Bike Sharing Demand

April 16, 2022

#

Prédiction de la Location de Vélos

Table of Contents

Entraînement et Évaluation de Modèles

Métriques d'Évaluation

Régression Linéaire

Decision Tree

Random Forest

Gradient Boosting

Comparaison des Modèles

Conclusion

0.1 Introduction

Le but de ce projet consiste à entraîner un modèle capable de prédire le nombre de vélos loués à n'importe quel temps t dans des bornes libres-services de la ville (système type Vélib'). La variable cible à prédire est ici la variable `count`.

Voici un descriptif de l'ensemble des variables du jeu de données: * `datetime` - date et heure du relevé * `season` - 1 = printemps , 2 = été, 3 = automne, 4 = hiver * `holiday` – indique si le jour est un jour de vacances scolaires * `workingday` - indique si le jour est travaillé (ni week-end ni vacances) * `weather` - 1: Dégagé à nuageux, 2 : Brouillard, 3 : Légère pluie ou neige, 4 : Fortes averses ou neige * `temp` – température en degrés Celsius * `atemp` – température ressentie en degrés Celsius * `humidity` – taux d'humidité * `windspeed` – vitesse du vent * `casual` - nombre de locations d'usagers non abonnés * `registered` – nombre de locations d'usagers abonnés * `count` – nombre total de locations de vélos

0.2 Import des Librairies

[1]:
```
import pprint
import calendar
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import metrics
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.model_selection import train_test_split, RandomizedSearchCV,␣
↪GridSearchCV
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.base import BaseEstimator
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
```

0.3 Configuration des Paramètres d'Affichage du Notebook

[2]:
```
pp = pprint.PrettyPrinter(indent=4)
sns.set_style("whitegrid")
```

0.4 Import du Dataset

[3]:
```
dataset = pd.read_csv("data/velo.csv")

dataset.info(verbose=True, show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   datetime    10886 non-null  object
 1   season      10886 non-null  int64
 2   holiday     10886 non-null  int64
 3   workingday  10886 non-null  int64
 4   weather     10886 non-null  int64
 5   temp        10886 non-null  float64
 6   atemp       10886 non-null  float64
 7   humidity    10886 non-null  int64
 8   windspeed   10886 non-null  float64
 9   casual      10886 non-null  int64
 10  registered  10886 non-null  int64
```

```
 11  count       10886 non-null  int64
dtypes: float64(3), int64(8), object(1)
memory usage: 1020.7+ KB
```

Le jeu de données ne comporte pas de valeurs manquantes.

Affichons les premières lignes pour nous faire une meilleure idée des données qu'il contient.

[4]:
```
dataset.head(5)
```

[4]:

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0	3	10	13
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0	0	1	1

0.5 Extraction d'Informations Temporelles de `datetime`

Avant de passer à l'exploration du dataset, nous extrayons l'année, le jour et l'heure des timestamps de la colonne `datetime`. Nous créons ainsi trois nouvelles colonnes `year`, `weekday` et `hour`.

Plutôt que d'extraire le jour en chiffres (1 à 30/31), nous extrayons le numéro du jour de la semaine. Cela nous permet d'avoir une colonne avec moins de catégories. Il nous apparait également plus pertinent de pouvoir distinguer les différents jours de la semaine. On s'attendra à avoir des différences dans le nombre de locations entre la semaine et le weekend. Il pourrait également y avoir une différence entre certains jours de la semaine. Les Mercredi et les Jeudi étant des jours préférés pour le télétravail, il se pourrait que les utilisateurs réguliers du service qui l'utilise pour leur commuting ne loue pas de vélo sur ces jours.

Nous n'exportons pas le mois car l'évolution de la demande de location au long de l'année est déjà portée par les colonnes `season` et `holiday`.

[5]:
```
# Conversion de la colonne `datetime` au type datetime
dataset["datetime"] = pd.to_datetime(dataset["datetime"])

# Extraction de l'année des timestamps et insertion de la nouvelle colonne␣
 ↪`year` dans le dataset
year = dataset["datetime"].dt.year
dataset.insert(loc=1, column="year", value=year)

# Extraction du jour de la semaine des timestamps et insertion de la nouvelle␣
 ↪colonne `weekday` dans le dataset
```

```
weekday = dataset["datetime"].apply(lambda x: x.weekday()+1)
dataset.insert(loc=3, column="weekday", value=weekday)

# Extraction de l'heure des timestamps et insertion de la nouvelle colonne␣
  ↪`hour` dans le dataset
hour = dataset["datetime"].dt.hour
dataset.insert(loc=6, column="hour", value=hour)

dataset
```

[5]:

	datetime	year	season	weekday	holiday	workingday	hour	\
0	2011-01-01 00:00:00	2011	1	6	0	0	0	
1	2011-01-01 01:00:00	2011	1	6	0	0	1	
2	2011-01-01 02:00:00	2011	1	6	0	0	2	
3	2011-01-01 03:00:00	2011	1	6	0	0	3	
4	2011-01-01 04:00:00	2011	1	6	0	0	4	
…	…	…	…	…	…	…	…	
10881	2012-12-19 19:00:00	2012	4	3	0	1	19	
10882	2012-12-19 20:00:00	2012	4	3	0	1	20	
10883	2012-12-19 21:00:00	2012	4	3	0	1	21	
10884	2012-12-19 22:00:00	2012	4	3	0	1	22	
10885	2012-12-19 23:00:00	2012	4	3	0	1	23	

	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	1	9.84	14.395	81	0.0000	3	13	16
1	1	9.02	13.635	80	0.0000	8	32	40
2	1	9.02	13.635	80	0.0000	5	27	32
3	1	9.84	14.395	75	0.0000	3	10	13
4	1	9.84	14.395	75	0.0000	0	1	1
…	…	…	…	…	…	…	…	
10881	1	15.58	19.695	50	26.0027	7	329	336
10882	1	14.76	17.425	57	15.0013	10	231	241
10883	1	13.94	15.910	61	15.0013	4	164	168
10884	1	13.94	17.425	61	6.0032	12	117	129
10885	1	13.12	16.665	66	8.9981	4	84	88

[10886 rows x 15 columns]

0.6 Conversion des Types des Colonnes

Nous convertissons ensuite le type des colonnes.

[6]:
```
categorical_cols = [
    "holiday",
    "workingday",
    "weather"
]
```

```
for col in categorical_cols:
    dataset[col] = dataset[col].astype("category")

dataset["temp"] = dataset["temp"].astype("int")
dataset["atemp"] = dataset["atemp"].astype("int")
dataset["windspeed"] = round(dataset["windspeed"], 1)

dataset.info(verbose=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10886 entries, 0 to 10885
Data columns (total 15 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   datetime    10886 non-null  datetime64[ns]
 1   year        10886 non-null  int64
 2   season      10886 non-null  int64
 3   weekday     10886 non-null  int64
 4   holiday     10886 non-null  category
 5   workingday  10886 non-null  category
 6   hour        10886 non-null  int64
 7   weather     10886 non-null  category
 8   temp        10886 non-null  int64
 9   atemp       10886 non-null  int64
 10  humidity    10886 non-null  int64
 11  windspeed   10886 non-null  float64
 12  casual      10886 non-null  int64
 13  registered  10886 non-null  int64
 14  count       10886 non-null  int64
dtypes: category(3), datetime64[ns](1), float64(1), int64(10)
memory usage: 1.0 MB
```

0.7 Nettoyage du Dataset

Comme nous avons pu le voir plus haut, le jeu de données ne présente pas de valeurs manquantes. Aucune action d'imputation ou de suppression de variables ou d'observations n'est donc nécessaire.

Pour ce qui est des outliers, nous calculons et affichons quelques statistiques sur les colonnes numériques pour nous permettre de détecter leur présence.

[7]:
```
dataset.describe(percentiles=[.25, .50, .75, .95])
```

[7]:

	year	season	weekday	hour	temp	\
count	10886.000000	10886.000000	10886.000000	10886.000000	10886.000000	
mean	2011.501929	2.506614	4.013963	11.541613	19.740492	
std	0.500019	1.116174	2.004585	6.915838	7.792108	
min	2011.000000	1.000000	1.000000	0.000000	0.000000	

```
25%     2011.000000      2.000000      2.000000      6.000000     13.000000
50%     2012.000000      3.000000      4.000000     12.000000     20.000000
75%     2012.000000      4.000000      6.000000     18.000000     26.000000
95%     2012.000000      4.000000      7.000000     22.000000     32.000000
max     2012.000000      4.000000      7.000000     23.000000     41.000000

              atemp      humidity     windspeed        casual    registered  \
count  10886.000000  10886.000000  10886.000000  10886.000000  10886.000000
mean      23.185468     61.886460     12.799100     36.021955    155.552177
std        8.500893     19.245033      8.164634     49.960477    151.039033
min        0.000000      0.000000      0.000000      0.000000      0.000000
25%       16.000000     47.000000      7.000000      4.000000     36.000000
50%       24.000000     62.000000     13.000000     17.000000    118.000000
75%       31.000000     77.000000     17.000000     49.000000    222.000000
95%       36.000000     93.000000     28.000000    141.000000    464.000000
max       45.000000    100.000000     57.000000    367.000000    886.000000

              count
count  10886.000000
mean     191.574132
std      181.144454
min        1.000000
25%       42.000000
50%      145.000000
75%      284.000000
95%      563.750000
max      977.000000
```

Il semble pas qu'il n'y ai pas de valeurs abhérantes dans notre jeu de données. Les valeurs maximales de `windspeed`, de `casual` et de `registered` sont plutôt élévées par rapport à leur 95e centile et leurs médianes respectives. Néanmoins, après quelques recherches, il semblerait qu'une vitesse du vent de 57mph est observable dans le cas de tempêtes d'intensité modérée. Pour ce qui est des maximums de location, il se peut qu'un évènement particulier est favorisé ceux-ci (grêve des transports, journée sans voitures, …).

Nous laissons ainsi le jeu de données tel quel et passons à son analyse.

0.8 Analyse du Dataset

Le jeu de données est désormais prêt à être analysé. Dans cette section, nous nous intéressons à l'évolution du nombre total de locations (variable à prédire `count`) en fonction des différentes variables prédictives. Nous effectuons aussi une analyse de corrélations.

0.8.1 Évolution temporelle du nombre de locations

Commençons par tracer l'évolution temporelle du nombre de locations en ne prenant en compte que les timestamps.

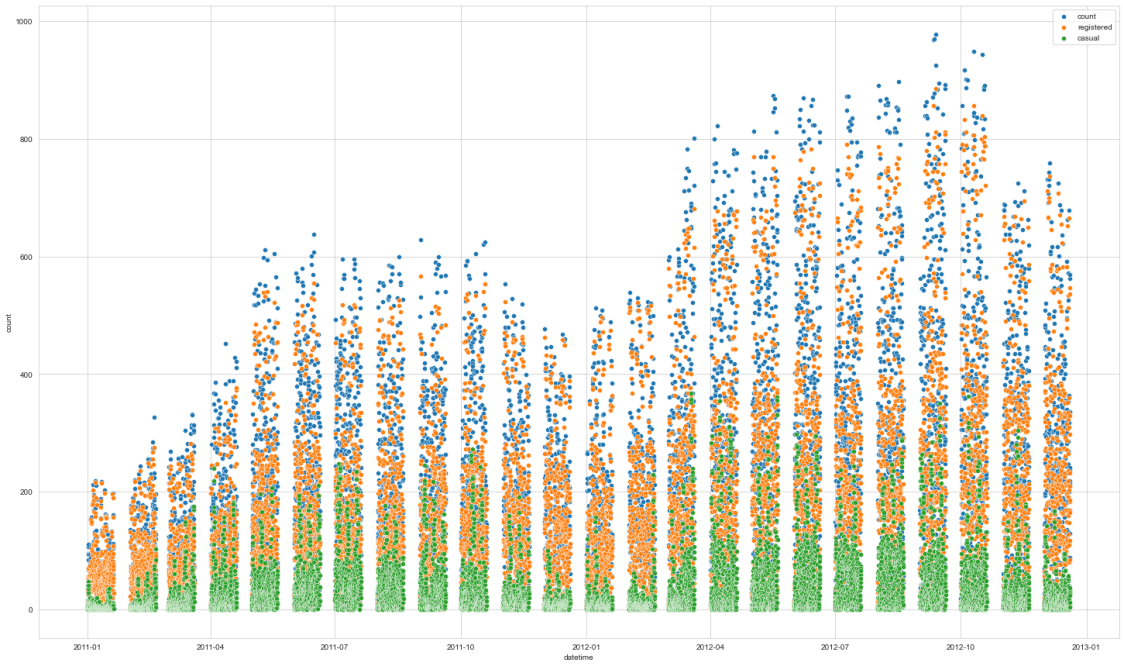
[8]:
```
first_ts = dataset["datetime"].min()
last_ts = dataset["datetime"].max()

print(f"First recorded timestamp: {first_ts}")
print(f"Last recorded timestamp: {last_ts}")
```

```
First recorded timestamp: 2011-01-01 00:00:00
Last recorded timestamp: 2012-12-19 23:00:00
```

Le dataset couvre une période d'environ deux ans, à raison d'un enregistrement par heure.

[9]:
```
plt.figure(figsize=(25,15))
sns.scatterplot(x="datetime", y="count", data=dataset)
sns.scatterplot(x="datetime", y="registered", data=dataset)
sns.scatterplot(x="datetime", y="casual", data=dataset)
plt.legend(["count", "registered", "casual"])
plt.show()
```



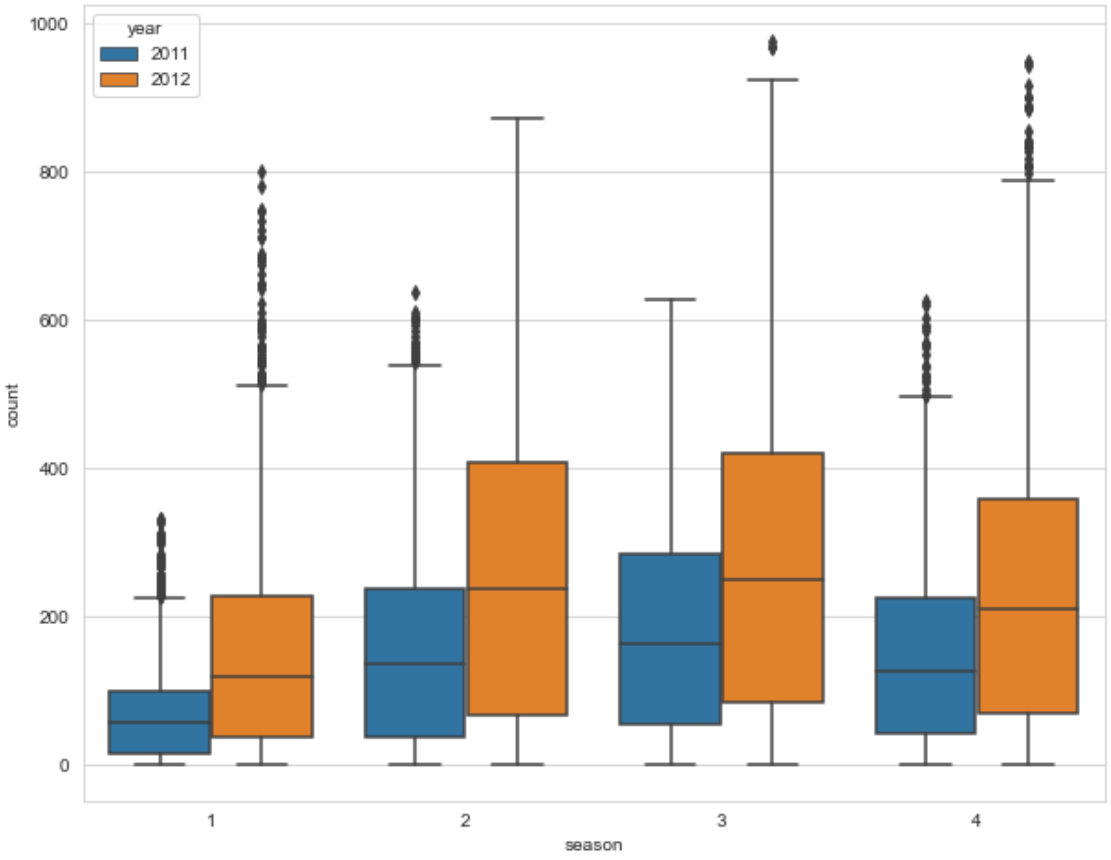
Nous pouvons clairement observer une saisonalité dans l'évolution du nombre total de locations au sein d'une même année, ainsi qu'un offset entre l'année 2011 et l'année 2012, cette dernière ayant enregistré significativement plus de locations.

Il apparaît que le nombre de locations `casual` a moins augmenté d'une année sur l'autre que le nombre de locations `registered`. Ce type de service est très axé sur les "commuters". Il semble donc logique que, suite à une campagne d'information et de publicité, le nombre d'utilisateurs réguliers ait augmenté d'une année sur l'autre.

0.8.2 Influence de la saison sur le nombre total de locations

Ci-dessous, nous traçons le nombre total de locations par jour pour chaque saison et ce pour l'année 2011 et 2012 pour voir si une tendance se répète d'une année sur l'autre.

[10]:
```
plt.figure(figsize=(10,8))
sns.boxplot(x="season", y="count", data=dataset, hue="year")
plt.show()
```

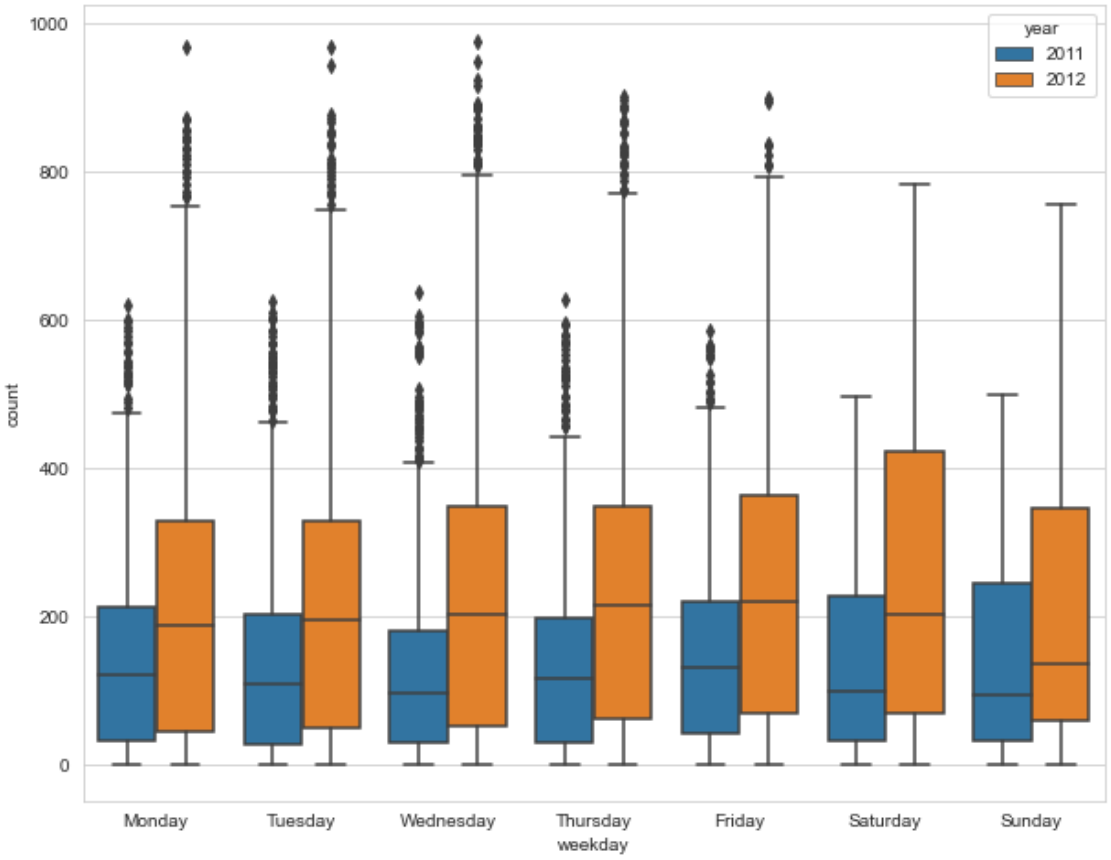


Nous observons bien la saisonalité que nous avions déjà observé sur le graphe précédent. Sur les deux années consécutives, le printemps et l'été ont, logiquement, été les deux saisons où le nombre de locations-jour étaient le plus élévé. Cela est surement dû à la météo plus favorable qui pousse plus de gens à utiliser ce moyen de transports tant pour le commuting que pour le loisir et à l'affluence de touristes sur ces deux saisons.

0.8.3 Influence du jour de la semaine sur le nombre total de locations

Intéressons-nous maintenant à l'évolution du nombre total de locations au cours d'une semaine. Quels sont les jours pour lesquels la demande est la plus forte?

[11]:
```
plt.figure(figsize=(10,8))
ax = sns.boxplot(x="weekday", y="count", data=dataset, hue="year")
ax.set_xticklabels(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",␣
 ↪"Saturday", "Sunday"])
plt.show()
```



Nous observons une tendance légèrement différente entre l'année 2011 et l'année 2012.

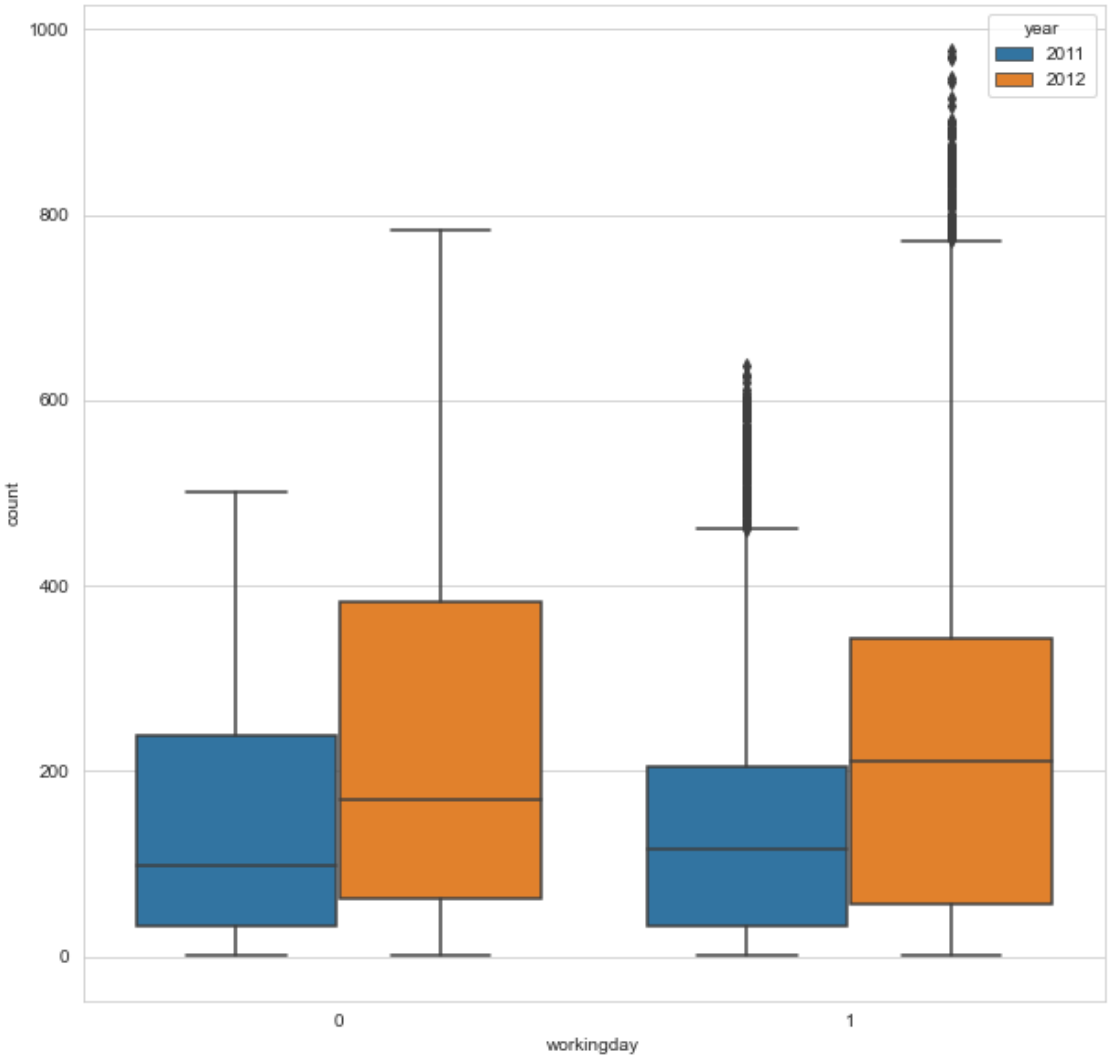
Sur l'année 2011, le nombre total de locations était, en se basant sur la médiane pour cette période, plus élevé les Lundi, Mardi, Jeudi et Vendredi. Le Mercredi étant un jour préféré pour le télé-travail, il est possible que les utilisateurs réguliers aient eu en moyenne moins recours au service de location ce jour-ci. Sur cette même année, la demande semblait plus faible mais plus variable les Samedi et Dimanche. Cela était peut-être dû à une part moins importante des utilisateurs `casual` par rapport aux utilisateurs `registered`.

Pour l'année 2012, la demande était relativement constante au cours des cinq jours travaillés de la semaine, avec une augmentation sensible à l'approche du week-end. La plus grande variation de la demande les Samedi traduit peut-être de l'influence de la météo sur l'utilisation de ce service pour le loisir sur ce jour de weekend. Les Dimanche, la demande était significativement plus faible.

Dans la continuité du graphe précédent, nous pouvons tracer le nombre total de locations en fonction seulement du statut du jour (travaillé ou non) en prenant en compte la colonne `workingday`,

originellement présente dans le jeu de données.

[12]:
```
plt.figure(figsize=(10,10))
sns.boxplot(x="workingday", y="count", data=dataset, hue="year")
plt.show()
```

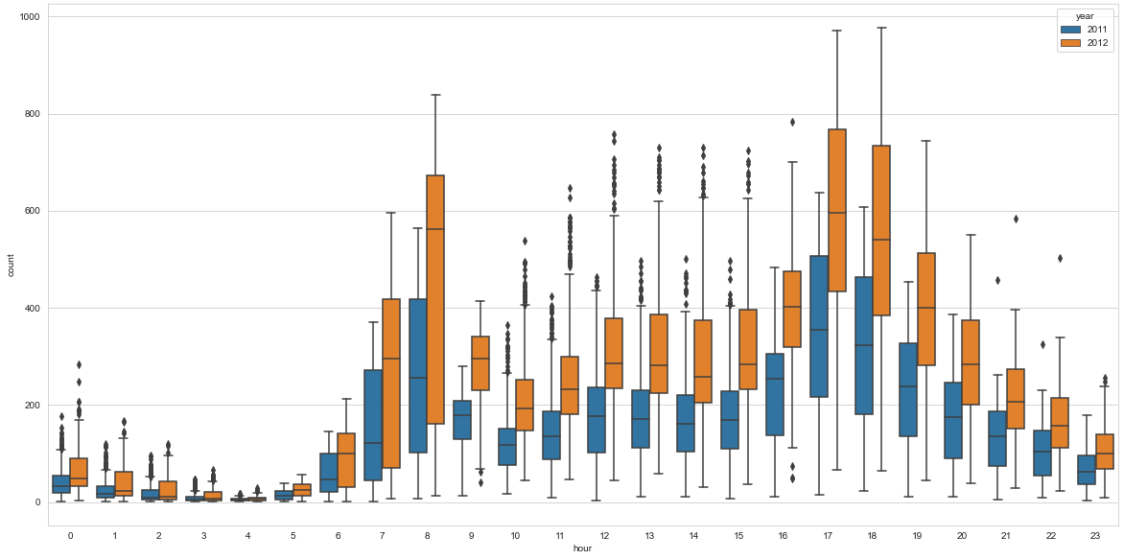


La tendance est similaire sur les deux années. Le nombre total de locations fût sensiblement plus élevé pendant les jours travaillés. Cela confirme la part importante de commuters parmi les utilisateurs de ce service.

0.8.4 Influence de l'heure de la journée sur le nombre total de locations

Il est également intéressant de regarder l'évolution du nombre total de locations en fonction de l'heure de la journée.

[13]:
```
plt.figure(figsize=(20,10))
sns.boxplot(x="hour", y="count", data=dataset, hue="year")
plt.show()
```

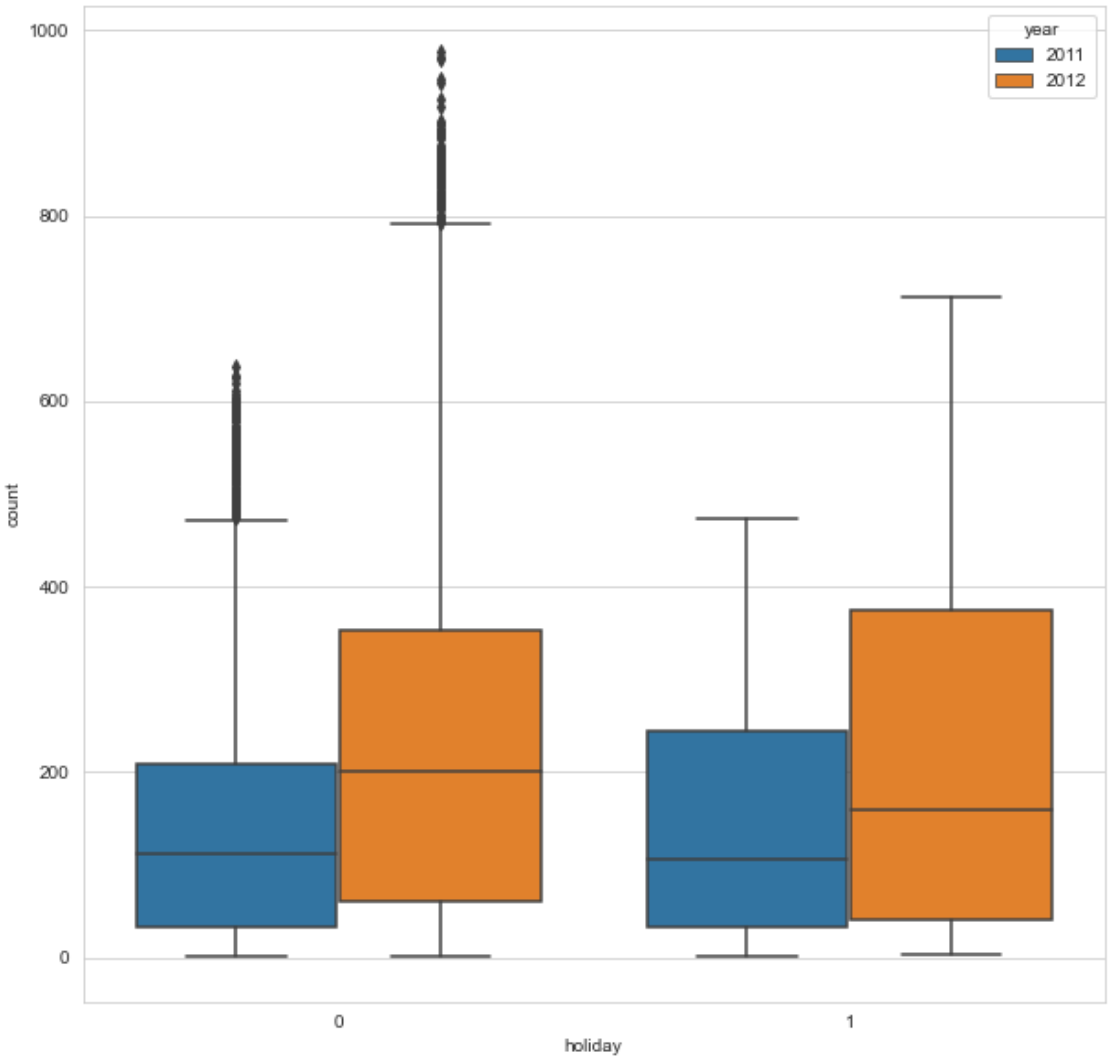


Nous observons une tendance similaire pour les années 2011 et 2012. Nous pouvons noter deux pics de demande sur les périodes horaires correspondant au commuting, de 6h à 9h et de 16h à 20h. Entre ces deux pics, la demande est moyennement élevée et peu variable. Après 20h, le nombre de locations ne cesse de baisser jusqu'à atteindre un minimum à 4h du matin.

0.8.5 Influence des vacances sur le nombre total de locations

Les vacances sont également spécifiées dans le dataset, par le biais de la colonne `holiday`.

[14]:
```
plt.figure(figsize=(10,10))
sns.boxplot(x="holiday", y="count", data=dataset, hue="year")
plt.show()
```

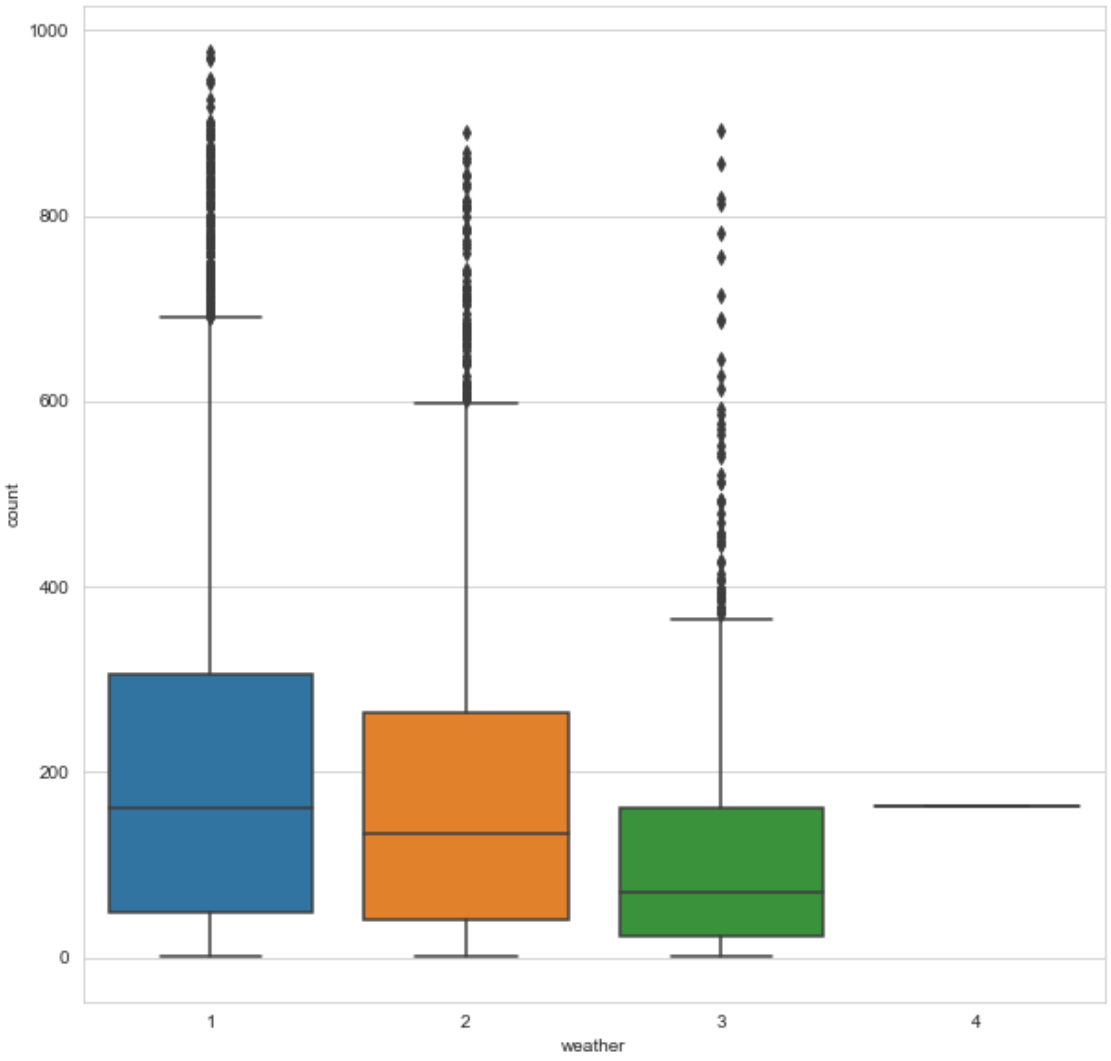


Le nombre total de locations n'est pas significativement plus important ou plus faible pendant les vacances. Pendant les périodes de vacances, il est possible que la location touristique ou de loisir compense la baisse du commuting.

0.8.6 Influence de la météo sur le nombre total de locations

L'information sur la météo est également dans le jeu de données. La variable `weather` donne une indication de la météo au moment de l'enregistrement en 4 niveaux.

[15]:
```
plt.figure(figsize=(10,10))
sns.boxplot(x="weather", y="count", data=dataset)
plt.show()
```

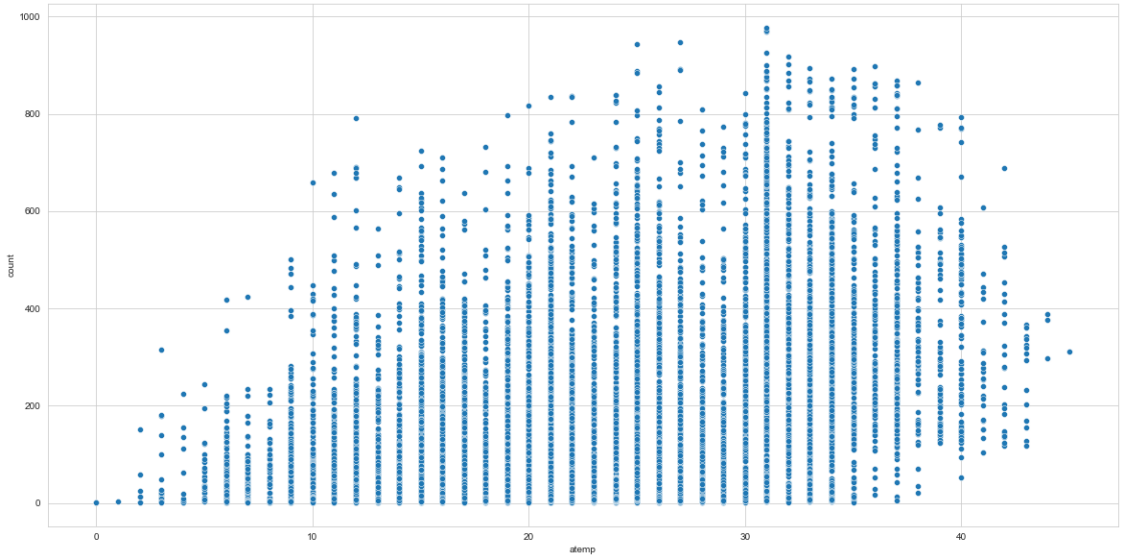


Comme nous pouvions nous y attendre, plus la météo est clémente, plus le nombre total de locations est élevé. À l'inverse, plus la météo est défavorable, moins la demande pour ce service est forte.

0.8.7 Influence de la température sur le nombre total de locations

Dans la lignée de l'analyse précédente, nous pouvons également observer l'évolution du nombre total de locations en fonction de la température.

[16]:
```
plt.figure(figsize=(20,10))
sns.scatterplot(x="atemp", y="count", data=dataset)
plt.show()
```

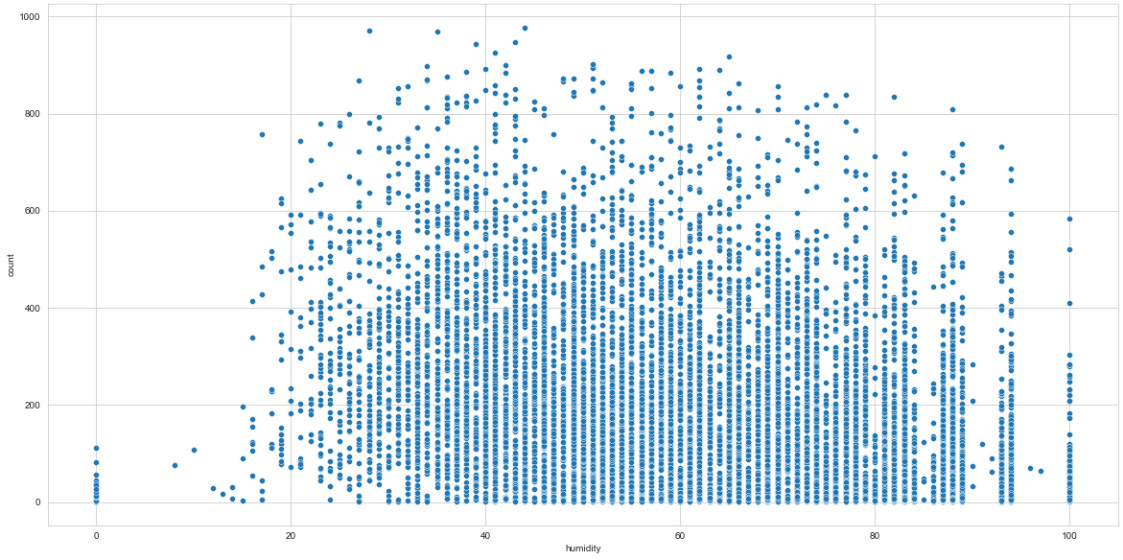


Globalement, des températures plus clémentes favorisent l'utilisation de ce service.

0.8.8 Influence de l'humidité sur le nombre total de locations

Nous pouvons également analyser l'évolution du nombre total de locations en fonction de l'humidité.

[17]:
```
plt.figure(figsize=(20,10))
sns.scatterplot(x="humidity", y="count", data=dataset)
plt.show()
```



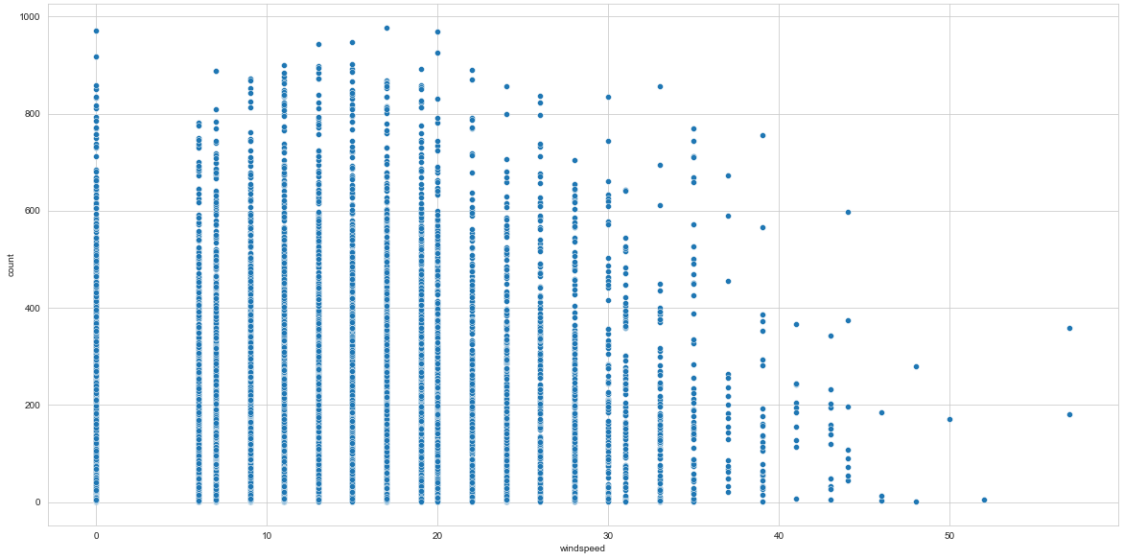
L'impact de l'humidité est moins flagrant, même si l'on peut tout de même déceler une baisse de

la demande pour des taux d'humidité plus élevés.

0.8.9 Influence de la vitesse du vent sur le nombre total de locations

Enfin, nous terminons par l'analyse de l'impact de la vitesse du vent sur le nombre total de locations.

[18]:
```
plt.figure(figsize=(20,10))
sns.scatterplot(x="windspeed", y="count", data=dataset)
plt.show()
```



Il apparaît que le nombre de locations est plus faible lorsque la vitesse du vent est plus élevée. La variable `windspeed` pourrait donc être pertinente pour notre tentative de prédiction du nombre de locations.

0.8.10 Résumé des observations

L'analyse de l'influence de chaque variable prédictice sur la variable à prédire `count` nous a permis de faire les observations suivantes: - le nombre total de locations a globalement augmenté d'une année sur l'autre. - la demande respecte une seasonalité au cours de l'année. - la saison impact la demande, probablement en lien avec la météo qui est plus clémente pour certaines saisons plutôt que pour d'autres. - le jour de la semaine peut être pertinent à prendre en compte car la demande pour ce service varie en fonction de celui-ci. - le caractère travaillé ou non du jour influe également sur la demande. - le nombre total de locations varie de façon encore plus importante en fonction de l'heure de la journée. - les vacances impactent moins la demande, et l'on peut supposer que les demandes pour le commuting et pour le loisir s'équilibrent. - La météo a également un fort impact sur le nombre total de locations. - Globalement, plusieurs variables de ce jeu de données portent des informations très proches (`season`, `weather` et `atemp` par exemple).

Dans la prochaine section, nous menons une étude de corrélations afin de voir quelles sont les variables prédictives qui ont une plus forte corrélation avec la variable à prédire `count`.

0.8.11 Analyse de Corrélations

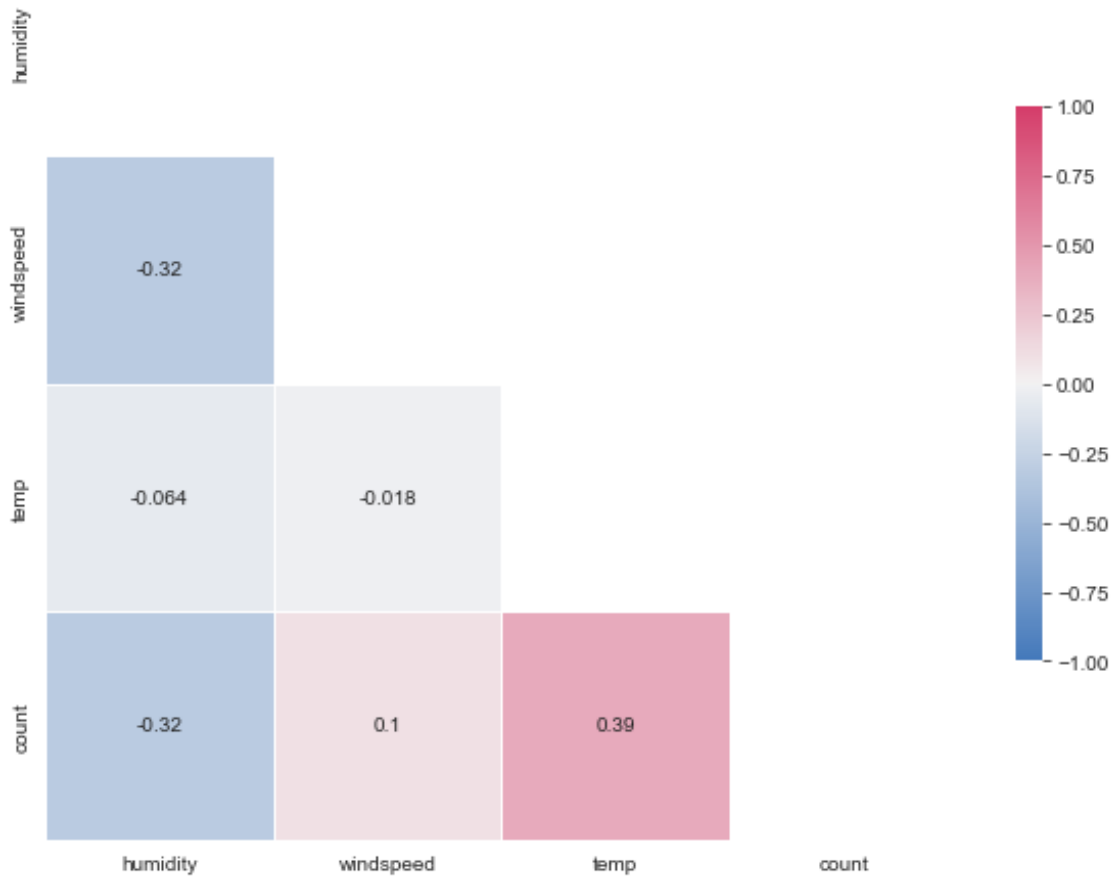
Pour compléter l'analyse du jeu de données, nous menons une analyse de corrélations sur les variables numériques. Il s'agit alors de voir à quel point les variables prédictives sont corrélées linéairement avec la variable à prédire.

[19]:
```
# Sélection des variables numériques
numerical_predictors = ["humidity", "windspeed", "temp", "count"]

# Calcul de la matrice de corrélation sur les variables numériques
corr = dataset[numerical_predictors].corr()

# Création d'un masque pour masquer la symmétrie de la matrice
mask = np.triu(np.ones_like(corr, dtype=bool))

# Traçage la matrice de corrélation sous forme de heatmap
f, ax = plt.subplots(figsize=(10, 10))
cmap = sns.diverging_palette(250, 0, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=1, vmin=-1, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5}, annot=True)
plt.show()
```



Il apparait que la température et l'humidité ont un coefficient de corrélation non négligeable, mais de signes opposés. La vitesse du vent, en revance, est très faiblement corrélée avec le nombre total de locations.

Cela ne reflète pas, en tout cas pour l'humidité et la vitesse du vent, les observations faites dans la section précédente, dans laquelle nous voyions un plus fort impact de la vitesse du vent que de l'humidité. Toutefois, il ne s'agit là que de corrélation linéaire.

Dans la prochaine section, nous préparons le jeu de données.

0.9 Préparation du Jeu de Données

Nous préparons maintenant le jeu de données pour le rendre exploitable pour l'entraînement d'algorithmes de prédiction.

Nous commençons par supprimer les colonnes suivantes: - `datetime`: car l'information temporelle est portée par d'autres variables créées à partir de celle-ci telles que `weekday` ou `hour`. - `year`: car l'évolution observée de la demande d'une année sur l'autre ne sera pas forcément répétable et ne paraît donc pas être une variable prédictive pertinente. - `temp`: car la température ressentie est plus

pertinente car la température théorique. - `casual` et `registered`: car elles servent directement à calculer la variable à prédire `count`.

[20]:
```
dataset.drop(columns=["datetime", "year", "temp", "casual", "registered"],␣
↪inplace=True)
```

0.9.1 Séparation en jeux d'entraînement et de test

Une fois le nombre de variables figé, nous pouvons séparer le jeu de données en un jeu de données d'entraînement et un jeu de données de test. Pour cela, nous décidons de consacrer 20% des observations au jeu de test.

[21]:
```
target = "count"

y = dataset[target]
X = dataset[dataset.columns.difference([target])]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

print(f"Number of train samples: {len(X_train)}")
print(f"Number of test samples: {len(X_test)}")
```

```
Number of train samples: 8708
Number of test samples: 2178
```

0.9.2 Feature Engineering

Une fois le jeu de données séparé, nous pouvons effectuer du feature engineering. Dans notre cas, nous allons transformer les variables temporelles `season`, `weekday` et `hour`. Leur nature cyclique et leur nombre élevé de catégories fait que l'encodage one-hot n'est pas le plus approprié. Au lieu d'encoder ces colonnes, nous allons créer à partir de chacune deux nouvelles colonnes qui seront le sinus et le cosinus de leurs valeurs.

Nous créons une fonction custom qui permet de créer ces nouvelles colonnes.

[22]:
```
def transformation(column):
    max_value = column.max()
    sin_values = [math.sin((2*math.pi*x)/max_value) for x in list(column)]
    cos_values = [math.cos((2*math.pi*x)/max_value) for x in list(column)]
    return sin_values, cos_values
```

Et nous l'appliquons aux variables temporelles cycliques, sur le jeu d'entraînement comme sur le jeu de test.

[23]:
```
cyclic_features = [
    "season",
    "weekday",
    "hour"
]
```

```
for feature in cyclic_features:
    X_train[f"sin_{feature}"], X_train[f"cos_{feature}"] =␣
 ↪transformation(X_train[feature])
    X_test[f"sin_{feature}"], X_test[f"cos_{feature}"] =␣
 ↪transformation(X_test[feature])
```

Enfin, nous supprimons les colonnes originelles.

[24]:
```
X_train.drop(cyclic_features, axis=1, inplace=True)
X_test.drop(cyclic_features, axis=1, inplace=True)
```

0.9.3 Création du preprocessor

Nous créons enfin un pré-processeur. Via celui-ci, nous spécifions les pré-traitements à appliquer à chaque variable en fonction de son type avant les étapes d'entraînement du modèle ou de prédiction.

Dans notre cas, nous spécifions de centrer et réduire toutes les variables numériques et d'opérer un encodage one-hot des variables catégorielles.

[25]:
```
# Opérations de transformation pour les variables numériques
numeric_transformer = StandardScaler()
# Opérations de transformation pour les variables catégorielles
categorical_transformer = OneHotEncoder(handle_unknown="ignore")

# Variables numériques
numeric_features = X_train.select_dtypes([np.number]).columns
# Variables catégorielles
categorical_features = X_train.select_dtypes(["category"]).columns

# Instantiation du pré-processeur
preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, numeric_features),
        ("cat", categorical_transformer, categorical_features)
    ]
)
```

0.10 Entraînement et Évaluation de Modèles

Nous sommes enfin prêt pour attaquer la phase de modélisation. Dans cette dernière section, nous entraînons plusieurs modèles de plus en plus complexes et comparons leurs performances en prédiction sur le jeu de test. Pour chaque modèle, nous chercherons à optimiser ses hyperparamètres pour obtenir la meilleure performance.

0.10.1 Métriques d'Évaluation

Nous choisissons de comparer les performances de nos modèles à l'aide de trois métriques: - RMSE - MAE - r^2

La fonction ci-dessous nous permettra de calculer et d'afficher ces scores sur le jeu de test pour chaque modèle.

[26]:
```
def metrics(targets, predictions):
    rmse = round(mean_squared_error(targets, predictions, squared=False), 1)
    mae = round(mean_absolute_error(targets, predictions), 1)
    r2 = round(r2_score(targets, predictions), 2)
    print(f"RMSE: {rmse}")
    print(f"MAE: {mae}")
    print(f"R2: {r2}")
    return {"rmse":rmse, "mae":mae, "r2":r2}
```

0.10.2 Régression Linéaire

Le premier modèle que nous entraînons est une simple régression linéaire multiple. Nous créons d'abord le pipeline qui combine le pré-processeur, suivi du modèle.

[27]:
```
lr_pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("model", LinearRegression()),
])
```

Puis nous entraînons le modèle sur le jeu d'entraînement.

[28]:
```
# Entraînement du modèle
lr_pipeline.fit(X_train, y_train)

# Prédiction sur le jeu d'entraînement
lr_train_predictions = lr_pipeline.predict(X_train)

# Calcul et affichage des métriques de performance sur le jeu d'entraînement
_ = metrics(y_train, lr_train_predictions)
```

```
RMSE: 137.1
MAE: 97.2
R2: 0.43
```

Le score r^2 obtenu sur le jeu d'entraînement est très moyen. Le modèle linéaire semble ne pas être assez complexe pour modéliser les relations entre les variables prédictives et la variable à prédire.

Nous évaluons le modèle sur le jeu de test.

[29]:
```
# Prédiction sur le jeu de test
lr_test_predictions = lr_pipeline.predict(X_test)

# Calcul et affichage des métriques de performance sur le jeu de test
lr_metrics = metrics(y_test, lr_test_predictions)
```

```
RMSE: 136.3
MAE: 95.8
R2: 0.44
```

Comme attendu, ce modèle ne performe pas bien en prédiction sur le jeu de test. Il est donc nécessaire de tester un modèle plus complexe qui arrive à capter les relations non-linéaires entre les variables prédictives et la variable à prédire.

0.10.3 Decision Tree

Nous essayons maintenant un arbre de décision en régression.

[30]:
```
dt_pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("model", DecisionTreeRegressor()),
])
```

Ce modèle possède une multitude d'hyper-paramètres dont l'optimisation peut permettre une amélioration de ses performances. Nous choisissons d'implémenter la méthode `GridSearchCV` qui permet de tester un espace d'hyperparamètres en validation croisée.

L'utilisation de `GridSearchCV` est un processus itératif. Étant limité en termes de puissance de calcul, nous ne pouvons pas nous permettre de tester chaque hyperparamètre finement et sur des plages trop grandes de valeurs. Nous ré-entraînons donc le modèle plusieurs fois en modifiant les bornes de nos listes d'hyperparamètres numériques jusqu'à arriver aux meilleurs valeurs. La liste d'hyperparamètres ci-dessous est la résultante de cette phase de recherche.

[31]:
```
dt_params_grid = {
    "model__criterion": ["squared_error", "friedman_mse", "absolute_error",␣
 ↪"poisson"],
    "model__splitter": ["best", "random"],
    "model__max_depth": [5, 10, 15, 20],
    "model__min_samples_split": [15, 20, 25, 30, 35, 40],
    "model__min_samples_leaf": [1, 2, 5, 10],
    "model__max_features": ["auto", "sqrt", "log2"]
}

pp.pprint(dt_params_grid)

total_conf = 1
for _, value in dt_params_grid.items():
    if isinstance(value, list):
        total_conf *= len(value)

print("\n")
print(f"Nombre de configurations à tester : {total_conf}")
```

```
{   'model__criterion': [   'squared_error',
                            'friedman_mse',
                            'absolute_error',
                            'poisson'],
    'model__max_depth': [5, 10, 15, 20],
    'model__max_features': ['auto', 'sqrt', 'log2'],
```

```
    'model__min_samples_leaf': [1, 2, 5, 10],
    'model__min_samples_split': [15, 20, 25, 30, 35, 40],
    'model__splitter': ['best', 'random']}
```

```
Nombre de configurations à tester : 2304
```

Nous testons chacune de ces configurations en validation croisée.

[32]:
```
dt_grid_search = GridSearchCV(
                                    estimator=dt_pipeline,
                                    param_grid=dt_params_grid,
                                    cv=5,
                                    scoring="r2",
                                    verbose=1,
                                    n_jobs=6
                                )

dt_grid_search.fit(X_train, y_train)
print(f"Best score: {round(dt_grid_search.best_score_, 2)}")
pp.pprint(dt_grid_search.best_params_)
```

```
Fitting 5 folds for each of 2304 candidates, totalling 11520 fits
Best score: 0.8
{   'model__criterion': 'squared_error',
    'model__max_depth': 20,
    'model__max_features': 'auto',
    'model__min_samples_leaf': 5,
    'model__min_samples_split': 30,
    'model__splitter': 'random'}
```

Le score affiché ci-dessus est le score moyen des 5 folds obtenu avec la meilleur configuration d'hyperparamètres dans l'espace défini.

Vérifions les performances de ce modèle en prédiction sur le jeu de test avec cette configuration.

[33]:
```
dt_predictions = dt_grid_search.predict(X_test)
dt_metrics = metrics(y_test, dt_predictions)
```

```
RMSE: 84.2
MAE: 55.4
R2: 0.79
```

Le score est significativement meilleur que celui obtenu avec la régression linéaire.

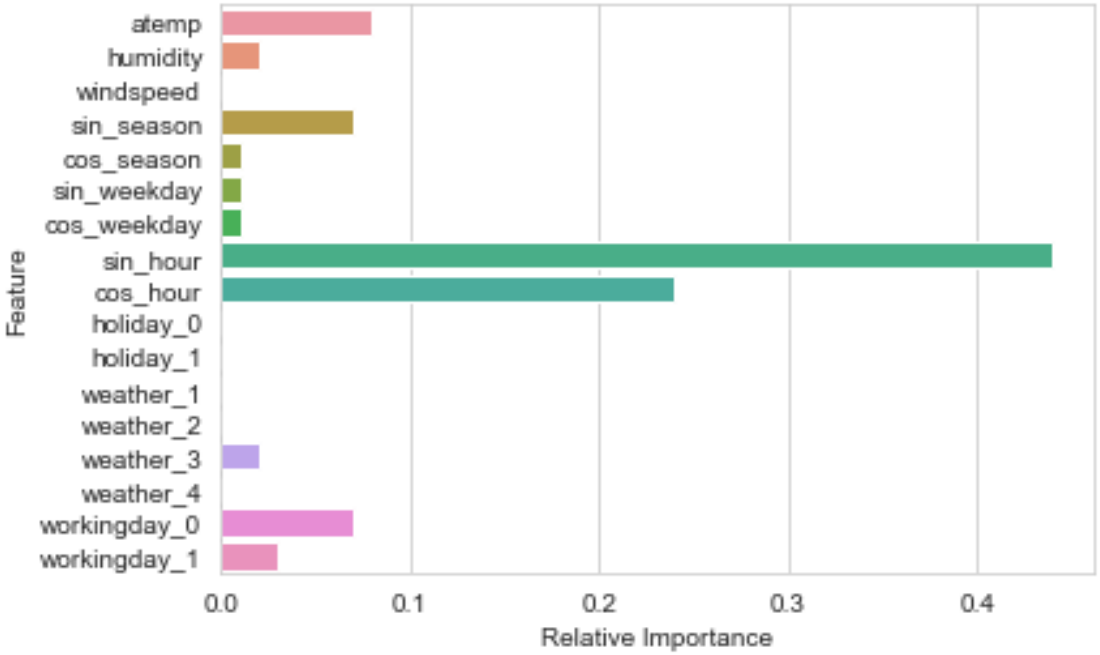
Il nous est enfin également possible d'afficher l'importance relative de chaque feature dans la décision du modèle.

[40]:
```
raw_features_name = dt_pipeline.named_steps["preprocessor"].
 ↪get_feature_names_out()
```

```
formatted_features_name = [raw_feature_name.split("__")[-1] for␣
 ↪raw_feature_name in raw_features_name]
dt_features_importance = np.around(dt_grid_search.best_estimator_.
 ↪named_steps["model"].feature_importances_, decimals=2)

dt_fi_df = pd.DataFrame(data={"name": formatted_features_name, "importance":␣
 ↪dt_features_importance})

sns.color_palette("pastel")
sns.barplot(x="importance", y="name", data=dt_fi_df, orient="h")
plt.xlabel("Relative Importance")
plt.ylabel("Feature")
plt.show()
```



Nous pouvons voir que les features liées à l'heure du jour sont très au-dessus du lot. La température, la saison ainsi que le statut du jour (travaillé ou non) sont également des features importantes pour le modèle.

0.10.4 Random Forest

Passer d'un simple modèle linéaire à un arbre de décision nous a pratiquement permis de doubler notre score r2. Nous décidons désormais d'essayer une méthode ensembliste et d'entraîner un Random Forest, modèle plus complexe, plus performant et qui permet en outre de mieux contrôler l'over-fitting.

Comme précédemment, nous commençons par créer le pipeline.

[35]:
```
rf_pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("model", RandomForestRegressor()),
])
```

Nous déterminons ensuite un espace d'hyper-paramètres.

[36]:
```
rf_params_grid = {
    "model__n_estimators": [100, 300, 500],
    "model__max_depth": [5, 10, 15],
    "model__min_samples_split": [5, 10, 15, 20],
    "model__min_samples_leaf": [5, 10, 15, 20],
    "model__max_features": ["auto", "sqrt"]
}

pp.pprint(rf_params_grid)

total_conf = 1
for _, value in rf_params_grid.items():
    if isinstance(value, list):
        total_conf *= len(value)

print("\n")
print(f"Total number of configurations: {total_conf}")
```

```
{   'model__max_depth': [5, 10, 15],
    'model__max_features': ['auto', 'sqrt'],
    'model__min_samples_leaf': [5, 10, 15, 20],
    'model__min_samples_split': [5, 10, 15, 20],
    'model__n_estimators': [100, 300, 500]}


Total number of configurations: 288
```

Puis nous entraînons le modèle en validation croisée pour un certain nombre de configurations d'hyper-paramètres.

[37]:
```
rf_grid_search = GridSearchCV(
                                    estimator=rf_pipeline,
                                    param_grid=rf_params_grid,
                                    scoring="r2",
                                    cv=5,
                                    verbose=1,
                                    n_jobs=6
                                )

rf_grid_search.fit(X_train, y_train)
print(f"Best score: {round(rf_grid_search.best_score_, 2)}")
pp.pprint(rf_grid_search.best_params_)
```

```
Fitting 5 folds for each of 288 candidates, totalling 1440 fits
Best score: 0.84
{   'model__max_depth': 15,
    'model__max_features': 'auto',
    'model__min_samples_leaf': 5,
    'model__min_samples_split': 10,
    'model__n_estimators': 500}
```

Nous observons un gain supplémentaire de performance par l'utilisation de ce modèle ensembliste. Vérifions ce score en prédiction sur le jeu de test.

[38]:
```
rf_predictions = rf_grid_search.predict(X_test)
rf_metrics = metrics(y_test, rf_predictions)
```

```
RMSE: 72.2
MAE: 48.8
R2: 0.84
```

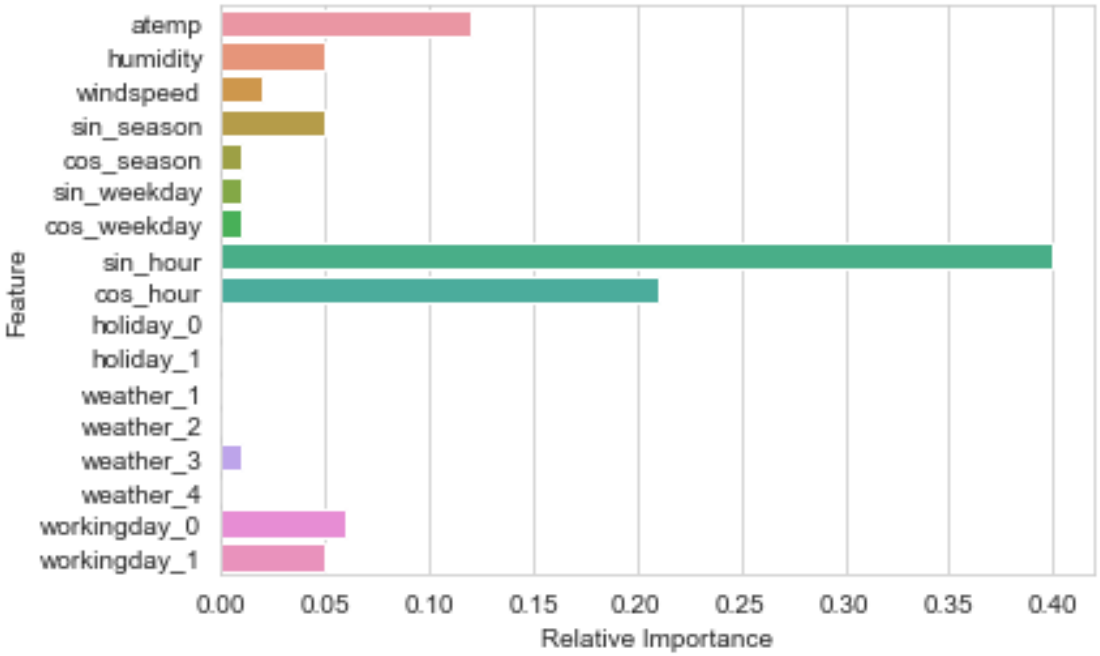
Ce modèle est également plus performant en prédiction sur le jeu de test.

Comme nous l'avons fait précédemment pour l'arbre de décision, affichons l'importance relative des features pour le modèle.

[41]:
```
raw_features_name = rf_pipeline.named_steps["preprocessor"].
 ↪get_feature_names_out()
formatted_features_name = [raw_feature_name.split("__")[-1] for␣
 ↪raw_feature_name in raw_features_name]
rf_features_importance = np.around(rf_grid_search.best_estimator_.
 ↪named_steps["model"].feature_importances_, decimals=2)

rf_fi_df = pd.DataFrame(data={"name": formatted_features_name, "importance":␣
 ↪rf_features_importance})

sns.color_palette("pastel")
sns.barplot(x="importance", y="name", data=rf_fi_df, orient="h")
plt.xlabel("Relative Importance")
plt.ylabel("Feature")
plt.show()
```



Pour ce modèle également, les features les plus importantes sont celles liées à l'heure du jour. La feature `atemp` a également du poids dans la prédiction du modèle.

0.10.5 Gradient Boosting

Pour finir, nous choisissons l'algorithme de Gradient Boosting. Celui-ci se base sur un ensemble d'arbres de décision mais sa méthode est différente de celle d'un Random Forest, lui permettant souvent de fournir un gain de performance.

Nous créons un nouveau pipeline.

[42]:
```
gb_pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("model", GradientBoostingRegressor()),
])
```

Nous définissons un espace d'hyper-paramètres spécifiques à ce modèle.

[44]:
```
gb_params_grid = {
    "model__n_estimators": [100, 300, 500],
    "model__learning_rate": [0.01, 0.05, 0.1],
    "model__min_samples_split": [10, 15, 20],
    "model__min_samples_leaf": [10, 15, 20],
    "model__max_depth": [5, 10, 15],
    "model__max_features": ["auto", "sqrt", "log2"]
}
```

```
pp.pprint(gb_params_grid)

total_conf = 1
for _, value in gb_params_grid.items():
    if isinstance(value, list):
        total_conf *= len(value)

print("\n")
print(f"Total number of configurations: {total_conf}")
```

```
{   'model__learning_rate': [0.01, 0.05, 0.1],
    'model__max_depth': [5, 10, 15],
    'model__max_features': ['auto', 'sqrt', 'log2'],
    'model__min_samples_leaf': [10, 15, 20],
    'model__min_samples_split': [10, 15, 20],
    'model__n_estimators': [100, 300, 500]}


Total number of configurations: 729
```

Nous entraînons le modèle en validation croisée sur cet espace d'hyperparamètres.

[45]:
```
gb_grid_search = GridSearchCV(
                                    estimator=gb_pipeline,
                                    param_grid=gb_params_grid,
                                    scoring="r2",
                                    cv=5,
                                    verbose=1,
                                    n_jobs=6
                                )

gb_grid_search.fit(X_train, y_train)
print(f"Best score: {round(gb_grid_search.best_score_, 2)}")
pp.pprint(gb_grid_search.best_params_)
```

```
Fitting 5 folds for each of 729 candidates, totalling 3645 fits
Best score: 0.87
{   'model__learning_rate': 0.05,
    'model__max_depth': 10,
    'model__max_features': 'sqrt',
    'model__min_samples_leaf': 20,
    'model__min_samples_split': 15,
    'model__n_estimators': 500}
```

Le meilleur score obtenu en validation croisée est supérieur à celui obtenu avec un RandomForest. Nous évaluons ce nouveau modèle avec la meilleur configuration d'hyper-paramètres en prédiction sur le jeu de test.

[47]:
```
gb_predictions = gb_grid_search.predict(X_test)
gb_metrics = metrics(y_test, gb_predictions)
```

```
RMSE: 64.0
MAE: 43.4
R2: 0.88
```

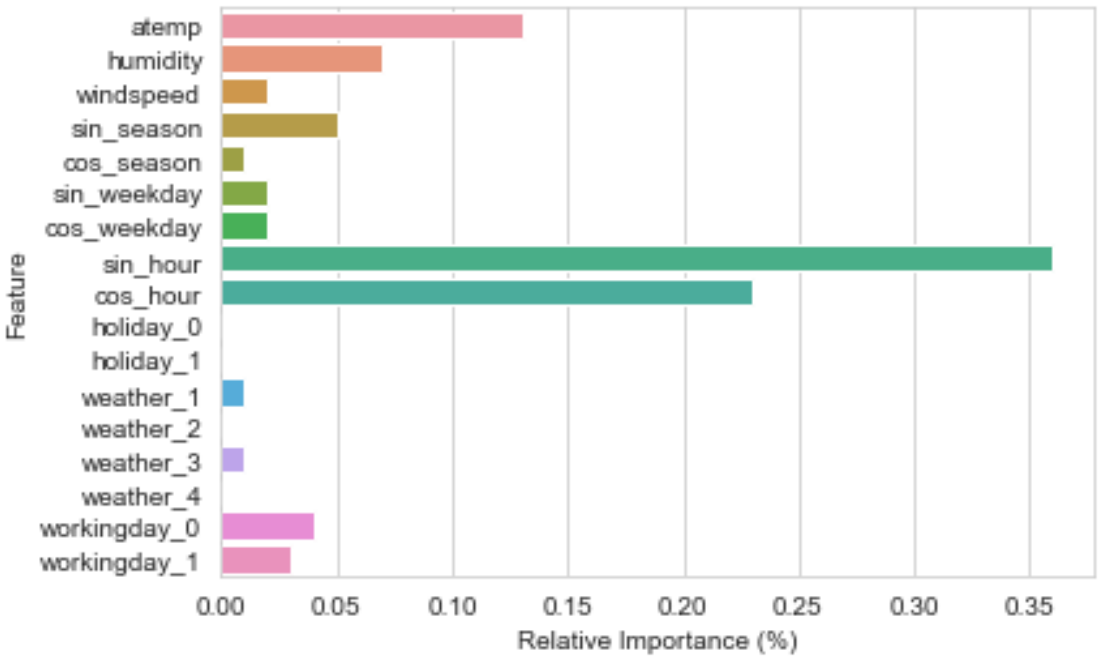
Le score obtenu en prédiction sur le jeu de test est également légèrement supérieur à celui obtenu avec le modèle RandomForest.

Nous affichons l'importance relative des features.

[49]:
```
raw_features_name = gb_pipeline.named_steps["preprocessor"].
 ↪get_feature_names_out()
formatted_features_name = [raw_feature_name.split("__")[-1] for␣
 ↪raw_feature_name in raw_features_name]
gb_features_importance = np.around(gb_grid_search.best_estimator_.
 ↪named_steps["model"].feature_importances_, decimals=2)

gb_fi_df = pd.DataFrame(data={"name": formatted_features_name, "importance":␣
 ↪gb_features_importance})

sns.color_palette("pastel")
sns.barplot(x="importance", y="name", data=gb_fi_df, orient="h")
plt.xlabel("Relative Importance (%)")
plt.ylabel("Feature")
plt.show()
```



Les relations d'importance de chaque feature sont les mêmes que précédemment.

0.10.6 Comparaison des Modèles

Afin de pouvoir mieux comparer les performances de chaque modèle, nous affichons dans un même dataframe les metrics calculées pour chaque modèle sur le jeu de test.

[50]:
```
models = [
    "Linear Regression",
    "Decision Tree",
    "Random Forest",
    "Gradient Boosting"
]

d = [
    lr_metrics,
    dt_metrics,
    rf_metrics,
    gb_metrics
]

pd.DataFrame(data=d, index=models)
```

[50]:

	rmse	mae	r2
Linear Regression	136.3	95.8	0.44
Decision Tree	84.2	55.4	0.79
Random Forest	72.2	48.8	0.84
Gradient Boosting	64.0	43.4	0.88

Comme nous avons pu le constater, le modèle le plus performant est le `GradientBoostingRegressor`. Celui-ci se base sur un ensemble d'arbres de décision comme peux le faire un Random Forest, mais ceux-ci sont moins profonds et la façon dont ils sont construits et leurs résultats aggrégés est différente. Cela lui permet souvent de mieux performer qu'un `RandomForest`, comme nous pouvons le voir ici.

0.11 Conclusion

Lors de ce projet, nous avons testé 4 modèles de complexité croissante. Pour chaque modèle (à part la régression linéaire), nous avons implémenté une méthode optimisation des hyper-paramètres en validation croisée afin d'optimiser leurs performances.

Le modèle le plus performant sur ce jeu de données est le `GradientBoostingRegressor`.

En analysant l'importance relative des features dans les prédictions de ces modèles, nous avons pu observer que les variables `hour`, `atemp`, `season` et `workingday` sortent du lot. Ces variables ont donc un poids plus fort dans la prédiction. Cela confirme nos analyses préliminaires, lors desquelles nous avions vu que le nombre total de locations est très dépendant de l'heure de la journée, des conditions météorologiques et du statut du jour (travaillé ou non).

Quelques axes d'amélioration pour la suite: - entraînement d'un modèle `XGBoost`, qui est une version plus régularisée du `GradientBoosting`, et pourrait ainsi apporter un gain de performance en prédiction. - affiner la recherche d'hyperparamètres. - tester d'autres méthodes de feature engineering.