progetto_programmazione_data_intensive

June 6, 2019

1 Determinare la presenza o meno di persone all'interno di un ufficio

Programmazione di Applicazioni Data Intensive
Laurea in Ingegneria e Scienze Informatiche
DISI - Università di Bologna, Cesena
Alessandro Lombardi
Citazioni

- Accurate occupancy detection of an office room from light, temperature, humidity and CO2 measurements using statistical learning models. L. M. Candanedo, V. Feldheim. Energy and Buildings. Volume 112, 15 January 2016, Pages 28-39.

1.1 Parte 1 - Descrizione del problema e analisi esplorativa

Si deve realizzare un modello che utilizzando i dati registrati da alcuni sensori posti all'interno di un ufficio sia in grado di determinare la presenza o meno di persone al suo interno.

Vengono importate le librerie necessarie per scaricare i file, organizzare le strutture dati e disegnare i grafici.

[1]:
```
%matplotlib inline
import os.path
import numpy as np
import pandas as pd
from pandas.compat import StringIO
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import seaborn as sb
```

1.1.1 Caricamento dei dati e preprocessing

[2]:
```
file_zip_url = "http://archive.ics.uci.edu/ml/machine-learning-databases/00357/
 ↪occupancy_data.zip"
file_zip_name = "occupancy_data.zip"
file_training_set_name = "datatraining.txt"
file_validation_set_name = "datatest.txt"
file_test_set_name = "datatest2.txt"
```

```
if not os.path.exists(file_zip_name):
    from urllib.request import urlretrieve
    urlretrieve(file_zip_url, file_zip_name)
    from zipfile import ZipFile
    with ZipFile(file_zip_name) as f:
        f.extractall()
```

[3]:
```
with open(file_training_set_name) as dataFile:
    data_raw = pd.read_csv(dataFile, sep=",")

data_raw.head()
```

[3]:

	date	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
1	2015-02-04 17:51:00	23.18	27.2720	426.0	721.25	0.004793	1
2	2015-02-04 17:51:59	23.15	27.2675	429.5	714.00	0.004783	1
3	2015-02-04 17:53:00	23.15	27.2450	426.0	713.50	0.004779	1
4	2015-02-04 17:54:00	23.15	27.2000	426.0	708.25	0.004772	1
5	2015-02-04 17:55:00	23.10	27.2000	426.0	704.50	0.004757	1

Di seguito sono riportate le dimensioni in memoria, il numero di istanze non nulle e il tipo delle feature che compongono i dati raccolti nel dataset (training set)

[4]:
```
data_raw.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8143 entries, 1 to 8143
Data columns (total 7 columns):
date             8143 non-null object
Temperature      8143 non-null float64
Humidity         8143 non-null float64
Light            8143 non-null float64
CO2              8143 non-null float64
HumidityRatio    8143 non-null float64
Occupancy        8143 non-null int64
dtypes: float64(5), int64(1), object(1)
memory usage: 1.0 MB
```

Sono rinominate le feature `date` in `Date` e `HumidityRatio` in `Humidity_Ratio`

[5]:
```
data = data_raw.rename(columns={"date" : "Date",
                                "HumidityRatio": "Humidity_Ratio"})
```

1.1.2 Significato delle feature

Riportiamo le informazioni sul dataset tratte dalla fonte
Dati di tipo categorico - `Date` timestamp della misurazione

- `Temperature` misura della temperatura misurata in gradi centigradi (Cř)
- `Humidity` misura dell'umidità relativa in percentuale
- `Light` misura della luminosità in Lux (lx)
- `CO2` misura di CO2 nell'aria in parti per milione (ppm)
- `Humidity_Ratio` misura derivata dalla umidità e dalla temperatura in Kg di vapore acqueo in Kg di aria
- `Occupancy` indica la presenza di persone nell'ufficio
  - 1 = presenti (stato occupato) 0 = assenti (stato non occupato)

Con il metodo *describe* è possibile avere una rappresentazione statistica delle feature numeriche, ottenendo media, deviazione standard e la distribuzione in termini di massimi, minimi e percentili.

La temperatura è abbastanza stabile intorno ai 20 gradi centigradi, il che ci fa supporre che le misure siano state fatte in un luogo protetto, dove non ci sono troppi sbalzi di temperatura e/o le misurazioni siano limitate ad una certa parte dell'anno. I valori della umidità sono invece più variegati, se si mantengono le medesime considerazioni sulla temaperatura, si può attribuire tale fenomeno ad una maggiore sensibilità alla presenza o meno di persone nell'ufficio. Come per la temperatura non è chiaro se la luce sia influenzata da fattori ambientali, come l'alternarsi fra giorno e notte, le ore di luce che variano durante l'anno oppure artificiali come l'uso di luci elettriche. Il valore minimo in Lux corrisponde al completo buio, il massimo ad una giornata abbastanza luminosa, la distribuzione è abbastanza binaria, facendo ipotizzare misurazioni frequenti anche durante la notte con valori costantemente pari a zero. I valori di anidride carbonica in parti per milione potrebbero essere influenzati dall'inquinamento atmosferico o dalla vegetazione nei pressi dell'ufficio.

1.1.3 Esplorazione singole feature

[6]:
```
num_features = data.columns.size
data.describe()
```

[6]:

	Temperature	Humidity	Light	CO2	Humidity_Ratio	\
count	8143.000000	8143.000000	8143.000000	8143.000000	8143.000000	
mean	20.619084	25.731507	119.519375	606.546243	0.003863	
std	1.016916	5.531211	194.755805	314.320877	0.000852	
min	19.000000	16.745000	0.000000	412.750000	0.002674	
25%	19.700000	20.200000	0.000000	439.000000	0.003078	
50%	20.390000	26.222500	0.000000	453.500000	0.003801	
75%	21.390000	30.533333	256.375000	638.833333	0.004352	
max	23.180000	39.117500	1546.333333	2028.500000	0.006476	

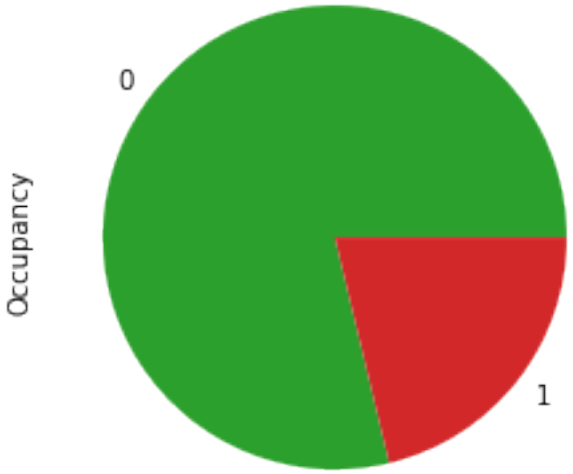
```
          Occupancy
count  8143.000000
mean      0.212330
std       0.408982
min       0.000000
25%       0.000000
50%       0.000000
75%       0.000000
max       1.000000
```

Il dataset contiene circa tre quarti delle righe labeled come ufficio non occupato. Questo potrebbe essere un problema nella fase di modellazione e valutazione del modello.

Viene creato un dizionario che associa al valore 0 il colore verde e indica lo stato "non occupato" dell'ufficio mentre al valore 1 il colore rosso che indica lo stato "occupato"

[7]:
```
free_color = "#2ca02c" # green
occupied_color = "#d32829" # red

class_colors = [free_color, occupied_color]
class_colors_map = {0:free_color, 1:occupied_color}

print(data["Occupancy"].value_counts())
_ = data["Occupancy"].value_counts().plot.pie(colors=class_colors)
```

```
0    6414
1    1729
Name: Occupancy, dtype: int64
```



La feature `Date` contiene il timestamp della misurazione, per questioni di ottimizzazione e preprocessing la data viene convertita in *numpy.datetime*. La distribuzione è uniforme e continua ma piuttosto limitata in un periodo relativamente breve.

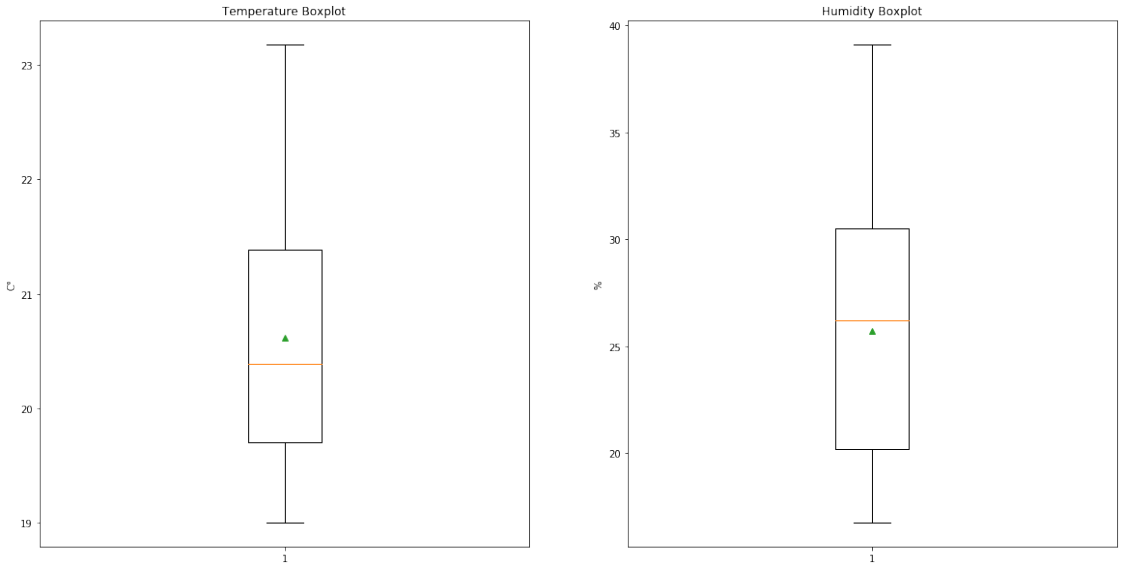
[8]:
```
def cast_date_time_and_show(dataset):
    dataset["Date"] = pd.to_datetime(dataset["Date"], format="%Y-%m-%d %H:%M:
 ↪%S")

    print(dataset["Date"].describe())
    #plt.plot(dataset["Date"])
    return dataset


data = cast_date_time_and_show(data)
```

```
count                    8143
unique                   8143
top       2015-02-07 17:43:00
freq                        1
first     2015-02-04 17:51:00
last      2015-02-10 09:33:00
Name: Date, dtype: object
```

Rappresentazione tramite boxplot dI `Temperature` e `Humidity`

[9]:
```
fig, axes = plt.subplots(nrows=1, ncols=2, sharex=False, sharey=False,␣
 ↪figsize=(20, 10))


axes[0].boxplot(data["Temperature"], showmeans=True)
axes[1].boxplot(data["Humidity"], showmeans=True)

axes[0].set_title("Temperature Boxplot")
axes[1].set_title("Humidity Boxplot")
axes[0].set_ylabel("Cř")
axes[1].set_ylabel("%")
```

[9]:
```
Text(0, 0.5, '%')
```

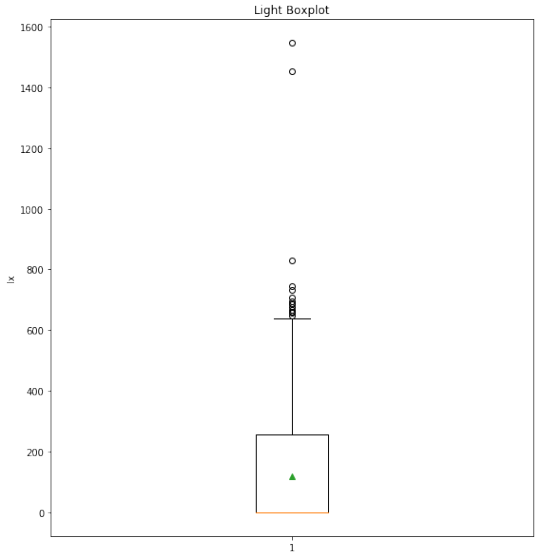


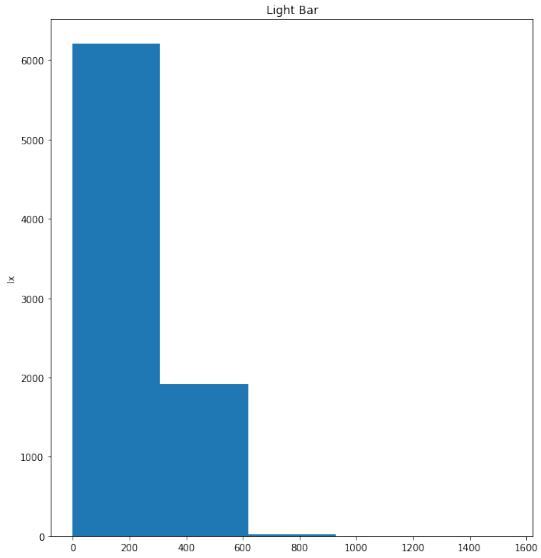
Distribuzioni delle altre feature con boxplot e istogrammi.

Nella distribuzione di `Light` vi sono alcuni **outliers** ma sono sempre all'interno di un ragionevole per l'unità di misura Lux. La distribuzione di `CO2` è particolare perchè presenta molti outliers spalmati in modo uniforme.

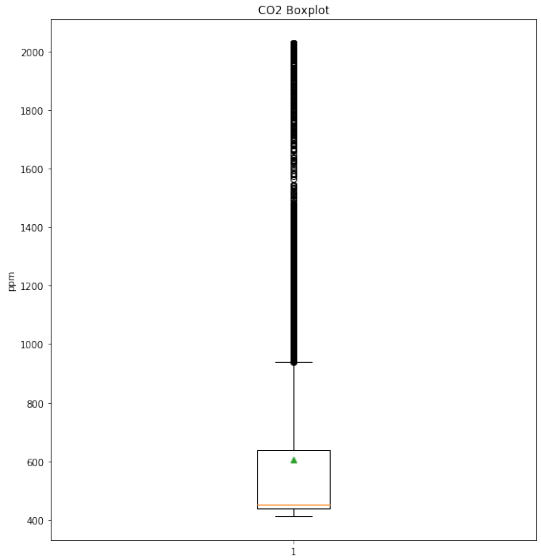
[10]:
```
def plot_box_and_hist(column_name, dataset, symbol=""):
    fig, axes = plt.subplots(nrows=1, ncols=2, sharex=False, sharey=False,␣
 ↪figsize=(20, 10))

    axes[0].boxplot(dataset["{}".format(column_name)], showmeans=True)
    axes[1].hist(dataset["{}".format(column_name)], bins=5)

    axes[0].set_title("{} Boxplot".format(column_name))
    axes[1].set_title("{} Bar".format(column_name))
    axes[0].set_ylabel(symbol)
    axes[1].set_ylabel(symbol)
```

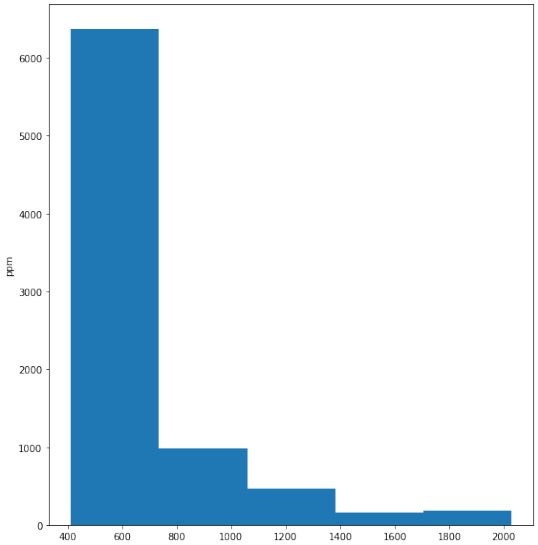
[11]:
```
plot_box_and_hist("Light", data, "lx")
```



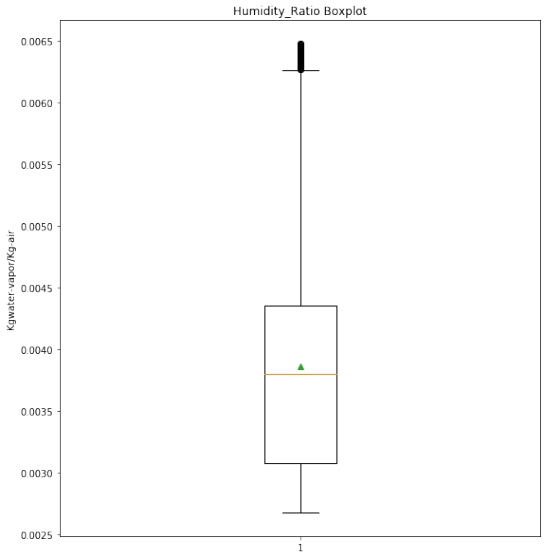


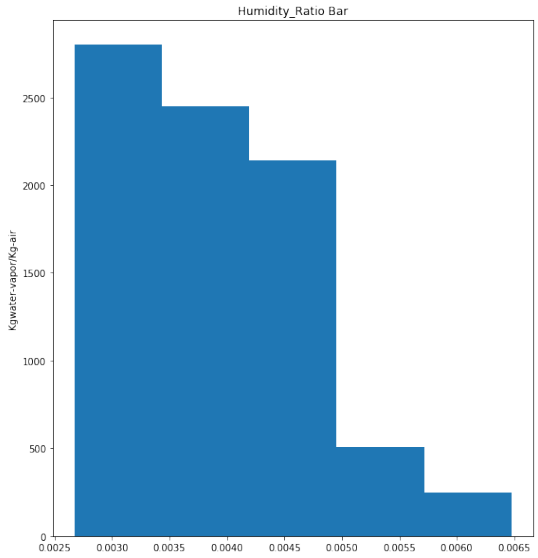
[12]:
```
plot_box_and_hist("CO2", data, "ppm")
```





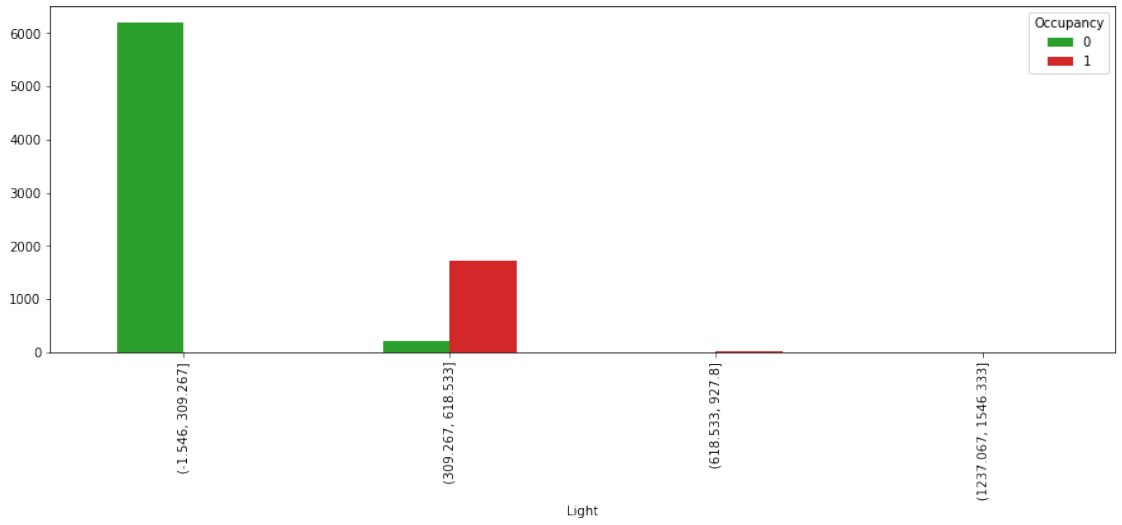
[13]:
```
plot_box_and_hist("Humidity_Ratio", data, "Kgwater-vapor/Kg-air")
```



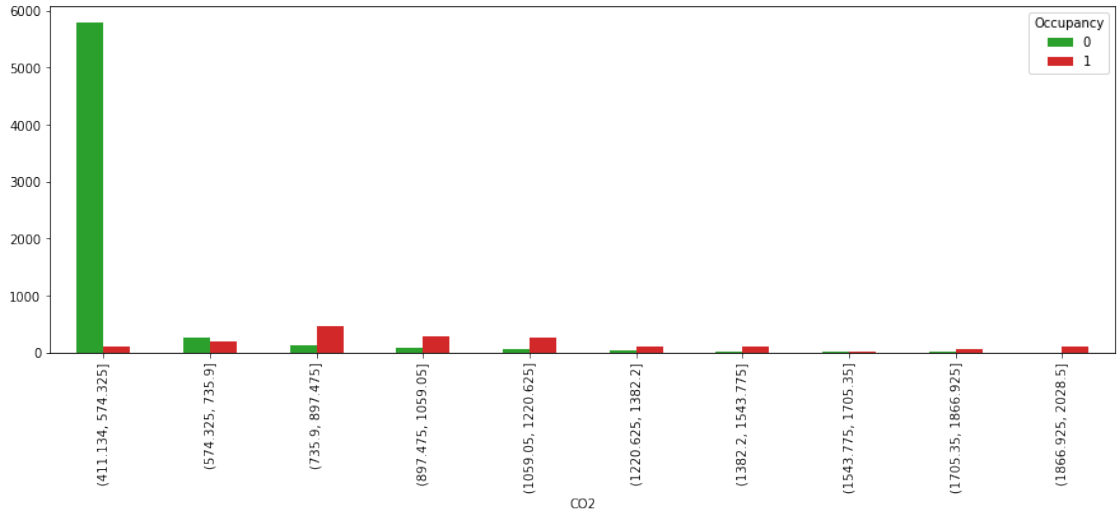


E' evidente che al crescere del valore della feature `Light` il numero dei record labeled come occupati è molto basso, mentre aumentando i livelli di `CO2` aumentano i record segnati come occupati e diminuiscono significativamente quelli che non lo sono. Si può ipotizzare che esista una soglia sulla quale poter binarizzare l'insieme dei valori per classificare i record.

[14]:
```
_ = data.groupby([pd.cut(data["Light"], bins=5), "Occupancy"]).size().
↪unstack("Occupancy").plot.bar(stacked=False, color=class_colors,␣
↪figsize=(15,5))
```



[15]:
```
_ = data.groupby([pd.cut(data["CO2"], bins=10), "Occupancy"]).size().
↪unstack("Occupancy").plot.bar(stacked=False, color=class_colors,␣
↪figsize=(15,5))
```

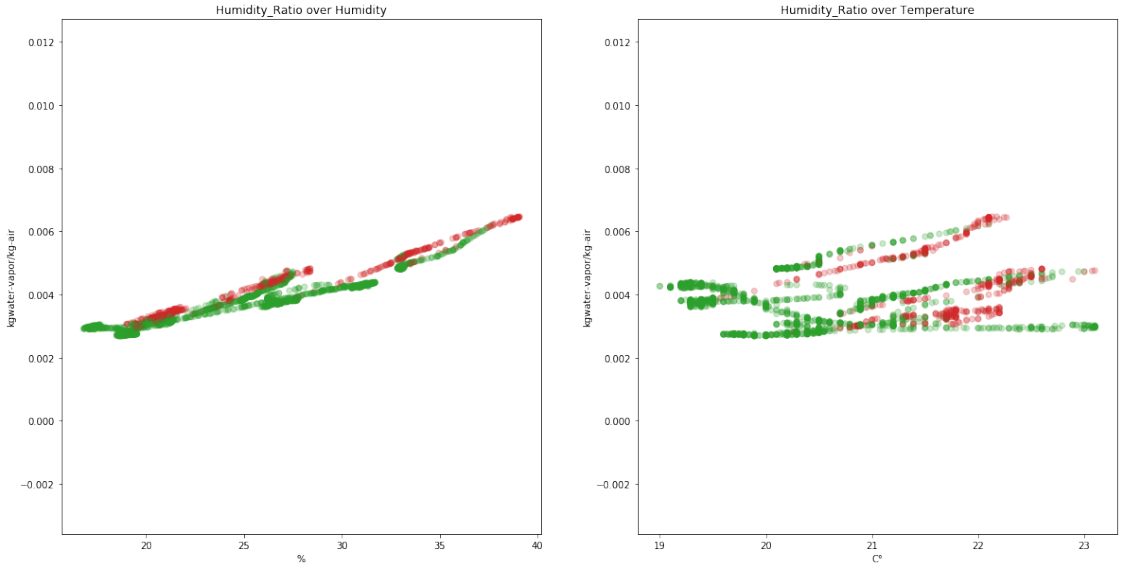


1.1.4 Esplorazione relazioni fra feature

Si vuole dimostrare che cresce al crescere di umidità e temperatura cresce anche il valore contenuto in `Humidity_Ratio` che rappresenta la massa di acqua sulla massa d'aria. Di seguito vengono riportati i valori classificati per colore sulla feature `Occupancy` della temperatura, che ricorda la tabella psicometrica, e dell'umidità.

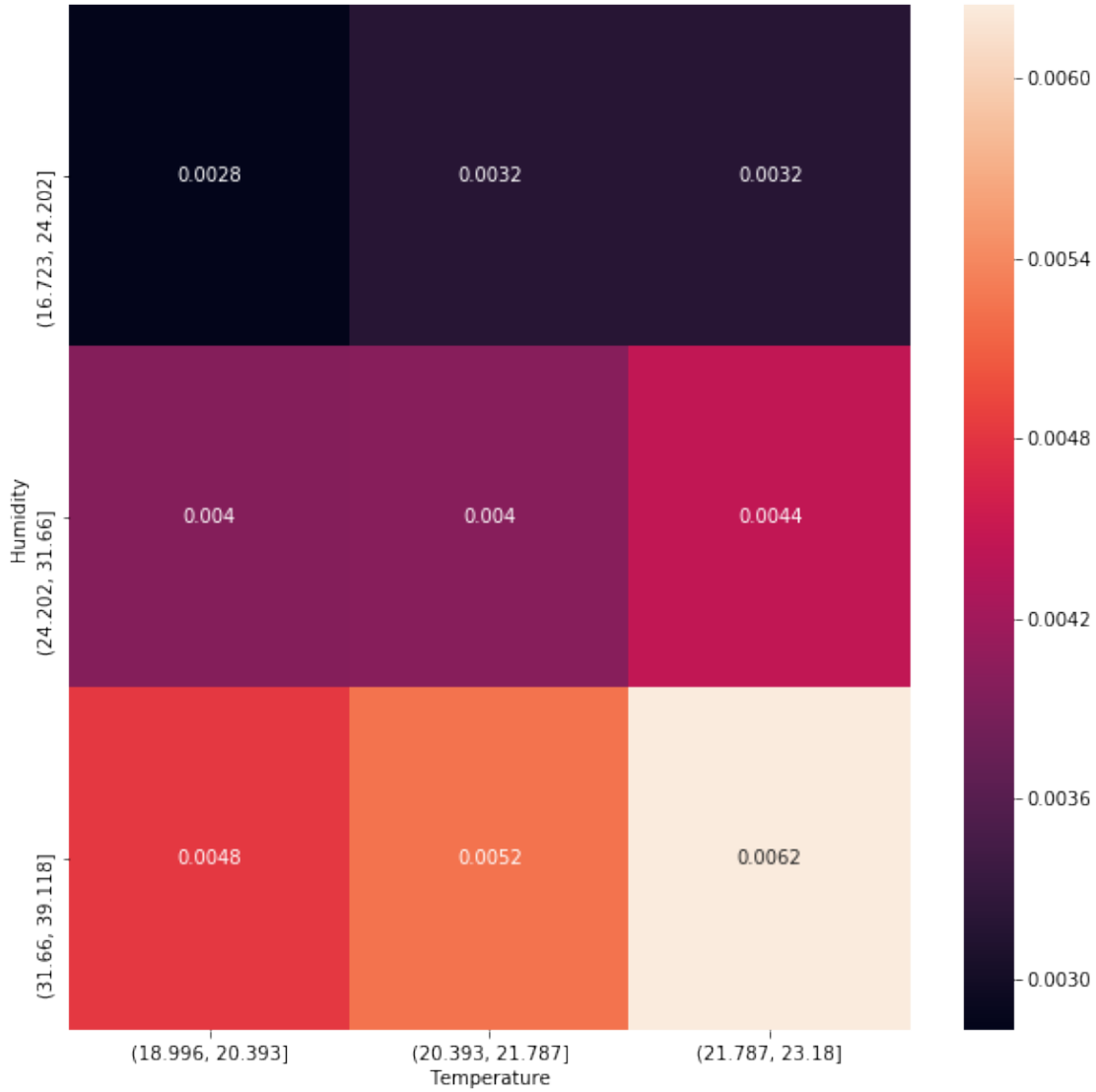
[16]:
```
fig, axes = plt.subplots(nrows=1, ncols=2, sharex=False, sharey=False,␣
 ↪figsize=(20, 10))

sample = data.sample(2500)
axes[0].scatter(sample["Humidity"], sample["Humidity_Ratio"],␣
 ↪c=sample["Occupancy"].map(class_colors_map), alpha=0.25)
axes[1].scatter(sample["Temperature"], sample["Humidity_Ratio"],␣
 ↪c=sample["Occupancy"].map(class_colors_map), alpha=0.25)

axes[0].set_title("Humidity_Ratio over Humidity")
axes[1].set_title("Humidity_Ratio over Temperature")
axes[0].set_xlabel("%")
axes[0].set_ylabel("kgwater-vapor/kg-air")
axes[1].set_xlabel("Cř")
axes[1].set_ylabel("kgwater-vapor/kg-air")
```

[16]:
```
Text(0, 0.5, 'kgwater-vapor/kg-air')
```



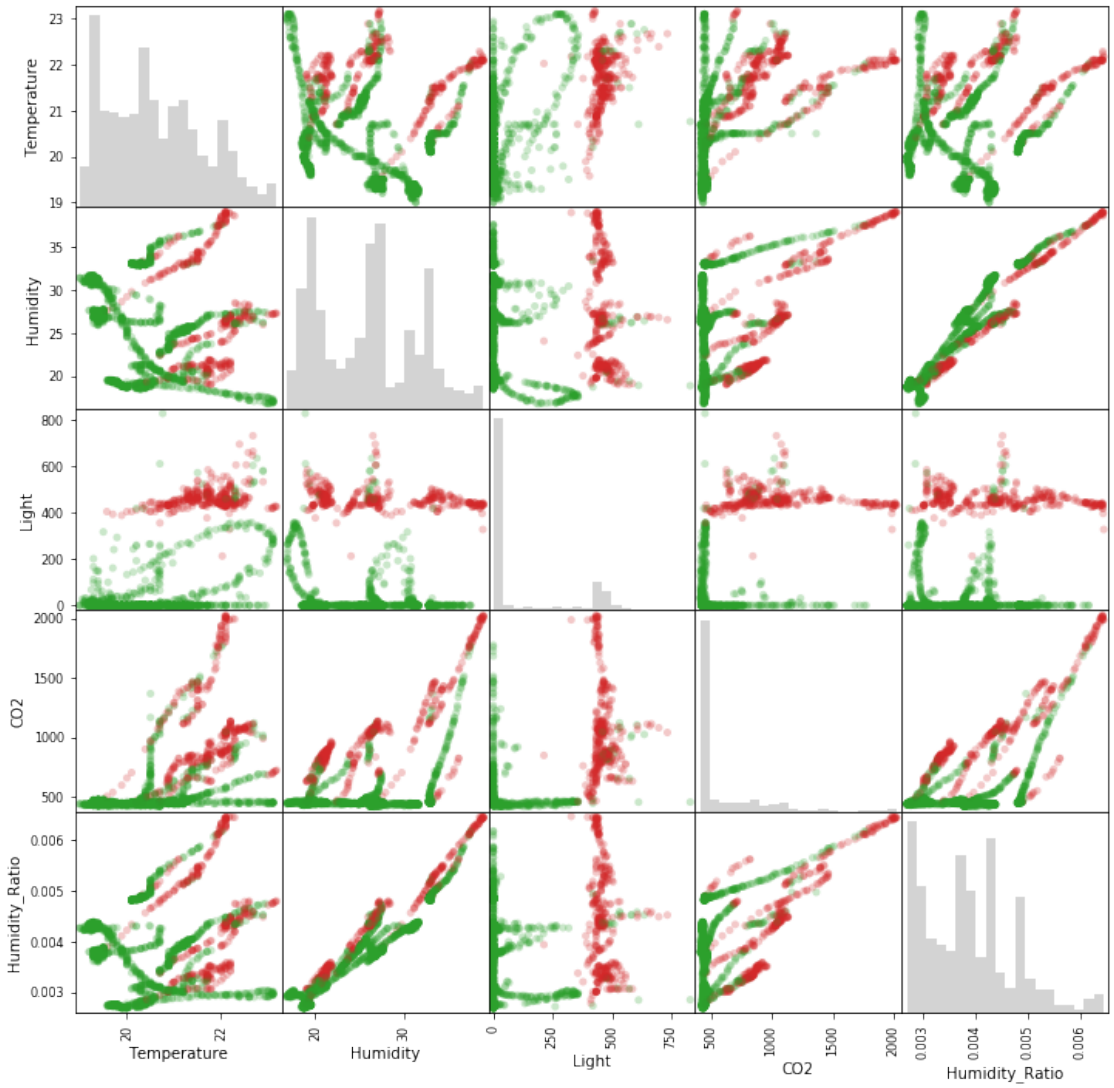
Di seguito viene riprodotta una semplificazione della tabella che lega temperatura e pressione con la massa dell'acqua per Kg di aria come questa riportata in questo sito https://www.engineeringtoolbox.com/water-vapor-air-d_854.html. Si conferma anche qui ciò che è stato ipotizzato precedentemente

[17]:
```
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111)
temperature_interval = pd.cut(data["Temperature"], bins=3)
humidity_interval = pd.cut(data["Humidity"], bins=3)

sb.heatmap(data.groupby([temperature_interval, humidity_interval]).mean().
 ↪unstack("Temperature")["Humidity_Ratio"], annot=True)

plt.show()
```



Di seguito per completezza vengono mostrate su scatter plot tutte le combinazioni fra le feature continue e numeriche del dataset, classificate su `Occupancy` per colore. In molti grafici è possibile confermare la tesi fatta precedentemente sulla binarizzazione, infatti in seguito viene visualizzato nel dettaglio la feature `Light` in relazione con `Temperature` e `CO2` dove è palese l'addensarsi delle classi in zone abbastanza distinte fra loro, in base alla combinazione dei due valori.

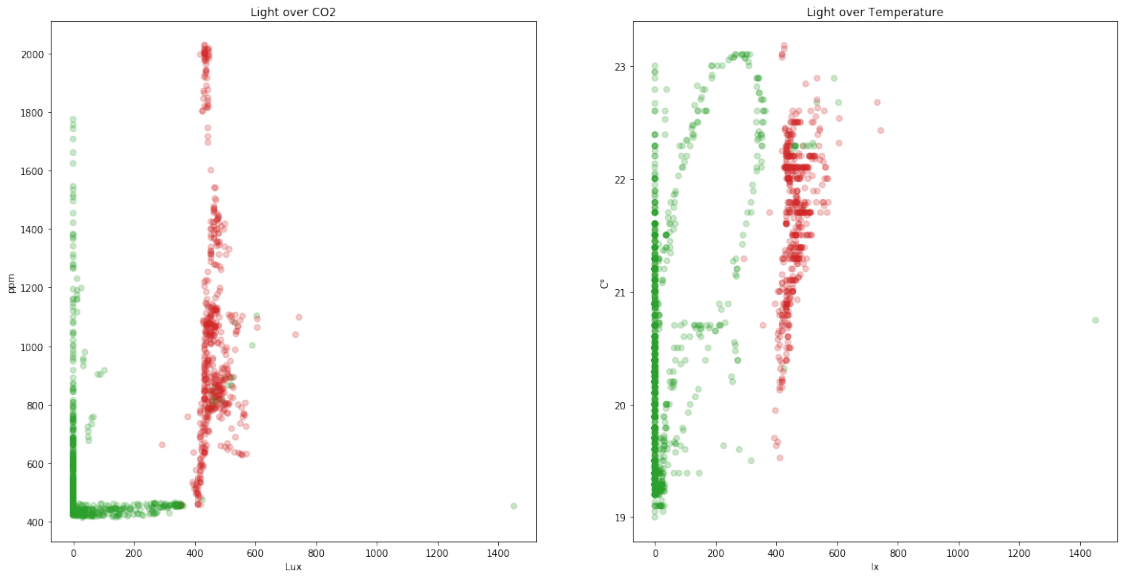
[18]:
```
sample = data.sample(2000)
_ = pd.plotting.scatter_matrix(sample.drop(["Occupancy"], axis=1),
                               c=sample["Occupancy"].map(class_colors_map),
                               alpha=0.25,
                               figsize=(12,12),
                               marker="o",
                               s=25,
                               hist_kwds={"bins": 20, "color": "lightgray"})
```



[19]:
```
fig, axes = plt.subplots(nrows=1, ncols=2, sharex=False, sharey=False,␣
 ↪figsize=(20, 10))

sample = data.sample(2500)
axes[0].scatter(sample["Light"], sample["CO2"], c=sample["Occupancy"].
 ↪map(class_colors_map), alpha=0.25)
axes[1].scatter(sample["Light"], sample["Temperature"], c=sample["Occupancy"].
 ↪map(class_colors_map), alpha=0.25)

axes[0].set_title("Light over CO2")
axes[1].set_title("Light over Temperature")
axes[0].set_xlabel("Lux")
axes[0].set_ylabel("ppm")
axes[1].set_xlabel("lx")
```

```
axes[1].set_ylabel("Cř")
```

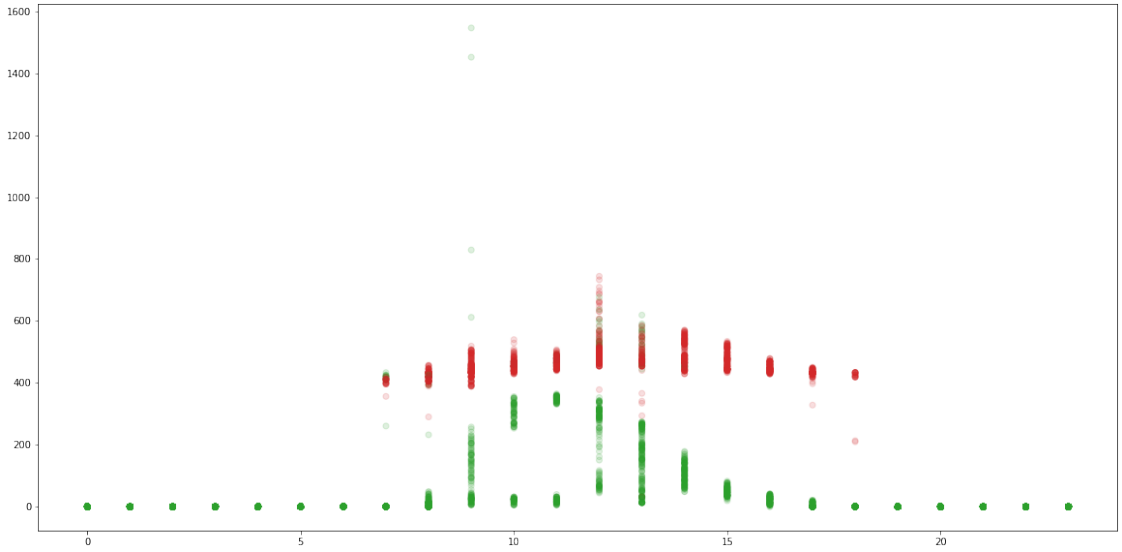
[19]:
```
Text(0, 0.5, 'Cř')
```



In seguito sono mostrate con scatter plot le distribuzioni di alcune misure nel tempo, classificate per colore. Le teorie precedenti riguardanti l'alternarsi fra giorno e notte sembrano essere in parte confermate. Non è scontato che l'orario di lavoro vada dalla mattina alla sera, nemmeno che lo stesso ufficio lo mantenga in futuro, ma nel breve periodo analizzato sembra che questa regola venga rispettata, in quanto non sembrano essere presenti record classificati come "occupato" all'infuori di certe fasce orarie. La feature `Light` distribuita durante le ore di una giornata, svela nuovamente che sotto una certa soglia è difficile che l'ufficio risulti occupato. Inoltre anche durante le ore diurne, quando l'ufficio non è occupato, i valori tendono ad alzarsi, facendo pensare a relazioni con l'ambiente esterno, anche se non è da escludere il semplice allontanamento del personale con l'uso costante della luce elettrica. I grafici per `Temperature` e `Humidity_Ratio` non sembrano avere soglie di binarizzazione molto visibili. La `CO2` invece cresce notevolmente quando classificata come occupata, il picco classificato come non occupato che ha durante le ore 13 potrebbe essere causato dalla pausa pranzo del personale, mentre quello delle ore 18 dall'inquinamento dell'ora di punta, anche se è totalmente assente in altri orari in cui il personale potrebbe uscire o entrare nell'ufficio.

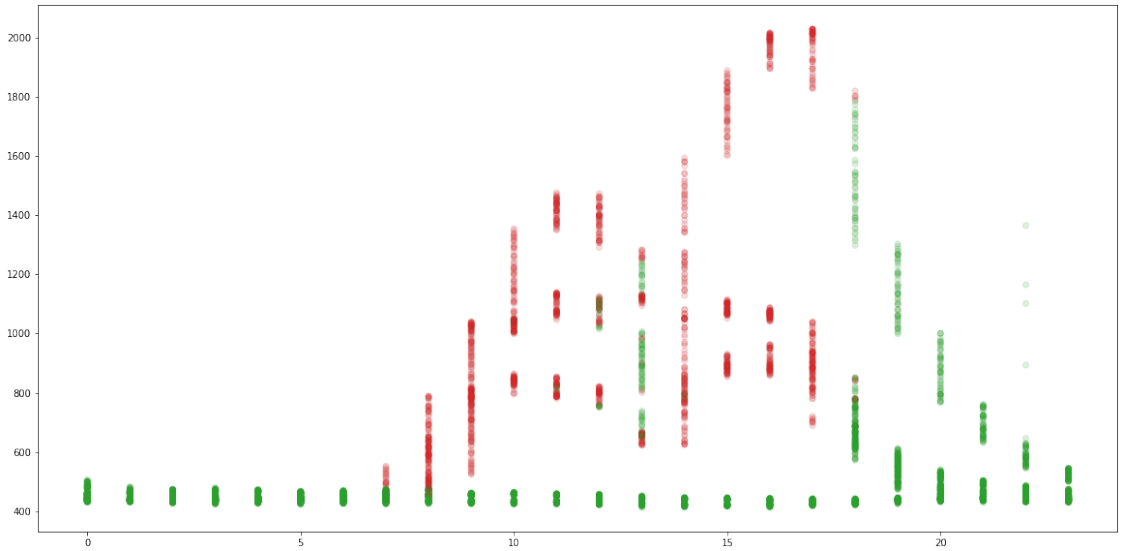
[20]:
```
fig = plt.figure(figsize=(20, 10))

plt.scatter(data["Date"].dt.hour, data["Light"], c=data["Occupancy"].
 ↪map(class_colors_map), alpha=0.15)
```

[20]:
```
<matplotlib.collections.PathCollection at 0x7f5acb7560b8>
```



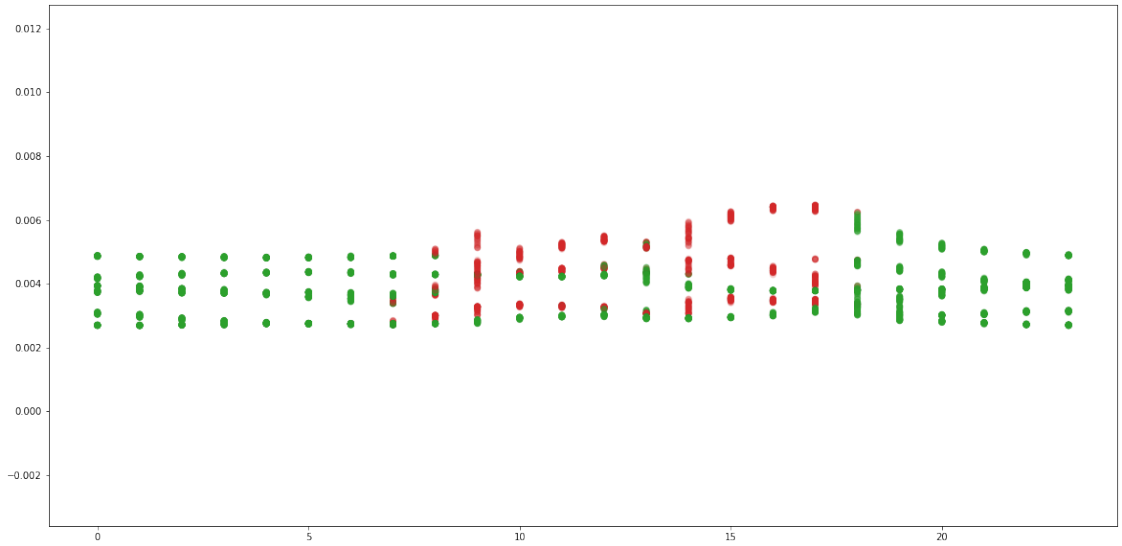
[21]:
```
fig = plt.figure(figsize=(20, 10))

plt.scatter(data["Date"].dt.hour, data["CO2"], c=data["Occupancy"].
 ↪map(class_colors_map), alpha=0.15)
```

[21]:
```
<matplotlib.collections.PathCollection at 0x7f5acb718080>
```



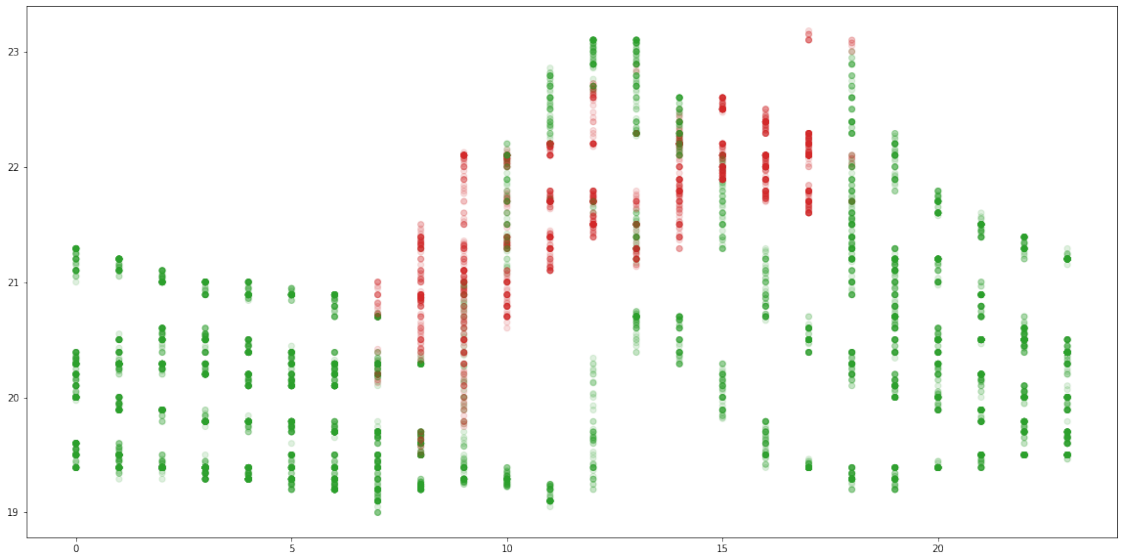
[22]:
```
fig = plt.figure(figsize=(20, 10))

plt.scatter(data["Date"].dt.hour, data["Humidity_Ratio"], c=data["Occupancy"].
 ↪map(class_colors_map), alpha=0.15)
```

[22]: <matplotlib.collections.PathCollection at 0x7f5ac93b1048>



[23]:
```
fig = plt.figure(figsize=(20, 10))

plt.scatter(data["Date"].dt.hour, data["Temperature"], c=data["Occupancy"].
 ↪map(class_colors_map), alpha=0.15)
```

[23]: <matplotlib.collections.PathCollection at 0x7f5ac939a2e8>



1.2 Parte 2 - Feature preprocessing

Inizialmente vengono importate le librerie e il validation set dal file zip precedentemente scaricato

[24]:
```
from sklearn.model_selection import train_test_split
import datetime as dt
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import Perceptron
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from scipy import stats
```

1.2.1 Validation set

[25]:
```
def import_dataset_and_preprocess_it(file_name):
    with open(file_validation_set_name) as dataFile:
        data_raw = pd.read_csv(dataFile, sep=",")

    return cast_date_time_and_show(data_raw.rename(columns={"date" : "Date",
                                    "HumidityRatio": "Humidity_Ratio"}))
```

[26]:
```
data_val = import_dataset_and_preprocess_it(file_validation_set_name)

data_val.head()
```

```
count                   2665
unique                  2665
top       2015-02-03 07:25:59
freq                       1
first     2015-02-02 14:19:00
last      2015-02-04 10:43:00
Name: Date, dtype: object
```

[26]:

	Date	Temperature	Humidity	Light	CO2	\
140	2015-02-02 14:19:00	23.7000	26.272	585.200000	749.200000	
141	2015-02-02 14:19:59	23.7180	26.290	578.400000	760.400000	
142	2015-02-02 14:21:00	23.7300	26.230	572.666667	769.666667	
143	2015-02-02 14:22:00	23.7225	26.125	493.750000	774.750000	
144	2015-02-02 14:23:00	23.7540	26.200	488.600000	779.000000	

	Humidity_Ratio	Occupancy
140	0.004764	1
141	0.004773	1
142	0.004765	1
143	0.004744	1
144	0.004767	1

Si può notare che i dati provenienti dal file del validation set rappresentano un insieme temporale molto vicino e in intersezione con quello di training, onde evitare che il modello sfrutti in

qualche modo questa caratteristica poco naturale, uniamo i due gruppi, mescoliamo il contenuto ed estraiamo due nuovi set di training e validation, che saranno usati per addestrare modelli con il metodo di **hold-out**

[27]:
```
dataset_joined = pd.concat([data_val, data])

print("Shape dei due dataset combinati = {}".format(dataset_joined.shape))

dataset_joined_unix = dataset_joined.copy()
dataset_joined_unix["Date"] = (dataset_joined_unix["Date"] - dt.
 ↪datetime(1970,1,1)).dt.total_seconds()
```

```
Shape dei due dataset combinati = (10808, 7)
```

[28]:
```
X_pre_train, X_pre_val, y_pre_train, y_pre_val = train_test_split(
    dataset_joined_unix.drop(["Occupancy"], axis=1),
    dataset_joined_unix["Occupancy"],
    test_size=1/3, random_state=42
)

print("Training set:   \n{}\n".format(X_pre_train["Date"].describe()))
print("Validation set: \n{}".format(X_pre_val["Date"].describe()))
```

```
Training set:
count    7.205000e+03
mean     1.423231e+09
std      1.952644e+05
min      1.422887e+09
25%      1.423075e+09
50%      1.423238e+09
75%      1.423398e+09
max      1.423561e+09
Name: Date, dtype: float64

Validation set:
count    3.603000e+03
mean     1.423229e+09
std      1.962623e+05
min      1.422887e+09
25%      1.423073e+09
50%      1.423234e+09
75%      1.423400e+09
max      1.423561e+09
Name: Date, dtype: float64
```

print_eval stampa tre indici di errore: - Con il metodo score è possibile sapere, utilizzando i dati del validation o del test set, il coefficiente di determinazione del modello appena costruito - Relative error ritorna la media degli scarti degli errori

[29]:
```
def get_coefficients(model, index, model_name="model"):
    return pd.Series(model.named_steps[model_name].coef_[0], index=index)

def fit_and_eval(model, X_train, y_train, X_val, y_val):
    model.fit(X_train, y_train)
    print("R-squared coefficient: {:.5}".format(model.score(X_val, y_val)))
```

1.2.2 Standardizzazione, regolarizzazione e multicollinearità

Si esegue una prova utilizzando i dati suddivisi precedentemente per allenare un semplice modello di classificazione basato sull'algoritmo Perceptron.

[30]:
```
model = Pipeline([
    ("model",  Perceptron(n_jobs=-1, random_state=42))
])
fit_and_eval(model, X_pre_train, y_pre_train, X_pre_val, y_pre_val)
```

```
R-squared coefficient: 0.25507
```

Sono visualizzati i coefficienti del modello per comprendere quali siano le feature più importanti. In questo caso è evidente che le misurazioni della luce e delle tracce di CO2 giochino il ruolo più importante rispetto alle altre misurazioni nella predizione di nuovi dati. La data ha l'impatto più alto sulla predizione, questo potrebbe aver causato così poca accuratezza. E'possibile provare a **standardizzare** i valori per vedere se il modello può migliorare.

[31]:
```
model_coeff = get_coefficients(model, X_pre_train.columns)
model_coeff
```

[31]:
```
Date              6.494821e+08
Temperature       1.570784e+04
Humidity          2.344418e+04
Light             4.805673e+06
CO2               5.660272e+06
Humidity_Ratio    7.778605e+00
dtype: float64
```

[32]:
```
std_model = Pipeline([
    ("scaler", StandardScaler()),
    ("model",  Perceptron())
])
fit_and_eval(std_model, X_pre_train, y_pre_train, X_pre_val, y_pre_val)
```

```
R-squared coefficient: 0.98751
```

[33]:
```
std_model_coeff = get_coefficients(std_model, X_pre_train.columns)
std_model_coeff
```

[33]:
```
Date             -1.074585
Temperature      -4.866701
Humidity         -4.974188
```

```
Light             9.154769
CO2               2.779431
Humidity_Ratio    7.833317
dtype: float64
```

La standardizzazioen comporta un grande miglioramento dello score. Si può provare ad aggiungere una **penalizzazione di tipo L1** per scoprire se tutte le variabili sono realmente utili al modello, e confermare alcune relazioni fra variabili scoperte durante la fase esplorativa.

[34]:
```
std_pen_model = Pipeline([
    ("scaler", StandardScaler()),
    ("model",  Perceptron(penalty="l1", alpha=0.0001))
])
fit_and_eval(std_pen_model, X_pre_train, y_pre_train, X_pre_val, y_pre_val)
```

```
R-squared coefficient: 0.98418
```

[35]:
```
std_pen_model_coeff = get_coefficients(std_pen_model, X_pre_train.columns)
std_pen_model_coeff
```

[35]:
```
Date             -3.001233
Temperature      -1.942962
Humidity          0.000000
Light             7.421260
CO2               1.431194
Humidity_Ratio    2.610306
dtype: float64
```

Il modello perde un pò di accuratezza, il peso delle varie feature si allinea, `Light` si conferma la feature più decisiva, confermando l'andamento di alcuni grafici in fase esplorativa. Si nota inoltre la poca utilità di `Humidity` che già in fase esplorativa non aveva raccolto grande interesse. L'annullamento di `Humidity` chiarisce definitivamente la dipendenza che questa feature ha con altre variabili, risolvendo il problema di **multicollinearità** con la feature `Humidity_Ratio`.

Di seguito si può notare quanto sia decisivo determinare il valore corretto per gli iperparametri, lo score può cambiare nettamente.

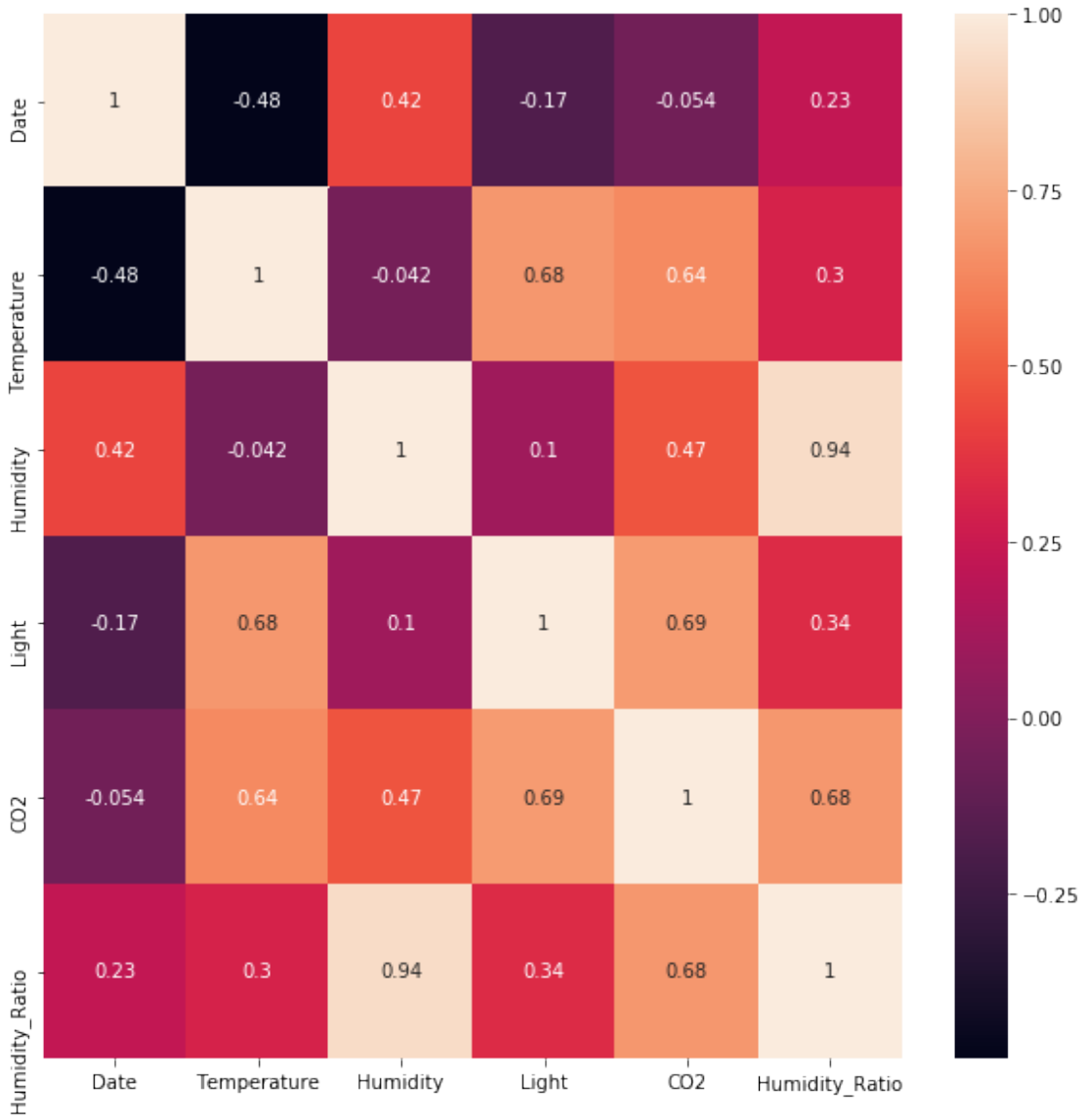
[36]:
```
model_coeffs = []
i = 0

for alpha in np.logspace(-3, 0, 4):
    print("alpha {} :\n".format(alpha))
    model = Pipeline([("scaler", StandardScaler()), ("model", ␣
 ↪Perceptron(penalty="l1", alpha=alpha))
    ])
    fit_and_eval(model, X_pre_train, y_pre_train, X_pre_val, y_pre_val)
    model_coeffs.append(get_coefficients(model, X_pre_train.columns))
    i += 1
    print("\n")
```

```
alpha 0.001 :
```

```
R-squared coefficient: 0.81238


alpha 0.01 :

R-squared coefficient: 0.98723


alpha 0.1 :

R-squared coefficient: 0.86511


alpha 1.0 :

R-squared coefficient: 0.74493
```

Per provare la dipendenza fra `Humidity` e `Humidity_Ratio` si usa la **Correlazione di Pearson**. Di seguito è presente un **heat map** di tutte le correlazioni fra le feature, che conferma l'altissima correlazione fra le due variabili. Meno accentuata è la correlazione fra `Humidity_Ratio` e `Temperature`, forse a causa della poca variazione di quest'ultima, mentre si possono notare altre sostanziali correlazioni fra `Temperature`, `CO2` e `Light` probabilmente per il fatto che tutti questi valori tendono a crescere con la presenza di persone nell'ufficio

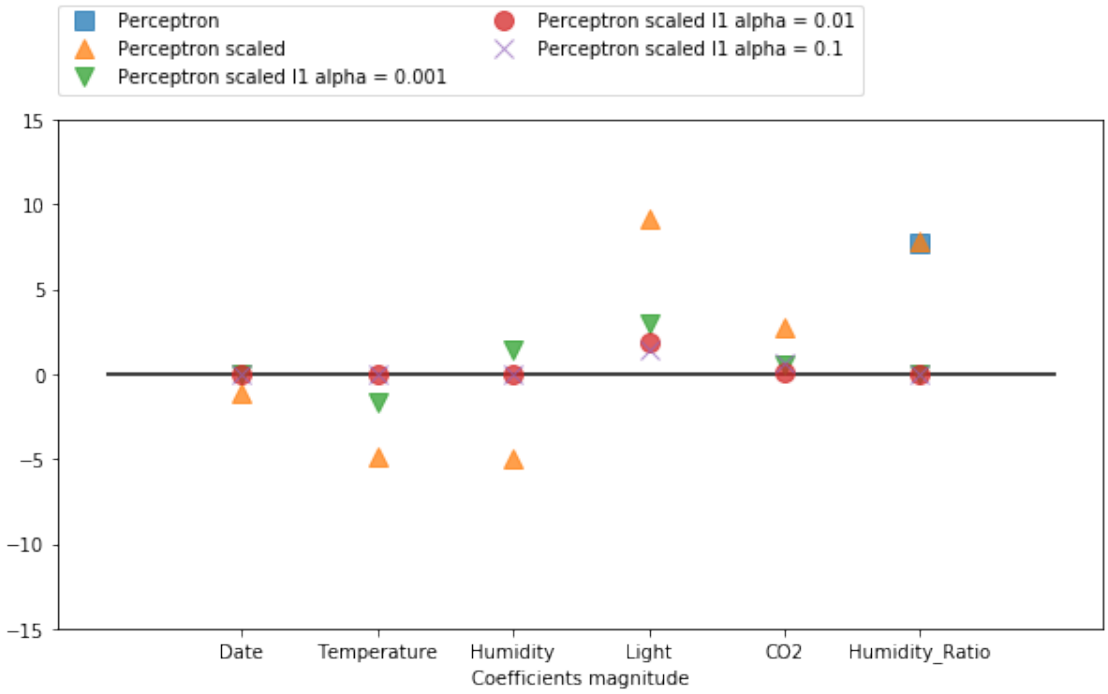
[37]:
```
pearson_correlation = X_pre_train.corr(method='pearson')

fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111)

sb.heatmap(pearson_correlation, annot=True)

plt.show()
```



Vengono di seguito mostrati su un grafico i valori dei coefficienti nelle varie regolarizzazioni con L1, le feature `Temperature`, `Humidity` e `Humidity_Ratio` tendono ad essere annullate in fretta per le dipendenze che hanno fra loro mentre altre come `Light` e `CO2` restano tendenzialmente molto significative

[38]:
```
fig = plt.figure(figsize=(10, 5))

plt.plot(model_coeff, 's', alpha=0.75, ms=10, label="Perceptron")
plt.plot(std_model_coeff, '^', alpha=0.75, ms=10, label="Perceptron scaled")
plt.plot(model_coeffs[0], 'v', alpha=0.75, ms=10, label="Perceptron scaled l1␣
 ↪alpha = 0.001")
plt.plot(model_coeffs[1], 'o', alpha=0.75, ms=10, label="Perceptron scaled l1␣
 ↪alpha = 0.01")
```

```
plt.plot(model_coeffs[2], 'x', alpha=0.75, ms=10, label="Perceptron scaled l1␣
 ↪alpha = 0.1")
plt.hlines(0, -1, len(model_coeff))
plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-15, 15)
plt.xlabel("Coefficients index")
_ = plt.xlabel("Coefficients magnitude")
```



Inizialmente può essere comodo utilizzare usare tutte le feature per scoprire con mezzi più analitici, come standardizzazione e penalizzazione, le feature meno utili o addirittura dannose per il modello, ma questo non deve offuscare il buon senso e il significato reale delle feature all'interno del modello. `Date`, precedentemente convertita in formato UNIX, contiene informazioni che nel dataset in possesso risultano ridondanti, come l'anno e il mese che è costante. Il timestamp inoltre ha una natura molto diversa dal resto delle feature, non è una misurazione scientifica assoluta, ma un dato influenzato da aspetti sociali ed esterni dal dataset e dall'ambito di studio. Ad esempio, si potrebbe supporre che se il modello fosse allenato su un dataset molto più ampio, contenente i dati di molti anni, potrebbe essere spinto ad associare ai periodi di festività l'assenza di persone nell'ufficio. Probabilmente non sarebbe completamente scorretto, ne irrimediabile con un allenamento e un giusto bilanciamento delle feature che tenga conto di questa situazione, ma è importante osservare queste caratteristiche.

Nel dataset: - gli anni non cambiano e non dovrebbero influenzare - i mesi non cambiano e non dovrebbero influenzare - i giorni sono pochi e non dovrebbero influenzare - le ore influenzano decisamente le altre misurazioni, è ben visibile il pattern della routine di un ufficio, gli orari di apertura e chiusura sono intuibili nel grafico temporale della feature `Light` - i minuti e i secondi

sono ripetuti ogni ora, sia in quelle dove è probabile che l'ufficio sia occupato che nelle altre, quindi è più difficile che siano determinanti nella classificazione

Ignorando la data si ha un notevole miglioramento con l'uso puro del Perceptron, ma un abbassamento in quello più complesso.

[39]:
```
fit_and_eval(std_model, X_pre_train.drop(["Date"], axis=1), y_pre_train,␣
↪X_pre_val.drop(["Date"], axis=1), y_pre_val)
```

```
R-squared coefficient: 0.97558
```

[40]:
```
fit_and_eval(std_pen_model, X_pre_train.drop(["Date"], axis=1), y_pre_train,␣
↪X_pre_val.drop(["Date"], axis=1), y_pre_val)
```

```
R-squared coefficient: 0.98723
```

Infine viene considerato solo: - il giorno, completamente penalizzato con `Humidity`, ma ottiene lo score migliore. - l'ora, penalizzato con `CO2`, secondo score migliore - i minuti, completamente penalizzati con `Humidity`, ma si ottiene lo score peggiore - i secondi, completamente penalizzati, terzo score migliore

[41]:
```
dataset_joined_daily = dataset_joined.copy()
dataset_joined_hourly = dataset_joined.copy()
dataset_joined_minutely = dataset_joined.copy()
dataset_joined_secondly = dataset_joined.copy()

dataset_joined_daily["Date"] = (dataset_joined_daily["Date"].dt.day).
↪astype("int64")
dataset_joined_hourly["Date"] = (dataset_joined_hourly["Date"].dt.hour).
↪astype("int64")
dataset_joined_minutely["Date"] = (dataset_joined_minutely["Date"].dt.minute).
↪astype("int64")
dataset_joined_secondly["Date"] = (dataset_joined_secondly["Date"].dt.second).
↪astype("int64")

datasets = {"Day" : dataset_joined_daily,
            "Hour" : dataset_joined_hourly,
            "Minute" : dataset_joined_minutely,
            "Second" : dataset_joined_secondly}

for name, value in datasets.items():
    X_train, X_val, y_train, y_val = train_test_split(
        value.drop(["Occupancy"], axis=1),
        value["Occupancy"],
        test_size=1/3, random_state=42
    )
    print(name + ": ")
    fit_and_eval(std_pen_model, X_train, y_train, X_val, y_val)
    print(get_coefficients(std_pen_model, X_train.columns))
```

```
    print("\n")
```

```
Day:
R-squared coefficient: 0.98113
Date               0.000000
Temperature       -1.564631
Humidity           0.000000
Light             10.224578
CO2                3.162925
Humidity_Ratio     1.055726
dtype: float64


Hour:
R-squared coefficient: 0.98029
Date               0.060890
Temperature       -2.335478
Humidity          -0.978834
Light             10.599593
CO2                0.000000
Humidity_Ratio     0.701627
dtype: float64


Minute:
R-squared coefficient: 0.92506
Date              0.000000
Temperature      -0.752823
Humidity          0.000000
Light             5.364673
CO2               2.861636
Humidity_Ratio    3.040504
dtype: float64


Second:
R-squared coefficient: 0.97419
Date              0.000000
Temperature      -0.956395
Humidity         -0.930250
Light             8.175963
CO2               2.959528
Humidity_Ratio    3.454902
dtype: float64
```

In generale non bisogna escludere che il modello generato possa venir usato in altri ambiti,

dove non solo gli orari di lavoro potrebbero essere diversi, ma anche l'uso degli impianti di illuminazione, i valori di inquinamento atmosferico e le temperature.

1.3 Parte 3 - Modellazione

Importiamo le librerie e l'ultimo file utilizzabile come test set. Anche in questo caso i dati forniti come test set sono un sottoinsime temporale del training come era avvenuto per quello di validation.

[42]:
```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
import math
from sklearn.ensemble import RandomForestClassifier
```

Sono testati altri modelli di classificazione, sul dataset in cui `Date` viene trasformato da timestamp completo ad uno orario. Infine tramite **Grid Search** e **K-fold cross validation** sono generati 5 modelli differenti in grado di stimare gli iperparametri e permettere il calcolo dell'accuratezza dei modelli studiati.

[43]:
```
models = {}
k_fold = KFold(n_splits=5, shuffle=True, random_state=42)

def print_k_cross_validation_scores(model, X, y, kf):
    scores = cross_val_score(model, X, y, cv=kf)
    print("            Scores: {}\n              Mean: {}\nStandard deviation:␣
 ↪{}".format(scores, scores.mean(), scores.std()))

def grid_search_with_cross_validation(model, grid, kf, dataset, scoring=None):
    grid_search = GridSearchCV(model, grid, scoring=scoring, cv=kf, n_jobs=-1)

    X_train, X_val, y_train, y_val = train_test_split(
        dataset.drop(["Occupancy"], axis=1),
        dataset["Occupancy"],
        test_size=1/3, random_state=42
    )

    grid_search.fit(X_train, y_train)

    score = grid_search.score(X_val, y_val)

    print("Best cross validation score: {}\n".format(grid_search.best_score_))
    print("             Test set score: {}\n".format(score))
    print("                Best params: {}\n".format(grid_search.best_params_))
    print("             Best estimator: {}\n".format(grid_search.
 ↪best_estimator_))
```

```
    return grid_search.best_estimator_, score
    #print(pd.DataFrame(grid_search.cv_results_))
```

1.3.1 Perceptron

Viene controllato nuovamente il punteggio che otterrebbe ottimizzando gli iperparametri e facendo cross-fold validation.

[44]:
```
%%time

per_model = Pipeline([
    ("scaler", StandardScaler()),
    ("per", Perceptron(n_jobs=-1, random_state=42))
])

#print(per_model.get_params())

per_grid = {
    "scaler": [None, StandardScaler()],
    "per__penalty": ["l2", "l1", "elasticnet"],
    "per__alpha": np.logspace(-3, 3, 7),
    "per__fit_intercept": [False, True]
}

per_model, score = grid_search_with_cross_validation(per_model, per_grid,␣
 ↪k_fold, dataset_joined_hourly)
models["Perceptron"] = {"Model": per_model, "Score": score}
```

```
Best cross validation score: 0.9719639139486468

            Test set score: 0.8318068276436303

               Best params: {'per__alpha': 0.01, 'per__fit_intercept': True,
'per__penalty': 'l2', 'scaler': StandardScaler(copy=True, with_mean=True,
with_std=True)}

            Best estimator: Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('per',
                 Perceptron(alpha=0.01, class_weight=None, early_stopping=False,
                            eta0=1.0, fit_intercept=True, max_iter=1000,
                            n_iter_no_change=5, n_jobs=-1, penalty='l2',
                            random_state=42, shuffle=True, tol=0.001,
                            validation_fraction=0.1, verbose=0,
                            warm_start=False))],
         verbose=False)
```

```
CPU times: user 816 ms, sys: 172 ms, total: 988 ms
Wall time: 2.8 s
```

1.3.2 Logistic Regression

Metodo per ottenere un piano di separazione non ottimale ma lineare. Si può decidere: - Standardizzazione i dati sottraendo la media e scalando con la varianza - Metodo di penalizzazione/regolarizzazione - il valore di C, ovvero dell'inverso dell'intensità di regolarizzazione - Aggiungere o meno la intercetta alla funzione

[45]:
```
%%time

log_model = Pipeline([
    ("scaler", StandardScaler()),
    ("lr", LogisticRegression(solver='liblinear', random_state=42))
])

#print(log_model.get_params())

log_grid = {
    "scaler": [None, StandardScaler()],
    "lr__penalty": ["l2", "l1"],
    "lr__C": np.logspace(-4, 2, 7),
    "lr__fit_intercept": [False, True]
}

log_model, score = grid_search_with_cross_validation(log_model, log_grid,␣
 ↪k_fold, dataset_joined_hourly)
models["Logistic Regression"] = {"Model" : log_model, "Score": score}
```

```
Best cross validation score: 0.9851492019430951

            Test set score: 0.9875104079933389

               Best params: {'lr__C': 0.001, 'lr__fit_intercept': False,
'lr__penalty': 'l2', 'scaler': None}

            Best estimator: Pipeline(memory=None,
         steps=[('scaler', None),
                ('lr',
                 LogisticRegression(C=0.001, class_weight=None, dual=False,
                                    fit_intercept=False, intercept_scaling=1,
                                    l1_ratio=None, max_iter=100,
                                    multi_class='warn', n_jobs=None,
                                    penalty='l2', random_state=42,
                                    solver='liblinear', tol=0.0001, verbose=0,
                                    warm_start=False))],
         verbose=False)
```

```
CPU times: user 655 ms, sys: 561 ms, total: 1.22 s
Wall time: 4.34 s
```

1.3.3 SVM

Support Vector Machine, è un algoritmo di classificazione che trasforma lo spazio dei dati in modo che le classi diventino separabili linearmente. - Il parametro C consente di controllare l'overfitting, aumentando tale valore il margine si restringe riducendo il numero di errori di training e causando overfitting. - Inoltre vi è la possibilità di specificare il tipo di kernel da utilizzare. - Il parametro gamma, ovvero il coefficiente del kernel basato su **radial basis function** è impostato al valore standard è 1 / (n_features * X.var()).

[46]:
```
%%time

svm_model = Pipeline([
    ("scaler", StandardScaler()),
    ("svc", SVC(random_state=42))
])

#print(svm_model.get_params())

svm_grid = [
  {'svc__C': np.logspace(3, 5, 3), 'svc__kernel': ['linear']},
  {'svc__C': np.logspace(3, 5, 3), 'svc__gamma': ['scale'], 'svc__kernel':␣
 ↪['rbf']},
]

svm_model, score= grid_search_with_cross_validation(svm_model, svm_grid,␣
 ↪k_fold, dataset_joined_hourly)
models["Support Vector Machine"] = {"Model": svm_model, "Score": score}
```

```
Best cross validation score: 0.9884802220680083

             Test set score: 0.9908409658617818

                Best params: {'svc__C': 100000.0, 'svc__gamma': 'scale',
'svc__kernel': 'rbf'}

             Best estimator: Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('svc',
                 SVC(C=100000.0, cache_size=200, class_weight=None, coef0=0.0,
                     decision_function_shape='ovr', degree=3, gamma='scale',
                     kernel='rbf', max_iter=-1, probability=False,
                     random_state=42, shrinking=True, tol=0.001,
                     verbose=False))],
```

```
         verbose=False)

CPU times: user 4.22 s, sys: 456 ms, total: 4.68 s
Wall time: 5min 53s
```

1.3.4 K Neighbors Classifier

Un algoritmo concettualmente semplice che sceglie la classe da attribuire ad un determinato punto, dello spazio generato dalla feature del problema, che presenta più entità, fra le k che sono state scelte in base ad un determinato sistema di misurazione della distanza. Viene fatto il tuning sulla base di: - Standardizzazione - Numero dei vicini da considerare - Metrica adottata

[47]:
```
%%time

knc_model = Pipeline([
    ("scaler", StandardScaler()),
    ("knc", KNeighborsClassifier(n_jobs=-1))
])

#print(svm_model.get_params())

knc_grid = {"scaler": [None, StandardScaler()],
            'knc__n_neighbors': range(1, 10, 1),
            'knc__weights': ['uniform', 'distance']}

knc_model, score = grid_search_with_cross_validation(knc_model, knc_grid,␣
 ↪k_fold, dataset_joined_hourly)
models["K-Neighbor"] = {"Model": knc_model, "Score" : score}
```

```
Best cross validation score: 0.9908396946564886

            Test set score: 0.9941715237302248

               Best params: {'knc__n_neighbors': 6, 'knc__weights': 'distance',
'scaler': StandardScaler(copy=True, with_mean=True, with_std=True)}

            Best estimator: Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('knc',
                 KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                      metric='minkowski', metric_params=None,
                                      n_jobs=-1, n_neighbors=6, p=2,
                                      weights='distance'))],
         verbose=False)

CPU times: user 1.01 s, sys: 59.7 ms, total: 1.07 s
Wall time: 3.76 s
```

1.3.5 Decision Tree

Modello creato deducendo semplici regole di decisione dai dati delle varie feature in input. E' un metodo concettualmente semplice e non richiede grande preparazione dei dati ma in certe situazioni potrebbe non avere grandi performance in termini di capacità di apprendimento e adattamento a nuovi dati. Vengono regolate: - il numero minimo di record necessario ad eseguire uno split di un nodo interno - il numero minimo di record per considerare una foglia valida - la profondità dell'albero - il numero massimo di feature da considerare

[48]:
```
%%time

tree_model = Pipeline([
    ("scaler", StandardScaler()),
    ("tree", DecisionTreeClassifier(random_state=42))
])

#print(tree_model.get_params())

tree_grid = {"scaler": [None, StandardScaler()],
             'tree__min_samples_split': range(2, 4, 1),
             'tree__min_samples_leaf': range(1, 4, 1),
             'tree__max_depth': [None] + [i for i in range(2, 7)],
             'tree__max_features': range(2, num_features, 1)}

tree_model, score = grid_search_with_cross_validation(tree_model, tree_grid,␣
 ↪k_fold, dataset_joined_daily)
models["Decision Tree"] = {"Model": tree_model, "Score" : score}
```

```
Best cross validation score: 0.9890353920888272

             Test set score: 0.990563419372745

                Best params: {'scaler': None, 'tree__max_depth': None,
'tree__max_features': 5, 'tree__min_samples_leaf': 1, 'tree__min_samples_split':
3}

             Best estimator: Pipeline(memory=None,
         steps=[('scaler', None),
                ('tree',
                 DecisionTreeClassifier(class_weight=None, criterion='gini',
                                        max_depth=None, max_features=5,
                                        max_leaf_nodes=None,
                                        min_impurity_decrease=0.0,
                                        min_impurity_split=None,
                                        min_samples_leaf=1, min_samples_split=3,
                                        min_weight_fraction_leaf=0.0,
                                        presort=False, random_state=42,
                                        splitter='best'))],
         verbose=False)
```

```
CPU times: user 2.38 s, sys: 38.6 ms, total: 2.42 s
Wall time: 4.61 s
```

1.3.6 Random Forest

E' un metodo di ensemble learning, che fa uso di un insieme di alberi decisionali che indipendentemente svolgono il loro algoritmo di classificazione, il risultato finale viene poi democraticamente scelto fra le varie predizioni ottenute dali estimatori. Questa tecnica generalmente tende a diminuire il tipico overfitting dei singoli alberi decisionali.

[49]:
```
%%time

forest_model = Pipeline([
    ("scaler", StandardScaler()),
    ("forest", RandomForestClassifier(n_jobs=-1, random_state=42))
])

#print(forest_model.get_params())

forest_grid = {"scaler": [None, StandardScaler()],
             'forest__n_estimators': range(5, 10),
             'forest__min_samples_split': range(2, 5),
             'forest__max_depth': [None] + [i for i in range(1, 3)],
             'forest__max_features': [int(math.sqrt(num_features)),␣
 ↪num_features - 1]}

forest_model, score = grid_search_with_cross_validation(forest_model,␣
 ↪forest_grid, k_fold, dataset_joined_hourly)
models["Random Forest"] = {"Model": forest_model, "Score" : score}
```

```
Best cross validation score: 0.9908396946564886

            Test set score: 0.9938939772411879

               Best params: {'forest__max_depth': None, 'forest__max_features':
2, 'forest__min_samples_split': 2, 'forest__n_estimators': 7, 'scaler': None}

            Best estimator: Pipeline(memory=None,
         steps=[('scaler', None),
                ('forest',
                 RandomForestClassifier(bootstrap=True, class_weight=None,
                                        criterion='gini', max_depth=None,
                                        max_features=2, max_leaf_nodes=None,
                                        min_impurity_decrease=0.0,
                                        min_impurity_split=None,
                                        min_samples_leaf=1, min_samples_split=2,
                                        min_weight_fraction_leaf=0.0,
```

```
                                       n_estimators=7, n_jobs=-1,
                                       oob_score=False, random_state=42,
                                       verbose=0, warm_start=False))],
         verbose=False)

CPU times: user 8.99 s, sys: 651 ms, total: 9.64 s
Wall time: 28.2 s
```

Riepilogo dll'accuratezza ottenuta

[50]:
```
for name, model in models.items():
    print(name + ":\n\t\t\t\t{}".format(model["Score"]))
```

```
Perceptron:
                                0.8318068276436303
Logistic Regression:
                                0.9875104079933389
Support Vector Machine:
                                0.9908409658617818
K-Neighbor:
                                0.9941715237302248
Decision Tree:
                                0.990563419372745
Random Forest:
                                0.9938939772411879
```

1.4 Parte 4 - Valutazione dei modelli di classificazione

Spesso l'accuratezza non è una buona stima della capacità predittiva di un modello di classificazione, in particolare nei casi in cui il dataset è sbilanciato, perchè contiene maggiori record appartenenti ad una classe. In questo caso se si costruisse un modello che predica sempre la stanza coma "non occupata", esso avrebbe una accuratezza di circa il 75% nel predirla non occupata.

[51]:
```
X_train, X_val, y_train, y_val = train_test_split(
    dataset_joined_hourly.drop(["Occupancy"], axis=1),
    dataset_joined_hourly["Occupancy"],
    test_size=1/3, random_state=42)

y_val.value_counts(normalize=True)
```

[51]:
```
0    0.744935
1    0.255065
Name: Occupancy, dtype: float64
```

[52]:
```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score, recall_score, f1_score
```

1.4.1 Precision, Recall e F1-Score

Si costruisce una **matrice di confusione** per ciascun modello creato in precedenza, calcolando la **precisione**, la **recall** e la **F1-score**. Il Perceptron ottiene nuovamente lo score peggiore, Logistic

Regression e SVM hanno all'incirca la medesima performance in F1-Score, ma la precisione della prima è nettamente peggiore. Anche K-Neighbor e Random Forest ottengono all'incirca il medesimo risultato. Il Decision Tree ottiene un risultato superiore al Perceptron, ma la recall peggiore.

[53]:
```
def calculate_precision_recall_f1(name, X, y) :
    model = models[name]
    y_pred = model["Model"].predict(X)

    model["Precision"] = precision_score(y, y_pred, pos_label=1)
    model["Recall"] = recall_score(y, y_pred)
    model["F1_Score"] = f1_score(y, y_pred, average="macro")

    print("{}:\n\n \
    Confusion matrix:\n{}\n\n \
           Precision: {}\n \
              Recall: {}\n \
    F1 Score (macro): {}\n\n\n\n".format(name,
                                pd.DataFrame(confusion_matrix(y, y_pred),␣
 ↪index=classes, columns=classes),
                                model["Precision"],
                                model["Recall"],
                                model["F1_Score"]))
```

[54]:
```
classes = ["Free", "Occupied"]

for name, model in models.items():
    calculate_precision_recall_f1(name, X_val, y_val)
```

```
Perceptron:

     Confusion matrix:
          Free  Occupied
Free      2132       552
Occupied    54       865

             Precision: 0.6104446012702893
                Recall: 0.941240478781284
     F1 Score (macro): 0.8080734367528339



Logistic Regression:

     Confusion matrix:
          Free  Occupied
Free      2640        44
Occupied     1       918

             Precision: 0.9542619542619543
```

```
               Recall: 0.998911860718172
     F1 Score (macro): 0.9838129253992858



Support Vector Machine:

     Confusion matrix:
          Free  Occupied
Free      2664        20
Occupied    13       906

            Precision: 0.978401727861771
               Recall: 0.985854189336235
     F1 Score (macro): 0.9879791265735824



K-Neighbor:

     Confusion matrix:
          Free  Occupied
Free      2674        10
Occupied    11       908

            Precision: 0.9891067538126361
               Recall: 0.9880304678998912
     F1 Score (macro): 0.9923284875076208



Decision Tree:

     Confusion matrix:
          Free  Occupied
Free      2666        18
Occupied    60       859

            Precision: 0.9794754846066135
               Recall: 0.9347116430903155
     F1 Score (macro): 0.9710762054925919



Random Forest:

     Confusion matrix:
          Free  Occupied
```

```
Free      2674        10
Occupied    12       907

            Precision: 0.9890948745910578
               Recall: 0.9869423286180631
     F1 Score (macro): 0.9919602974647339
```

1.4.2 Confronto modelli con intervallo di confidenza

Si verifica con **confidenza** al 95% se la F1-Score dei modelli è statisticamente significativa una dall'altra. I modelli basati su K-Neighbor e Random Forest ottengono risultati praticamente uguali infatti, l'intervallo non garantisce che la K-Neighbor sia migliore della Random Forest, stessa cosa accade fra Logistic Regression e SVM.

Si denotano quindi due coppie di modelli in sretta competizione fra loro e un modello che si trova a livello di performance sotto entrambe: - Random Forest e K-Neighbor (score 0.99..) - SVM e Logistic Regression (score 0.98..) - Decision Tree (score 0.97..)

[55]:
```
def difference_between_two_models(error1, error2, confidence):
    z_half_alfa = stats.norm.ppf(confidence)
    variance = (((1 - error1) * error1) / len(y_val)) + (((1 - error2) *␣
 ↪error2) / len(y_val))
    d_minus = abs(error1 - error2) - z_half_alfa * (pow(variance, 0.5))
    d_plus = abs(error1 - error2) + z_half_alfa * (pow(variance, 0.5))
    print("Valore minimo: {}\nValore massimo: {}\n".format(d_minus, d_plus))

svm_error = 1 - models["Support Vector Machine"]["F1_Score"]
lre_error = 1 - models["Logistic Regression"]["F1_Score"]
knc_error = 1 - models["K-Neighbor"]["F1_Score"]
tree_error = 1 - models["Decision Tree"]["F1_Score"]
forest_error = 1 - models["Random Forest"]["F1_Score"]

print("Support Vector Machine vs Logistic Regression, intervallo di confidenza:
 ↪")
difference_between_two_models(svm_error, lre_error, 0.95)

print("Support Vector Machine vs K-Neighbor, intervallo di confidenza:")
difference_between_two_models(svm_error, knc_error, 0.95)

print("Decision Tree vs Support Vector Machine, intervallo di confidenza:")
difference_between_two_models(tree_error, svm_error, 0.95)

print("Decision Tree vs K-Neighbor, intervallo di confidenza:")
difference_between_two_models(tree_error, knc_error, 0.95)

print("Decision Tree vs Random Forest, intervallo di confidenza:")
```

```
difference_between_two_models(tree_error, forest_error, 0.95)

print("Random Forest vs K-Neighbor, intervallo di confidenza:")
difference_between_two_models(forest_error, knc_error, 0.95)
```

```
Support Vector Machine vs Logistic Regression, intervallo di confidenza:
Valore minimo: -0.0004028777093146021
Valore massimo: 0.00873528005790794

Support Vector Machine vs K-Neighbor, intervallo di confidenza:
Valore minimo: 0.0005238428022691808
Valore massimo: 0.00817487906580756

Decision Tree vs Support Vector Machine, intervallo di confidenza:
Valore minimo: 0.011424854299745102
Valore massimo: 0.022380987862236024

Decision Tree vs K-Neighbor, intervallo di confidenza:
Valore minimo: 0.016074682713422037
Valore massimo: 0.02642988131663583

Decision Tree vs Random Forest, intervallo di confidenza:
Valore minimo: 0.015680278966726885
Valore massimo: 0.02608790497755717

Random Forest vs K-Neighbor, intervallo di confidenza:
Valore minimo: -0.0030530733366320715
Valore massimo: 0.003789453422405885
```

Riepilogo F1-Score

[56]:
```
for name, model in models.items():
    print(name + ":\n\t\t\t\t\t{}".format(model["F1_Score"]))
```

```
Perceptron:
                                0.8080734367528339
Logistic Regression:
                                0.9838129253992858
Support Vector Machine:
                                0.9879791265735824
K-Neighbor:
                                0.9923284875076208
Decision Tree:
                                0.9710762054925919
Random Forest:
                                0.9919602974647339
```

I modelli che apparentemente sembrano essere i migliori in base allo score ottenuto durante il tuning degli iper parametri in grid search con cross validation e F1-score sono: - K-Neighbor - Random Forest - Support Vector Machine

1.5 Parte 5 - Analisi del modello migliore

[57]:
```
from sklearn.dummy import DummyClassifier
```

1.5.1 Confronto con modello casuale

Viene creato un modello casuale

[58]:
```
random = DummyClassifier(strategy="uniform", random_state=42)
random.fit(X_train, y_train)

random_score = random.score(X_val, y_val)
models["Dummy"] = {"Model" : random, "Score": random_score}
random_score
```

[58]:
```
0.4948653899528171
```

Ovviamente tutti i modelli scelti precedentemente sono statisticamente migliori di uno casuale, di seguito sono riportati gli intervalli di confidenza

[59]:
```
calculate_precision_recall_f1("Dummy", X_val, y_val)

difference_between_two_models(1 - models["Support Vector Machine"]["F1_Score"],␣
↪1 - models["Dummy"]["F1_Score"], 0.99)

difference_between_two_models(1 - models["Random Forest"]["F1_Score"], 1 -␣
↪models["Dummy"]["F1_Score"], 0.99)

difference_between_two_models(1 - models["K-Neighbor"]["F1_Score"], 1 -␣
↪models["Dummy"]["F1_Score"], 0.99)
```

```
Dummy:

     Confusion matrix:
          Free  Occupied
Free      1337      1347
Occupied   473       446

            Precision: 0.2487451199107641
               Recall: 0.485310119695321
     F1 Score (macro): 0.46196206544812946



Valore minimo: 0.5062388228227107
Valore massimo: 0.5457952994281953
```

```
Valore minimo: 0.5103686944711007
Valore massimo: 0.5496277695621082

Valore minimo: 0.5107507550432632
Valore massimo: 0.5499820890757195
```

1.5.2 Precision Recall Curve

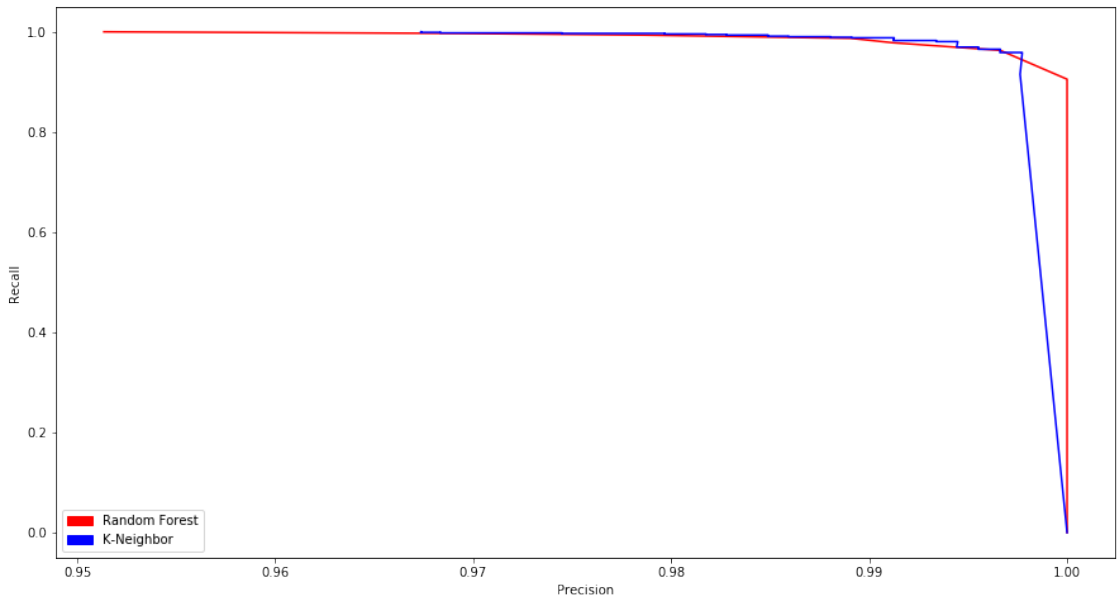
Escludendo SVM che ottiene il punteggio peggiore fra i tre e da un punto di vista computazionale è il più oneroso e richiede molto più tempo per l'addestramento bisogna scegliere il modello migliore fra K-Neighbor e Random Forest.

Non essendo chiaro il motivo per il quale si vuole determinare la presenza o meno di persone di un ufficio è difficile scegliere il migliore, se il sistema dovesse essere usato per gestire questioni di sicurezza, a supporto di un sistema di sorveglianza, potrebbe essere meglio una maggiore recall. Se fosse a supporto nella coordinazione del personale nell'evacuazione dall'edificio in caso di incendio sarebbe meglio una maggiore precisione per ottimizzare la tempestività dei soccorsi.

Non essendo noto il campo di applicazione che avrà il modello addestrato e non potendo quindi decidere una soglia minima di precision o recall, si deve procede analizzando tutte le possibili soglie e trade-off fra precision e recall utilizzando una **precision-recall curve**, che è consideata migliore della **ROC** per i problemi sbilanciati sul numero di istanze di una classe rispetto all'altra.

[60]:
```
from sklearn.metrics import precision_recall_curve

fig = plt.figure(figsize=(15, 8))

red_patch = mpatches.Patch(color='red', label='Random Forest')
blue_patch = mpatches.Patch(color='blue', label='K-Neighbor')
plt.legend(handles=[red_patch, blue_patch])

#predict_proba ritorna la probabilità per ciascun record che appartenga alla␣
 ↪classe 1 (colonna 1) o 0 (colonna 0)
#print("Media probabilità classe 1: {}".format(models["Random Forest"]["Model"].
 ↪predict_proba(X_val)[:, 1].mean()))

#valori
#print(np.unique(models["Random Forest"]["Model"].predict_proba(X_val)[:, 1]))

precision_rf, recall_rf, tresholds_rf = precision_recall_curve(y_val,
    models["Random Forest"]["Model"].predict_proba(X_val)[:, 1])

_ = plt.plot(precision_rf, recall_rf, color="red")

precision_knn, recall_knn, tresholds_knn = precision_recall_curve(y_val,
    models["K-Neighbor"]["Model"].predict_proba(X_val)[:, 1])

_ = plt.plot(precision_knn, recall_knn, color="blue")
```

```
_ = plt.xlabel("Precision")
_ = plt.ylabel("Recall")
```



Maggiore è l'area sotto la curva migliore è il modello, è evidente che non esistono sostanziali differenze fra i Random Forest e K-Neighbor, la precisione rimane alta anche con valori molto elevati di recall, si può notare che che la Random Forest si avvicina di più all'angolo in alto a destra e ottiene visivamente l'area maggiore sotto la curva.

A seguito di queste considerazioni, è stato scelto il modello basato su Random Forest

1.5.3 Test finale

Viene importata una piccola parte del dataset per fare una finale valutazione utilizzando dei dati che il modello non ha mai utilizzato per addestrarsi ne fare il tuning degli iperparametri. I dati seppur nuovi, rappresentano sempre un insieme temporale che si trova in buona parte in intersezione con quelli usati precedentemente.

[61]:
```
data_test = import_dataset_and_preprocess_it(file_test_set_name)

data_test.head()
```

```
count                    2665
unique                   2665
top       2015-02-03 07:25:59
freq                        1
first     2015-02-02 14:19:00
last      2015-02-04 10:43:00
Name: Date, dtype: object
```

[61]:
```
                  Date  Temperature  Humidity       Light         CO2  \
140 2015-02-02 14:19:00      23.7000    26.272  585.200000  749.200000
141 2015-02-02 14:19:59      23.7180    26.290  578.400000  760.400000
142 2015-02-02 14:21:00      23.7300    26.230  572.666667  769.666667
143 2015-02-02 14:22:00      23.7225    26.125  493.750000  774.750000
144 2015-02-02 14:23:00      23.7540    26.200  488.600000  779.000000

     Humidity_Ratio  Occupancy
140        0.004764          1
141        0.004773          1
142        0.004765          1
143        0.004744          1
144        0.004767          1
```

[62]:
```
data_test["Date"] = (data_test["Date"].dt.hour).astype("int64")

X = data_test.drop(["Occupancy"], axis=1)
y = data_test["Occupancy"]
```

I punteggi ottentuti con Random Forest sul test set confermano che il modello non è in overfit, in quanto gli score sono migliori dei precedenti ottenuti in fase di training e tuning degli iperparametri.

[64]:
```
y_pred = models["Random Forest"]["Model"].predict(X)

print("Accuracy  {}".format(models["Random Forest"]["Model"].score(X, y)))
print("Precision {} ".format(precision_score(y, y_pred, pos_label=1)))
print("Recall    {} ".format(recall_score(y, y_pred)))
print("F1-Score  {} ".format(f1_score(y, y_pred, average="macro")))
```

```
Accuracy  0.9962476547842402
Precision 0.9948559670781894
Recall    0.9948559670781894
F1-Score  0.9959513149035365
```