ANNEXE : Code

1 Librairies

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import scipy.stats as st
from scipy.stats import chi2_contingency
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, RobustScaler
from sklearn.model_selection import GridSearchCV
from xgboost import plot_importance
```

2 Aperçu Préliminaire des Données

[ ]:
```
data = pd.read_csv("INNHotelsGroup.csv")
data = data.set_index("Booking_ID")
data.head()
```

[ ]:
```
data.shape
```

[ ]:
```
data.info()
```

[ ]:
```
data[["required_car_parking_space", "repeated_guest"]] =␣
↪data[["required_car_parking_space", "repeated_guest"]].astype("category")
```

[ ]:
```
data.isnull().sum()
```

2.1 Aperçu de la variable "booking_status"

[ ]:
```
print(data["booking_status"].value_counts())

plt.figure(figsize=(6,4))
sns.countplot(x=data["booking_status"], palette="coolwarm")
plt.title("Distribution des réservations annulées et non annulées")
plt.xlabel("Statut de la réservation")
plt.ylabel("Nombre de réservations")
plt.show()
```

2.2 Aperçu des variables explicatives

[ ]:
```
# Transformer les variables temporelles en type 'category'
data['arrival_year'] = data['arrival_year'].astype('category')
data['arrival_month'] = data['arrival_month'].astype('category')
data['arrival_date'] = data['arrival_date'].astype('category')
```

[ ]:
```
var_num = data.select_dtypes(include=["int64", "float64"]).columns
var_num = [col for col in var_num ]
var_num
```

[ ]:
```
data[var_num].describe()
```

[ ]:
```
var_char = data.select_dtypes(include=["object", "category"]).columns
var_char= [col for col in var_char ]
var_char
```

[ ]:
```
occurrences = {var: data[var].value_counts(dropna=False) for var in var_char}

for var, counts in occurrences.items():
    print(f"Occurrences pour la variable '{var}':")
    print(counts)
    print("-" * 40)
```

3 Statistiques Descriptives

3.1 Analyse univarié

3.1.1 Analyse des variables numeriques

[ ]:
```
n_cols = 3
n_rows = (len(var_num) + n_cols - 1) // n_cols

plt.figure(figsize=(n_cols * 5, n_rows * 5))

for i, col in enumerate(var_num):
```

```
    plt.subplot(n_rows, n_cols, i + 1)
    sns.histplot(data[col], kde=True, color='skyblue', bins=20)
    plt.title(f'Distribution de {col}')
    plt.xlabel(col)
    plt.ylabel('Fréquence')

plt.tight_layout()
plt.show()
```

[ ]:
```
n_cols = 3
n_rows = (len(var_num) + n_cols - 1) // n_cols

plt.figure(figsize=(n_cols * 5, n_rows * 5))

for i, col in enumerate(var_num):
    plt.subplot(n_rows, n_cols, i + 1)
    sns.boxplot(y=data[col], palette='Set2')
    plt.title(f'Distribution de {col}')
    plt.ylabel(col)

plt.tight_layout()

plt.show()
```

- **Analyse de la variable lead_time**

[ ]:
```
def hist_box(data, var):
    fig, (ax_box, ax_hist) = plt.subplots(nrows=2, sharex=True,␣
↪gridspec_kw={"height_ratios": (.15, .85)})

    # Boxplot
    sns.boxplot(data=data, x=var, ax=ax_box)
    ax_box.set(xlabel='')
    ax_box.set_title(f"Histogramme et Boxplot pour la variable '{var}'",␣
↪fontsize=14)

    # Histogramme
    sns.histplot(data=data, x=var, kde=True, ax=ax_hist)
    ax_hist.set(xlabel=var, ylabel='Fréquence')

    plt.tight_layout()
    plt.show()

hist_box(data, 'lead_time')
```

[ ]:
```
summary_400 = data.loc[data["lead_time"] > 400, "booking_status"].value_counts()
summary_300 = data.loc[data["lead_time"] > 300, "booking_status"].value_counts()
```

```
summary_df = pd.DataFrame({
    'lead_time > 400': summary_400,
    'lead_time > 300': summary_300
}).fillna(0).astype(int)
print(summary_df)
```

Analyse de la variable avg_price_per_room

[ ]:
```
hist_box(data, 'avg_price_per_room')
```

[ ]:
```
sns.boxplot(x='market_segment_type', y='avg_price_per_room', data=data)
plt.xlabel('market_segment_type')
plt.ylabel('Prix moyen par chambre')
plt.xticks(rotation=45)
plt.show()
```

[ ]:
```
print(data[data["avg_price_per_room"] == 0].shape[0]/data.shape[0] *100)
data.loc[data["avg_price_per_room"] == 0, "market_segment_type"].value_counts()
```

[ ]:
```
data[data["avg_price_per_room"] > 500]
```

[ ]:
```
plt.figure(figsize=(12, 6))
sns.boxplot(x='room_type_reserved', y='avg_price_per_room', data=data)
plt.title('Distribution des prix moyens par chambre en fonction du type de␣
↪chambre réservée')
plt.xlabel('Type de chambre réservée')
plt.ylabel('Prix moyen par chambre')
plt.xticks(rotation=45)
plt.show()
```

Gestion des valeurs aberrantes à l'aide de l'intervalle interquartile (IQR)

[ ]:
```
Q1 = data["avg_price_per_room"].quantile(0.25)
Q3 = data["avg_price_per_room"].quantile(0.75)
IQR = Q3 - Q1
Upper_Whisker = Q3 + 1.5 * IQR
data.loc[data["avg_price_per_room"] >= 500, "avg_price_per_room"] =␣
↪Upper_Whisker

data.loc["INN33115"]
```

- ***Analyse de la variable no_of_children**

[ ]:
```
hist_box(data, 'no_of_children')
```

[ ]:
```
data['no_of_children'].value_counts(normalize=True)
```

```
[ ]: data["no_of_children"] = data["no_of_children"].replace([9, 10], 3)
     data['no_of_children'].value_counts(normalize=True)
```

3.1.2 Analyse des variables categorielles

```
[ ]: var_char = [col for col in var_char if col != "booking_status"]
     for col in var_char:

         plt.figure(figsize=(8,4))
         sns.countplot(y=col, data=data, order=data[col].value_counts().index,␣
      ↪palette='viridis')
         plt.title(f'Distribution de {col}')
         plt.xlabel('Nombre d\'observations')
         plt.ylabel(col)
         plt.grid(axis='x', linestyle='--', alpha=0.7)
         plt.show()
```

3.2 Analyse multivariré

3.2.1 Analyse des variables numeriques en fonction de la variable cible :

```
[ ]: num_vars = len(var_num)
     rows = (num_vars // 3) + (num_vars % 3 > 0)
     plt.figure(figsize=(21, 7 * rows))

     for i, var in enumerate(var_num[:-1], start=1):
         plt.subplot(rows, 3, i)

         # kdeplot
         sns.kdeplot(
             data=data,
             x=var,
             hue="booking_status",
             fill=True,
             palette="Purples",
             common_norm=True
         )
         sns.despine(top=True, right=True, bottom=True, left=True)
         plt.tick_params(axis="both", which="both", bottom=False, top=False,␣
      ↪left=False)
         plt.xlabel("")
         plt.title(var, fontsize=14)

     plt.tight_layout()
     plt.show()
```

3.2.2 Analyse des variables catégorielles en fonction de la variable cible :

[ ]:
```
var_char = [col for col in var_char if col != "booking_status"]

for col in var_char:
    plt.figure(figsize=(10, 5))
    sns.countplot(y=col, hue='booking_status', data=data,
                  order=data[col].value_counts().index, palette='Set2')

    plt.title(f'Répartition de {col} selon booking_status', fontsize=15)
    plt.xlabel('Nombre d\'observations')
    plt.ylabel(col)
    plt.legend(title='booking_status')
    plt.grid(axis='x', linestyle='--', alpha=0.7)

    plt.show()
```

3.3 Analyse de correlation et liaison avec la variable cible

3.3.1 Analyse de la correlation entre les variables

Variables Quantitatives : Matrice de Correlation

[ ]:
```
corr_matrix = data[var_num].corr()

plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm', center=0)
plt.title('Matrice de Corrélation des Variables Continues')
plt.show()
```

Variables Qualitatives : test de Khi-deux et V de cramer

[ ]:
```
results = {}

for i in range(len(var_char)):
    for j in range(i + 1, len(var_char)):
        var1, var2 = var_char[i], var_char[j]
        contingency_table = pd.crosstab(data[var1], data[var2])

        chi2_stat, p_value, dof, expected = chi2_contingency(contingency_table)

        n = contingency_table.sum().sum()
        k = min(contingency_table.shape)
        cramer_v = np.sqrt(chi2_stat / (n * (k - 1))) if k > 1 else 0

        results[f'{var1} & {var2}'] = {
            "chi2_statistic": chi2_stat,
            "p_value": p_value,
            "cramer_v": cramer_v
```

```
        }

results_df = pd.DataFrame(results).T
results_df.columns = ['Chi2 Statistic', 'P-Value', 'Cramer V']

results_df
```

3.3.2 Liens entre "booking_status" et les variables explicatives :

Variables Quantitatives : test de Student

[ ]:
```
results = []
for var in var_num:

    groupe_annule = data[data['booking_status'] == 'Canceled'][var]
    groupe_non_annule = data[data['booking_status'] == 'Not_Canceled'][var]
    t_stat, p_value = st.ttest_ind(groupe_annule, groupe_non_annule,␣
↪equal_var=False)

    results.append({
        "Variable": var,
        "Statistique t": t_stat,
        "P-Value": p_value
    })


results_df = pd.DataFrame(results)
results_df = results_df.sort_values(by="P-Value")

print(results_df.round(4))
```

Variables catégorielles : test de Chi2

[ ]:
```
var_char
results_chi2 = []

for col in var_char:
    contingency_table = pd.crosstab(data[col], data['booking_status'])
    chi2_stat, p_val, dof, expected = chi2_contingency(contingency_table)

    results_chi2.append({
        "Variable": col,
        "Chi2 Statistique": chi2_stat,
        "P-Value": p_val
    })

chi2_results_df = pd.DataFrame(results_chi2).sort_values(by="P-Value")
```

```
print(chi2_results_df.round(4))
```

4 Préparation des données pour la modélisation

4.1 Préparation variables catégorielles et de la variable cible

4.1.1 Preparation et Encodage des variables categorielles

[ ]:
```
# X['lead_time'] = np.log1p(X['lead_time'])
# X['avg_price_per_room'] = np.log1p(X['avg_price_per_room'])
```

[ ]:
```
df = data.copy()
```

[ ]:
```
df.info()
```

[ ]:
```
# Recodage de la variable cible :
df['booking_status'] = df['booking_status'].map({'Canceled': 1, 'Not_Canceled':␣
↪0})
```

[ ]:
```
q1 = np.quantile(df['lead_time'], 0.25)
q3 = np.quantile(df['lead_time'], 0.75)
diff = q3 - q1
df['lead_time_out'] = df['lead_time'] > q3 + (1.5 * diff)

freq_map = df['lead_time_out'].value_counts(normalize=True)
df['lead_time_out'] = df['lead_time_out'].map(freq_map)
```

[ ]:
```
# Definir la variable cible
target = 'booking_status'
```

[ ]:
```
df.drop(['repeated_guest', 'arrival_year'], axis=1, inplace=True)
```

[ ]:
```
df.info()
```

- **Encodage des variables temporelles : Cyclical encoding**

[ ]:
```
df['arrival_date'] = df['arrival_date'].astype('int64')
df['arrival_month'] = df['arrival_month'].astype('int64')

# we assum that the madx is  31 days
df['arrival_date_sin'] = np.sin(2 * np.pi * df['arrival_date'] / 31)
df['arrival_date_cos'] = np.cos(2 * np.pi * df['arrival_date'] / 31)

#
df['arrival_month_sin'] = np.sin(2 * np.pi * df['arrival_month'] / 12)
df['arrival_month_cos'] = np.cos(2 * np.pi * df['arrival_month'] / 12)
```

```
[ ]: df.drop(['arrival_date','arrival_month'], axis=1,inplace=True)
```

```
[ ]: df.columns
```

- **Encodage des variables avec plus de 4 modalités:**

```
[ ]: categorical_columns = df.select_dtypes(include=['object', 'category']).columns
     categorical_columns
```

```
[ ]: var_cat_plus_4 = []
     var_cat_moins_4 = []

     for var in categorical_columns :
         print(df[var].value_counts())
         print(len(df[var].value_counts()))
         if len(df[var].value_counts()) >= 4 :
             var_cat_plus_4.append(var)
         else :
             var_cat_moins_4.append(var)

     print(var_cat_plus_4,var_cat_moins_4)
```

```
[ ]: # Les variables catégorielles a plus de 4 modalités sont remplacées par la␣
      ↪variable moyenne de la variable cible par modalité

     for v in var_cat_plus_4:
         tmp = pd.DataFrame(df.groupby(by=[v])[target].mean())
         df= df.join(tmp, on=v, how='left', lsuffix='', rsuffix= "_%_"+ v ,␣
      ↪sort=False)
```

Encodage des variables binaires

```
[ ]: df["required_car_parking_space"] = df["required_car_parking_space"].
      ↪astype('int64')
```

```
[ ]: df.info()
```

4.1.2 Separtaion et encodage de la variable cible

```
[ ]: var_continues = list(df.select_dtypes(include=['int64', 'float64']).columns)
     len(var_continues)
```

```
[ ]: var_continues
```

```
[ ]: X_var_continues = var_continues.remove(target)
```

```
[ ]: Y = df[target]
```

```
[ ]: colonnes_presentes = [col for col in var_continues if col in df.columns]
     X = df[colonnes_presentes]
```

```
[ ]: X.columns
```

```
[ ]: Y.head()
```

```
[ ]: X.head()
```

4.2 Standardisation des variables

```
[ ]: scaler = RobustScaler()
     X_scaled = scaler.fit_transform(X)
     X_scaled = pd.DataFrame(X_scaled, columns=X.columns)
     X_scaled.head()
```

```
[ ]: X_scaled.shape
```

4.3 Séparation en train/test

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(
         X_scaled, Y,
         test_size=0.2,
         stratify=Y,
         random_state=42
     )

     # Vérification des dimensions
     print("X_train shape :", X_train.shape)
     print("y_train shape :", y_train.shape)
     print("X_test shape :", X_test.shape)
     print("y_test shape :", y_test.shape)
```

5 Modélisation

```
[ ]: from sklearn.metrics import accuracy_score
     from sklearn.metrics import classification_report
     from sklearn.metrics import confusion_matrix
     from sklearn.metrics import log_loss
     from sklearn.metrics import precision_recall_curve
     from sklearn.metrics import average_precision_score
     from sklearn.metrics import roc_curve, auc

     # from imblearn.over_sampling import SMOTE
     from sklearn.linear_model import LogisticRegression
```

```
from sklearn.linear_model import lasso_path, LassoCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import plot_tree

import random
from skopt import BayesSearchCV
from skopt.space import Real, Integer,Categorical
from sklearn.model_selection import cross_validate
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV

from xgboost import XGBClassifier
import optuna
```

5.1 Arbre de décision :

5.1.1 Modèle d'initialisation

[ ]:
```
tree = DecisionTreeClassifier(criterion='gini',
                              splitter='best',
                              min_samples_leaf=20, # augmentation de ce␣
↪parametre pour forcer l'arbre à ne pas trop se spécialiser sur des petits␣
↪groupes d'observations.
                              max_depth=5, # imiter la profondeur maximale de␣
↪l'arbre pour éviter qu'il apprenne trop de détails spécifiques à␣
↪l'échantillon d'entraînement.
                              random_state=42)
# il faut optimiser ces parametres
```

[ ]:
```
tree.fit(X_train,y_train)
print(tree)
```

[ ]:
```
y_train_predict = tree.predict(X_train)
print(y_train_predict)
```

[ ]:
```
y_test_predict = tree.predict(X_test)
```

[ ]:
```
y_train_predict_proba = tree.predict_proba(X_train)[:,1]# On conserve en␣
↪mémoire uniquement la probabilité de l'événement cible pour nos graphiques
print(y_train_predict_proba)
y_test_predict_proba = tree.predict_proba(X_test)[:,1]
```

[ ]:
```
plt.figure(figsize=(100,100))
plot_tree(tree, feature_names = var_continues,  max_depth=5, filled = True,␣
↪fontsize=50)
plt.show()
```

[ ]:
```python
# Importance des variables
importance_variable = pd.DataFrame()
importance_variable["Variable"]= var_continues
importance_variable["Feature Importance"] = tree.feature_importances_
importance_variable.sort_values(by = "Feature Importance", axis=0,␣
 ↪ascending=False, inplace=True)

print("Les 5 variables les plus importantes : ")
importance_variable.head(5)
```

[ ]:
```python
importance_variable
```

[ ]:
```python
select_var = ["lead_time",
"booking_status_%_market_segment_type",
"no_of_special_requests",
"avg_price_per_room"]
```

Évaluation du modele :

[ ]:
```python
fpr_train, tpr_train, _ = roc_curve(y_train, y_train_predict_proba)
roc_auc_train = auc(fpr_train, tpr_train)
fpr_test, tpr_test, _ = roc_curve(y_test, y_test_predict_proba)
roc_auc_test = auc(fpr_test, tpr_test)

plt.figure()
lw = 2
plt.plot(fpr_train, tpr_train, color='darkorange',
         lw=lw, label='Train - ROC curve (area = %0.2f)' % roc_auc_train)

plt.plot(fpr_test, tpr_test, color='darkgreen',
         lw=lw, label='Test - ROC curve (area = %0.2f)' % roc_auc_test)

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Comparaison courve ROC (TRAIN / TEST)')
plt.legend(loc="lower right")
plt.show()
```

[ ]:
```python
print("log loss app : " + str(log_loss(y_train, y_train_predict_proba)))
print("log loss test : " + str(log_loss(y_test, y_test_predict_proba)))
```

[ ]:
```python
def metrics_score(actual, predicted):
    print(classification_report(actual, predicted))
```

```
    cm = confusion_matrix(actual, predicted)
    plt.figure(figsize=(8,5))

    sns.heatmap(cm, annot=True,  fmt='.2f', xticklabels=['Not Cancelled',␣
 ↪'Cancelled'], yticklabels=['Not Cancelled', 'Cancelled'])
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.show()
```

[ ]:
```
y_train_predict = tree.predict(X_train)
y_test_predict = tree.predict(X_test)
```

[ ]:
```
metrics_score(y_train, y_train_predict)
```

[ ]:
```
metrics_score(y_test, y_test_predict)
```

5.1.2 Optimisation du modele

[ ]:
```
# Création du dictionnaire des indicateurs que nous souhaitons testés pour la␣
 ↪méthode Random ou GridSearch
param_dict = {
    'criterion': ['gini', 'entropy'], # Le critère de split des arbres
    'splitter': ['best','random'],    # Est-ce que l'on teste un échantillon de␣
 ↪variable (random)
                                        #ou toutes les variables (best) à␣
 ↪chaque neoud
    'max_depth': [3,4,10],  # Profondeur maximum de l'arbre
    'min_samples_split': [2,4], # # Nombre d'observations mimnimum pour créer␣
 ↪un split
    'min_samples_leaf': [1,5,10],     # Nombre d'observations minimum dans une␣
 ↪feuille
    'min_weight_fraction_leaf': [0,0.01], # Proportion minimum des observations␣
 ↪dans une feuille
    'max_features': ['log2',"sqrt"]}

#Création  du dictionnaire de recherche pour la méthode d'optimisation␣
 ↪bayesienne
clf = DecisionTreeClassifier()
param_dict_bayes = {
    'criterion': Categorical(['gini', 'entropy']),
    'splitter': Categorical(['best','random']),
    'max_depth': Integer(3,30),
    'min_samples_split': Integer(2,50),
    'min_samples_leaf': Integer(1,20),
    'min_weight_fraction_leaf': Real(0,0.5, prior='uniform')}
```

```
NB_ITER = 5



def random_parameter(clf,param_dict,n_iter,X_train,y_train,nb_cv) :
    res = pd.DataFrame()
    compt = 0
    num_iter = []
    auc=[]
    param = []
    while compt <n_iter :
        compt = compt +1
        params = {key: random.sample(value, 1)[0] for key, value in param_dict.items()}
        clf.set_params(**params)
        scores = cross_validate(clf, X_train, y_train, cv=5,
                                scoring = ['roc_auc'])
        num_iter.append(compt)
        param.append(params)
        auc.append(scores['test_roc_auc'].mean())

    res["Num_ITER"] = num_iter
    res["Param"] = param
    res["Auc"] = auc

    return res

Random_Res_Tree = random_parameter(DecisionTreeClassifier() ,param_dict,NB_ITER,X_train,y_train,5)
print(" #### RECHERCHE ALEATOIRE #### ")
Random_Res_Tree.sort_values('Auc', ascending = False, inplace = True)
Random_Res_Tree.head()
best_param_random_search = list(Random_Res_Tree["Param"])[0]
print("\n Paramètres recherche aléatoire : ")
print(best_param_random_search)

print("\n Résultats recherche aléatoire : " + str(Random_Res_Tree['Auc'].max()))




Grid_Search = GridSearchCV(DecisionTreeClassifier(),param_dict,scoring='roc_auc',cv=5)
Grid_Search.fit(X_train,y_train)
print(" #### RECHERCHE GRID SEARCH #### ")
print("\n Paramètres grid search : ")
```

```
best_param_gid_search = Grid_Search.best_params_
print(best_param_gid_search)
best_score_grid_search = Grid_Search.best_score_
print("\n Résultats grid search  : " + str(best_score_grid_search))



opt = BayesSearchCV(clf,param_dict_bayes , n_iter=NB_ITER,cv=5,scoring =␣
↪'roc_auc')
opt.fit(X_train, y_train)
print(" #### RECHERCHE OPTIMISATION #### ")
print("\n Paramètres grid search : ")
best_param_opti_bayes =opt.best_params_
print(best_param_opti_bayes)
best_score_opti_bayes = opt.best_score_
print("\n Résultats grid search  : " + str(best_score_opti_bayes))
```

[ ]:
```
tree = DecisionTreeClassifier(criterion='entropy',
                              splitter='best',
                              max_depth=10,
                              min_samples_leaf=5,
                              min_samples_split=2,
                              min_weight_fraction_leaf=0,
                              max_features='sqrt',
                              random_state=42)

tree.fit(X_train, y_train)
```

[ ]:
```
y_train_predict = tree.predict(X_train)
print(y_train_predict)
```

[ ]:
```
y_test_predict = tree.predict(X_test)
```

[ ]:
```
y_train_predict_proba = tree.predict_proba(X_train)[:,1]
print(y_train_predict_proba)
y_test_predict_proba = tree.predict_proba(X_test)[:,1]
```

[ ]:
```
plt.figure(figsize=(100,100))
plot_tree(tree, feature_names = var_continues,  max_depth=5, filled = True,␣
↪fontsize=50)
plt.show()
```

[ ]:
```
# Importance des variables
importance_variable = pd.DataFrame()
importance_variable["Variable"]= var_continues
importance_variable["Feature Importance"] = tree.feature_importances_
```

```
importance_variable.sort_values(by = "Feature Importance", axis=0,␣
 ↪ascending=False, inplace=True)

print("Les 5 variables les plus importantes :")
importance_variable.head(5)
```

[ ]:
```
importance_variable
```

Évaluation du modele :

[ ]:
```
fpr_train, tpr_train, _ = roc_curve(y_train, y_train_predict_proba)
roc_auc_train = auc(fpr_train, tpr_train)
fpr_test, tpr_test, _ = roc_curve(y_test, y_test_predict_proba)
roc_auc_test = auc(fpr_test, tpr_test)

plt.figure()
lw = 2
plt.plot(fpr_train, tpr_train, color='darkorange',
         lw=lw, label='Train - ROC curve (area = %0.2f)' % roc_auc_train)

plt.plot(fpr_test, tpr_test, color='darkgreen',
         lw=lw, label='Test - ROC curve (area = %0.2f)' % roc_auc_test)

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Comparaison courve ROC (TRAIN / TEST)')
plt.legend(loc="lower right")
plt.show()
```

[ ]:
```
print("log loss app : " + str(log_loss(y_train, y_train_predict_proba)))
print("log loss test : " + str(log_loss(y_test, y_test_predict_proba)))
```

[ ]:
```
def metrics_score(actual, predicted):
    print(classification_report(actual, predicted))

    cm = confusion_matrix(actual, predicted)
    plt.figure(figsize=(8,5))

    sns.heatmap(cm, annot=True,  fmt='.2f', xticklabels=['Not Cancelled',␣
 ↪'Cancelled'], yticklabels=['Not Cancelled', 'Cancelled'])
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.show()
```

```
[ ]: y_train_predict = tree.predict(X_train)
     y_test_predict = tree.predict(X_test)
```

```
[ ]: metrics_score(y_train, y_train_predict)
```

```
[ ]: metrics_score(y_test, y_test_predict)
```

Seuils

```
[ ]: #Choix du seuil
     precision_train, recall_train, thresholds_train =␣
      ↪precision_recall_curve(y_train,                                                     ␣

      ↪y_train_predict_proba)
     precision_test, recall_test, thresholds_test = precision_recall_curve(y_test,
                                                                                    ␣

      ↪y_test_predict_proba)
     table_choix_seuil = pd.DataFrame()
     table_choix_seuil["SEUIL"] = [0] + list(thresholds_train)
     table_choix_seuil["Precision_train"] = precision_train
     table_choix_seuil["Recall_train"] = recall_train

     f1_scores = 2 * (precision_train * recall_train) / (precision_train +␣
      ↪recall_train)
     table_choix_seuil["f1_scores"] = f1_scores


     table_choix_seuil.sort_values(by = "SEUIL", axis=0, ascending=False,␣
      ↪inplace=True)
     print(table_choix_seuil)
```

```
[ ]: max_f1_score_row = table_choix_seuil.loc[table_choix_seuil['f1_scores'].
      ↪idxmax()]
     print(max_f1_score_row)
```

```
[ ]: table_choix_seuil
```

```
[ ]: # seuil optimal
     y_proba = tree.predict_proba(X_test)[:, 1]
     y_pred = (y_proba >= 0.409).astype(int)
```

```
[ ]: metrics_score(y_test, y_test_predict)
```

5.2 Régression logistique

5.2.1 Model d'initialisation

[ ]:
```
regLog1 = LogisticRegression(max_iter=500, solver='lbfgs')
regLog1.fit(X_train,y_train)

y_train_predict_proba = regLog1.predict_proba(X_train)[:,1]
y_test_predict_proba = regLog1.predict_proba(X_test)[:,1]
```

[ ]:
```
fpr_train, tpr_train, _ = roc_curve(y_train, y_train_predict_proba)
roc_auc_train = auc(fpr_train, tpr_train)
fpr_test, tpr_test, _ = roc_curve(y_test, y_test_predict_proba)
roc_auc_test = auc(fpr_test, tpr_test)

plt.figure()
lw = 2
plt.plot(fpr_train, tpr_train, color='darkorange',
         lw=lw, label='Train - ROC curve (area = %0.2f)' % roc_auc_train)

plt.plot(fpr_test, tpr_test, color='darkgreen',
         lw=lw, label='Test - ROC curve (area = %0.2f)' % roc_auc_test)

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Comparaison courve ROC (TRAIN / TEST)')
plt.legend(loc="lower right")
plt.show()
```

[ ]:
```
y_predict = regLog1.predict(X_test)
metrics_score(y_test, y_predict)
```

[ ]:
```
table_coeff = pd.DataFrame()
table_coeff["Variable"]=X_train.columns
table_coeff["Coefficient"] = regLog1.coef_[0]
table_coeff["Odds Ratio"] = np.exp(table_coeff["Coefficient"])

print(table_coeff.sort_values(by='Coefficient', key=abs, ascending=False))

print("Intercept : " + str(regLog1.intercept_))
```

5.2.2 Selection de variable avec Lasso

[ ]:
```python
alphas_lasso, coefs_lasso, _ = lasso_path(X_train.values, y_train)
```

[ ]:
```python
feature_names = X_train.columns

plt.figure(figsize=(12, 8))

for i, coef in enumerate(coefs_lasso):
    plt.plot(alphas_lasso, coef, label=feature_names[i])

plt.xlabel('Alpha')
plt.ylabel('Coefficients')
plt.title('LASSO Path')
plt.xscale('log')
plt.gca().invert_xaxis()


plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', fontsize='small')
plt.tight_layout()

plt.show()
```

[ ]:
```python
lasso_cv = LassoCV(alphas=np.logspace(-4, 0, 50), cv=5)
lasso_cv.fit(X_train, y_train)
```

[ ]:
```python
threshold = 0.01

selected_features = X_train.columns[np.abs(lasso_cv.coef_) > threshold]
coefficients = lasso_cv.coef_[np.abs(lasso_cv.coef_) > threshold]

print(f"Variables sélectionnées : {selected_features.tolist()}")
```

[ ]:
```python
selected_features = ['no_of_children', 'no_of_weekend_nights',
 ↪'no_of_week_nights', 'required_car_parking_space', 'lead_time',
 ↪'no_of_previous_cancellations', 'no_of_previous_bookings_not_canceled',
 ↪'avg_price_per_room', 'no_of_special_requests', 'lead_time_out',
 ↪'arrival_month_sin', 'arrival_month_cos',
 ↪'booking_status_%_type_of_meal_plan', 'booking_status_%_market_segment_type']
X_train_lasso = X_train[selected_features]
X_test_lasso = X_test[selected_features]
```

[ ]:
```python
coef_df = pd.Series(coefficients, index=selected_features).sort_values(key=abs,
 ↪ascending=False)

plt.figure(figsize=(8, 5))
sns.barplot(x=coef_df.values, y=coef_df.index, palette="viridis")
```

```
plt.title("Importance des variables issues de la régression Lasso")
plt.xlabel("Coefficient")
plt.ylabel("Variables")
plt.axvline(0, color='gray', linestyle='--', lw=1)  # Ligne verticale pour␣
 ↪séparer les coefficients positifs et négatifs
plt.show()
```

[ ]:
```
regLog_lasso = LogisticRegression()
regLog_lasso.fit(X_train_lasso, y_train)

y_pred = regLog_lasso.predict(X_test_lasso)

conf_matrix = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

metrics_score(y_test, y_pred)
```

5.2.3 Sélection de variable par regression logistique statsmodels

[ ]:
```
X_train = X_train.reset_index(drop=True)
y_train = y_train.reset_index(drop=True)
```

[ ]:
```
import statsmodels as sm

from statsmodels.api import Logit

X_train["const"] =1
X_test["const"]=1 # pour ajouter l'intercept
lr = Logit(endog=y_train,exog=X_train)

reg = lr.fit()
print(reg.summary())
```

[ ]:
```
p_values = reg.pvalues
```

[ ]:
```
variables_significatives = p_values[p_values < 0.05].index.tolist()
variables_non_significatives = p_values[p_values >= 0.05].index.tolist()
```

[ ]:
```
variables_significatives
```

[ ]:
```
variables_non_significatives
```

[ ]:
```
X_train_sel = X_train[variables_significatives]
X_test_sel = X_test[variables_significatives]
```

```
regLog_sel = LogisticRegression()
regLog_sel.fit(X_train_sel, y_train)

y_pred = regLog_sel.predict(X_test_sel)

conf_matrix = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

metrics_score(y_test, y_pred)
```

5.2.4 Optimisation du modele par Grid Search et Poids

```
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Regularization strength
    'penalty': ['l1', 'l2'],  # Type de régularisation
    'solver': ['liblinear']  # Solver adapté à l1/l2
}

log_reg = LogisticRegression(max_iter=500, random_state=42)

grid_search = GridSearchCV(log_reg, param_grid, cv=5, scoring='accuracy',␣
 ↪verbose=1)
grid_search.fit(X_train_lasso, y_train)

best_params = grid_search.best_params_
print(f"Best Params: {best_params}")


best_log_reg = grid_search.best_estimator_
y_pred = best_log_reg.predict(X_test_lasso)
```

```
metrics_score(y_test, y_pred)
```

Ajouter des poids pour les classes pour mieux gerer le déséquilibre de classes

```
log_reg_balanced = LogisticRegression(
    C=best_params['C'],
    penalty=best_params['penalty'],
    solver=best_params['solver'],
    class_weight='balanced',
    max_iter=500,
    random_state=42
)


log_reg_balanced.fit(X_train_lasso, y_train)
```

```
y_pred_balanced = log_reg_balanced.predict(X_test_lasso)
```

[ ]:
```
metrics_score(y_test, y_pred_balanced)
```

5.3 XGBoost

[ ]:
```
from xgboost import XGBClassifier
```

5.3.1 Modèle d'initialisation

[ ]:
```
xgb_model = XGBClassifier(
                objective="multi:softmax",
                num_class=2,
                booster="gbtree",
                eval_metric="mlogloss"
)
```

[ ]:
```
xgb_model.fit(X_train, y_train)
```

[ ]:
```
y_pred_proba = xgb_model.predict_proba(X_test)
```

[ ]:
```
initial_log_loss = log_loss(y_test, y_pred_proba)
print("Log Loss initiale :", initial_log_loss)
```

[ ]:
```
y_pred = xgb_model.predict(X_test)
```

[ ]:
```
metrics_score(y_test, y_pred)
```

5.3.2 Optimisation du modèle

Optimisation du modèle par GridSearch

[ ]:
```
params = {
    'n_estimators': [100, 200, 300, 500],
    'max_depth': [3, 5, 7, 10],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'subsample': [0.6, 0.7, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.7, 0.8, 1.0],
    'gamma': [0, 0.1, 0.2, 0.5],
    'min_child_weight': [1, 3, 5, 7]
}

#random_search = RandomizedSearchCV(
#    XGBClassifier(objective="multi:softmax", num_class=2,␣
 ↪eval_metric="mlogloss", random_state=42),
#    param_distributions=params,
```

```
#    n_iter=30,
#    scoring='f1_weighted',
#    cv=5,
#    verbose=1,
#    n_jobs=-1,
#    random_state=42
#)

#random_search.fit(X_train, y_train)

print("Best parameters:", random_search.best_params_)
print("Best accuracy:", random_search.best_score_)
```

[ ]:
```
#Best model
optimized_xgb = XGBClassifier(
    objective="multi:softmax",
    num_class=2,
    eval_metric="mlogloss",
    subsample=0.8,
    n_estimators=300,
    min_child_weight=1,
    max_depth=10,
    learning_rate=0.05,
    gamma=0.1,
    colsample_bytree=0.7,
    random_state=42,
    use_label_encoder=False
)


optimized_xgb.fit(X_train, y_train)

y_pred = optimized_xgb.predict(X_test)
metrics_score(y_test, y_pred)
```

[ ]:
```
fig, axes = plt.subplots(1, 2, figsize=(18, 6))  # 1 ligne, 2 colonnes

# Premier graphique : importance selon le 'gain'
plot_importance(optimized_xgb, ax=axes[0], importance_type='gain')
axes[0].set_title("Importance des caractéristiques (Gain)")

# Deuxième graphique : importance selon le 'weight'
plot_importance(optimized_xgb, ax=axes[1], importance_type='weight')
axes[1].set_title("Importance des caractéristiques (Weight)")

# Ajuster l'espace entre les graphiques
plt.tight_layout()
```

```
plt.show()
```

```
import shap
shap.initjs()
```

```
explainer = shap.TreeExplainer(optimized_xgb)
shap_values = explainer.shap_values(X_test)
```

```
classe_index = 1  # Classe positive
observation_index = 0  # Première observation

# Correction de force_plot
shap.force_plot(
    explainer.expected_value[classe_index],  # Base value pour la classe 1
    shap_values[observation_index, :, classe_index],  # SHAP values pour cette␣
↪observation et classe
    X_test.iloc[observation_index]  # Caractéristiques de l'observation
)
```

```
shap.summary_plot(shap_values[..., 1], X_test)  # Classe positive (1)
```

Optimisation du modèle avec Optuna

```
def objective(trial):
    selected_features = trial.suggest_categorical('features', [list(X.columns)])
    X_train_selected = X_train[selected_features]
    X_test_selected = X_test[selected_features]

    param = {
        'objective': 'multi:softmax',
        'num_class': 2,
        'booster': 'gbtree',
        'eval_metric': 'mlogloss',
        'n_estimators': trial.suggest_int('n_estimators', 50, 1000),
        'max_depth': trial.suggest_int('max_depth', 3, 30),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.1),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 15),
        'subsample': trial.suggest_float('subsample', 0.5, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.01, 1.0),
        'reg_alpha': trial.suggest_float('reg_alpha', 0.0, 1),
        'reg_lambda': trial.suggest_float('reg_lambda', 0.0, 10)
    }

    mod = XGBClassifier(**param, random_state=42)
    scores = cross_validate(mod, X_train_selected, y_train, cv=3,␣
↪scoring='accuracy')
```

```python
    return np.mean(scores['test_score'])  # Fixed line

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=50)

print("Meilleurs paramètres:", study.best_params)
```

[ ]:
```python
xgb_tunned = XGBClassifier(
                objective="multi:softmax",
                num_class=2,
                booster="gbtree",
                eval_metric="mlogloss",
                n_estimators=876,
                max_depth=14,
                learning_rate=0.012036591105960953,
                min_child_weight=1,
                subsample=0.9303558848762992,
                colsample_bytree=0.7479507299406605,
                reg_alpha=0.022053370844735434,
                reg_lambda=7.723870704304475,
                random_state=42
            )
```

[ ]:
```python
xgb_tunned.fit(X_train, y_train)
```

[ ]:
```python
y_pred_proba = xgb_tunned.predict_proba(X_test)
```

[ ]:
```python
final_log_loss = log_loss(y_test, y_pred_proba)
print("Log Loss finale :", final_log_loss)
```

[ ]:
```python
y_pred = xgb_tunned.predict(X_test)
```

[ ]:
```python
metrics_score(y_test, y_pred)
```

[ ]:
```python

```