Random Forrest

- Phase: Data Modelling -

Das Ziel dieses Notebooks ist die Erstellung eines Modells mit Hilfe des Random Forrest Algorithmus. Beim Random Forest handelt es sich um ein sogenanntes Ensemble Modell. Dieses kombiniert mehrere Entscheidungsbäume miteinander und berechnet deren Prognosemittelwert um die Vorhersagegenauigkeit des Modells im Gegensatz zu einem einzelnen Entscheidungsbaum zu verbessern. Dazu verwendet der Algorithmus Bootstrap Aggregation (bagging). Hierbei werden aus den Trainingsdaten zufällig verschiedene Stichproben gezogen (mit Zurücklegen) anhand derer dann ein Entscheidungsbaum erstellt wird. Aus allen Bäumen wird dann die durchschnittliche Vorhersage ermittelt, welche das Ergebnis des Random Forest ist [1]

Dieses Notebook verwendet die folgenden Dateien counts_prepared.pkl und Feiertage.xlsx.

Es werden keine Dateien durch dieses Notebook erzeugt.

In [1]:
```
import pandas as pd
import numpy as np
import datetime
from pandas.tseries.holiday import USFederalHolidayCalendar as calendar
import matplotlib.pyplot as plt
from matplotlib import style
import seaborn as sn
from sklearn import metrics
import math
from sklearn.model_selection import GridSearchCV
```

In [2]:
```
DATA_PATH = '../data/'
HOLIDAYS = DATA_PATH + 'Feiertage.xlsx'
```

In [3]:
```
df_count=pd.read_pickle(DATA_PATH+'counts_prepared.pkl')
```

In [4]:
```
df_count.reset_index()
```

Out[4]:

	index	date	count_in	count_out	day_of_week	dewpoint	humidity	precipitation	pressure	temperature	...	hour_19	hour_20	hour_
0	0	2015-01-01	42	54	1	-14.0	29.0	0.0	1026.7	2.2	...	0	0	
1	1	2015-01-01	98	114	1	-12.3	36.0	0.0	1026.5	1.1	...	0	0	
2	2	2015-01-01	116	100	1	-11.0	40.0	0.0	1026.3	1.1	...	0	0	
3	3	2015-01-01	27	16	1	-11.8	39.0	0.0	1025.6	0.6	...	0	0	
4	4	2015-01-01	7	8	1	-11.2	41.0	0.0	1025.1	0.6	...	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
25966	25966	2017-12-31	54	59	31	-17.1	38.0	0.0	1028.3	-5.0	...	1	0	
25967	25967	2017-12-31	33	30	31	-18.3	36.0	0.0	1028.4	-5.6	...	0	1	
25968	25968	2017-12-31	46	46	31	-17.2	41.0	0.0	1028.5	-6.1	...	0	0	
25969	25969	2017-12-31	22	25	31	-14.9	52.0	0.0	1028.7	-6.7	...	0	0	
25970	25970	2017-12-31	18	14	31	-14.5	56.0	0.0	1028.9	-7.2	...	0	0	

25971 rows × 46 columns

In [5]:
```
df_count.head()
```

Out[5]:

	date	count_in	count_out	day_of_week	dewpoint	humidity	precipitation	pressure	temperature	winddirection	...	hour_19	hour_20	ho
0	2015-01-01	42	54	1	-14.0	29.0	0.0	1026.7	2.2	220.0	...	0	0	
1	2015-01-01	98	114	1	-12.3	36.0	0.0	1026.5	1.1	210.0	...	0	0	
2	2015-01-01	116	100	1	-11.0	40.0	0.0	1026.3	1.1	230.0	...	0	0	
3	2015-01-01	27	16	1	-11.8	39.0	0.0	1025.6	0.6	250.0	...	0	0	
4	2015-01-01	7	8	1	-11.2	41.0	0.0	1025.1	0.6	170.0	...	0	0	

5 rows × 45 columns

In [6]:
```
df_count = df_count.drop(['count_in'], axis=1)
```

In [7]:
```
df_count.reset_index(inplace=True)
```

Erstellen von Trainings- und Testdaten

Trainingsdaten von 2015 und 2016, Testdaten von 2017

In [8]:
```
array = [2015, 2016]
```

In [9]:
```
dataTrain = df_count.loc[df_count['year'].isin(array)]
dataTrain = dataTrain.sort_values(by=['date'])
dataTest = df_count.loc[df_count['year']==2017]
dataTest = dataTest.sort_values(by=['date'])

datetimecol = dataTest['date']
yLabels = df_count["count_out"]
```

In [10]:
```
# Trennen von Trainings- und Testdaten in unabhängige Variablen (X) und abhängige Variable (count_out y)
drop_cols = ['date', 'count_in', 'index']
y_cols =['count_out']
feature_cols = [col for col in df_count.columns if (col not in y_cols) & (col not in drop_cols)]

X_train = dataTrain[feature_cols]
X_test = dataTest[feature_cols]

y_train = dataTrain['count_out']
y_test = dataTest['count_out']
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

```
(17402, 42) (17402,) (8569, 42) (8569,)
```

In [11]:
```
dataTest.columns
```

Out[11]:
```
Index(['index', 'date', 'count_out', 'day_of_week', 'dewpoint', 'humidity',
       'precipitation', 'pressure', 'temperature', 'winddirection',
       'windspeed', 'workingday', 'year', 'Season_Fall', 'Season_Spring',
       'Season_Summer', 'hour_0', 'hour_1', 'hour_2', 'hour_3', 'hour_4',
       'hour_5', 'hour_6', 'hour_7', 'hour_8', 'hour_9', 'hour_10', 'hour_11',
       'hour_12', 'hour_13', 'hour_14', 'hour_15', 'hour_16', 'hour_17',
       'hour_18', 'hour_19', 'hour_20', 'hour_21', 'hour_22', 'weekday_Friday',
       'weekday_Monday', 'weekday_Saturday', 'weekday_Thursday',
       'weekday_Tuesday', 'weekday_Wednesday'],
      dtype='object')
```

Definition einer geeigneten Kostenfunktion

Um das Ergebnis eines Algorithmus bewerten zu können, wird eine sog. Kostenfunktion eingesetzt. Kostenfunktionen messen den Unterschied zwischen dem vorhergesagten Wert und dem tatsächlichen Wert[2] und geben so Auskunft über die Performance des Modells. Je nach Problemstellung kommen verschiedene Kostenfunktionen in Frage. Hier soll nun eine der am häufigsten verwendeten Funktionen, der Root Mean Squared Log Error (RMSLE) verwendet werden . Berechnet wird der RMSLE aus der logarithmierten Quadratwurzel des durchschnittlichen Prognosefehlers. Durch die Verwendung des Logarithmus ist RMSLE recht stabil gegenüber Ausreißern. Je größer der RMSLE ist, desto schlechter ist die Anpassung des Modells. Ziel ist es folglich RMSLE zu minimieren um so die Güte des Modells zu steigern. Im Folgenden wird zuerst eine Funktion zur Berechnung des RMSLE definiert. Anschließend wird eine Funktion implementiert, die basierend auf einem Regressor ein Modell trainiert und den RMSLE-Wert berechnet.

In [12]:
```
# Implementierung des RMSE
def rmsle(y_test, y_pred):
    rmsle = np.sqrt(metrics.mean_squared_log_error( y_test, y_pred ))
    return (rmsle)
```

In [13]:
```
from sklearn.metrics import mean_squared_error as mse
from sklearn.metrics import r2_score
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold

def score_model(model):
    # Model wird basierend auf Trainingsdaten gefitted. Dann wird Prognose anhand der Testdaten durchgeführt und RM
SLE berechnet
    model.fit(X_train, y_train)
    yhat = model.predict(X_test)
    r2 = r2_score(y_test, yhat)
    me = rmsle(y_test, yhat)
    print("Ergebnis von {}: \nr2={:0.3f} \nRMSLE={:0.3f}".format(model, r2, me))
```

Baseline Prediction

Um einen Vergleichswert zu erhalten, an dem die im weiteren implementierten Algorithmen gemessen werden können, wird ein Baseline Modell erstellt. Hierbei handelt es sich um ein einfaches, leicht nachzuvollziehendes Modell, dessen Performance die anderen Modelle übertreffen sollen. Für Regressions-Probleme eignen sich statistische Maße wie Median oder Mittelwert[3].

Als Baseline Modell wurde der Mittelwert der Variable 'count_out' gewählt. Das Modell wird mit Hilfe der DummyRegressor-Bibliothek von sklearn implementiert.

In [14]:
```
from sklearn.dummy import DummyRegressor
dummy_regr = DummyRegressor(strategy="mean")
dummy_regr.fit(X_train, y_train)
DummyRegressor()

score_model(dummy_regr)
```

```
Ergebnis von DummyRegressor(constant=None, quantile=None, strategy='mean'):
r2=-0.012
RMSLE=1.675
```

Das Baseline-Model liefert einen RMSLE-Wert von 1,675 und einen r2-Wert von -0,012. Dies spricht für eine sehr schlechte Anpassung des Modells.Ausgehend vom Data Understanding, bei dem stunden- und tagesabhängige Schwankungen (Saisonalität) identizifiert wurden, war dieses Ergebnis zu erwarten. Die reine Vorhersage unter Verwendung des Mittelwerts kann diese Schwankungen nicht abbilden.

Random Forest mit Default-Parametern

Um einen ersten Eindruck der Performance des Random Forest Regressors zu gewinnen, wird zunächst ein Modell mit Standard-Parametern trainiert. Es wird der Random Forest Regressor von sklearn verwendet. Folgende Parameter sind definiert:

1. n_estimators: Anzahl der Entscheidungsbäume, die erstellt werden (default: 100).
2. criterion: Misst Güte eines Splits (default: Mean Squared Error).
3. max_depth: Maximale Tiefe eines Baumes. Wenn kein Wert definiert ist, werden die daten solange gesplittet, bis jedes Blatt weniger als in min_samples_split definierte Datensätze enthält (default: none).
4. min_samples_split: Anzahl an Datensätzen die nötig sind, damit ein Split durchgeführt wird (default:2).
5. min_samples_leaf: Anzahl der Datensätze, die mindestens in einem Blattknoten enthalten sein müssen. Ein Split wird nur dann durchgeführt, wenn mindestens min_samples_leaf Datensätze nach dem Split im linken und rechten Knoten enthalten wären (default:1).
6. max_features: Anzahl an Variablen, die bei einem Split beachtet werden (default: auto = alle).

In [14]:
```
from sklearn.ensemble import RandomForestRegressor
```

In [15]:
```
rf_base = RandomForestRegressor()
rf_base.fit(X_train, y_train)
```

Out[15]:
```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=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=100, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False)
```

In [16]:
```
score_model(rf_base)
```

```
Ergebnis von RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=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=100, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False):
r2=0.796
RMSLE=0.534
```

Hyperparameter tuning

Die Ausführung des Random Forest mit Standardparametern liefert einen RMSLE-Score von 0,53. Um diesen Wert noch zu verbessern, können die sog. Hyperparameter, sprich die Einstellungen für den Algorithmus optimiert werden. Um diese zu identifizieren wird zunächst eine Random Search durchgeführt. Diese kombiniert zuvor definierte Werte der verschiedenen Parameter zufällig miteinander und ermittelt die optimale Kombination. In der untenstehenden Implementierung wird mittels n_iter = 100 und cv= 3 definiert, dass 100 verschiedene Kombinationen (der 1260 möglichen Kombinationen) trainiert und diese auf 3 verschiedenen Teilmengen der Trainingsdaten validiert werden. Die optimale Kombination wird dann für die Prognose verwendet und der RMSLE ermittelt.

In [19]:
```
from sklearn.model_selection import RandomizedSearchCV
```

In [20]:
```
# Festlegen der zu testenden n_estimators
n_estimators = [int(x) for x in np.linspace(start = 100, stop = 600, num = 6)]
# Maximale Anzahl der Features die bei jedem Split verwendet werden
max_features = ['auto', 'sqrt']
# Maximale Tiefe der Bäume
max_depth = [int(x) for x in np.linspace(30, 80, num = 6)]
max_depth.append(None)
# Minimale Anzahl an Datenpunkten bevor Split durchgeführt wird
min_samples_split = [2, 5, 10]
# Minimale Anzahl an Datenpunkten in einem "Blatt"
min_samples_leaf = [1, 2, 4]
# Methode zur Stichprobenauswahl (mit oder ohne Zurücklegen)
bootstrap = [True, False]

random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}

print(random_grid)
```

```
{'n_estimators': [100, 200, 300, 400, 500, 600], 'max_features': ['auto', 'sqrt'], 'max_depth': [30, 40, 50, 60, 7
0, 80, None], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4], 'bootstrap': [True, False]}
```

In [21]:
```
#Zuvor definierte Parameter werden verwendet um die beste Kombination an Hyperparametern zu identifizieren.
rf = RandomForestRegressor()
#Zufällige Suche nach Parametern. Verwendung von 3-facher Kreuzvalidierung --> Trainingsdaten werden in 3 Sets unte
rteilt
#Testen von 100 verschiedenen, zufälligen Kombinationen (n_iter)
rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid, n_iter = 100, cv = 3, verbose=2,
random_state=42, n_jobs = -1)# Fit the random search model
rf_random.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 100 candidates, totalling 300 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:  1.4min
[Parallel(n_jobs=-1)]: Done 146 tasks      | elapsed: 15.2min
[Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed: 27.8min finished
```

Out[21]:
```
RandomizedSearchCV(cv=3, error_score=nan,
                   estimator=RandomForestRegressor(bootstrap=True,
                                                   ccp_alpha=0.0,
                                                   criterion='mse',
                                                   max_depth=None,
                                                   max_features='auto',
                                                   max_leaf_nodes=None,
                                                   max_samples=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=100,
                                                   n_jobs=None, oob_score=Fals...
                                                   warm_start=False),
                   iid='deprecated', n_iter=100, n_jobs=-1,
                   param_distributions={'bootstrap': [True, False],
                                        'max_depth': [30, 40, 50, 60, 70, 80,
                                                      None],
                                        'max_features': ['auto', 'sqrt'],
                                        'min_samples_leaf': [1, 2, 4],
                                        'min_samples_split': [2, 5, 10],
                                        'n_estimators': [100, 200, 300, 400,
                                                         500, 600]},
                   pre_dispatch='2*n_jobs', random_state=42, refit=True,
                   return_train_score=False, scoring=None, verbose=2)
```

random_rf = rf_random.best_estimator_rf_random.best_estimator_

Ergebnisse der Random Search:
'n_estimators': 100,
'max_features': 'auto',
'max_depth': 70,
'min_samples_split': 2,
'min_samples_leaf': 1,
'bootstrap': True

In [19]:
```
# Durch Random Search ermittelte Paramter werden Random Forest übergeben. So muss Random Search nicht erneut durchg
eführt werden
rf_random = RandomForestRegressor(bootstrap = True, max_depth = 70, max_features = 'auto', max_leaf_nodes = None, m
ax_samples = None,
               min_samples_leaf=1, min_samples_split=2, n_estimators=100)
```

In [20]:
```
# Ermitteln des RMSLE für das Modell mit der durch Random Search ermittelten Hyperparameter-Kombination
score_model(rf_random)
```

```
Results from RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=70, max_features='auto', max_leaf_nodes=None,
                      max_samples=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=100, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False):
r2=0.795
RMSLE=0.539
```

In [21]:
```
from sklearn.model_selection import cross_val_score
accuracy = cross_val_score(estimator = rf_random, X = X_train, y = y_train, cv =10)
accuracy.mean()
```

Out[21]:
```
0.739275049982572
```

Im Vergleich zum Random Forest mit Default-Parametern, konnte der RMSLE-Score von 0,536 auf 0,53 reduziert werden. Auch der Wert von r2 und Accuracy hat sich minimal verbessert. Insgesamt hat das Hyperparameter Tuning durch Random Search aber nicht zu einer nennenswerten Verbesserung der Prognose geführt.

In einem nächsten Schritt sollen die Hyperparameter weiter angepasst werden, um das Modell noch weiter zu verbessern. Hierzu wird eine sog. Grid Search durchgeführt. Im Gegensatz zur Random Search wird hier nicht eine bestimmte Anzahl zufälliger Kombinationen verwendet, sondern es werden Modelle zu allen Kombinationen trainiert. Um die Anzahl möglicher Kombinationen einzuschränken und die Performance zu erhöhen werden die durch die Random Search ermittelten Parameter verwendet um das Parameter Grid für die Grid Search zu definieren.
Es ergeben sich 216 verschiedene Kombinationsmöglichkeiten, aus denen im Folgenden wiederum die beste Kombination ermittelt wird.

param_grid = {'n_estimators': [500,600,700], 'max_features':['auto', 'sqrt', 'log2'], 'max_depth':[60,70,80], 'min_samples_leaf':[1,2], 'min_samples_split':[5,10], 'bootstrap': [True, False]}

regressor = RandomForestRegressor() parameters = [{'n_estimators' : [150,200,250,300], 'max_features' : ['auto','sqrt','log2']}] grid_search = GridSearchCV(estimator = regressor, param_grid = param_grid, n_jobs=-1)

grid_search = grid_search.fit(X_train, y_train) best_parameters = grid_search.best*params* best_accuracy = grid_search.best*score*

best_parameters

Ergebnis der Grid Search:
bootstrap: True
max_depth: 70
max_features: auto
min_samples_leaf: 2
min_samples_split: 10
n_estimators: 300

In [17]:
```
# Durch Random Search ermittelte Paramter werden Random Forest übergeben. So muss Random Search nicht erneut durchgeführt werden
rf_grid = RandomForestRegressor(bootstrap = True, max_depth = 70, max_features = 'auto', max_leaf_nodes = None, max_samples = None,
                                min_samples_leaf=2, min_samples_split=10, n_estimators=300)
```

In [18]:
```
score_model(rf_grid)
```

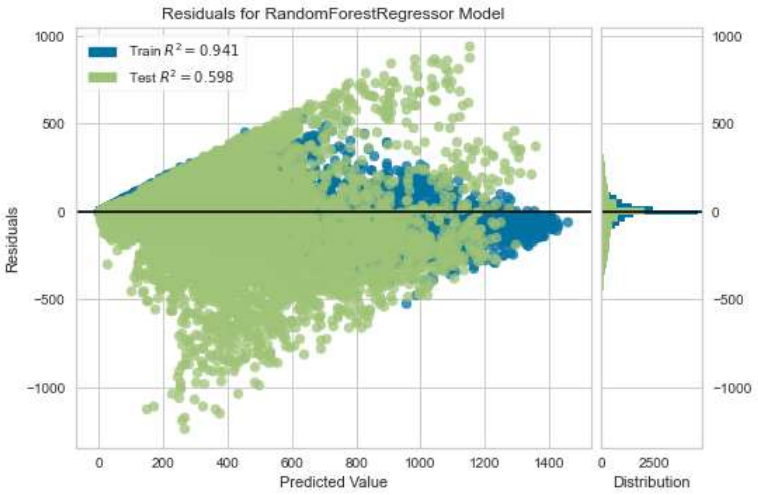
```
Ergebnis von RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=70, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=2,
                      min_samples_split=10, min_weight_fraction_leaf=0.0,
                      n_estimators=300, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False):
r2=0.796
RMSLE=0.533
```

accuracy = cross_val_score(estimator = rf_grid, X = X_train, y = y_train, cv =10) accuracy.mean()

Vergleicht man die RMSLE-Werte der Random Search bzw. Grid Search mit dem Ergebnis des Random Forest mit Default-Parametern ist zu erkennen, dass keine Verbesserung der Güte des Modells erreicht werden konnte. Diese Ergebnis lässt zwei Schlüsse zu: Entweder die Konfiguration der zu testenden Hyperparameter ist noch nicht optimal oder aber mit Random Forest kann kein besseres Prognoseergebnis erzielt werden.

Im Folgenden sollen mögliche Gründe identifiziert werden, die verhindern, dass Random Forest eine bessere Prognose erstellen kann. Hierzu wird die Yellowbrick-Bibliothek verwendet. Als erstes wird ein Residual-Plot erstellt, der die prognostizierten Werte den tatsächlichen Werten gegenüberstellt. Betrachtet man die Residuen der Test-Daten ist auffällig, dass das Modell dazu tendiert, zu hohe Werte vorherzusagen (Punkte im negativen Bereich --> actual - predicted <0). Auch wird deutlich, dass mit steigenden Ausleihzahlen die Güte der Vorhersage abnimmt.
Ein weiterer Aspekt der sichtbar wird, ist, dass das Modell zum Overfitting zu neigen scheint. Für die Trainingsdaten liegt ein r2-Wert von 0,97 vor, wobei eine perfekte Prognose durch einen Wert von 1 repräsentiert wird. Das Modell scheint die Trainingsdaten nahezu perfekt prognostizieren zu können. Für die Test-Daten wird hingegen nur ein Wert von 0,791 erreicht.

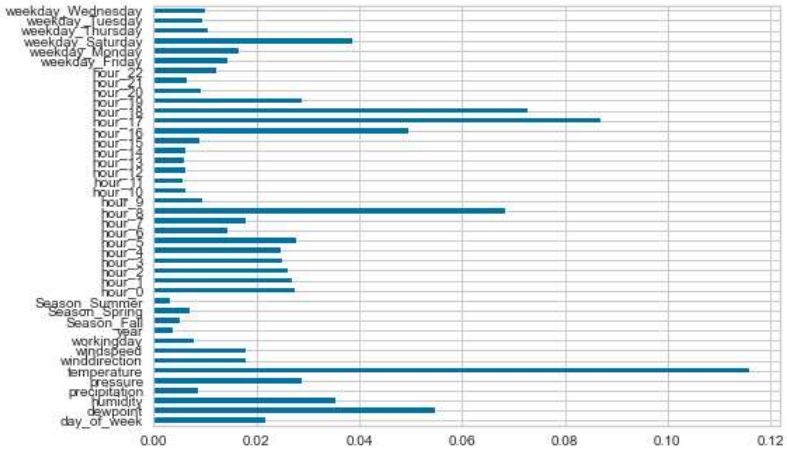
In [74]:
```
from yellowbrick.regressor import ResidualsPlot

resplot = ResidualsPlot(rf_grid)

resplot.fit(X_train, y_train)
resplot.score(X_test, y_test)
g = resplot.poof()
```



Um das Problem des Overfitting zu reduzieren, wird im nächsten Schritt die Feature-Importance ermittelt. Diese gibt an, wie hoch der Einfluss einer Variablen auf das Prognosemodell ist. Da Overfitting dann auftritt, wenn zu viele Variablen für die Prognose verwendet werden, können mittels Feature-Importance nur die Variablen ausgewählt werden, die einen Beitrag zur Erstellung der Prognose leisten

In [48]:
```
import matplotlib.pyplot as plt

print(rf_grid.feature_importances_)
feat_importances = pd.Series(rf_grid.feature_importances_, index=X_train.columns)
feat_importances.plot(kind='barh')
plt.show()
```

```
[0.02177627 0.0548653  0.03535251 0.00858136 0.02871017 0.11597862
 0.01782972 0.0179101  0.00784087 0.00379851 0.00497404 0.00708574
 0.00305828 0.0275135  0.02682225 0.02617202 0.02511341 0.0248568
 0.02776578 0.01424083 0.01782363 0.06840709 0.00945992 0.00620731
 0.00554725 0.00609947 0.005976   0.00628449 0.00891897 0.04943163
 0.08696595 0.07287596 0.02878123 0.00919559 0.00632949 0.01231653
 0.01425871 0.01653194 0.03857809 0.01043471 0.00935253 0.00997744]
```



In [83]:
```
X_train_red = X_train[['day_of_week','dewpoint','weekday_Monday', 'weekday_Friday', 'weekday_Thursday','weekday_Saturday','hour_7', 'hour_6', 'hour_5', 'hour_4', 'hour_3', 'hour_2', 'hour_1', 'hour_0', 'hour_22','hour_19', 'hour_18', 'hour_17', 'hour_16','hour_8','temperature', 'pressure', 'humidity', 'winddirection', 'windspeed']]
X_test_red = X_test[['day_of_week','dewpoint','weekday_Monday', 'weekday_Friday', 'weekday_Thursday','weekday_Saturday','hour_7', 'hour_6', 'hour_5', 'hour_4', 'hour_3', 'hour_2', 'hour_1', 'hour_0', 'hour_22','hour_19', 'hour_18', 'hour_17', 'hour_16','hour_8','temperature', 'pressure', 'humidity', 'winddirection', 'windspeed']]
```

In [84]:
```
def score_model_red(model):

    # Modell wird gefittet. Anschließend wird Prognose basierend auf Testdaten berechnet und Metrics berechnet.
    model.fit(X_train_red, y_train)
    yhat = model.predict(X_test_red)
    r2 = r2_score(y_test, yhat)
    me = rmsle(y_test, yhat)
    print("Ergebnisse von {}: \nr2={:0.3f} \nRMSLE={:0.3f}".format(model, r2, me))
```

In [85]:
```
rf_grid_red = RandomForestRegressor(bootstrap = True, max_depth = 80, max_features = 'auto', max_leaf_nodes = None,
max_samples = None,
               min_samples_leaf=1, min_samples_split=5, n_estimators=500)
```

In [86]:
```
score_model_red(rf_grid_red)
```

```
Results from RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=80, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=5, min_weight_fraction_leaf=0.0,
                      n_estimators=500, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False):
r2=0.625
RMSLE=0.707
```

Die Zahl der Variablen wurde basierend auf ihrer Wichtigkeit eingeschränkt und erneut ein Modell trainiert. In einem ersten Schritt wurden nur Variablen ausgewählt, deren Wichtigkeit größer 0,04 ist (siehe Diagramm). Hierbei wurde ein RMSLE-Wert > 1 erzielt. Anschließend wurden immer mehr Variablen dem Modell hinzugefügt, mit dem Ziel, den RMSLE-Wert zu senken. Unter Verwendung aller Variablen mit einer Wichtigkeit >0,02 konnte eine RMSLE-Wert von 0,75 erreicht werden. In einem letzten Schritt wurden anschließend alle Variablen verwendet, deren Wichtigkeit >0,01 ist. Hierbei konnte ein RMSLE-Wert von 0,684 erzielt werden.

Die vorhergehenden Schritte haben gezeigt, dass mit den vorhandenen Daten die Prognosegüte des Modells nicht weiter verbessert werden kann. Als weiterführender Schritt kann versucht werden die Datenqualität während der Data Preparation Phase weiter zu steigern. Möglich wäre die genauere Untersuchung möglicher Korrelationen zwischen Variablen und die Skalierung der Daten. Auch das Sicherstellen der Stationarität der Daten sowie das Ergänzen zeitverschobender Variablen sind eine Möglichkeit. Somit kann auf die Schwächen von Random Forest im Umgang mit Trends und die fehlende Fähigkeit, zurückliegende Werte für die Prognose zu verwenden, eingewirkt werden [4].

Als letzter Schritt wird die Prognose den tatsächlichen Werten der Testdaten gegenüber gestellt und in einem Liniendiagramm dargestellt.

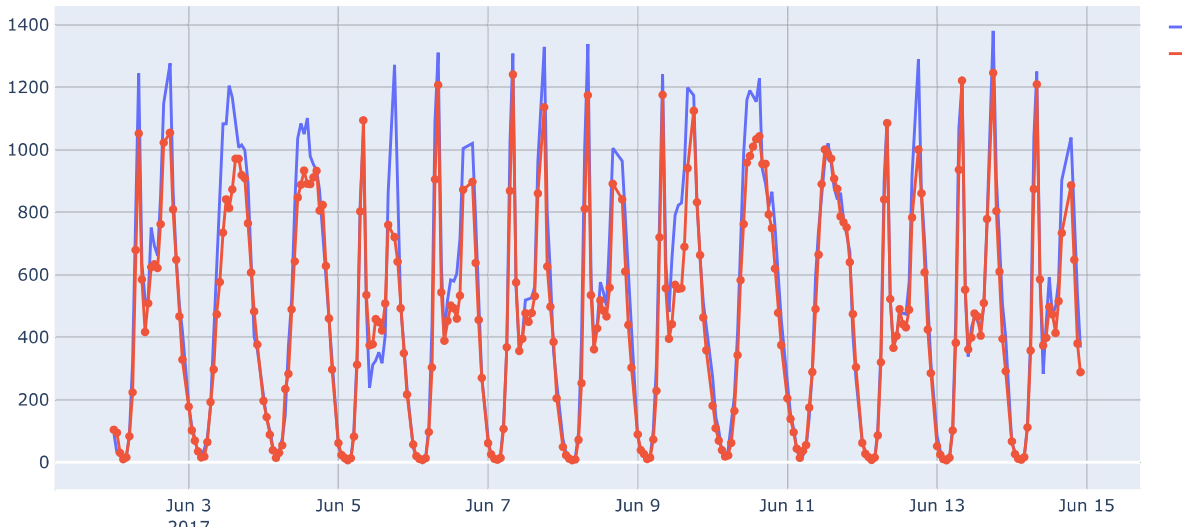
In [19]:
```
pred = rf_grid.predict(X_test)
```

In [20]:
```
# Die folgende Funktion kombiniert  Testdaten mit Vorhersagewerten. Dazu benötigt werden X_test, y_test sowie ein s
ortiertes Array
# mit Datumswerten und ein Array mit den durch den ALgorithmus vorhergesagten werten
def CombineTestPrediction (X_test, y_test, datetimevalues, prediction):
    y_test = pd.DataFrame(y_test)
    X_test_combined=X_test.join(y_test)
    X_test_combined = X_test_combined.join(pd.DataFrame(datetimevalues))
    X_test_combined.reset_index(inplace = True)
    X_test_combined['prediction']=pd.DataFrame(prediction)
    X_test_combined['date'] = pd.to_datetime(X_test_combined.date, format="%Y-%M-%D %H:%M:%S")
    X_test_combined['month'] = X_test_combined.date.dt.month
    X_test_combined['day'] = X_test_combined.date.dt.day

    return (X_test_combined)
```

In [21]:
```
X_test_combined = CombineTestPrediction(X_test, y_test, datetimecol, pred)
```

In [22]:
```
#Datenset wird auf einen bestimmten Zeitraum eingeschränkt um tatsächliche Werte und Vorhersagewerte als Liniendiag
ramm
# miteinander zu vergleichen

#Zeitraum definieren
start_date = '2017-06-01'
end_date = '2017-06-14'

# Datenset auf Zeitraum einschränken
date_range = (X_test_combined['date'] > start_date) & (X_test_combined['date'] <= end_date)
X_test_range = X_test_combined.loc[date_range]

# Dummy-Variablen der Uhrzeit in Spalte zurück umwandeln
ndf = pd.wide_to_long(X_test_range.reset_index(), stubnames='hour_',i='index',j='hour')
X_test_range = ndf[ndf['hour_'].ne(0)].reset_index(level='hour').drop('hour_',1)

# Spalten für Monat und Tag erstellen
X_test_range['month'] = X_test_range.date.dt.month
X_test_range['day'] = X_test_range.date.dt.day

# DateTime aus Jahr, Monat, Tag und Stunde generieren und nach Datum sortieren
X_test_range['DateTime'] = pd.to_datetime(X_test_range[['year', 'month', 'day', 'hour']], format ='%Y-%M-%D %H:%M:%
S')
X_test_range=X_test_range.sort_values(by='DateTime')

X_test_range.to_pickle(DATA_PATH+'Prognosedaten.pkl')
```

In [23]:
```
# Liniendiagramm mit tatsächlichen vs. vorhergesagten Werten erstellen
import plotly.graph_objects as go

fig = go.Figure()
fig.add_trace(go.Scatter(x=X_test_range['DateTime'], y=X_test_range['count_out'],
                    mode='lines',
                    name='actual'))
fig.add_trace(go.Scatter(x=X_test_range['DateTime'], y=X_test_range['prediction'],
                   mode='lines+markers',
                         name='predicted'))

fig.show()
```



Quellen

1. Irizarry, Rafael A.:Introduction to Data Science: Data Analysis and Prediction Algorithms with R, S.566.
2. Saleh, Hyatt: The Machine Learning Workshop - Get ready to develop your own high-performance machine learning algorithms with scikit-learn (2020), S.168
3. Moocarme, M.; Abdolahnejad, M.:The Deep Learning with Keras Workshop - An Interactive Approach to Understanding Deep Learning with Keras (2020), S.47
4. https://towardsdatascience.com/multivariate-time-series-forecasting-using-random-forest-2372f3ecbad1