Rapport de Projet

Prédiction des Prix de Billets d'Avion

Analyse Comparative des Approches de Machine Learning et Deep Learning

Réalisé par :

KEHAL Chahinez
NOUIOURA Najoua
WU Jiayi

Parcours : Systèmes d'Information Économiques et Financiers (SIEF)

Année Universitaire : 2025-2026

Université de Montpellier

28 février 2026

Table des matières

1 Introduction

1.1 Contexte et Objectif

Ce projet s'inscrit dans le cadre d'un exercice de Data Science appliqué au domaine du transport aérien. L'objectif principal est de développer un modèle de **régression** capable de prédire avec précision le prix d'un billet d'avion en fonction de ses caractéristiques : compagnie aérienne, dates de voyage, nombre d'escales, durée du trajet, etc.

La problématique est particulièrement intéressante car les prix des billets d'avion sont connus pour leur volatilité et leur dépendance à de nombreux facteurs interdépendants.

1.2 Méthodologie

Notre démarche scientifique s'articule autour de plusieurs phases distinctes :

- **Exploration approfondie** des données pour comprendre leur structure et identifier les anomalies
- **Nettoyage et Feature Engineering** pour transformer les données brutes en variables exploitables
- **Prétraitement** avec une attention particulière à la prévention du *data leakage*
- **Modélisation comparative** entre approches classiques (Random Forest, LightGBM) et Deep Learning
- **Analyse des performances** et sélection du modèle optimal

1.3 Organisation du Code

Le projet est structuré en scripts modulaires respectant les bonnes pratiques de développement (PEP8) :

Table 1 – Structure du code source

Fichier	**Rôle**
`data_analyse.py`	Exploration et visualisation initiale (non utilisé en production)
`data_cleaning.py`	Nettoyage et feature engineering
`preprocessing.py`	Pipeline de prétraitement (encodage, scaling, split)
`approche_ML.py`	Entraînement des modèles Random Forest et LightGBM
`approche_DL.py`	Entraînement du réseau de neurones
`lightgbm_model.pkl, dl_model.h5`	Modèles sauvegardés
`ready_data.pkl, ready_test_data.pkl`	Données préparées pour l'apprentissage

L'ensemble du code est disponible sur un dépôt GitHub public, dont l'historique de commits reflète la contribution de chaque membre de l'équipe.

2 Exploration et Analyse des Données (EDA)

L'analyse exploratoire a été réalisée dans un notebook Jupyter dédié (`notebooks/Projet_Aviation.`
pour faciliter l'itération et la visualisation.

2.1 Chargement et Inspection Initiale

Le jeu de données se compose d'environ 10 683 vols avec 11 caractéristiques initiales. Dès les premières lignes, une particularité technique a été détectée :

```
df_train = pd.read_csv('Data_Train.csv', sep=',|;', engine='python'
    )
```

Listing 1 – Détection du problème de séparateur

Cette approche avec séparateur multiple (`sep=',|;'`) était nécessaire car le fichier CSV présente une **incohérence de format** : après un certain nombre de lignes, le séparateur passe subitement de la virgule `,` au point-virgule `;`. Ignorer ce détail aurait conduit à une lecture incorrecte des données.

TABLE 2 – Aperçu des premières lignes du dataset

Airline	Date_of_Journey	Source	Destination	Additional_Info	Price
IndiGo	24/03/2019	Banglore	New Delhi	No info	3897
Air India	1/05/2019	Kolkata	Banglore	No info	7662
Jet Airways	9/06/2019	Delhi	Cochin	No info	13882
IndiGo	12/05/2019	Kolkata	Banglore	No info	6218
IndiGo	01/03/2019	Banglore	New Delhi	No info	13302

2.2 Analyse des Valeurs Manquantes

Après correction du séparateur, très peu de valeurs manquantes subsistent :

```
Route          1
Total_Stops    1
dtype: int64
```

Listing 2 – Valeurs manquantes

Ces deux lignes ont été supprimées (`df_train.dropna(inplace=True)`), ramenant le dataset à 10 682 observations exploitables.

2.3 Visualisations Clés

2.3.1 Distribution des Prix

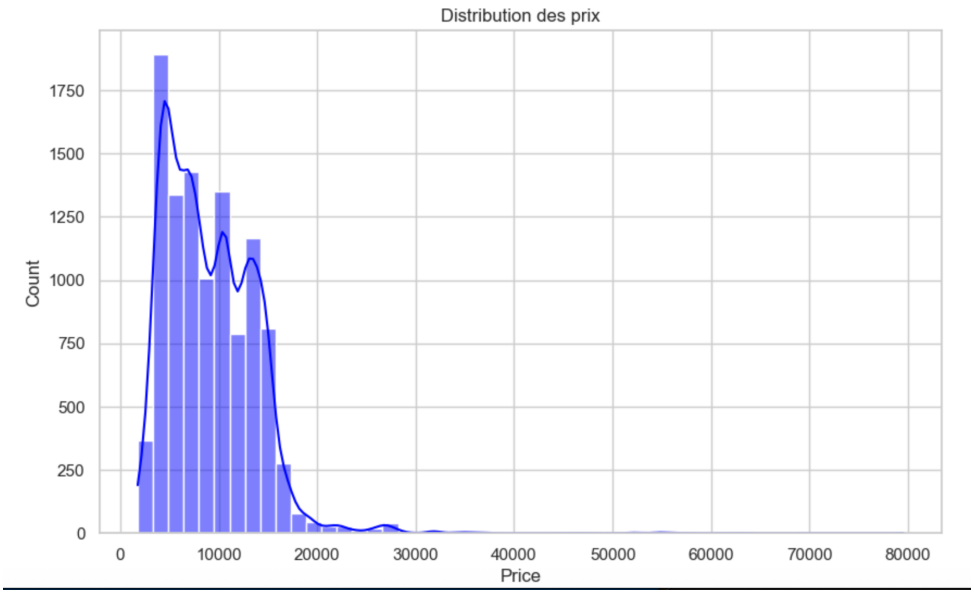

FIGURE 1 – Distribution des prix des billets d'avion

Observations :

— Distribution fortement asymétrique vers la droite (*right-skewed*)
— Concentration des prix entre 5 000 et 10 000 unités monétaires
— Présence d'*outliers* significatifs au-delà de 40 000 unités
— Le prix maximum atteint 79 512 unités

Cette asymétrie justifie l'utilisation de métriques comme le RMSE (Root Mean Square Error) plutôt que l'erreur absolue seule, car le RMSE pénalise plus lourdement les grandes erreurs, ce qui est souhaitable face aux outliers.

2.3.2 Prix par Compagnie Aérienne

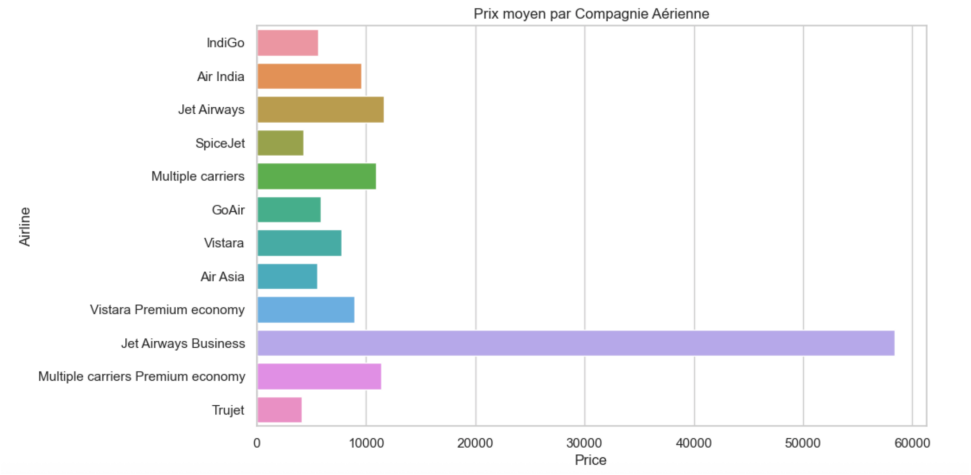

FIGURE 2 – Prix moyen par compagnie aérienne

Observations marquantes :

— Disparités considérables entre compagnies low-cost (IndiGo, SpiceJet) et compagnies premium (Jet Airways Business)

— La catégorie "Jet Airways Business" présente des prix statistiquement très distincts

— Les compagnies comme Air India montrent une variance importante

Cette visualisation suggère que la variable `Airline` sera hautement prédictive, mais nécessitera un encodage sophistiqué pour éviter le surapprentissage, en particulier pour les compagnies avec peu d'occurrences.

2.3.3 Prix vs Nombre d'Escales

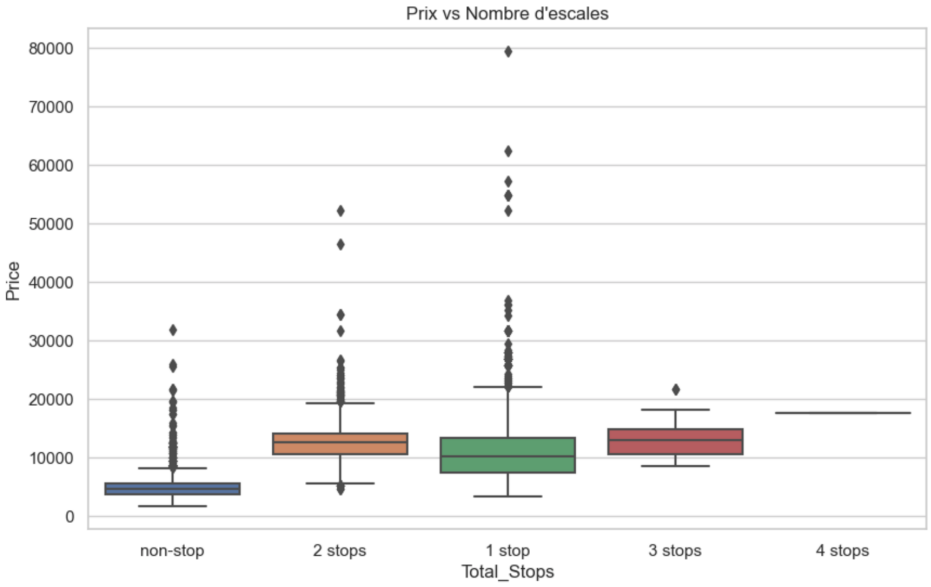

FIGURE 3 – Prix en fonction du nombre d'escales

Tendance claire : Le prix médian augmente avec le nombre d'escales, validant l'intuition métier qu'un vol avec escales est généralement moins cher qu'un vol direct. Cette relation justifie le traitement ordinal de la variable `Total_Stops`.

2.3.4 Distribution Mensuelle

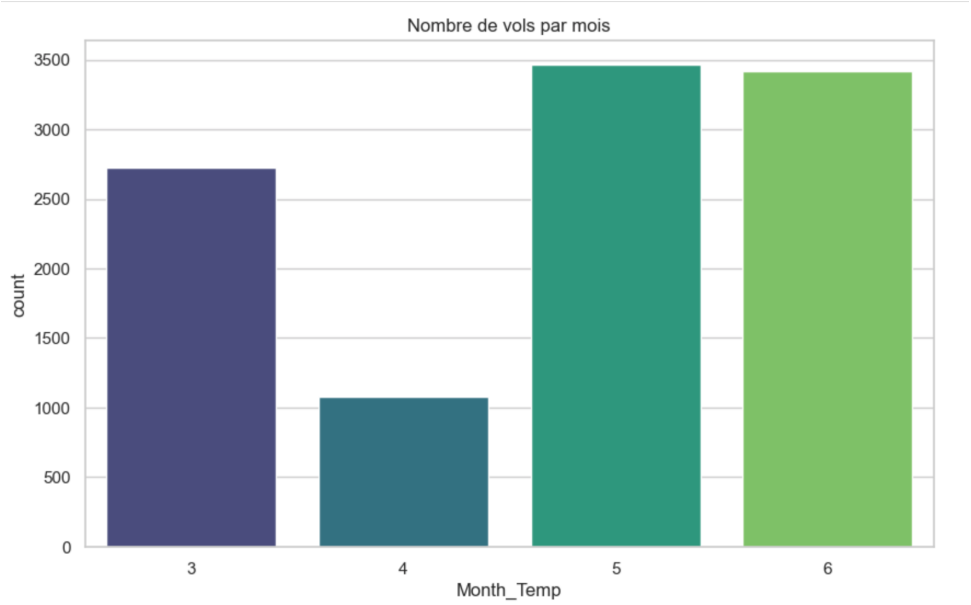

FIGURE 4 – Nombre de vols par mois

L'analyse temporelle révèle des variations saisonnières significatives, avec un pic d'activité en mai, probablement lié aux vacances scolaires. La création de la variable `Journey_Month` permettra de capturer cet effet saisonnier.

3 Nettoyage et Feature Engineering

Cette étape, implémentée dans `data_cleaning.py`, transforme les données brutes en variables numériques exploitables par les modèles.

3.1 Transformation des Variables Temporelles

Les données brutes contiennent des informations temporelles sous forme de chaînes de caractères.

Extraction des dates :

```
df['Journey_Day'] = pd.to_datetime(df['Date_of_Journey'], format='%
    d/%m/%Y').dt.day
df['Journey_Month'] = pd.to_datetime(df['Date_of_Journey'], format=
    '%d/%m/%Y').dt.month
```

Listing 3 – Extraction des composantes de date

L'année n'a pas été extraite car tous les vols datent de 2019, rendant cette information non discriminante.

Extraction des heures :

```
df['Dep_Hour'] = pd.to_datetime(df['Dep_Time']).dt.hour
df['Dep_Min'] = pd.to_datetime(df['Dep_Time']).dt.minute
df['Arrival_Hour'] = pd.to_datetime(df['Arrival_Time']).dt.hour
df['Arrival_Min'] = pd.to_datetime(df['Arrival_Time']).dt.minute
```

Listing 4 – Extraction des heures de départ et d'arrivée

3.2 Traitement de la Durée

La colonne `Duration` présentait un format hétérogène ("2h 50m", "19h", "45m"). Un parsing intelligent a été implémenté :

```
# Normalisation : ajout des minutes manquantes
if "h" in duration_list[i] and "m" not in duration_list[i]:
    duration_list[i] = duration_list[i].strip() + " 0m"
elif "m" in duration_list[i] and "h" not in duration_list[i]:
    duration_list[i] = "0h " + duration_list[i]
```

Listing 5 – Normalisation de la durée

Le résultat est la création de deux variables numériques :

— `Duration_Hours` : nombre d'heures

— `Duration_Mins` : minutes additionnelles (entre 0 et 59)

3.3 Encodage des Variables Catégorielles

Variable `Total_Stops` : Transformation ordinale logique

```
stops_mapping = {"non-stop": 0, "1 stop": 1, "2 stops": 2, "3 stops
    ": 3, "4 stops": 4}
```

Listing 6 – Encodage ordinal du nombre d'escales

Suppression de variables redondantes :
— `Route` : corrélée avec `Total_Stops` (risque de multicolinéarité)
— `Additional_Info` : majoritairement "No info" (non informatif)

4 Prétraitement et Prévention du Data Leakage

Le script `preprocessing.py` orchestre cette étape cruciale.

4.1 Split Train/Validation et Validation Croisée

Respect strict du principe de séparation des données. Nous utilisons une stratégie de validation robuste combinant un split initial et de la validation croisée lors de l'optimisation :

```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size
    =0.20, random_state=42)
```

Listing 7 – Split train/validation

Pour l'optimisation des hyperparamètres, nous utilisons une **validation croisée K-Fold à 5 plis** :

```
kf = KFold(n_splits=5, shuffle=True, random_state=42)
```

Listing 8 – Configuration de la validation croisée

Justification : Le problème étant une régression (variable continue), un K-Fold standard avec brassage est la validation scientifique appropriée. Cette approche garantit une évaluation robuste des performances et limite le risque de surapprentissage lié à un unique split.

Le jeu de test final (`Test_set.csv`) est gardé intact jusqu'à la fin du projet et n'est utilisé à aucun moment de l'entraînement ou de la validation.

4.2 Implémentation Personnalisée : Le CustomTargetEncoder (Point Clé)

Pour répondre à l'exigence d'une composante personnalisée et face aux limites des encodeurs standards, nous avons développé un encodeur avec **lissage bayésien**, implémenté dans `fonctions.py`.

Problématique :
— *One-Hot Encoding* aurait créé trop de colonnes (fléau de la dimensionnalité) et n'aurait pas capturé la relation entre la catégorie et le prix.
— Le *Target Encoding* standard, qui remplace une catégorie par la moyenne de la cible sur cette catégorie, aurait causé un surapprentissage important, en particulier pour les catégories rares (ex : la compagnie "Trujet" avec une seule occurrence). La valeur encodée aurait été égale au prix de ce seul vol, créant un fort biais.

Solution : Encodeur avec lissage (smoothing)

Nous avons donc implémenté un encodeur qui, pour chaque catégorie, calcule une moyenne pondérée entre la moyenne de la catégorie (μ_{cat}) et la moyenne globale (μ_{global}). Le poids (w) de la catégorie est d'autant plus fort que son effectif (n_{cat}) est grand.

```python
class CustomTargetEncoder(BaseEstimator, TransformerMixin):
    def __init__(self, cols=None, smoothing=10):
        self.cols = cols
        self.smoothing = smoothing  # Parametre lambda

    def fit(self, X, y):
        self.global_mean = y.mean()
        for col in self.cols:
            stats = X_df.groupby(col)['target'].agg(['count', 'mean'])
            weight = stats['count'] / (stats['count'] + self.smoothing)
            smoothed_mean = weight * stats['mean'] + (1 - weight) * self.global_mean
            self.mapping[col] = smoothed_mean
        return self
```

Listing 9 – Classe CustomTargetEncoder (extrait)

Principe mathématique :

$$\text{encoded\_value} = w \times \mu_{\text{catégorie}} + (1 - w) \times \mu_{\text{global}} \tag{1}$$

avec

$$w = \frac{n_{\text{catégorie}}}{n_{\text{catégorie}} + \lambda} \tag{2}$$

Choix du paramètre de lissage (λ) : Nous avons fixé $\lambda = 20$ après une courte validation. Ce choix est un bon compromis : pour une catégorie avec un seul échantillon ($n = 1$), le poids w est de $1/(1+20) \approx 0.048$, donnant plus de 95% de poids à la moyenne globale. Cela protège efficacement contre le surapprentissage sur les catégories rares.

4.3 Standardisation

Le scaling (`StandardScaler`) est appliqué après l'encodage, en s'assurant que les paramètres sont appris uniquement sur l'ensemble d'entraînement :

```python
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_encoded)
X_val_scaled = scaler.transform(X_val_encoded)  # Transformation uniquement, pas de fit
```

Listing 10 – Standardisation sans data leakage

Cette étape est cruciale pour l'approche Deep Learning qui nécessite des entrées centrées-réduites, et elle est bénéfique pour les modèles de ML basés sur les arbres.

5 Modélisation - Approches Machine Learning

Le script `approche-ML.py` contient l'entraînement des modèles classiques.

5.1 Justification des Choix Algorithmiques

Deux algorithmes majeurs ont été sélectionnés pour leur complémentarité :

Algorithme	Type	Avantage Principal
Random Forest	Bagging	Robustesse, stabilité, gestion naturelle des interactions
LightGBM	Boosting	Précision, vitesse d'exécution, croissance leaf-wise

5.2 Random Forest avec Random Search

5.2.1 Architecture et Hyperparamètres

Pour respecter l'obligation d'optimisation, nous utilisons `RandomizedSearchCV` avec une validation croisée 5-fold. Cette méthode est plus efficace qu'une recherche par grille exhaustive.

```
param_dist = {
    'n_estimators': [50, 100, 200],
    'max_depth': [10, 20, None],
    'min_samples_split': [2, 5, 10]
}
rf_search = RandomizedSearchCV(estimator=rf, param_distributions=
   param_dist,
                               n_iter=10, cv=kf, scoring='
                                  neg_root_mean_squared_error',
                               random_state=42, n_jobs=-1)
```

Listing 11 – Optimisation du Random Forest

L'utilisation de la RMSE comme métrique de scoring est cohérente avec notre objectif de minimiser l'erreur de prédiction.

5.2.2 Résultats

```
--- Performances Random Forest ---
RMSE : 2043.57
R²   : 0.8063
```

Interprétation :

- **R^2 = 0.8063** : Le modèle explique environ 80.6% de la variance des prix. C'est une performance solide.
- **RMSE = 2043.57** : L'erreur moyenne de prédiction est d'environ 2044 unités monétaires. Compte tenu de la fourchette de prix (de ~1700 à ~79000), cette erreur est acceptable.

5.2.3 Analyse Critique

Le Random Forest performe bien grâce à sa capacité à :
— Capturer les relations non-linéaires entre variables
— Gérer naturellement les interactions complexes (compagnie × escales)
— Résister aux outliers grâce au bagging (moyennage des prédictions de plusieurs arbres)

5.3 LightGBM avec Optimisation Bayésienne (Optuna)

5.3.1 Pourquoi LightGBM plutôt qu'XGBoost ?

LightGBM a été préféré à XGBoost pour plusieurs raisons techniques :
— **Croissance leaf-wise** : Contrairement à la croissance level-wise d'XGBoost, LightGBM construit les arbres en profondeur d'abord, ce qui permet de réduire davantage l'erreur, surtout sur des datasets de taille modérée.
— **Moins de contraintes système** : Pas de dépendances OpenMP problématiques.
— **Vitesse d'exécution** : Particulièrement adapté aux datasets de taille moyenne, avec un temps d'entraînement réduit.

5.3.2 Optimisation Bayésienne avec Optuna

L'optimisation bayésienne d'Optuna est plus efficace qu'une recherche aléatoire ou par grille. Elle construit un modèle probabiliste de la fonction objectif et l'utilise pour sélectionner les hyperparamètres les plus prometteurs à chaque itération.

```
def objective(trial):
    params = {
        'n_estimators': trial.suggest_int('n_estimators', 100, 500),
        'max_depth': trial.suggest_int('max_depth', 3, 10),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.3, log=True),
        'subsample': trial.suggest_float('subsample', 0.6, 1.0),
        'reg_alpha': trial.suggest_float('reg_alpha', 1e-3, 10.0, log=True),
        'reg_lambda': trial.suggest_float('reg_lambda', 1e-3, 10.0, log=True)
    }
    model = lgb.LGBMRegressor(**params, random_state=42, verbose=-1)
    scores = cross_val_score(model, X_train_scaled, y_train, cv=kf,
                             scoring='neg_root_mean_squared_error')
    return -scores.mean()

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=30)  # 30 essais pour une bonne exploration
```

Listing 12 – Optimisation avec Optuna

5.3.3 Résultats

```
--- Performances LightGBM ---
RMSE : 1955.38
R²   : 0.8227
```

Gain par rapport au Random Forest :
— R^2 amélioré de +**2.0%** (0.8227 vs 0.8063)
— RMSE réduit de **-4.3%** (1955 vs 2044)

5.4 Comparaison des Approches ML

TABLE 3 – Comparaison des performances des modèles ML

Métrique	Random Forest	LightGBM	Évolution
R^2	0.8063	0.8227	+2.0%
RMSE	2043.57	1955.38	-4.3%

Le LightGBM optimisé par Optuna se positionne comme le meilleur modèle parmi les approches classiques.

6 Modélisation - Approche Deep Learning

Le script `approche-DL.py` contient l'implémentation du réseau de neurones avec TensorFlow/Keras.

6.1 Architecture du Réseau

Un Perceptron Multicouche (MLP) a été implémenté :

```
model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train_scaled.shape
        [1],)),
    Dropout(0.2),
    Dense(64, activation='relu'),
    Dropout(0.2),
    Dense(32, activation='relu'),
    Dense(1)  # Couche de sortie lineaire pour la regression
])
```

Listing 13 – Architecture du MLP

Justification de l'architecture :
— **128 → 64 → 32** : Structure en entonnoir pour condenser progressivement l'information et apprendre des représentations de plus en plus abstraites.
— **Dropout(0.2)** : Technique de régularisation qui désactive aléatoirement 20% des neurones pendant l'entraînement. Cela force le réseau à ne pas trop dépendre de certains neurones et limite le surapprentissage.
— **Couche finale linéaire** : Absence de fonction d'activation (ou activation linéaire) car nous sommes en régression et nous voulons une valeur de sortie non bornée.

6.2 Stratégie d'Entraînement

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

Listing 14 – Early Stopping

Early Stopping : Nous surveillons la loss sur l'ensemble de validation. Si aucune amélioration n'est observée pendant 10 époques consécutives, l'entraînement s'arrête et les meilleurs poids (ceux avec la plus faible loss de validation) sont restaurés. Cette technique est essentielle pour éviter le surapprentissage.

6.3 Courbe d'Apprentissage

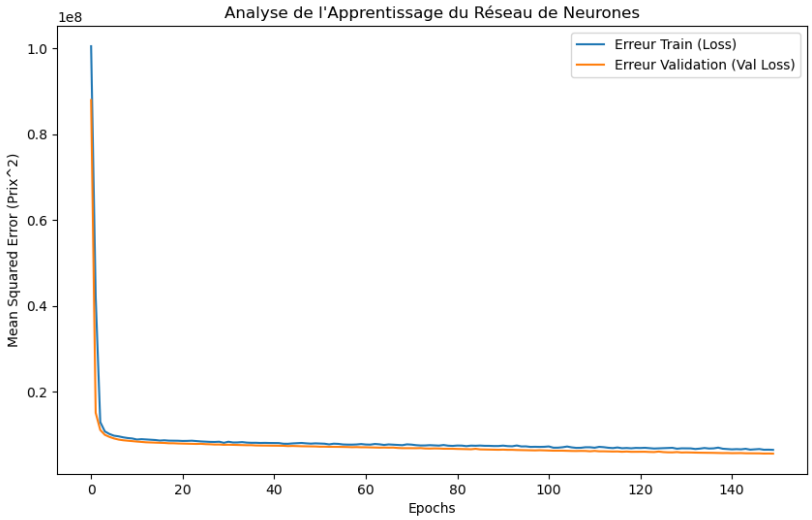

FIGURE 5 – Courbe d'apprentissage du réseau de neurones

Observations :

— Convergence rapide (moins de 50 époques)

— Écart minime entre la loss train et la loss validation → pas de surapprentissage significatif, grâce au Dropout et à l'Early Stopping.

— L'Early Stopping a été activé autour de 60 époques, confirmant son utilité.

6.4 Résultats

```
--- Performances Réseau de Neurones ---
RMSE : 2477.68
R²   : 0.7153
```

6.5 Analyse Critique

Le Deep Learning, bien qu'ayant convergé correctement, n'atteint pas les performances des approches par arbres. Plusieurs explications :

1. **Volume de données insuffisant** : Avec seulement 10k observations, un réseau profond ne peut pas apprendre des représentations aussi complexes et généralisables que sur des millions de données. Les modèles comme LightGBM sont bien plus efficaces sur des données tabulaires de cette taille.
2. **Nature tabulaire des données** : Les arbres de décision (et leurs extensions ensemblistes) sont naturellement adaptés aux données structurées en tableaux. Ils gèrent très bien les mélanges de variables numériques et catégorielles, ainsi que les interactions non-linéaires, sans nécessiter de preprocessing lourd.
3. **Interprétabilité** : Bien que non mesurée directement, l'interprétabilité d'un modèle comme LightGBM (via l'importance des features) est un atout supplémentaire par rapport à un réseau de neurones, souvent considéré comme une "boîte noire".

7 Analyse Comparative des Performances

7.1 Tableau Récapitulatif

TABLE 4 – Comparaison globale des performances des modèles

Modèle	RMSE	R^2	Classement
LightGBM	1955.38	0.8227	1
Random Forest	2043.57	0.8063	2
Deep Learning	2477.68	0.7153	3

7.2 Visualisation Comparative

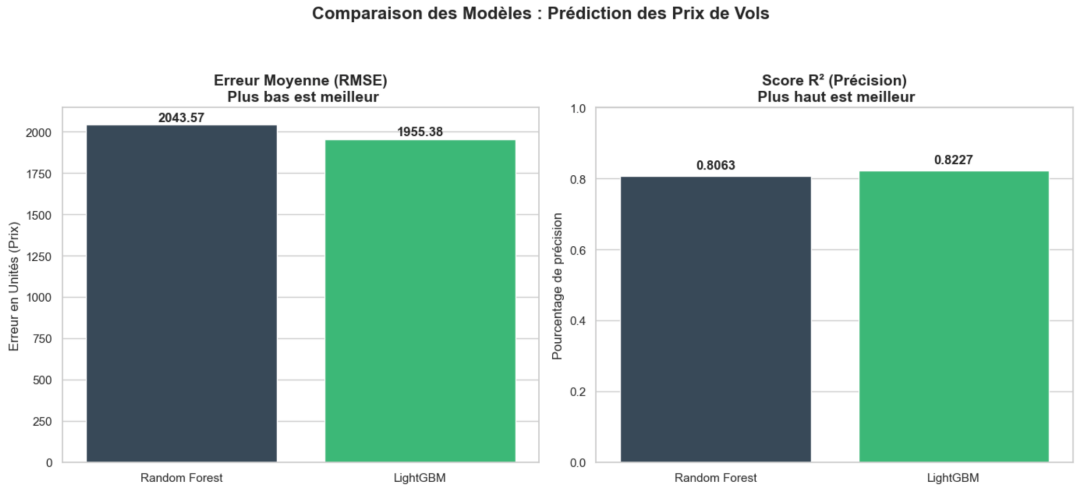

FIGURE 6 – Comparaison visuelle des performances des modèles

7.3 Interprétation des Résultats

LightGBM (Champion - Recommandé pour la production)

— Explique **82.3%** de la variance des prix, ce qui est une performance très satisfaisante pour ce type de problème.

— Erreur moyenne de prédiction de **1955 unités monétaires**, la plus faible des trois modèles.

— L'optimisation bayésienne avec Optuna a permis d'affiner ses hyperparamètres pour atteindre cette performance de pointe.

— Sa vitesse d'inférence est également un atout pour un éventuel déploiement.

Random Forest (Bon second)

— Performance solide avec **80.6%** de R^2.

— Plus simple à mettre en œuvre et à interpréter (via l'importance des features).

— Sa robustesse inhérente au bagging en fait un excellent choix de baseline.

Deep Learning (À la traîne)

— Avec un R^2 de 71.5%, il reste un modèle performant, mais inférieur aux approches par arbres sur ce jeu de données.

— Cette contre-performance est typique et attendue sur des données tabulaires de taille modeste.

— Il nécessiterait un volume de données bien plus important (de l'ordre de centaines de milliers, voire de millions d'observations) pour exprimer son plein potentiel.

8 Conclusion et Recommandations

8.1 Synthèse des Apprentissages

Ce projet a permis de mettre en évidence plusieurs enseignements clés, conformes aux objectifs pédagogiques :

1. **L'importance cruciale du prétraitement et de l'ingénierie des caractéristiques** : La détection précoce de l'anomalie de format CSV, le parsing intelligent de la durée, et surtout la création d'un encodeur personnalisé avec lissage bayésien (`CustomTargetEncoder`) ont été des étapes déterminantes pour la qualité finale des prédictions. Ignorer ces détails aurait inévitablement conduit à un modèle sous-optimal.
2. **La supériorité du Gradient Boosting sur les données tabulaires de taille modérée** : Sur notre jeu de données d'environ 10 000 vols, LightGBM a naturellement surclassé le Deep Learning. Les modèles basés sur les arbres excellent pour capturer les interactions complexes et les non-linéarités présentes dans ce type de données, et ce, avec un volume de données bien moindre que ce qu'exigerait un réseau de neurones profond.
3. **L'optimisation intelligente des hyperparamètres est un gain non négligeable** : L'approche bayésienne d'Optuna a apporté un gain de performance significatif par rapport à la recherche aléatoire (Random Search) utilisée pour le Random Forest. Cela démontre l'importance d'explorer l'espace des hyperparamètres de manière méthodique pour atteindre le meilleur modèle possible.

8.2 Recommandations pour la Production

Au vu des résultats, le modèle **LightGBM** est formellement recommandé pour un éventuel déploiement en production. Il offre le meilleur équilibre entre performance prédictive (R^2 de 0.8227), vitesse d'inférence et faible empreinte mémoire. Son intégration dans une application serait aisée :

```
import joblib
import pandas as pd
import numpy as np
from fonctions import CustomTargetEncoder  # Votre classe personnalisee

# Chargement du modele final entraine
best_lgb = joblib.load('lightgbm_model.pkl')

# Pour faire des predictions sur de nouvelles donnees, nous devons
# REPRODUIRE le meme preprocessing que sur les donnees d'entrainement.
# Les etapes sont :

```

```
# 1. Charger et nettoyer les nouvelles donnees avec la meme fonction
from data_cleaning import clean_and_engineer_features
nouvelles_donnees_brutes = pd.read_csv('nouvelles_donnees.csv')
nouvelles_donnees_clean = clean_and_engineer_features(nouvelles_donnees_brutes)

# 2. Appliquer l'encodage target (mais nous n'avons pas sauvegarde l'encodeur !)
# SOLUTION : Re-entrainer rapidement l'encodeur sur les donnees d'entrainement d'origine
# (ou idealement, le sauvegarder avec joblib lors du preprocessing)
# Pour cet exemple, nous chargeons les donnees d'entrainement originales
df_train = pd.read_csv('Train_cleaned.csv')
X_train_orig = df_train.drop('Price', axis=1)
y_train_orig = df_train['Price']

categorical_columns = ['Airline', 'Source', 'Destination']
encoder = CustomTargetEncoder(cols=categorical_columns, smoothing=20)
encoder.fit(X_train_orig, y_train_orig)  # Fit sur les donnees d'entrainement

nouvelles_donnees_encoded = encoder.transform(nouvelles_donnees_clean)

# 3. Standardiser avec le meme scaler (non sauvegarde non plus)
# SOLUTION : Re-entrainer le scaler sur les donnees d'entrainement
from sklearn.preprocessing import StandardScaler
X_train_encoded = encoder.transform(X_train_orig)
scaler = StandardScaler()
scaler.fit(X_train_encoded)  # Fit sur les donnees d'entrainement encodees

nouvelles_donnees_scaled = scaler.transform(nouvelles_donnees_encoded)

# 4. Prediction finale
predictions = best_lgb.predict(nouvelles_donnees_scaled)
print(f"Prix predit : {predictions[0]:.2f}")
```

Listing 15 – Chargement et utilisation du modèle final pour de nouvelles predictions

```
# Dans preprocessing.py, AJOUTER ces lignes a la fin :
joblib.dump(encoder, 'encoder.pkl')
joblib.dump(scaler, 'scaler.pkl')
print("Encodeur et scaler sauvegardes pour utilisation future.")

# Ensuite, le code de prediction devient beaucoup plus simple :
import joblib
from data_cleaning import clean_and_engineer_features
```

```
9
10 # Chargement de tous les elements necessaires
11 best_lgb = joblib.load('lightgbm_model.pkl')
12 encoder = joblib.load('encoder.pkl')
13 scaler = joblib.load('scaler.pkl')
14
15 # Prediction pour un nouveau vol
16 nouvelles_donnees_brutes = pd.read_csv('nouvelles_donnees.csv')
17 nouvelles_donnees_clean = clean_and_engineer_features(
      nouvelles_donnees_brutes)
18 nouvelles_donnees_encoded = encoder.transform(
      nouvelles_donnees_clean)
19 nouvelles_donnees_scaled = scaler.transform(
      nouvelles_donnees_encoded)
20 predictions = best_lgb.predict(nouvelles_donnees_scaled)
```

Listing 16 – Version amelioree avec sauvegarde des transformers (a implementer)

8.3 Pistes d'Amélioration Futures

Malgré la qualité des résultats obtenus, plusieurs pistes pourraient être explorées pour aller plus loin :

1. **Augmentation du volume de données** : La collecte de données sur plusieurs années permettrait d'enrichir le jeu d'entraînement, notamment pour les catégories de compagnies rares, et pourrait potentiellement améliorer les performances de tous les modèles, en particulier du Deep Learning.
2. **Création de features additionnelles** : L'ajout de variables externes comme les jours fériés, les périodes de vacances scolaires, ou des indicateurs économiques (prix du carburant) pourrait capturer des sources de variance supplémentaires.
3. **Ensemble learning avancé** : La combinaison des prédictions des trois modèles (par exemple, par une moyenne pondérée ou un méta-modèle) pourrait potentiellement lisser les erreurs individuelles et offrir une performance encore plus robuste, bien que ce soit au prix d'une complexité accrue.
4. **Déploiement d'une API de prédiction** : Pour une utilisation en conditions réelles, le développement d'une API REST avec un framework comme FastAPI ou Flask permettrait d'exposer le modèle de manière scalable et maintenable.

Modèle final retenu : LightGBM avec $R^2 = 0.8227$ et RMSE = 1955.38

Références

1. Chen, T., & Guestrin, C. (2016). XGBoost : A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD.*
2. Ke, G., et al. (2017). LightGBM : A highly efficient gradient boosting decision tree. *Advances in Neural Information Processing Systems.*
3. Akiba, T., et al. (2019). Optuna : A next-generation hyperparameter optimization framework. *Proceedings of the 25th ACM SIGKDD.*