 1]*3, head_width=0.1, head_length=0.1, fc='red', ec='red') plt.text(loadings[i, 0]*3.2, loadings[i, 1]*3.2, feature, fontsize=12, fontweight='bold') plt.xlabel(f'PC1 ({variance_expliquee[0]*100:.1f}%)') plt.ylabel(f'PC2 ({variance_expliquee[1]*100:.1f}%)') plt.title('Biplot PCA - Variables et Observations') plt.grid(True, alpha=0.3) plt.axhline(y=0, color='k', linewidth=0.5) plt.axvline(x=0, color='k',

linewidth=0.5) plt.show() ``` --- ## 3.6 ÉVALUATION DES MODÈLES ### 3.6.1 Métriques de Performance pour la Classification ```python from sklearn.metrics import (accuracy_score, precision_score, recall_score, f1_score, matthews_corrcoef, cohen_kappa_score) print("=" * 80) print("MÉTRIQUES D'ÉVALUATION - CLASSIFICATION") print("=" * 80) # Utiliser le modèle de régression logistique du Titanic y_pred = log_reg.predict(X_test_scaled) y_pred_proba = log_reg.predict_proba(X_test_scaled)[:, 1] # Calculer toutes les métriques metriques = { 'Accuracy': accuracy_score(y_test, y_pred), 'Precision': precision_score(y_test, y_pred), 'Recall (Sensibilité)': recall_score(y_test, y_pred), 'F1-Score': f1_score(y_test, y_pred), 'MCC': matthews_corrcoef(y_test, y_pred), 'Cohen Kappa': cohen_kappa_score(y_test, y_pred), } print("\nMÉTRIQUES DE PERFORMANCE:") for metric, value in metriques.items(): print(f"{metric:.<30} {value:.4f}") # Matrice de confusion détaillée tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel() print("\nMATRICE DE CONFUSION DÉTAILLÉE:") print(f"Vrais Négatifs (TN): {tn}") print(f"Faux Positifs (FP): {fp}") print(f"Faux Négatifs (FN): {fn}") print(f"Vrais Positifs (TP): {tp}") print("\nMÉTRIQUES CALCULÉES MANUELLEMENT:") print(f"Accuracy = (TP + TN) / Total = {(tp + tn) / (tp + tn + fp + fn):.4f}") print(f"Precision = TP / (TP + FP) = {tp / (tp + fp):.4f}") print(f"Recall = TP / (TP + FN) = {tp / (tp + fn):.4f}") print(f"F1-Score = 2 * (Precision * Recall) / (Precision + Recall)") # Courbe Precision-Recall from sklearn.metrics import precision_recall_curve, average_precision_score precision, recall, thresholds = precision_recall_curve(y_test, y_pred_proba) ap_score = average_precision_score(y_test, y_pred_proba) fig, axes = plt.subplots(1, 2, figsize=(14, 5)) # Courbe Precision-Recall axes[0].plot(recall, precision, linewidth=2, label=f'AP = {ap_score:.2f}') axes[0].set_xlabel('Recall') axes[0].set_ylabel('Precision') axes[0].set_title('Courbe Precision-Recall') axes[0].legend() axes[0].grid(True, alpha=0.3) # Impact du seuil f1_scores = 2 * (precision[:-1] * recall[:-1]) / (precision[:-1] + recall[:-1]) axes[1].plot(thresholds, precision[:-1], label='Precision', linewidth=2) axes[1].plot(thresholds, recall[:-1], label='Recall', linewidth=2) axes[1].plot(thresholds, f1_scores, label='F1-Score', linewidth=2) axes[1].set_xlabel('Seuil de décision') axes[1].set_ylabel('Score') axes[1].set_title('Impact du Seuil sur les Métriques') axes[1].legend() axes[1].grid(True, alpha=0.3) plt.tight_layout() plt.show() ``` ### 3.6.2 Métriques de Performance pour la Régression ```python from sklearn.metrics import mean_absolute_percentage_error, max_error print("=" * 80) print("MÉTRIQUES D'ÉVALUATION - RÉGRESSION") print("=" * 80) # Utiliser le modèle de régression des salaires y_pred_test = model.predict(X_test) # Calculer les métriques mae = mean_absolute_error(y_test, y_pred_test) mse = mean_squared_error(y_test, y_pred_test) rmse = np.sqrt(mse) r2 = r2_score(y_test, y_pred_test) mape = mean_absolute_percentage_error(y_test, y_pred_test) max_err = max_error(y_test, y_pred_test) print("\nMÉTRIQUES DE RÉGRESSION:") print(f"MAE (Mean Absolute Error):..... {mae:.2f}") print(f"MSE (Mean Squared Error):..... {mse:.2f}") print(f"RMSE (Root Mean Squared Error):... {rmse:.2f}") print(f"R² Score:..... {r2:.4f}") print(f"MAPE (Mean Abs Percentage Error):. {mape*100:.2f}%") print(f"Max Error:..... {max_err:.2f}") print("\nINTERPRÉTATION:") print(f"- En moyenne, nos prédictions s'écartent de {mae:.2f} de la réalité") print(f"- Le modèle explique {r2*100:.2f}% de la variance des données") print(f"- L'erreur relative moyenne est de {mape*100:.2f}%") ``` ### 3.6.3 Validation Croisée (Cross-Validation) La validation croisée évalue la performance du modèle sur différents sous-ensembles des données. ```python from sklearn.model_selection import cross_val_score, cross_validate, KFold print("=" * 80) print("VALIDATION CROISÉE") print("=" * 80) # K-Fold Cross-Validation kfold = KFold(n_splits=5, shuffle=True, random_state=42) # Classification avec Régression Logistique scores = cross_val_score(LogisticRegression(max_iter=1000), X_train_scaled, y_train, cv=kfold, scoring='accuracy') print("\nCLASSIFICATION - Régression Logistique (5-fold CV):") print(f"Scores par fold: {scores}") print(f"Accuracy moyenne: {scores.mean():.4f} (+/- {scores.std() * 2:.4f})") # Cross-validation avec plusieurs métriques scoring = ['accuracy', 'precision', 'recall', 'f1', 'roc_auc'] cv_results = cross_validate(LogisticRegression(max_iter=1000), X_train_scaled, y_train, cv=kfold, scoring=scoring) print("\nMÉTRIQUES MULTIPLES:") for metric in scoring: scores = cv_results[f'test_{metric}'] print(f"{metric:.<20} {scores.mean():.4f} (+/- {scores.std() * 2:.4f})") # Visualisation plt.figure(figsize=(10, 6)) metrics_means = [cv_results[f'test_{m}'].mean() for m in scoring] metrics_stds = [cv_results[f'test_{m}'].std() for m in scoring] x_pos = np.arange(len(scoring)) plt.bar(x_pos, metrics_means, yerr=metrics_stds, alpha=0.7,

capsize=10) plt.xticks(x_pos, scoring) plt.ylabel('Score') plt.title('Performance avec Validation Croisée (5-fold)') plt.ylim([0, 1]) plt.grid(True, alpha=0.3, axis='y') plt.show() ``` ### 3.6.4 Surapprentissage et Sous-apprentissage **Surapprentissage (Overfitting) :** Le modèle performe bien sur les données d'entraînement mais mal sur de nouvelles données. **Sous-apprentissage (Underfitting) :** Le modèle ne capture pas les patterns des données. ```python from sklearn.model_selection import learning_curve print("=" * 80) print("ANALYSE SURAPPRENTISSAGE / SOUS-APPRENTISSAGE") print("=" * 80) # Calculer les courbes d'apprentissage train_sizes, train_scores, val_scores = learning_curve(LogisticRegression(max_iter=1000), X_train_scaled, y_train, train_sizes=np.linspace(0.1, 1.0, 10), cv=5, scoring='accuracy', random_state=42) # Calculer moyennes et écarts-types train_mean = train_scores.mean(axis=1) train_std = train_scores.std(axis=1) val_mean = val_scores.mean(axis=1) val_std = val_scores.std(axis=1) # Visualisation plt.figure(figsize=(10, 6)) plt.plot(train_sizes, train_mean, label='Score Train', marker='o', linewidth=2) plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.15) plt.plot(train_sizes, val_mean, label='Score Validation', marker='s', linewidth=2) plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.15) plt.xlabel('Taille de l\'ensemble d\'entraînement') plt.ylabel('Accuracy') plt.title('Courbes d\'Apprentissage - Détection du Surapprentissage') plt.legend(loc='best') plt.grid(True, alpha=0.3) plt.show() print("\nINTERPRÉTATION:") gap = train_mean[-1] - val_mean[-1] if gap > 0.1: print(f"⚠️ SURAPPRENTISSAGE détecté (gap = {gap:.4f})") print("Solutions: régularisation, plus de données, réduire la complexité") elif val_mean[-1] < 0.7: print("⚠️ SOUS-APPRENTISSAGE détecté") print("Solutions: modèle plus complexe, plus de features, moins de régularisation") else: print("✅ Le modèle semble bien équilibré") # Exemple de régularisation print("\n" + "="*80) print("IMPACT DE LA RÉGULARISATION") print("="*80) C_values = [0.001, 0.01, 0.1, 1, 10, 100] train_scores = [] test_scores = [] for C in C_values: lr = LogisticRegression(C=C, max_iter=1000, random_state=42) lr.fit(X_train_scaled, y_train) train_scores.append(lr.score(X_train_scaled, y_train)) test_scores.append(lr.score(X_test_scaled, y_test)) plt.figure(figsize=(10, 6)) plt.plot(C_values, train_scores, label='Train', marker='o', linewidth=2) plt.plot(C_values, test_scores, label='Test', marker='s', linewidth=2) plt.xscale('log') plt.xlabel('Paramètre C (inverse de la régularisation)') plt.ylabel('Accuracy') plt.title('Impact de la Régularisation sur la Performance') plt.legend() plt.grid(True, alpha=0.3) plt.show() print(f"\nMeilleur C: {C_values[np.argmax(test_scores)]}") ``` --- ## 3.7 OPTIMISATION DES HYPERPARAMÈTRES ### 3.7.1 Grid Search Grid Search teste toutes les combinaisons possibles d'hyperparamètres. ```python from sklearn.model_selection import GridSearchCV print("=" * 80) print("OPTIMISATION PAR GRID SEARCH") print("=" * 80) # Définir la grille de paramètres param_grid = { 'C': [0.001, 0.01, 0.1, 1, 10, 100], 'penalty': ['l1', 'l2'], 'solver': ['liblinear'] } # Créer le Grid Search grid_search = GridSearchCV(LogisticRegression(max_iter=1000, random_state=42), param_grid, cv=5, scoring='accuracy', n_jobs=-1, verbose=1) # Entraîner print("\nRecherche des meilleurs paramètres en cours...") grid_search.fit(X_train_scaled, y_train) # Résultats print("\nMEILLEURS PARAMÈTRES:") print(grid_search.best_params_) print(f"\nMeilleur score (CV): {grid_search.best_score_:.4f}") print(f"Score sur test: {grid_search.score(X_test_scaled, y_test):.4f}") # Visualiser les résultats results_df = pd.DataFrame(grid_search.cv_results_) pivot_table = results_df.pivot_table(values='mean_test_score', index='param_C', columns='param_penalty') plt.figure(figsize=(10, 6)) sns.heatmap(pivot_table, annot=True, fmt='.4f', cmap='YlGnBu') plt.title('Grid Search - Accuracy par Combinaison de Paramètres') plt.xlabel('Penalty') plt.ylabel('C') plt.show() ``` ### 3.7.2 Random Search Random Search échantillonne aléatoirement l'espace des hyperparamètres. ```python from sklearn.model_selection import RandomizedSearchCV from scipy.stats import uniform, randint print("=" * 80) print("OPTIMISATION PAR RANDOM SEARCH") print("=" * 80) # Définir les distributions de paramètres param_distributions = { 'n_estimators': randint(50, 500), 'max_depth': randint(3, 20), 'min_samples_split': randint(2, 20), 'min_samples_leaf': randint(1, 10), 'max_features': ['sqrt', 'log2', None] } # Créer le Random Search random_search = RandomizedSearchCV(RandomForestClassifier(random_state=42), param_distributions, n_iter=50, # Nombre d'itérations cv=5, scoring='accuracy', random_state=42, n_jobs=-1, verbose=1) # Entraîner print("\nRecherche aléatoire en cours...") random_search.fit(X_train, y_train) # Résultats print("\nMEILLEURS PARAMÈTRES:") for

param, value in random_search.best_params_.items(): print(f" {param}: {value}") print(f"\nMeilleur score (CV): {random_search.best_score_:.4f}") print(f"Score sur test: {random_search.score(X_test, y_test):.4f}") # Comparer modèle de base vs optimisé rf_base = RandomForestClassifier(random_state=42) rf_base.fit(X_train, y_train) print("\nCOMPARAISON:") print(f"Modèle de base (test): {rf_base.score(X_test, y_test):.4f}") print(f"Modèle optimisé (test): {random_search.score(X_test, y_test):.4f}") print(f"Amélioration: {(random_search.score(X_test, y_test) - rf_base.score(X_test, y_test))*100:.2f}%") ``` --- ## TRAVAUX PRATIQUES 2 : PROJET COMPLET DE MACHINE LEARNING ### Objectif Développer un système complet de prédiction du churn client pour une entreprise de télécommunications. ### Dataset ```python import pandas as pd import numpy as np import matplotlib.pyplot as plt import seaborn as sns from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV from sklearn.preprocessing import StandardScaler, LabelEncoder from sklearn.linear_model import LogisticRegression from sklearn.tree import DecisionTreeClassifier from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score, roc_curve print("=" * 80) print("PROJET ML: PRÉDICTION DU CHURN CLIENT - TÉLÉCOMMUNICATIONS") print("=" * 80) # Charger les données url = "https://raw.githubusercontent.com/IBM/telco-customer-churn-on-icp4d/master/data/Telco-Customer-Churn.csv" df_churn = pd.read_csv(url) print("\n1. EXPLORATION DES DONNÉES") print("-" * 80) print(f"Shape: {df_churn.shape}") print(f"\nPremières lignes:") print(df_churn.head()) # Informations sur les colonnes print(f"\nInfo:") df_churn.info() # Statistiques descriptives print(f"\nStatistiques:") print(df_churn.describe()) # Distribution du churn print(f"\nDistribution du Churn:") print(df_churn['Churn'].value_counts(normalize=True)) print("\n2. NETTOYAGE DES DONNÉES") print("-" * 80) # Convertir TotalCharges en numérique df_churn['TotalCharges'] = pd.to_numeric(df_churn['TotalCharges'], errors='coerce') # Gérer les valeurs manquantes print(f"Valeurs manquantes:") print(df_churn.isnull().sum()[df_churn.isnull().sum() > 0]) df_churn['TotalCharges'].fillna(df_churn['TotalCharges'].median(), inplace=True) # Supprimer customerID df_churn.drop('customerID', axis=1, inplace=True) print("\n3. FEATURE ENGINEERING") print("-" * 80) # Créer de nouvelles features df_churn['ChargePerMonth'] = df_churn['TotalCharges'] / (df_churn['tenure'] + 1) df_churn['HasMultipleServices'] = ((df_churn['OnlineSecurity'] == 'Yes') | (df_churn['OnlineBackup'] == 'Yes') | (df_churn['DeviceProtection'] == 'Yes')).astype(int) print("Nouvelles features créées: ChargePerMonth, HasMultipleServices") print("\n4. ENCODAGE DES VARIABLES") print("-" * 80) # Variables catégorielles binaires binary_cols = ['gender', 'Partner', 'Dependents', 'PhoneService', 'PaperlessBilling', 'Churn'] le = LabelEncoder() for col in binary_cols: df_churn[f'{col}_encoded'] = le.fit_transform(df_churn[col]) # Variables catégorielles multi-classes multi_cols = ['InternetService', 'Contract', 'PaymentMethod'] df_churn = pd.get_dummies(df_churn, columns=multi_cols, drop_first=True) print(f"Nombre de colonnes après encodage: {df_churn.shape[1]}") print("\n5. PRÉPARATION POUR LA MODÉLISATION") print("-" * 80) # Sélectionner les features feature_cols = [col for col in df_churn.columns if col not in ['gender', 'Partner', 'Dependents', 'PhoneService', 'PaperlessBilling', 'Churn', 'MultipleLines', 'OnlineSecurity', 'OnlineBackup', 'DeviceProtection', 'TechSupport', 'StreamingTV', 'StreamingMovies']] X = df_churn[feature_cols] y = df_churn['Churn_encoded'] print(f"Features: {len(feature_cols)}") print(f"Observations: {len(X)}") # Séparer train/test X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y) # Standardiser scaler = StandardScaler() X_train_scaled = scaler.fit_transform(X_train) X_test_scaled = scaler.transform(X_test) print(f"Train set: {X_train.shape}") print(f"Test set: {X_test.shape}") print("\n6. ENTRAÎNEMENT DE PLUSIEURS MODÈLES") print("-" * 80) # Dictionnaire de modèles models = { 'Logistic Regression': LogisticRegression(max_iter=1000, random_state=42), 'Decision Tree': DecisionTreeClassifier(random_state=42), 'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42), 'Gradient Boosting': GradientBoostingClassifier(random_state=42) } # Entraîner et évaluer chaque modèle results = {} for name, model in models.items(): print(f"\nEntraînement: {name}") # Entraîner if name == 'Logistic Regression': model.fit(X_train_scaled, y_train) y_pred = model.predict(X_test_scaled) y_pred_proba = model.predict_proba(X_test_scaled)[:, 1] else: model.fit(X_train, y_train) y_pred = model.predict(X_test) y_pred_proba = model.predict_proba(X_test)[:, 1] # Métriques from

sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score results[name] = { 'Accuracy': accuracy_score(y_test, y_pred), 'Precision': precision_score(y_test, y_pred), 'Recall': recall_score(y_test, y_pred), 'F1-Score': f1_score(y_test, y_pred), 'ROC-AUC': roc_auc_score(y_test, y_pred_proba) } print(f" Accuracy: {results[name]['Accuracy']:.4f}") print(f" ROC-AUC: {results[name]['ROC-AUC']:.4f}") # Tableau récapitulatif results_df = pd.DataFrame(results).T print("\n7. COMPARAISON DES MODÈLES") print("-" * 80) print(results_df) # Visualiser les résultats fig, axes = plt.subplots(1, 2, figsize=(161 Introduction au Machine Learning Le **Machine Learning (ML)** est une branche de l'intelligence artificielle qui permet aux ordinateurs d'apprendre à partir de données sans être explicitement programmés. **Différence avec la programmation traditionnelle :** | Programmation Traditionnelle | Machine Learning | |----------|----------| | Règles explicites définies par le développeur | Règles apprises automatiquement | | Logique if-then-else | Apprentissage par patterns | | Difficile à adapter | S'améliore avec plus de données | | Exemple : Calcul de TVA | Exemple : Détection de spam | **Types d'apprentissage :** 1. **Apprentissage Supervisé** : Apprendre à partir de données étiquetées - Régression : Prédire une valeur continue - Classification : Prédire une catégorie 2. **Apprentissage Non Supervisé** : Découvrir des structures dans des données non étiquetées - Clustering : Regrouper des observations similaires - Réduction de dimensionnalité : Simplifier les données 3. **Apprentissage par Renforcement** : Apprendre par essai-erreur (hors programme) ### 3.2 Préparation des Données pour le ML **Workflow typique :** ```python import pandas as pd import numpy as np from sklearn.model_selection import train_test_split from sklearn.preprocessing import StandardScaler, LabelEncoder import matplotlib.pyplot as plt import seaborn as sns # Charger des données url = "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv" df = pd.read_csv(url) print("=" * 80) print("PRÉPARATION DES DONNÉES POUR LE MACHINE LEARNING") print("=" * 80) # 1. EXPLORATION INITIALE print("\n1. APERÇU DES DONNÉES") print("-" * 80) print(df.head()) print(f"\nDimensions: {df.shape}") print(f"\nTypes de données:\n{df.dtypes}") # 2. GESTION DES VALEURS MANQUANTES print("\n2. VALEURS MANQUANTES") print("-" * 80) print(df.isnull().sum()) # Stratégies de traitement # a) Imputation par la médiane pour Age df['Age'].fillna(df['Age'].median(), inplace=True) # b) Imputation par le mode pour Embarked df['Embarked'].fillna(df['Embarked'].mode()[0], inplace=True) # c) Supprimer Cabin (trop de valeurs manquantes) df.drop('Cabin', axis=1, inplace=True) print("\nAprès traitement:") print(df.isnull().sum()) # 3. ENCODAGE DES VARIABLES CATÉGORIELLES print("\n3. ENCODAGE DES VARIABLES CATÉGORIELLES") print("-" * 80) # Label Encoding pour variables binaires le = LabelEncoder() df['Sex_encoded'] = le.fit_transform(df['Sex']) # One-Hot Encoding pour variables avec plusieurs catégories df = pd.get_dummies(df, columns=['Embarked'], prefix='Embarked') print("Variables encodées:") print(df[['Sex', 'Sex_encoded']].head()) print(df[[col for col in df.columns if 'Embarked' in col]].head()) # 4. FEATURE ENGINEERING print("\n4. FEATURE ENGINEERING") print("-" * 80) # Créer de nouvelles features df['FamilySize'] = df['SibSp'] + df['Parch'] + 1 df['IsAlone'] = (df['FamilySize'] == 1).astype(int) df['Title'] = df['Name'].str.extract(' ([A-Za-z]+)\.', expand=False) print("Nouvelles features créées:") print(df[['SibSp', 'Parch', 'FamilySize', 'IsAlone']].head()) print(f"\nTitres uniques: {df['Title'].unique()}") # 5. SÉLECTION DES FEATURES print("\n5. SÉLECTION DES FEATURES") print("-" * 80) # Features pour la modélisation features = ['Pclass', 'Sex_encoded', 'Age', 'SibSp', 'Parch', 'Fare', 'FamilySize', 'IsAlone'] + \ [col for col in df.columns if 'Embarked' in col] X = df[features] y = df['Survived'] print(f"Features sélectionnées: {features}") print(f"Shape X: {X.shape}, Shape y: {y.shape}") # 6. SÉPARATION TRAIN/TEST print("\n6. SÉPARATION DONNÉES D'ENTRAÎNEMENT/TEST") print("-" * 80) X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y) print(f"Train set: {X_train.shape}, {y_train.shape}") print(f"Test set: {X_test.shape}, {y_test.shape}") print(f"\nRépartition des classes dans train:") print(y_train.value_counts(normalize=True)) # 7. NORMALISATION/STANDARDISATION print("\n7. NORMALISATION DES DONNÉES") print("-" * 80) scaler = StandardScaler() X_train_scaled = scaler.fit_transform(X_train) X_test_scaled = scaler.transform(X_test) print("Statistiques avant standardisation:") print(f"Moyenne: {X_train['Age'].mean():.2f}, Std: {X_train['Age'].std():.2f}") print("\nStatistiques après standardisation:") print(f"Moyenne: {X_train_scaled[:, 2].mean():.2f}, Std: {X_train_scaled[:, 2].std():.2f}") print("\n" + "=" * 80) print("DONNÉES PRÊTES POUR LA MODÉLISATION")

print("=" * 80) ``` --- ## 3.3 APPRENTISSAGE SUPERVISÉ - RÉGRESSION ### 3.3.1 Régression Linéaire Simple La régression linéaire modélise la relation entre une variable dépendante y et une variable indépendante x : $$y = \beta_0 + \beta_1 x + \epsilon$$ Où : - β_0 : ordonnée à l'origine (intercept) - β_1 : pente (coefficient) - ϵ : erreur **Cas d'usage en entreprise :** - Prédire les ventes en fonction du budget marketing - Estimer le salaire en fonction de l'expérience - Prévoir le chiffre d'affaires ```python from sklearn.linear_model import LinearRegression from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error import numpy as np # Charger des données de salaires url = "https://raw.githubusercontent.com/FlipRoboTechnologies/ML-Datasets/main/Salary%20Prediction/Salary_Data.csv" df_salary = pd.read_csv(url) print("=" * 80) print("RÉGRESSION LINÉAIRE SIMPLE - PRÉDICTION DE SALAIRE") print("=" * 80) # Préparer les données X = df_salary[['YearsExperience']].values y = df_salary['Salary'].values # Séparer train/test X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42) # Créer et entraîner le modèle model = LinearRegression() model.fit(X_train, y_train) # Faire des prédictions y_pred_train = model.predict(X_train) y_pred_test = model.predict(X_test) # Évaluer le modèle print("\nPARAMÈTRES DU MODÈLE:") print(f"Intercept (β₀): {model.intercept_:.2f}") print(f"Coefficient (β₁): {model.coef_[0]:.2f}") print(f"\nÉquation: Salaire = {model.intercept_:.2f} + {model.coef_[0]:.2f} × Années") print("\nPERFORMANCE SUR TRAIN:") print(f"R² Score: {r2_score(y_train, y_pred_train):.4f}") print(f"RMSE: {np.sqrt(mean_squared_error(y_train, y_pred_train)):.2f}") print(f"MAE: {mean_absolute_error(y_train, y_pred_train):.2f}") print("\nPERFORMANCE SUR TEST:") print(f"R² Score: {r2_score(y_test, y_pred_test):.4f}") print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred_test)):.2f}") print(f"MAE: {mean_absolute_error(y_test, y_pred_test):.2f}") # Visualisation fig, axes = plt.subplots(1, 2, figsize=(14, 5)) # Graphique 1: Données et ligne de régression axes[0].scatter(X_train, y_train, alpha=0.6, label='Train', color='blue') axes[0].scatter(X_test, y_test, alpha=0.6, label='Test', color='green') axes[0].plot(X, model.predict(X), color='red', linewidth=2, label='Régression') axes[0].set_xlabel('Années d\'expérience') axes[0].set_ylabel('Salaire') axes[0].set_title('Régression Linéaire: Salaire vs Expérience') axes[0].legend() axes[0].grid(True, alpha=0.3) # Graphique 2: Résidus residus = y_test - y_pred_test axes[1].scatter(y_pred_test, residus, alpha=0.6) axes[1].axhline(y=0, color='red', linestyle='--', linewidth=2) axes[1].set_xlabel('Valeurs prédites') axes[1].set_ylabel('Résidus') axes[1].set_title('Analyse des Résidus') axes[1].grid(True, alpha=0.3) plt.tight_layout() plt.show() # Faire une prédiction pour un nouveau cas nouvelle_experience = np.array([[5.0]]) salaire_predit = model.predict(nouvelle_experience) print(f"\nPRÉDICTION: Avec 5 ans d'expérience, salaire estimé: {salaire_predit[0]:.2f}") ``` ### 3.3.2 Régression Linéaire Multiple La régression multiple utilise plusieurs variables indépendantes : $$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n + \epsilon$$ ```python # Charger des données immobilières url = "https://raw.githubusercontent.com/selva86/datasets/master/BostonHousing.csv" df_housing = pd.read_csv(url) print("=" * 80) print("RÉGRESSION LINÉAIRE MULTIPLE - PRIX IMMOBILIERS") print("=" * 80) print("\nVariables disponibles:") print(df_housing.columns.tolist()) # Sélectionner les features features = ['crim', 'rm', 'age', 'dis', 'tax', 'ptratio', 'lstat'] X = df_housing[features] y = df_housing['medv'] # Séparer et standardiser X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42) scaler = StandardScaler() X_train_scaled = scaler.fit_transform(X_train) X_test_scaled = scaler.transform(X_test) # Entraîner le modèle model_multi = LinearRegression() model_multi.fit(X_train_scaled, y_train) # Prédictions y_pred_train = model_multi.predict(X_train_scaled) y_pred_test = model_multi.predict(X_test_scaled) # Évaluation print("\nPERFORMANCE:") print(f"R² Train: {r2_score(y_train, y_pred_train):.4f}") print(f"R² Test: {r2_score(y_test, y_pred_test):.4f}") print(f"RMSE Test: {np.sqrt(mean_squared_error(y_test, y_pred_test)):.2f}") # Importance des features importance = pd.DataFrame({ 'Feature': features, 'Coefficient': model_multi.coef_ }).sort_values('Coefficient', key=abs, ascending=False) print("\nIMPORTANCE DES VARIABLES:") print(importance) # Visualisation fig, axes = plt.subplots(1, 2, figsize=(14, 5)) # Valeurs réelles vs prédites axes[0].scatter(y_test, y_pred_test, alpha=0.5) axes[0].plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', linewidth=2) axes[0].set_xlabel('Prix réel') axes[0].set_ylabel('Prix prédit') axes[0].set_title('Prédictions vs Réalité') axes[0].grid(True, alpha=0.3) # Importance des coefficients

axes[1].barh(importance['Feature'], np.abs(importance['Coefficient'])) axes[1].set_xlabel('|Coefficient|') axes[1].set_title('Importance des Variables') axes[1].grid(True, alpha=0.3) plt.tight_layout() plt.show() ``` ### 3.3.3 Régression Logistique Malgré son nom, la régression logistique est utilisée pour la **classification binaire**. **Fonction sigmoïde :** $$P(y=1|x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$ **Cas d'usage :** - Prédire le churn client (partir/rester) - Détection de fraude (frauduleux/légitime) - Scoring de crédit (accepter/rejeter) ```python from sklearn.linear_model import LogisticRegression from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc # Utiliser les données Titanic préparées précédemment print("=" * 80) print("RÉGRESSION LOGISTIQUE - PRÉDICTION DE SURVIE TITANIC") print("=" * 80) # Créer et entraîner le modèle log_reg = LogisticRegression(random_state=42, max_iter=1000) log_reg.fit(X_train_scaled, y_train) # Prédictions y_pred_train = log_reg.predict(X_train_scaled) y_pred_test = log_reg.predict(X_test_scaled) y_pred_proba = log_reg.predict_proba(X_test_scaled)[:, 1] # Évaluation print("\nPERFORMANCE SUR TEST:") print(f"Accuracy: {log_reg.score(X_test_scaled, y_test):.4f}") print("\nMATRICE DE CONFUSION:") cm = confusion_matrix(y_test, y_pred_test) print(cm) print("\nRAPPORT DE CLASSIFICATION:") print(classification_report(y_test, y_pred_test, target_names=['Décédé', 'Survécu'])) # Visualisation fig, axes = plt.subplots(1, 2, figsize=(14, 5)) # Matrice de confusion sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=axes[0], xticklabels=['Décédé', 'Survécu'], yticklabels=['Décédé', 'Survécu']) axes[0].set_ylabel('Réalité') axes[0].set_xlabel('Prédiction') axes[0].set_title('Matrice de Confusion') # Courbe ROC fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba) roc_auc = auc(fpr, tpr) axes[1].plot(fpr, tpr, color='darkorange', linewidth=2, label=f'ROC curve (AUC = {roc_auc:.2f})') axes[1].plot([0, 1], [0, 1], color='navy', linewidth=2, linestyle='--') axes[1].set_xlim([0.0, 1.0]) axes[1].set_ylim([0.0, 1.05]) axes[1].set_xlabel('Taux de Faux Positifs') axes[1].set_ylabel('Taux de Vrais Positifs') axes[1].set_title('Courbe ROC') axes[1].legend(loc="lower right") axes[1].grid(True, alpha=0.3) plt.tight_layout() plt.show() # Interprétation des coefficients feature_importance = pd.DataFrame({ 'Feature': features, 'Coefficient': log_reg.coef_[0] }).sort_values('Coefficient', ascending=False) print("\nIMPACT DES VARIABLES SUR LA SURVIE:") print(feature_importance) ``` --- ## 3.4 APPRENTISSAGE SUPERVISÉ - CLASSIFICATION ### 3.4.1 K-Nearest Neighbors (KNN) KNN classifie un point en fonction des k plus proches voisins. **Principe :** 1. Calculer la distance entre le point à classer et tous les points d'entraînement 2. Sélectionner les k voisins les plus proches 3. Attribuer la classe majoritaire parmi ces voisins ```python from sklearn.neighbors import KNeighborsClassifier from sklearn.metrics import accuracy_score # Charger des données de vin url = "https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data" columns = ['Class', 'Alcohol', 'Malic_acid', 'Ash', 'Alcalinity', 'Magnesium', 'Phenols', 'Flavanoids', 'Nonflavanoid', 'Proanthocyanins', 'Color_intensity', 'Hue', 'OD280', 'Proline'] df_wine = pd.read_csv(url, names=columns) print("=" * 80) print("K-NEAREST NEIGHBORS - CLASSIFICATION DE VINS") print("=" * 80) # Préparer les données X = df_wine.drop('Class', axis=1) y = df_wine['Class'] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y) # Standardiser (important pour KNN!) scaler = StandardScaler() X_train_scaled = scaler.fit_transform(X_train) X_test_scaled = scaler.transform(X_test) # Trouver le meilleur k k_values = range(1, 21) train_scores = [] test_scores = [] for k in k_values: knn = KNeighborsClassifier(n_neighbors=k) knn.fit(X_train_scaled, y_train) train_scores.append(knn.score(X_train_scaled, y_train)) test_scores.append(knn.score(X_test_scaled, y_test)) # Visualiser les performances plt.figure(figsize=(10, 6)) plt.plot(k_values, train_scores, label='Train', marker='o') plt.plot(k_values, test_scores, label='Test', marker='s') plt.xlabel('Nombre de voisins (k)') plt.ylabel('Accuracy') plt.title('Performance du KNN en fonction de k') plt.legend() plt.grid(True, alpha=0.3) plt.show() # Meilleur k best_k = k_values[np.argmax(test_scores)] print(f"\nMeilleur k: {best_k}") # Entraîner avec le meilleur k knn_best = KNeighborsClassifier(n_neighbors=best_k) knn_best.fit(X_train_scaled, y_train) # Évaluation y_pred = knn_best.predict(X_test_scaled) print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}") print("\nRAPPORT DE CLASSIFICATION:") print(classification_report(y_test, y_pred)) ``` ### 3.4.2 Support Vector Machine (SVM) SVM trouve l'hyperplan optimal qui sépare les classes avec la marge maximale. **Cas d'usage :** - Classification de textes - Détection

d'anomalies - Reconnaissance de patterns ```python from sklearn.svm import SVC print("=" * 80) print("SUPPORT VECTOR MACHINE - CLASSIFICATION") print("=" * 80) # Tester différents noyaux kernels = ['linear', 'rbf', 'poly'] results = {} for kernel in kernels: svm = SVC(kernel=kernel, random_state=42) svm.fit(X_train_scaled, y_train) train_score = svm.score(X_train_scaled, y_train) test_score = svm.score(X_test_scaled, y_test) results[kernel] = {'train': train_score, 'test': test_score} print(f"\nKernel: {kernel}") print(f" Train Accuracy: {train_score:.4f}") print(f" Test Accuracy: {test_score:.4f}") # Meilleur modèle best_kernel = max(results, key=lambda k: results[k]['test']) print(f"\nMeilleur noyau: {best_kernel}") # Visualisation kernels_list = list(results.keys()) train_accs = [results[k]['train'] for k in kernels_list] test_accs = [results[k]['test'] for k in kernels_list] x = np.arange(len(kernels_list)) width = 0.35 fig, ax = plt.subplots(figsize=(10, 6)) ax.bar(x - width/2, train_accs, width, label='Train') ax.bar(x + width/2, test_accs, width, label='Test') ax.set_ylabel('Accuracy') ax.set_title('Performance SVM par type de noyau') ax.set_xticks(x) ax.set_xticklabels(kernels_list) ax.legend() ax.grid(True, alpha=0.3, axis='y') plt.show() ``` ### 3.4.3 Arbres de Décision et Forêts Aléatoires **Arbre de Décision :** Structure hiérarchique de décisions basées sur les features. **Forêt Aléatoire :** Ensemble d'arbres de décision qui votent pour la prédiction finale. ```python from sklearn.tree import DecisionTreeClassifier from sklearn.ensemble import RandomForestClassifier from sklearn.tree import plot_tree print("=" * 80) print("ARBRES DE DÉCISION ET FORÊTS ALÉATOIRES") print("=" * 80) # Arbre de Décision dt = DecisionTreeClassifier(max_depth=5, random_state=42) dt.fit(X_train, y_train) # Forêt Aléatoire rf = RandomForestClassifier(n_estimators=100, random_state=42) rf.fit(X_train, y_train) # Évaluation print("\nARBRE DE DÉCISION:") print(f"Train Accuracy: {dt.score(X_train, y_train):.4f}") print(f"Test Accuracy: {dt.score(X_test, y_test):.4f}") print("\nFORÊT ALÉATOIRE:") print(f"Train Accuracy: {rf.score(X_train, y_train):.4f}") print(f"Test Accuracy: {rf.score(X_test, y_test):.4f}") # Importance des features feature_importance = pd.DataFrame({ 'Feature': X.columns, 'Importance': rf.feature_importances_ }).sort_values('Importance', ascending=False) print("\nIMPORTANCE DES VARIABLES (Forêt Aléatoire):") print(feature_importance.head(10)) # Visualisation fig, axes = plt.subplots(1, 2, figsize=(16, 6)) # Arbre de décision plot_tree(dt, feature_names=X.columns, class_names=[str(c) for c in y.unique()], filled=True, ax=axes[0], fontsize=8) axes[0].set_title('Arbre de Décision (profondeur=5)') # Importance des features top_features = feature_importance.head(10) axes[1].barh(range(len(top_features)), top_features['Importance']) axes[1].set_yticks(range(len(top_features))) axes[1].set_yticklabels(top_features['Feature']) axes[1].set_xlabel('Importance') axes[1].set_title('Top 10 Features - Forêt Aléatoire') axes[1].invert_yaxis() axes[1].grid(True, alpha=0.3, axis='x') plt.tight_layout() plt.show() ``` --- ## 3.5 APPRENTISSAGE NON SUPERVISÉ ### 3.5.1 Clustering : K-Means K-Means regroupe les données en k clusters en minimisant la variance intra-cluster. **Cas d'usage en entreprise :** - Segmentation client (RFM) - Segmentation produits - Détection d'anomalies ```python from sklearn.cluster import KMeans from sklearn.metrics import silhouette_score # Charger des données clients url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv" df_iris = pd.read_csv(url) print("=" * 80) print("CLUSTERING K-MEANS - SEGMENTATION") print("=" * 80) # Préparer les données X = df_iris.drop('species', axis=1) # Standardiser scaler = StandardScaler() X_scaled = scaler.fit_transform(X) # Méthode du coude pour trouver k optimal inertias = [] silhouette_scores = [] k_range = range(2, 11) for k in k_range: kmeans = KMeans(n_clusters=k, random_state=42, n_init=10) kmeans.fit(X_scaled) inertias.append(kmeans.inertia_) silhouette_scores.append(silhouette_score(X_scaled, kmeans.labels_)) # Visualisation fig, axes = plt.subplots(1, 2, figsize=(14, 5)) # Méthode du coude axes[0].plot(k_range, inertias, marker='o', linewidth=2) axes[0].set_xlabel('Nombre de clusters (k)') axes[0].set_ylabel('Inertie') axes[0].set_title('Méthode du Coude') axes[0].grid(True, alpha=0.3) # Score de silhouette axes[1].plot(k_range, silhouette_scores, marker='s', linewidth=2, color='green') axes[1].set_xlabel('Nombre de clusters (k)') axes[1].set_ylabel('Score de Silhouette') axes[1].set_title('Score de Silhouette par k') axes[1].grid(True, alpha=0.3) plt.tight_layout() plt.show() # Choisir k=3 (on sait qu'il y a 3 espèces) kmeans_final = KMeans(n_clusters=3, random_state=42, n_init=10) clusters = kmeans_final.fit_predict(X_scaled) # Ajouter les clusters au dataframe df_iris['Cluster'] = clusters print(f"\nNombre d'observations par cluster:") print(df_iris['Cluster'].value_counts().sort_index()) # Caractériser les clusters print("\nCaractéristiques moyennes par cluster:") cluster_profiles =

df_iris.groupby('Cluster')[X.columns].mean() print(cluster_profiles) # Visualisation 2D (2 premières features) plt.figure(figsize=(10, 6)) scatter = plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap='viridis', alpha=0.6, s=50) plt.scatter(kmeans_final.cluster_centers_[:, 0], kmeans_final.cluster_centers_[:, 1], c='red', marker='X', s=200, edgecolors='black', linewidths=2, label='Centroïdes') plt.xlabel(X.columns[0]) plt.ylabel(X.columns[1]) plt.title('Clustering K-Means (3 clusters)') plt.colorbar(scatter, label='Cluster') plt.legend() plt.grid(True, alpha=0.3) plt.show() ``` ### 3.