



BASKA Benjamin  
GUILLON Antonin  
HOTTON Robin

## Vue d'ensemble

L'objectif de ce projet est de permettre une prédiction de la quantité d'alcool et de l'amertume d'une bière en utilisant plusieurs facteurs. Cette prédiction doit être facile d'accès grâce à un formulaire web.

## Objectifs

1. Prédire l'amertume et la teneur en alcool d'un bière
2. Concevoir et développer une IA pour faire la prédiction
3. Faire une interface web pour que les utilisateurs aient accès aux prédictions

## Caractéristiques

Nous allons utiliser du python pour notre projet avec la librairie scikit-learn pour les IA, pandas pour la gestion des données et Flask pour le web.

## Structure des données

Entête	Type	Description
BeerID	Integer	Identifiant de la bière
Name	String	Nom de la bière
Url	String	URL de la recette de la bière
Style	String	Style de bière
StyleID	Integer	Identifiant du style

Size(L)	Float	Quantité brassée de la bière
OG	Float	Densité de moût avant fermentation
FG	Float	Densité de moût après fermentation
ABV	Float	Alcool par volume
IBU	Float	Unité internationale d'amertume
Color	Float	Méthode de Référence Standard
BoilSize	Float	Volume de liquide au début de l'ébullition
BoilTime	Integer	Temps d'ébullition du moût
BoilGravity	Float	Densité de moût avant ébullition
Efficiency	Float	Efficacité d'extraction de la macération de la bière
MashThickness	Float	Quantité d'eau par livre de grain
SugarScale	String	Échelle pour déterminer la concentration des solides dissous dans le moût
BrewMethod	String	Diverses techniques de brassage
PitchRate	Float	Levure ajoutée au fermenteur par unité de densité
PrimaryTemp	Float	Température à l'étape de la fermentation
PrimingMethod	String	Méthode d'amorçage



PrimingAmount	String	Quantité de sucre d'amorçage utilisée
UserId	Float	Identifiant de l'utilisateur

## I. Pré-traitement

La première étape fut d'ouvrir les deux csv qui proviennent de Kaggle et de prendre connaissance de la structure des données.

Le premier "recipeData.csv" :

Beer ID ▼	Name ▼	URL ▼	Style ▼	Style ID ▼	Size(L) ▼	OG ▼	FG ▼	ABV ▼	IBU ▼	Color ▼
1	Vanilla Cream	/homebrew/recipe/1	Cream Ale	45	21.77	1.06	1.01	5.48	17.65	4.83
2	Southern Tier	/homebrew/recipe/2	Holiday/Winter	85	20.82	1.08	1.02	8.16	60.65	15.64
3	Zombie Dust	/homebrew/recipe/3	American IPA	7	18.93	1.06	1.02	5.91	59.25	8.98
4	Zombie Dust	/homebrew/recipe/4	American IPA	7	22.71	1.06	1.02	5.8	54.48	8.5
5	Bakke Brygg	/homebrew/recipe/5	Belgian Blond	20	50	1.06	1.01	6.48	17.84	4.57
6	Sierra Nevada	/homebrew/recipe/6	American Pale	10	24.61	1.06	1.01	5.58	40.12	8

Boil Size ▼	Boil Time ▼	Boil Gravity ▼	Efficiency ▼	Mash Thickness ▼	Sugar Scale ▼	Brew Method ▼	Pitch Rate ▼	Primary Temp ▼	Priming Meth ▼	Priming Amount ▼	User Id ▼
28.39	75	1.04	70	N/A	Specific Gravity	All Grain	N/A	17.78	corn sugar	4.5 oz	116
24.61	60	1.07	70	N/A	Specific Gravity	All Grain	N/A	N/A	N/A	N/A	955
22.71	60	N/A	70	N/A	Specific Gravity	extract	N/A	N/A	N/A	N/A	
26.5	60	N/A	70	N/A	Specific Gravity	All Grain	N/A	N/A	N/A	N/A	
60	90	1.05	72	N/A	Specific Gravity	All Grain	N/A	19	Sukkerlake	6-7 g sukker/l	18325
29.34	70	1.05	79	N/A	Specific Gravity	All Grain	1	N/A	N/A	N/A	5889

Ce premier fichier décrit toutes les datas dont nous allons avoir besoin pour l'entraînement de notre IA. Le principal problème de ce dataset est qu'il contient une multitude de valeurs N/A qui peuvent poser problème. Nous devons donc en tenir compte pour l'exploration de données et leurs pré-traitements.

Le deuxième "styleData.csv" :

Style ▼	Style ID ▼
Altbier	1
Alternative Grain Beer	2
Alternative Sugar Beer	3
American Amber Ale	4
American Barleywine	5
American Brown Ale	6
American IPA	7
American Lager	8

Ce fichier est simplement le détail de deux colonnes déjà présentes dans le fichier précédent.

Une fois la première lecture faite, nous allons commencer par lire notre fichier csv dans un fichier python.

```
df = pd.read_csv("./beers/recipeData.csv", encoding="latin1")
```

Maintenant qu'on a ouvert le fichier on va pouvoir retirer les valeurs manquantes du fichier.

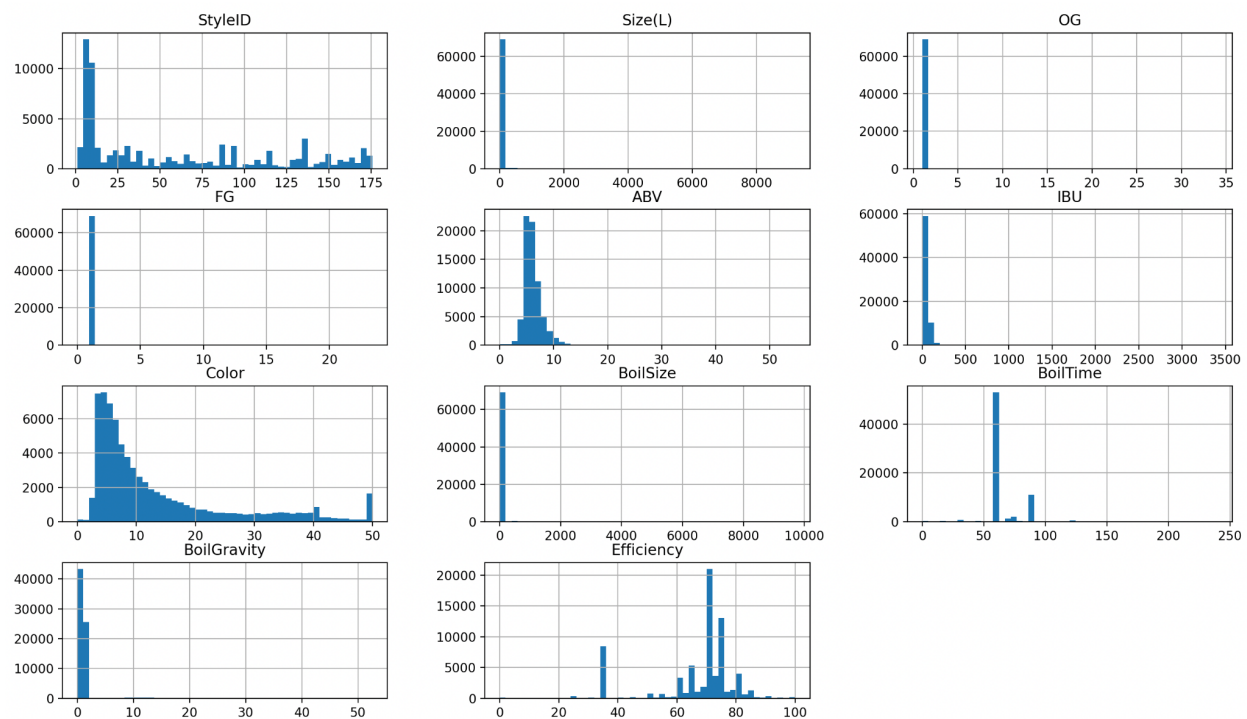
```
df.dropna(inplace=True)
```

Pour ce qui est du traitement des données inutiles, on cherche à afficher les données présentes et voir lesquelles peuvent nous servir à une prédiction de l'ABV et l'IBU.

Pour cela on drop celles qui ne nous servent pas.

```
df = df.drop(
    columns=["BeerID", "UserID", "URL", "Name", "Style", "PrimingMethod", "PrimingAmount", "PitchRate", "MashThickness",
            "PrimaryTemp", "SugarScale"])
```

Ensuite, on continue avec les données aberrantes.



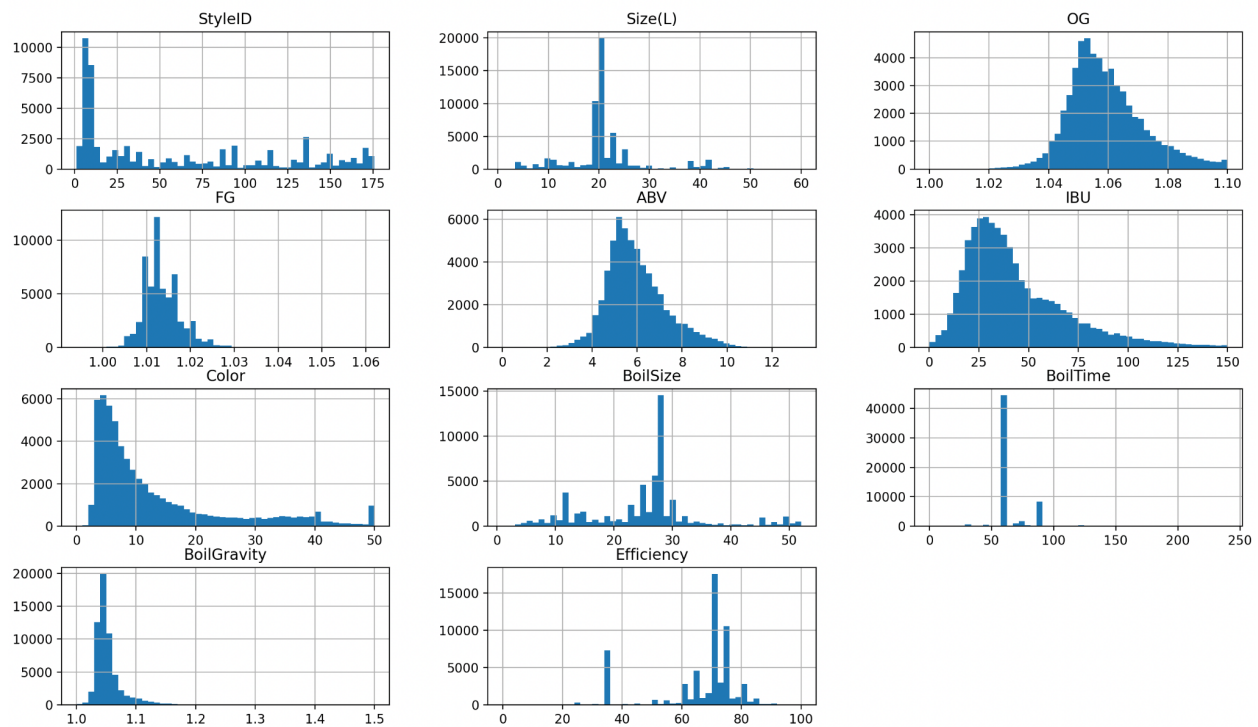
Les pics les plus hauts correspondent à des valeurs erronées ou inutilisables, nous allons donc essayer de faire coller les données avec les valeurs réellement attendues.

	StyleID	Size(L)	OG	FG	ABV	IBU	Color	BoilSize	BoilTime	BoilGravity	Efficiency	BrewMethod
count	70871.000000	70871.000000	70871.000000	70871.000000	70871.000000	70871.000000	70871.000000	70871.000000	70871.000000	70871.000000	70871.000000	70871
unique	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4
top	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	All Grain
freq	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	47778
mean	59.801738	44.849624	1.416692	1.077640	6.130476	44.782874	13.338915	50.795041	65.126046	1.353955	66.214736	NaN
std	56.793392	183.982380	2.229167	0.438967	1.877925	42.708998	11.896019	197.085525	15.037039	1.930989	14.157417	NaN
min	1.000000	1.000000	1.000000	-0.003000	0.000000	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	NaN
25%	10.000000	18.930000	1.051000	1.011000	5.070000	23.900000	5.150000	21.000000	60.000000	1.040000	65.000000	NaN
50%	34.000000	20.820000	1.058000	1.013000	5.790000	36.160000	8.350000	28.000000	60.000000	1.047000	70.000000	NaN
75%	109.000000	24.000000	1.068000	1.017000	6.820000	56.740000	16.670000	30.000000	60.000000	1.060000	75.000000	NaN
max	176.000000	9200.000000	34.034500	23.424600	54.720000	3409.300000	50.000000	9700.000000	240.000000	52.600000	100.000000	NaN

On regarde l'écart-type et les valeurs à 75% puis on essaie de voir lesquelles sont aberrantes puis on les dé-sélectionnent de notre DataFrame.

```
df = df[df["Size(L)"] <= df["Size(L)"].quantile(0.95)]
df = df[df["OG"] <= df["OG"].quantile(0.95)]
df = df[(df["IBU"] <= 150) & (df["IBU"] > 0)] # IBU max == 150 selon wikipedia
df = df[df["BoilSize"] <= df["BoilSize"].quantile(0.95)]
```

On se retrouve alors avec des courbes de valeurs beaucoup plus cohérentes et utilisables par nos futurs modèles.





Grâce à nos données épurées, on peut enfin commencer à chercher les différentes corrélations qui existent entre toutes les colonnes et IBU et ABV.

```
# Calculer la matrice de corrélation (ici, la méthode Pearson)
corr_matrix = df.corr(method='pearson')

# Créer la figure et l'axe
fig, ax = plt.subplots(figsize=(8, 6))

# Afficher la matrice de corrélation avec plt.matshow()
cax = ax.matshow(corr_matrix, cmap='coolwarm')

# Ajouter une barre de couleur
fig.colorbar(cax)

# Définir les étiquettes des axes
ax.set_xticks(range(len(corr_matrix.columns)))
ax.set_yticks(range(len(corr_matrix.columns)))
ax.set_xticklabels(corr_matrix.columns, rotation=45)
ax.set_yticklabels(corr_matrix.columns, rotation=45)

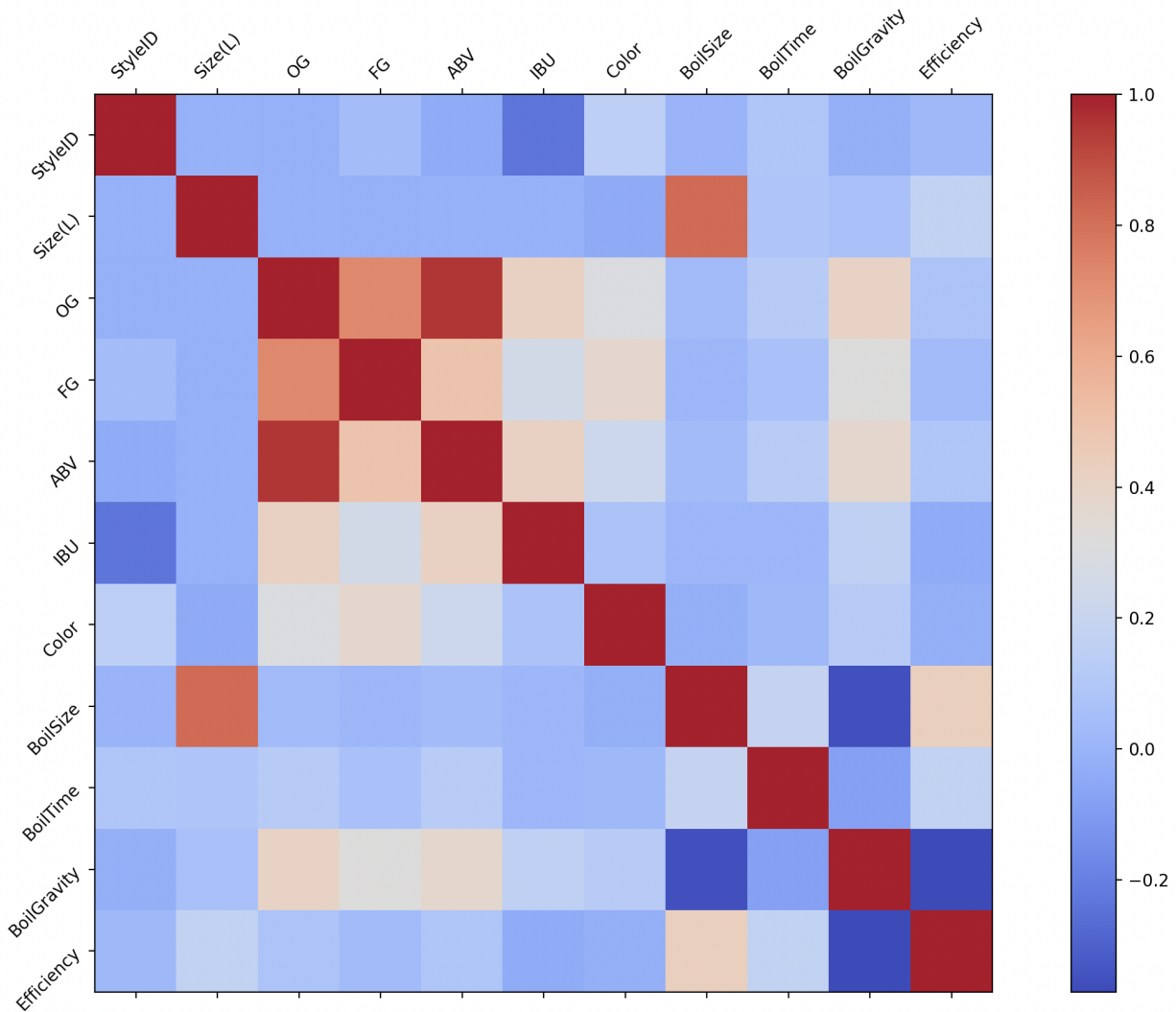
fig, axes = plt.subplots(nrows=10, ncols=20)
for ax, col in zip(axes.flatten(), df.columns):
    ax.scatter(df[col], df["IBU"])
    ax.set_title("Column: %s" % col)

df = df.drop(columns=["ABV"])
# Calculer la matrice de corrélation entre IBU et les autres variables
correlation_with_ibu = df.corr()["IBU"]

# Afficher les corrélations avec IBU
print(correlation_with_ibu)
```



Pour cela on trace une matrice de corrélation, avec la méthode pearson, qui nous permet d'avoir :



On voit la corrélation qu'il existe entre OG et ABV. Cela vient du fait que l'alcool est produit par le sucre pendant la fermentation. Donc si on prend OG-FG, le delta correspond au moût transformé en alcool pendant la fermentation, on obtient donc une corrélation parfaite entre le delta et ABV.

Malgré ça, nous avons décidé de ne pas nous en servir pour une question de logique. FG et aussi la couleur de la bière sont des données obtenues dans les dernières étapes de la fabrication de la bière or on veut que notre prédiction puisse être faite avant la fabrication totale.

Nous ajoutons donc aussi FG et Color dans les colonnes drop de notre dataset.

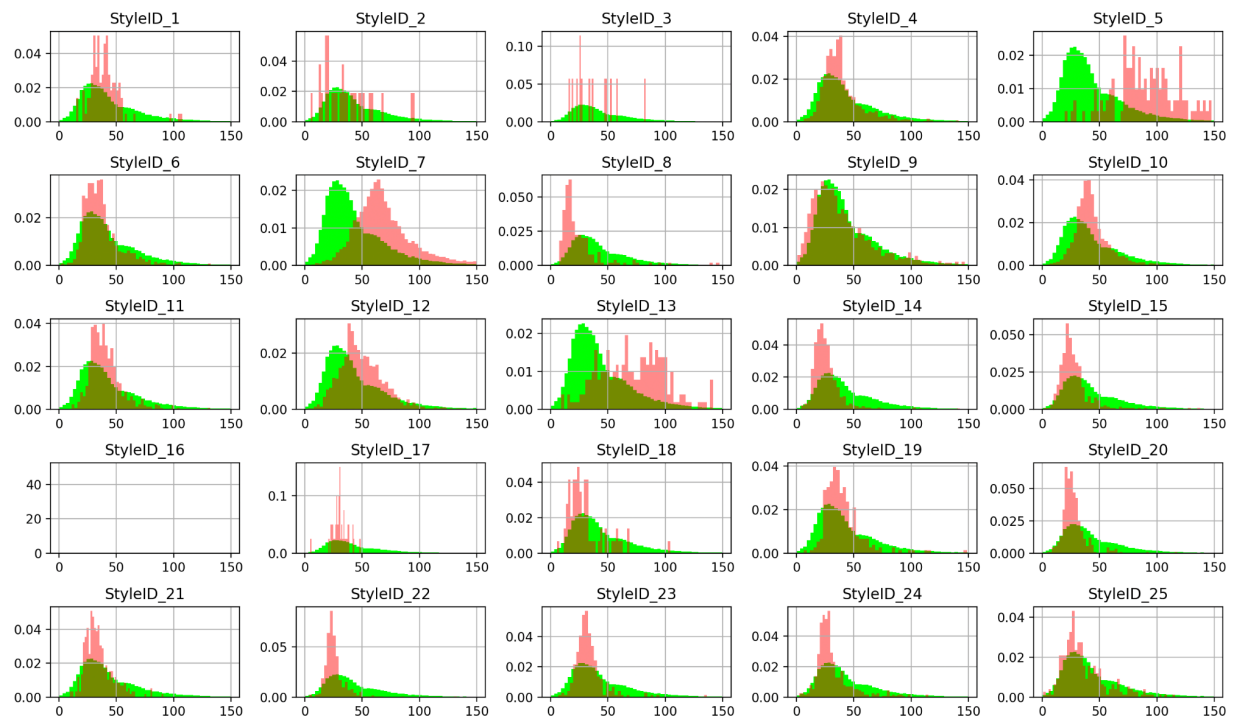
```
df = df.drop(
    columns=["BeerID", "UserID", "URL", "Name", "Style", "PrimingMethod", "PrimingAmount", "PitchRate", "MashThickness",
            "PrimaryTemp", "SugarScale", "FG", "Color"])
```

Après avoir découvert que OG et ABV sont fortement corrélés. On a cherché pour l'IBU ce qui pouvait être corrélé cependant rien ne sortait du lot et permettait de faire une prédiction précise de l'amertume.

Nous avons donc cherché sur internet quelle était la raison de l'amertume de la bière et ça nous a ressortis que la principale raison était le houblon. Or nous n'avions pas cette donnée dans notre dataset.

Afin de trouver un lien plus précis on a vu en essayant nos modèles que ce qui ressortait le plus, pour la prédiction de l'amertume, était le style de bière choisi. Nous avons donc fait un OneHot sur cette colonne afin de voir tous les styles de bières représenté par une colonne True/False et ainsi voir les styles les plus amères et possiblement améliorer nos prédictions.

```
one_hot = pd.get_dummies(df[["StyleID"]], columns=["StyleID"])
df = df.join(one_hot)
```



On superpose la courbe IBU avec celle de chacun des styles de bières.

On peut alors voir que le style 17 (Australian Sparkling Ale) est peu amère alors que le style 5 (American Barleywine) l'est beaucoup plus.

Cela permet d'améliorer un petit peu nos data. Mais malheureusement le volume de style différent rend quand même la prédiction peu efficace.

## II. IA

Pour la création de nos modèles, nous avons, dans un premier temps, choisi une régression linéaire. Ça nous a donné des résultats plutôt correct pour ABV mais assez catastrophique pour IBU :

```
df_X = df.drop(columns=["ABV", "IBU"])
df_X_IBU = df.drop(columns=['Size(L)', 'ABV', 'IBU', 'BoilSize', 'BoilTime'])
df_y = df[["ABV", "IBU"]]
df_y_ABV = df["ABV"]
df_y_IBU = df["IBU"]

X_train_ABV, X_test_ABV, y_train_ABV, y_test_ABV = train_test_split(df_X, df_y_ABV)
X_train_IBU, X_test_IBU, y_train_IBU, y_test_IBU = train_test_split(df_X_IBU, df_y_IBU)

reg_ABV = LinearRegression().fit(X_train_ABV, y_train_ABV)
reg_IBU = LinearRegression().fit(X_train_IBU, y_train_IBU)
```

```
Erreur MSE RL_ABV : 0.1462873748466414
Erreur MSE RL_IBU : 566.9539796536155
```

Nous avons alors décidé d'utiliser d'autre type de régression, nous avons donc essayé le random forest, qui crée un arbre de décision, et Multi-layer Perceptron, qui est un réseau neuronal.

Nous avons alors comparé leurs résultats pour voir lequel donnait, en général, de meilleurs résultats.

```
Erreur MSE RM_ABV : 0.43644803689326545
Erreur MSE RM_IBU : 325.7915052491038
Erreur MSE RA_ABV : 0.14555731709004008
Erreur MSE RA_IBU : 343.29581802755837
```

Une fois le choix des random forest en guise de regressor, nous devons ensuite définir les hyper paramètre.

Pour cela, nous utilisons HalvingGridSearch afin de les déterminer.

```
# Define the hyperparameter grids for each model
param_grid_rf = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2']
}

# Split the data into training and testing sets
X_train_IBU, X_test_IBU, y_train_IBU, y_test_IBU = train_test_split(df_X_IBU, df_y_IBU)

# Create the models
rf = RandomForestRegressor()

# Create HalvingGridSearchCV instances for each model
search_rf = HalvingGridSearchCV(rf, param_grid_rf, scoring='neg_mean_squared_error', n_jobs=-1)

# Fit the models with hyperparameter tuning
search_rf.fit(X_train_IBU, y_train_IBU)

# Get the best models with optimized hyperparameters
best_rf = search_rf.best_estimator_

print("Best parameters :", best_rf)

# Evaluate the models
mse_rf = mean_squared_error(y_test_IBU, best_rf.predict(X_test_IBU))

print("Best Random Forest MSE:", mse_rf)
```

On lui donne plusieurs hyper paramètre et on essaie à chaque fin d'itération de les rendre plus précis.

En plus des hyper paramètres, on choisit sur quel type de score on va évaluer notre modèle et on choisit ici le MSE donc l'erreur moyenne quadratique. Cela permet de voir surtout à quel point les erreurs sont loin de la valeur réelle au carré.

```
Best parameters : RandomForestRegressor(max_features='log2', min_samples_split=10,
                                         n_estimators=200)
Best Random Forest MSE: 301.6441951006349
```

Après plusieurs itérations, et l'utilisation de `range()` afin de tester plus hyper paramètre on obtient :

```
param_grid_rf = {
    'n_estimators': range(150, 250, 20),
    'min_samples_split': range(6, 22, 2),
}
```

```
Best parameters : RandomForestRegressor(max_features='log2', min_samples_split=14,
                                         n_estimators=230)
Best Random Forest MSE: 308.7365054906372
```

Pour finir notre IA, on va mettre enfin en forme tout ce que nous avons fait pour notre IA et la sauvegarder dans un point pickle (.pkl) afin de ne pas refaire l'entraînement à chaque prédiction.

```
# Save your trained models
joblib.dump(regr_ABV, 'models/random_forest_ABV.pkl')
joblib.dump(regr_IBU, 'models/random_forest_IBU.pkl')
```

Et finalement, on va le tester afin de vérifier qu'il est bien utilisable :

```
# Load trained models
ABV_model = joblib.load('models/random_forest_ABV.pkl')
IBU_model = joblib.load('models/random_forest_IBU.pkl')

# Test the models
y_pred_ABV = ABV_model.predict(X_test_ABV)
y_pred_IBU = IBU_model.predict(X_test_IBU)

# Calculate Mean Squared Error for both models
mse_ABV = mean_squared_error(y_test_ABV, y_pred_ABV)
mse_IBU = mean_squared_error(y_test_IBU, y_pred_IBU)

print("MSE for ABV Model:", mse_ABV)
print("MSE for IBU Model:", mse_IBU)
```

On obtient ces résultats:

```
MSE for ABV Model: 0.0552960697754256
MSE for IBU Model: 226.5760522835589
```

### III. application web

Pour la création de l'interface on a d'abord dû récupérer les champs utilisé pour la prédiction des valeurs ABV et IBU soit 'StyleID', 'Size(L)', 'OG', 'BoilSize', 'BoilTime', 'BoilGravity', et 'Efficiency'.

Le 'StyleID' est choisi par un selector alors que les autres champs sont saisis dans des inputs classiques afin de limiter le choix des ids à ceux disponibles à la prédiction.

## Formulaire

Style :

Size(L) :

OG :

BoilSize :


BoilTime :

BoilGravity :

Efficiency :

Les champs sont requis afin de lancer la prédiction, puisque chaque variable manquante risquerait de réduire la fiabilité de la prédiction.

Size(L) :

OG :  Veuillez renseigner ce champ.

Le selector utilise un tableau dans lequel se trouve tous les différents styles utilisables avec chaque style ayant pour id, "son index dans le tableau" + 1.

```
beers_style = [
    "Altbier",
    "Alternative Grain Beer",
    "Alternative Sugar Beer",
    "American Amber Ale",
    "American Barleywine",
    "American Brown Ale",
    "American IPA",
    "American Lager",
    "American Light Lager",
    "American Pale Ale",
    "American Porter",
    "American Stout",
    "American Strong Ale",
```



Les prédictions sont ensuite affichées sous forme de simple labels en dessous du formulaire.

## Formulaire

Style :

Size(L) :

OG :

BoilSize :

BoilTime :

BoilGravity :

Efficiency :

Prédiction ABV : [10.2873]  
Prédiction IBU : [38.2764992]

Pour faire le lien entre l'interface html nous avons utilisé [FLASK](#) pour faire une api REST avec une simple route en GET et en POST pour accéder au formulaire.

```
app = Flask(__name__)

@app.route("/", methods=["GET", "POST"])
def formulaire():
    if request.method == "POST":...

    # Si la méthode est GET ou si le formulaire n'a pas encore été soumis, afficher le formulaire vide
    return render_template(
        "formulaire.html",
        style="",
        size_l="",
        og="",
        boil_size="",
        boil_time="",
        boil_gravity="",
        efficiency="",
        beers_style=beers_style,
    )

if __name__ == "__main__":
    app.run()
```

Comme indiqué sur le screen précédent, la méthode GET renvoie un formulaire vide, tandis que pour la méthode POST, nous avons tout le traitement de prédiction, de la récupération des données dans la request au renvoi du formulaire avec les

prédictions, en passant par la transformation des données en dataframe ainsi que le chargement des modèles entraînés et la prédiction d'ABV et d'IBU.

```

if request.method == "POST":
    # Récupérer les données soumises par le formulaire
    style = request.form["style"]
    style_id = beers_style.index(style) + 1
    size_l = request.form["size_l"]
    og = request.form["og"]
    boil_size = request.form["boil_size"]
    boil_time = request.form["boil_time"]
    boil_gravity = request.form["boil_gravity"]
    efficiency = request.form["efficiency"]

    data = {
        'Size(L)': [size_l],
        'OG': [og],
        'BoilSize': [boil_size],
        'BoilTime': [boil_time],
        'BoilGravity': [boil_gravity],
        'Efficiency': [efficiency],
        'OGPoly': [float(og) + float(og) ** 2], # Vous avez mentionné 'OG2'
    }

    for i in range(1,177):
        if style_id == i:
            data[f'StyleID_{i}'] = True
        else:
            data[f'StyleID_{i}'] = False

    df = pd.DataFrame(data)

    # Load trained models
    ABV_model = joblib.load('models/random_forest_ABV.pkl')
    IBU_model = joblib.load('models/random_forest_IBU.pkl')

    # Test the models
    y_pred_ABV = ABV_model.predict(df)
    y_pred_IBU = IBU_model.predict(df)

    # Retourner les données soumises dans le formulaire
    return render_template(
        "formulaire.html",
        style = style,
        size_l = size_l,
        og = og,
        boil_size = boil_size,
        boil_time = boil_time,
        boil_gravity = boil_gravity,
        efficiency = efficiency,
        beers_style = beers_style,
        y_pred_ABV = y_pred_ABV,
        y_pred_IBU = y_pred_IBU
    )

```