

Dossier Technique - Projet Machine Learning Vin

Table des matières

1. Présentation du projet.....	1
2. Architecture et organisation du code	2
3. Dépendances techniques	2
4. Description des modules	3
exploration.py	4
traitement.py	5
entrainement.py	5
evaluation.py	6
deep_learning.py	7
app.py	8
5. Scripts utilitaires	9
6. Fonctionnalités implémentées	9
7. Conception et maintenance du projet	9
8. Modèles d'intelligence artificielle utilisés.....	10
9. Mode d'emploi pour l'utilisateur	12
10. Ressources et références	12

1. Présentation du projet

Ce projet vise à développer une application interactive en Python avec Streamlit qui présente un pipeline complet de Machine Learning appliqué à un dataset de vins (vin.csv).

L'application permet :

- L'exploration des données
- Le prétraitement avancé (gestion des valeurs manquantes par différentes stratégies, suppression de colonnes)
- La modélisation avec différents algorithmes ML (Logistic Regression, Random Forest, SVM) et Deep Learning (Keras)

- L'évaluation détaillée des modèles via métriques, visualisations, et comparaison rapide avec LazyPredict
- L'optimisation automatique des hyperparamètres avec GridSearchCV
- La prédiction interactive sur de nouvelles données, avec gestion automatique des valeurs manquantes

2. Architecture et organisation du code

```

├── data/
│   └── vin.csv          # Jeu de données CSV (à fournir manuellement)
├── models/             # Modèles ML sauvegardés (joblib / Keras)
│   ├── logistic_regression.joblib
│   ├── random_forest.joblib
│   ├── label_encoder.joblib
│   ├── scaler.pkl
│   ├── encoder.pkl
│   ├── model_dl.h5
│   └── feature_names.txt
├── models_dl/
│   ├── scaler.pkl
│   ├── encoder.pkl
│   ├── model_dl.h5
├── pages/
│   ├── exploration.py
│   ├── pretraitement.py
│   ├── training.py
│   ├── evaluation.py
│   ├── deep_learning.py
│   └── app.py
├── setup.py
└── run.py
└── requirements.txt
└── README.md
└── .gitignore

```

3. Dépendances techniques

Le projet repose sur un ensemble de bibliothèques Python bien établies, assurant à la fois **robustesse, performance, et facilité d'utilisation**. La version exacte des dépendances est figée dans requirements.txt afin de garantir la **reproductibilité** de l'environnement.

Interface utilisateur

- streamlit==1.31.1
 - Permet de créer une interface web interactive et réactive sans développement front-end.

Manipulation et visualisation de données

- numpy==1.23.5
 - Traitement des tableaux multidimensionnels, calculs vectorisés rapides
- pandas==2.2.1
 - Manipulation de données tabulaires avec DataFrame, nettoyage et transformation
- seaborn==0.13.2
 - Visualisations statistiques avancées (boxplots, heatmaps, etc.)
- matplotlib==3.8.4
 - Visualisation de données personnalisée et compatible avec seaborn

Machine Learning

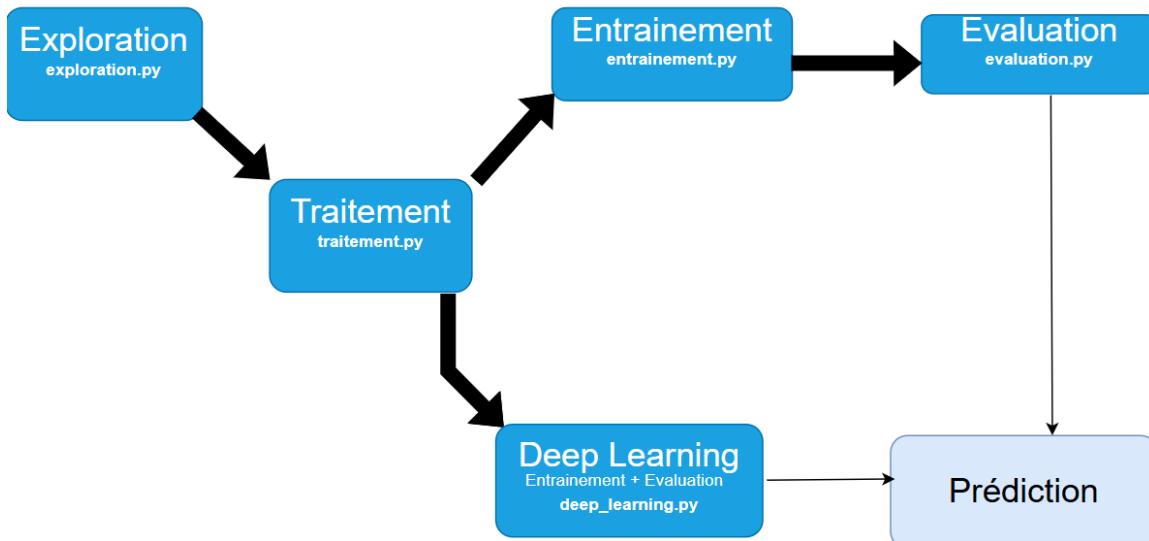
- scikit-learn==1.2.2
 - Modèles supervisés classiques (Logistic Regression, Random Forest, SVM)
 - Pipeline d'entraînement, encodage, standardisation, GridSearchCV
- lazypredict==0.2.12
 - Comparaison automatisée d'un grand nombre de modèles sans configuration manuelle

Deep Learning

- tensorflow==2.12.0
 - Framework de Deep Learning utilisé avec Keras (modèles denses, Dropout, EarlyStopping)
 - Optimisation avec Adam, gestion des callbacks, sauvegarde en .h5

4. Description des modules

Pipeline générale du projet :



Chaque module du projet correspond à une étape clé du pipeline Machine Learning. Ils sont organisés de manière modulaire et sont appelés dynamiquement dans l'application Streamlit.

exploration.py

1. Affichage du DataFrame et statistiques descriptives

- Chargement du fichier (data/vin.csv ou autre).
- Affichage des premières lignes :

```
st.dataframe(df.head())
```

2. Visualisations interactives

Sélection dynamique des colonnes numériques

- Sélection de colonnes numériques via interface :

```
selected_cols = st.multiselect("Colonnes numériques",
df.select_dtypes(...).columns.tolist())
```

- Histogrammes, boxplots, heatmaps, etc. affichés avec Matplotlib/Seaborn selon sélection :

```
sns.histplot(df[col])
plt.show()
```

3. Interface Streamlit

- Charger les données
- Choisir les colonnes à analyser
- Sélectionner le type de graphique

traitement.py

1. Nettoyage des données

Chargement des données brutes (data/vin.csv).

Correction automatique ("Vin équilibré" → "Vin équilibré") + correction manuelle possible

```
df[correction_col] = df[correction_col].replace(correction_old_value,  
correction_new_value)
```

2. Imputation configurable (moyenne, médiane, mode)

Numérique (mean, median, most_frequent) :

```
df[numeric_cols] =  
SimpleImputer(strategy=num_strategy).fit_transform(df[numeric_cols])
```

Catégorique (most_frequent ou constant) :

```
df[categorical_cols] = SimpleImputer(strategy=cat_strategy,  
fill_value="Inconnu").fit_transform(...)
```

3. Suppression de colonnes

- Suppression de colonnes sélectionnées :

```
df = df.drop(columns=drop_cols)
```

- Standardisation des colonnes numériques :

```
df[cols] = StandardScaler().fit_transform(df[cols])
```

entraînement.py

1. Prétraitement des données

-Chargement des données

```
if data_source == "Données nettoyées":  
    df = pd.read_csv(path_clean)  
else:  
    df = pd.read_csv(path_raw)
```

L'utilisateur choisit via un bouton radio la source des données. Si le fichier n'est pas disponible, un message d'erreur est affiché.

-Séparation des features et de la cible + encodage

```
X = df.drop(columns=["target"])
y = df["target"]

if y.dtype == 'object':
    le = LabelEncoder()
    y_enc = le.fit_transform(y)
else:
    y_enc = y
```

2. Entraînement ML (Logistic Regression, Random Forest, SVM)

```
model_options = {
    "Random Forest": make_pipeline(SimpleImputer(strategy='mean'),
RandomForestClassifier(random_state=42)),
    "Régression Logistique": make_pipeline(SimpleImputer(strategy='mean'),
LogisticRegression(max_iter=2000)),
    "SVM": make_pipeline(SimpleImputer(strategy='mean'), SVC(probability=True))
}
```

Chaque modèle est intégré dans un pipeline avec une **imputation automatique** (valeurs manquantes remplies par la moyenne).

```
model.fit(X_train, y_train)
y_pred = model.predict(X_test_reindexed)
report = classification_report(y_test, y_pred, output_dict=True)
cm = confusion_matrix(y_test, y_pred)
```

Chaque modèle sélectionné est entraîné puis évalué avec les prédictions (y_pred), un rapport de classification (classification_report) et une matrice de confusion (confusion_matrix).

3. Sauvegarde des artefacts

```
with open("models/feature_names.txt", "w") as f:
    for feat in feature_names:
        f.write(feat + "\n")
```

evaluation.py

1. Chargement des modèles et encodeurs

```
def load_models():
    model_files = [...] # Tous les modèles .joblib sauf l'encodeur
    models = {name: joblib.load(...) for name in model_files}
    label_encoder = joblib.load(...) # Encodeur de labels
    feature_names = open(...).read().splitlines() # Liste des colonnes
    return models, label_encoder, feature_names
```

Charge automatiquement tous les modèles stockés, l'encodeur de classes, et les noms des colonnes utilisées pour la prédiction.

2.Interface prédiction manuelle/CSV

```
mode = st.radio("Mode de saisie :", ["Manuel", "CSV"])
```

3.Imputation automatique

```
input_data[feat] = float(val) if val.strip() != "" else np.nan
```

En cas de valeurs manquantes (champs vides en mode manuel ou colonnes absentes dans CSV), ces valeurs sont remplacées automatiquement par des np.nan.

4.Prédiction avec décodage des labels

Pour chaque modèle chargé, la prédiction est réalisée sur les données d'entrée :

```
y_pred = model.predict(df_input) # ou X_input
```

Si disponible, la probabilité associée à chaque classe est calculée et affichée, avec un score de confiance en % :

```
if hasattr(model, "predict_proba"):
    proba = model.predict_proba(df_input)
    # Construction tableau probabilités et confiance max
```

Les labels prédits (souvent numériques) sont convertis en noms lisibles via l'encodeur inverse :

```
df_result["Nom du vin prédit"] = label_encoder.inverse_transform(y_pred)
```

deep_learning.py

1. GridSearchCV, LazyPredict :

```
# Exemple Random Forest pour GridSearchCV
param_grid = {"n_estimators": [50, 100], "max_depth": [5, 10, None]}
grid = GridSearchCV(RandomForestClassifier(), param_grid, cv=3, n_jobs=-1)
grid.fit(X_train, y_train)
```

et LazyPredict :

```
clf = LazyClassifier(verbose=0, ignore_warnings=True)
models, _ = clf.fit(X_train, X_test, y_train, y_test)
st.dataframe(models)
```

2.Construction et entraînement du modèle Keras

```
# Construction du modèle Deep Learning
model_dl = Sequential([
    Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)),
    Dropout(0.3),
```

```

        Dense(32, activation='relu'),
        Dense(num_classes, activation='softmax')
    ])

model_dl.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

early_stop = EarlyStopping(patience=5, restore_best_weights=True)

# Entraînement du modèle
history = model_dl.fit(
    X_train_scaled, y_train_cat,
    validation_split=0.2,
    epochs=50,
    batch_size=32,
    callbacks=[early_stop],
    verbose=0
)

```

3.Sauvegarde/chargement du modèle, scaler, encodeur

```

def save_dl_artifacts(model, scaler, encoder):
    folder = "models_dl"
    os.makedirs(folder, exist_ok=True)
    model.save(os.path.join(folder, "model_dl.h5"))
    joblib.dump(scaler, os.path.join(folder, "scaler.pkl"))
    joblib.dump(encoder, os.path.join(folder, "encoder.pkl"))

def load_dl_artifacts():
    folder = "models_dl"
    model = load_model(os.path.join(folder, "model_dl.h5"))
    scaler = joblib.load(os.path.join(folder, "scaler.pkl"))
    encoder = joblib.load(os.path.join(folder, "encoder.pkl"))
    return model, scaler, encoder

```

app.py

- Point d'entrée Streamlit

```

import streamlit as st
st.set_page_config(page_title="🍇 Application Machine Learning - Vin", layout="wide")
st.title("🍇 Application Machine Learning - Vin")

```

- Navigation fluide via sidebar

```

page = st.sidebar.radio("Choisissez une page :", [
    "Accueil", "1.Exploration", "2.Traitement", "3.Entrainement", "4.Deep Learning",
    "5.Évaluation"
])

```

5. Scripts utilitaires

setup.py :

- Installe automatiquement les dépendances Python
- Prépare l'environnement virtuel

run.py :

- Script principal pour démarrer l'application Streamlit

Usage :

- python setup.py
- python run.py
- ou streamlit run run.py

6. Fonctionnalités implémentées

- Chargement dynamique du dataset
- Exploration interactive avec graphiques et statistiques
- Prétraitement avancé : gestion fine des valeurs manquantes
- Suppression de colonnes personnalisable
- Modélisation ML complète avec split, fit, predict
- Comparaison automatique de modèles avec LazyPredict
- Optimisation d'hyperparamètres avec GridSearchCV
- Deep Learning avec Keras (Dropout, EarlyStopping)
- Sauvegarde/chargement de tous les artefacts
- Prédiction interactive, gestion automatique des valeurs manquantes
- Interface utilisateur fluide avec sidebar

7. Conception et maintenance du projet

L'architecture du projet a été pensée pour assurer à la fois clarté, évolutivité et facilité de maintenance. Chaque étape du pipeline (exploration, nettoyage, modélisation, évaluation...) est isolée dans un module distinct, rendant le code plus lisible et facilement modifiable sans impact sur les autres composants.

L'utilisation de Streamlit permet de naviguer intuitivement entre les fonctionnalités via une interface fluide. Le dossier pages/ organise les scripts comme une suite d'étapes logiques, ce qui facilite leur exécution dans un ordre cohérent. Tous les artefacts produits (modèles, encodeurs, scalers, features) sont systématiquement enregistrés dans des dossiers dédiés, ce qui assure leur réutilisabilité immédiate lors des prédictions ou des ajustements futurs.

Pour garantir une bonne maintenabilité et reproductibilité de l'environnement :

- Un fichier requirements.txt recense toutes les dépendances nécessaires.
- Le script setup.py automatise l'installation des bibliothèques.
- Le code est conçu pour s'adapter facilement à de nouveaux jeux de données similaires (autres types de vins...).
- Des tests manuels sont effectués pour valider les étapes critiques (imputation, prédiction, entraînement).
- L'architecture modulaire permet d'ajouter facilement de nouveaux algorithmes, visualisations ou options de traitement.

Les performances sont suivies grâce à des logs internes et des visualisations permettant d'identifier rapidement les anomalies ou dérives. À long terme, le projet pourrait intégrer une interface d'administration plus avancée ou une base de suivi versionné des modèles.

8. Modèles d'intelligence artificielle utilisés

Le cœur du projet repose sur deux grandes familles de modèles d'apprentissage supervisé : les algorithmes de **Machine Learning classiques** et les **réseaux de neurones profonds (Deep Learning)**. Chaque approche a été sélectionnée pour sa complémentarité, ses performances et sa capacité à traiter efficacement des données tabulaires.

Modèles de Machine Learning (Scikit-Learn)

- **Régression Logistique** : simple, interprétable, idéale pour une première baseline.
- **Random Forest** : robuste aux valeurs aberrantes, efficace pour détecter les relations non linéaires.
- **SVM (Support Vector Machines)** : performant pour les marges étroites et les petits jeux de données.

Les étapes de traitement incluent :

- Sélection des variables et encodage (OneHot/LabelEncoder)
- Mise à l'échelle avec StandardScaler
- Optimisation par GridSearchCV
- Évaluation automatisée via **LazyPredict**, pour comparer plusieurs modèles en un clic

Tous les modèles et outils (scalers, encodeurs, features) sont sauvegardés dans le dossier models/, assurant leur réutilisation sans retraitement.

Prétraitement :

```
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

Optimisation:

```
from sklearn.model_selection import GridSearchCV
grid = GridSearchCV(RandomForestClassifier(), param_grid, cv=5)
grid.fit(X_train, y_train)
```

Modèle de Deep Learning (Keras)

Un réseau de neurones dense (Fully Connected) conçu avec Keras. Il intègre :

- Plusieurs **couches Dense** avec activation ReLU
- **Dropout** pour éviter le surapprentissage
- **EarlyStopping** pour stopper l'entraînement dès stagnation des performances
- Optimisation avec **Adam**
- Perte : categorical_crossentropy ou binary_crossentropy selon le type de classification

Extrait code Keras :

```
from keras.models import Sequential
from keras.layers import Dense, Dropout

model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])
```

Avant l'entraînement, les données sont :

- Normalisées (StandardScaler)
- Encodées (LabelEncoder)
- Séparées en jeux d'entraînement/test

Les performances du modèle sont évaluées par :

- Précision, rappel, F1-score
- Matrice de confusion
- Courbes ROC et courbes d'apprentissage

Le modèle final est sauvegardé au format .h5 et utilisé pour des prédictions interactives via l'interface Streamlit.

9. Mode d'emploi pour l'utilisateur

1. Cloner le dépôt Git
2. Installer les dépendances avec :
`python setup.py`
3. Lancer l'application :
`python run.py`
ou
`streamlit run run.py`
4. Utiliser la sidebar pour accéder aux sections :
 - Exploration
 - Prétraitement
 - Modélisation (ML et Deep Learning)
 - Évaluation et Prédiction
5. Suivre les instructions à l'écran pour charger des données, entraîner les modèles, faire des prédictions

10. Ressources et références

- Documentation officielle Streamlit : <https://docs.streamlit.io>
- Documentation scikit-learn : <https://scikit-learn.org>
- Keras Documentation : <https://keras.io>
- Cours LinkedIn Learning : Python pour la Data Science
- LazyPredict GitHub : <https://github.com/shankarpandala/lazypredict>