# linear_regression_XYZ_MET-running-new

January 12, 2021

[1]:
```
from helpers import pandas_helper as pdh
from helpers import math_helper as mth
from sensors.activpal import *
from utils import read_functions

from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import RFE
from sklearn.model_selection import train_test_split, KFold, cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

from math import sqrt
from numpy import mean, absolute

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import datetime

activpal = Activpal()

activity_focus = 'rennen'
respondents = ['BMR002', 'BMR004', 'BMR011', 'BMR012', 'BMR014', 'BMR018',␣
↪'BMR031', 'BMR032', 'BMR033', 'BMR034', 'BMR036', 'BMR041', 'BMR042',␣
↪'BMR043', 'BMR044', 'BMR052', 'BMR053', 'BMR055', 'BMR058', 'BMR064',␣
↪'BMR097', 'BMR098']
test_respondents = ['BMR008', 'BMR040', 'BMR030']
```

## 1 Defining functions

#### 1.0.1 Method to retrieve DataFrame containing all information for regression

[2]:
```
def get_regression_df(respondent, activity):
    start, stop = get_timestamps(respondent, activity)
    res_number = get_respondent_number(respondent)
```

```python
    # read in all dataframes necessary
    respondents_df = pdh.read_csv_respondents_cleaned()
    vyntus_df, min_index, max_index = get_vyntus_df(respondent, respondents_df.
↪loc[res_number, 'weight_kg'], start, stop)
    activpal20_df = get_activpal20_df(respondent, min_index, max_index)

    # add met, mag acc, and mean speed to new dataframe
    new_df = pd.DataFrame(index=activpal20_df.index)
    new_df['mean_met'] = vyntus_df['met']

    new_df['sum_mag_acc'] = activpal20_df['sum_mag_acc']
    new_df['mean_mag_acc'] = activpal20_df['mean_mag_acc']
    new_df['std_mag_acc'] = activpal20_df['std_mag_acc']
    new_df['var_mag_acc'] = activpal20_df['var_mag_acc']

    new_df['sum_speed'] = activpal20_df['sum_speed']
    new_df['mean_speed'] = activpal20_df['mean_speed']
    new_df['std_speed'] = activpal20_df['std_speed']
    new_df['var_speed'] = activpal20_df['var_speed']

    # add features to new dataframe
    new_df['estimated_level'] = respondents_df.loc[res_number,␣
↪'estimated_level']
    new_df['length_cm'] = respondents_df.loc[res_number, 'length_cm']
    new_df['weight_kg'] = respondents_df.loc[res_number, 'weight_kg']
    new_df['waist_circumference'] = respondents_df.loc[res_number,␣
↪'waist_circumference']
    new_df['gender'] = respondents_df.loc[res_number, 'gender']
    new_df['age_category'] = respondents_df.loc[res_number, 'age_category']

    return new_df


def get_regression_dfs(respondents, activity):
    all_df = pd.DataFrame(index=pd.to_datetime([]))

    for cor in respondents:
        df = get_regression_df(cor, activity)
        all_df = pd.concat([all_df, df])


    all_df.sort_index(inplace=True)
    return all_df
```

### 1.0.2 Helper method, returning start and stop timestamps for an activity

[3]:
```python
def get_timestamps(respondent, activity):
    activities_df = read_functions.read_activities(respondent)
    start = activities_df.loc[activity].start
    stop = activities_df.loc[activity].stop

    return (start, stop)
```

### 1.0.3 Helper method, returning the number of respondent code

[4]:
```python
def get_respondent_number(respondent):
    if ('BMR0' in respondent):
        return int(respondent.replace('BMR0', ''))

    if ('BMR' in respondent):
        return int(respondent.replace('BMR', ''))
```

### 1.0.4 Helper method, returning Vyntus (lab data)

The DataFrame contains MET and other information necessary for regression

The DataFrame is resampled to minutes by mean

[5]:
```python
def get_vyntus_df(respondent, weight, start, stop):
    vyntus_df = pdh.read_csv_vyntus(respondent)
    mask = (vyntus_df.index >= start) & (vyntus_df.index < stop)
    vyntus_df = vyntus_df.loc[mask]

    min_index = vyntus_df.index.min()
    max_index = vyntus_df.index.max()

    vyntus_df['vyn_VO2'] = [float(vo2.replace(',', '.')) if type(vo2) == str␣
 ↪else vo2 for vo2 in vyntus_df['vyn_VO2']]
    vyntus_df['met'] = mth.calculate_met(vyntus_df['vyn_VO2'], weight)

    vyntus_df = vyntus_df.resample('60s').mean()[:-1]

    return (vyntus_df, min_index, max_index)
```

### 1.0.5 Helper method, returning Activpal20

The DataFrame contains sumation of magnitude of acceleration

The DataFrame is resampled to minutes by sumation

[6]:
```python
def get_activpal20_df(respondent, start, stop):
    df = activpal.read_data(respondent, start, stop)

    mask = (df.index >= start) & (df.index < stop)
    df = df.loc[mask]
    df = df[['pal_accX', 'pal_accY', 'pal_accZ']].apply(mth.convert_value_to_g)

    mag_acc = mth.to_mag_acceleration(df['pal_accX'], df['pal_accY'],␣
↪df['pal_accZ'])
    speed = get_speed(df['pal_accX'], df['pal_accY'], df['pal_accZ'])

    df['sum_mag_acc'] = mag_acc
    df['mean_mag_acc'] = mag_acc
    df['std_mag_acc'] = mag_acc
    df['var_mag_acc'] = mag_acc

    df['sum_speed'] = speed
    df['mean_speed'] = speed
    df['std_speed'] = speed
    df['var_speed'] = speed


    agg_dict = {
        'sum_mag_acc':'sum', 'mean_mag_acc': 'mean', 'std_mag_acc': 'std',␣
↪'var_mag_acc': 'var',
        'sum_speed':'sum', 'mean_speed': 'mean', 'std_speed': 'std',␣
↪'var_speed': 'var'
    }


    df = df.resample('60s').agg(agg_dict)[:-1]

    return df
```

### 1.0.6 Helper method, returns speed

[7]:
```python
def get_speed(x_acc, y_acc, z_acc):
    activpal_time = 0.05

    x_vel = x_acc.diff()**2
    y_vel = y_acc.diff()**2
    z_vel = z_acc.diff()**2

    return (x_vel + y_vel + z_vel) / activpal_time
```

### 1.0.7 Helper method to plot prediction results

[8]:
```
def plot_results_bar(pred_y, valid_y):
    plt.figure(figsize=(10,5))
    bar_width = 0.35

    pred_index = np.arange(len(pred_y))
    y_index = np.arange(len(valid_y)) + bar_width
    plt.bar(pred_index, pred_y, bar_width, color='blue', label='Prediction')
    plt.bar(y_index, valid_y, bar_width, color='red', label='Ground Truth')

    plt.xlabel('# Minute')
    plt.ylabel("MET")
    plt.title('MET predictions on validation per minute')
    plt.xticks(pred_index + 0.15, pred_index)
    plt.legend(loc='best')
    plt.grid()
    plt.show()
```

# 2 Data

Load the data and check for null values, in this case there are no null values

[9]:
```
data = get_regression_dfs(respondents, activity_focus)
data.isnull().sum()
```

[9]:
```
mean_met               0
sum_mag_acc            0
mean_mag_acc           0
std_mag_acc            0
var_mag_acc            0
sum_speed              0
mean_speed             0
std_speed              0
var_speed              0
estimated_level        0
length_cm              0
weight_kg              0
waist_circumference    0
gender                 0
age_category           0
dtype: int64
```

Split the dataset into training and validation (80/20)

[10]:
```
all_features = data.columns.drop('mean_met')
train_X, valid_X, train_y, valid_y = train_test_split(data[all_features],␣
↪data['mean_met'], test_size=0.2, random_state=0)
```

[11]:
```
data_test = get_regression_dfs(test_respondents, activity_focus)
data_test.isnull().sum()
```

[11]:
```
mean_met               0
sum_mag_acc            0
mean_mag_acc           0
std_mag_acc            0
var_mag_acc            0
sum_speed              0
mean_speed             0
std_speed              0
var_speed              0
estimated_level        0
length_cm              0
weight_kg              0
waist_circumference    0
gender                 0
age_category           0
dtype: int64
```

Split the test dataset into test X and Y

[12]:
```
test_X = data_test.drop('mean_met', axis=1)
test_y = data_test.mean_met
```

### 2.0.1 Scaling

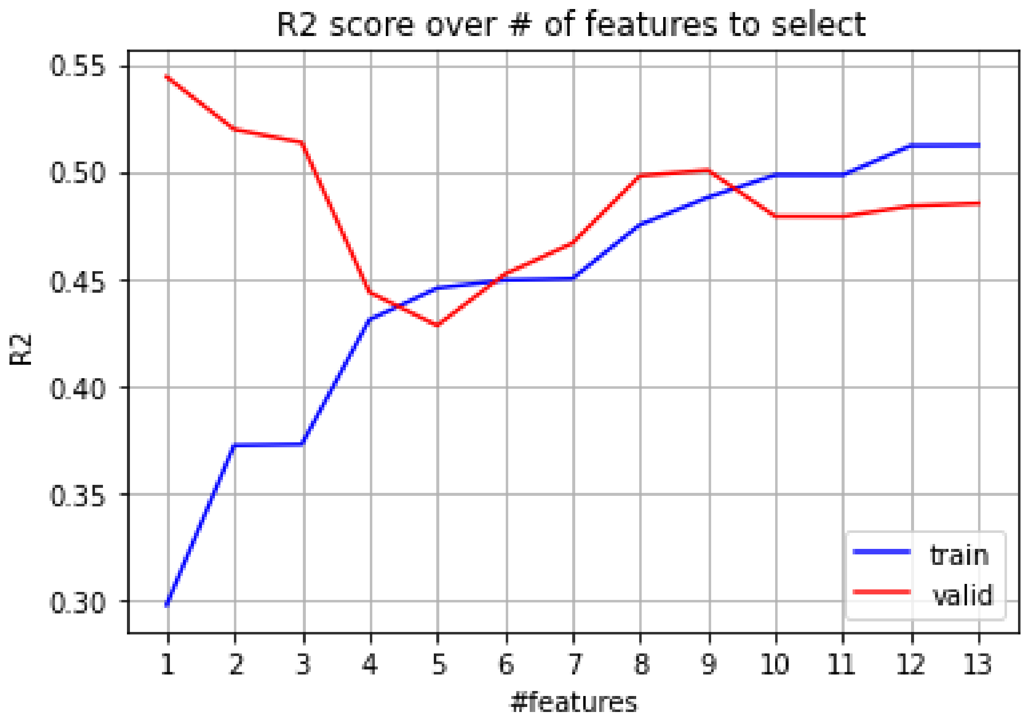
Because this notebook is working with a Linear Regression model and the feature values differ greatly, we scale the data to prevent non-convergence and oscillation

[13]:
```
scaler = StandardScaler()
scaler.fit(train_X)
train_X = scaler.transform(train_X)
valid_X = scaler.transform(valid_X)
data_X = scaler.transform(data[all_features])
test_X = scaler.transform(test_X)
```

### 2.0.2 RFE

Use RFE iteratively on training and validation set to find the optimal minimum number of features and the features themselves for the model

[14]:
```
r2t = []
r2v = []

features_support = None

for i in range(1, train_X.shape[1]):
    model = LinearRegression()
    rfe = RFE(model, n_features_to_select=i)

    rfe.fit(train_X, train_y)
    model.fit(rfe.transform(train_X), train_y)

    r2t.append(model.score(rfe.transform(train_X), train_y))
    r2v.append(model.score(rfe.transform(valid_X), valid_y))

    # if optimal number of features reached, save the best features
    if i == 9:
        features_support = rfe.support_


plt.title('R2 score over # of features to select')
plt.plot(range(1, train_X.shape[1]), r2t, c='b', label='train')
plt.plot(range(1, valid_X.shape[1]), r2v, c='r', label='valid')
plt.xlabel('#features')
plt.xticks(range(1, len(train_X[0])))
plt.ylabel('R2');
plt.grid()
plt.legend(loc='best')
plt.show()

print(all_features[features_support])
```




```
Index(['mean_mag_acc', 'std_mag_acc', 'var_mag_acc', 'sum_speed', 'mean_speed',
       'std_speed', 'estimated_level', 'waist_circumference', 'gender'],
      dtype='object')
```

Here we can see that the minimum optimal number of features is 9

Remove all columns except best features

[15]:
```
train_X = np.delete(train_X, features_support, axis=1)
valid_X = np.delete(valid_X, features_support, axis=1)
data_X = np.delete(data_X, features_support, axis=1)
test_X = np.delete(test_X, features_support, axis=1)
```

# 3 Model

[16]:
```
model = LinearRegression()
```

# 4 Training

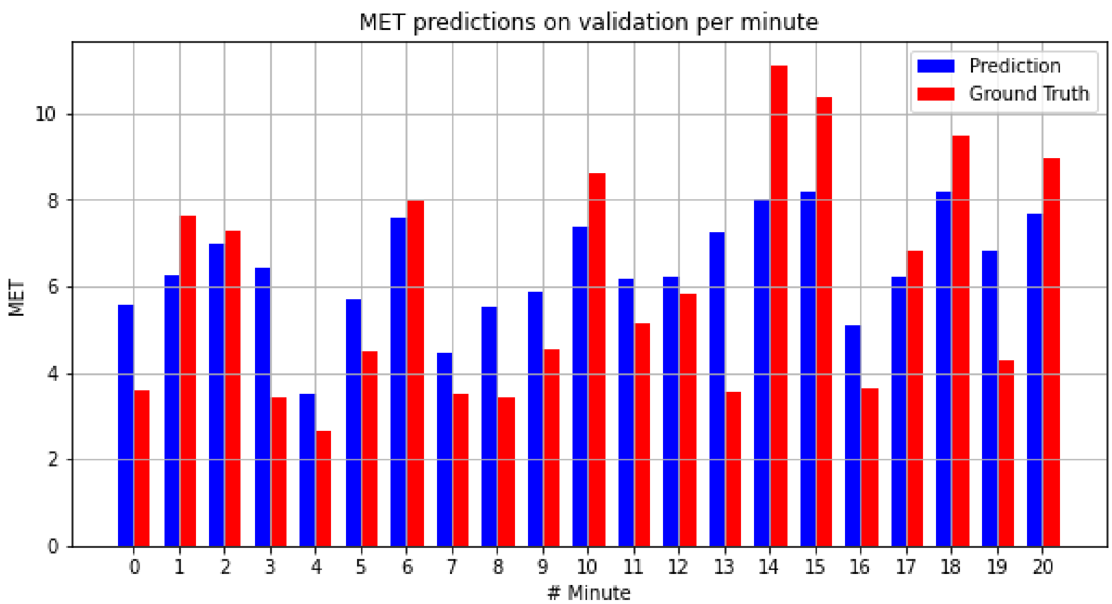
[17]:
```
model.fit(train_X, train_y)
```

[17]: LinearRegression()

# 5 Evaluation

### 5.0.1 Results Validation Set

[18]:
```
pred_y = model.predict(valid_X)

plot_results_bar(pred_y, valid_y)
print('MSE: %.2f' % mean_squared_error(valid_y, pred_y))
print('R2: %.2f' % r2_score(valid_y, pred_y))
```




```
MSE: 3.19
R2: 0.51
```

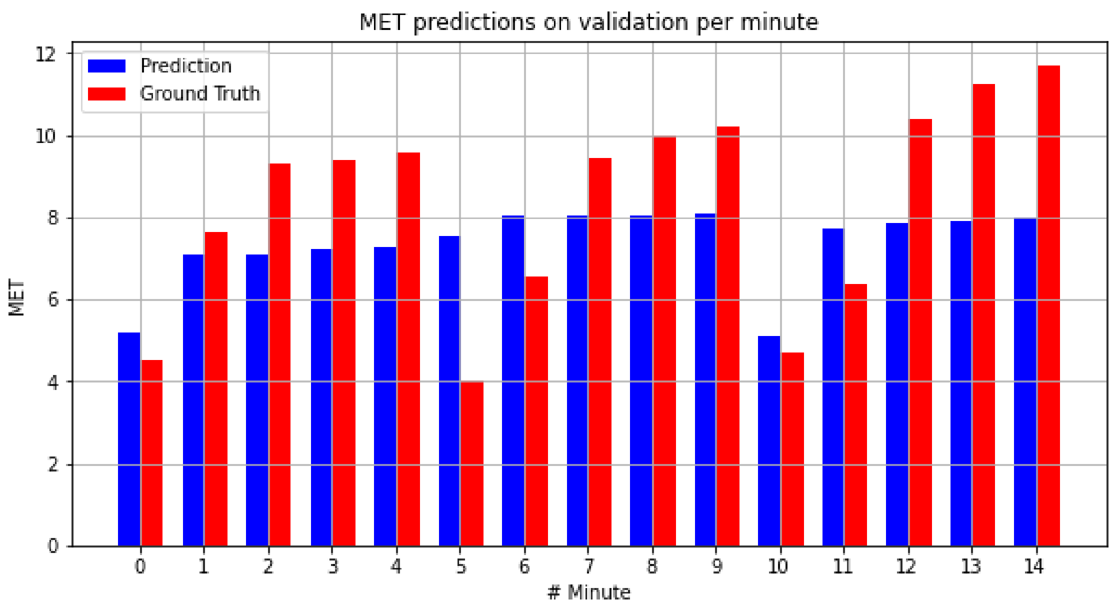
### 5.0.2 Results 5-fold Cross Validation

[19]:
```
model2 = LinearRegression()

mse = cross_val_score(model2, data_X, data.mean_met,␣
 ↪scoring='neg_mean_squared_error')
r2 = cross_val_score(model2, data_X, data.mean_met, scoring='r2')

print('Mean MSE: %.2f +/- %.2f' % (mse.mean(), mse.std()))
print('Mean R2: %.2f +/- %.2f' % (r2.mean(), r2.std()))
```

```
Mean MSE: -3.49 +/- 0.83
Mean R2: 0.33 +/- 0.14
```

### 5.0.3 Results test set

[20]:
```
test_pred_y = model.predict(test_X)

plot_results_bar(test_pred_y, test_y)

print('MSE: %.2f' % mean_squared_error(test_y, test_pred_y))
print('R2: %.2f' % r2_score(test_y, test_pred_y))
```




```
MSE: 4.95
R2: 0.17
```

# 6 Conclusion

The Linear Regression model does not score well and is probably underfitting because the model is too simplistic for this dataset.

```
[ ]:
```