LetsFaceIt_S06Ex1

December 14, 2021

Solution for Sheet 06 - Exercise 1 (sligtly adapted based on Hani's comments)

Contributors:

- Annika Bätz: annika.baetz@uni-potsdam.de
- Max Serra: maximilia-manuel.serra.lasierra@uni-potsdam.de
- Adrián de Miguel: adrian.de.miguel.palacio@uni-potsdam.de
- Ignacio Llorca: llorcarodriguez@uni-potsdam.de

[ ]:
```
import numpy as np
import pandas as pd
from numpy.linalg import inv
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
plt.style.use('seaborn')
```

0.0.1 Load data

[ ]:
```
X_potential_drivers = np.array(pd.read_csv('X_potential_drivers.txt',␣
↪header=None))
Y_snow = np.array(pd.read_csv('Y_snow.txt', header=None))
```

1 1 Preprocessing

Target analysis (Y_{snow}):

[ ]:
```
df_Y_snow = pd.DataFrame(Y_snow)

print("- First 5 target values:")
print(df_Y_snow.head())
print("\n- Type:")
print(df_Y_snow.dtypes)
print("\n- Main statistics:")
print(df_Y_snow.describe())
```

```
- First 5 target values:
        0
0  44.747
1  43.015
```

```
2  43.397
3  41.650
4  41.919

- Type:
0    float64
dtype: object

- Main statistics:
                0
count  201.000000
mean   -13.905716
std     33.601792
min    -70.328000
25%    -44.056000
50%    -13.604000
75%     16.012000
max     44.747000
```

Features analysis (x_1, x_2, x_3, x_4, x_5):

[ ]:
```
feature_names = ['1.feature','2.feature','3.feature','4.feature','5.feature']
df_X_potential_drivers = pd.read_csv('X_potential_drivers.txt',␣
 ↪names=feature_names)

print("- First 5 realisations:")
print(df_X_potential_drivers.head())
print("\n- Type:")
print(df_X_potential_drivers.dtypes)
print("\n- Main statistics:")
print(df_X_potential_drivers.describe())
```

```
- First 5 realisations:
   1.feature  2.feature  3.feature       4.feature  5.feature
0     11.728   -0.64836      -5.00  2.3026+3.1416i      -23.0
1     11.771   -0.51455      -4.95  2.2925+3.1416i      -22.7
2     11.817   -0.39388      -4.90  2.2824+3.1416i      -22.4
3     11.864   -0.28239      -4.85  2.2721+3.1416i      -22.1
4     11.913   -0.17704      -4.80  2.2618+3.1416i      -21.8

- Type:
1.feature    float64
2.feature    float64
3.feature    float64
4.feature     object
5.feature    float64
dtype: object

- Main statistics:
```

```
        1.feature     2.feature   3.feature   5.feature
count  201.000000  2.010000e+02  201.000000  201.000000
mean    12.000000 -7.070077e-17    0.000000    7.000000
std      0.345947  9.658693e+00    2.908393   17.450358
min     11.500000 -8.071300e+01   -5.000000  -23.000000
25%     11.669000 -8.855600e-01   -2.500000   -8.000000
50%     12.000000  0.000000e+00    0.000000    7.000000
75%     12.331000  8.855600e-01    2.500000   22.000000
max     12.500000  8.071300e+01    5.000000   37.000000
```

1.0.1 Deal with complex values

There might be many different alternatives to the one we have followed below to deal with complex values, for instance using Complex Matrices. This approach has also been covered in this topic and whose summary can be found in the section Annex: Complex Matrices

Our idea here: Split the fourth feature into a real and imaginary part. Additionally, we can compute the absolute value of the complex value and can use it as a third additional feature.

[ ]:
```
# lists to save our extraxted features:
complex_feature=[] # original values
real_feature=[]    # real part
imag_feature=[]    # imaginary part
abs_feature=[]     # absolute value

for s in X_potential_drivers[:,3]:
    s = s.replace('i', 'j')
    c = complex(s)
    complex_feature.append(c)
    real_feature.append(np.real(c))
    imag_feature.append(np.imag(c))
    abs_feature.append(np.abs(c))
```

[ ]:
```
print('complex feature:')
print(complex_feature[:5])
print('...')
print(complex_feature[-5:])
```

```
complex feature:
[(2.3026+3.1416j), (2.2925+3.1416j), (2.2824+3.1416j), (2.2721+3.1416j),
(2.2618+3.1416j)]
...
[(2.2618+0j), (2.2721+0j), (2.2824+0j), (2.2925+0j), (2.3026+0j)]
```

[ ]:
```
print('real feature:')
print(real_feature[:5])
print('...')
print(real_feature[-5:])
```

```
real feature:
[2.3026, 2.2925, 2.2824, 2.2721, 2.2618]
...
[2.2618, 2.2721, 2.2824, 2.2925, 2.3026]
```

[ ]:
```
print('imaginary feature:')
print(imag_feature[:5])
print('...')
print(imag_feature[-5:])
```

```
imaginary feature:
[3.1416, 3.1416, 3.1416, 3.1416, 3.1416]
...
[0.0, 0.0, 0.0, 0.0, 0.0]
```

[ ]:
```
print('absolute feature:')
print(abs_feature[:5])
print('...')
print(abs_feature[-5:])
```

```
absolute feature:
[3.8950760352013667, 3.8891138849357447, 3.883168850307697, 3.877123801221725,
3.8710967179857443]
...
[2.2618, 2.2721, 2.2824, 2.2925, 2.3026]
```

[ ]:
```
np.unique(imag_feature)
```

[ ]:
```
array([0.    , 3.1416])
```

Because the imaginary feature only takes on two unique values, i.e. it is a binary feature, we could also reformulate it as: * 0: no imaginary part * 1: imaginary part (instead of 3.1416) But we will leave it with the original implementation for now.

1.0.2 Dealing with infinity

We have one instance were the value of the original 4.feature is -inf.

[ ]:
```
np.unique(real_feature)
```

[ ]:
```
array([    -inf, -2.3026 , -1.6094 , -1.204  , -0.91629, -0.69315,
       -0.51083, -0.35667, -0.22314, -0.10536,  0.     ,  0.09531,
        0.18232,  0.26236,  0.33647,  0.40547,  0.47   ,  0.53063,
        0.58779,  0.64185,  0.69315,  0.74194,  0.78846,  0.83291,
        0.87547,  0.91629,  0.95551,  0.99325,  1.0296 ,  1.0647 ,
        1.0986 ,  1.1314 ,  1.1632 ,  1.1939 ,  1.2238 ,  1.2528 ,
        1.2809 ,  1.3083 ,  1.335  ,  1.361  ,  1.3863 ,  1.411  ,
        1.4351 ,  1.4586 ,  1.4816 ,  1.5041 ,  1.5261 ,  1.5476 ,
        1.5686 ,  1.5892 ,  1.6094 ,  1.6292 ,  1.6487 ,  1.6677 ,
        1.6864 ,  1.7047 ,  1.7228 ,  1.7405 ,  1.7579 ,  1.775  ,
```

```
1.7918 ,  1.8083 ,  1.8245 ,  1.8405 ,  1.8563 ,  1.8718 ,
1.8871 ,  1.9021 ,  1.9169 ,  1.9315 ,  1.9459 ,  1.9601 ,
1.9741 ,  1.9879 ,  2.0015 ,  2.0149 ,  2.0281 ,  2.0412 ,
2.0541 ,  2.0669 ,  2.0794 ,  2.0919 ,  2.1041 ,  2.1163 ,
2.1282 ,  2.1401 ,  2.1518 ,  2.1633 ,  2.1748 ,  2.1861 ,
2.1972 ,  2.2083 ,  2.2192 ,  2.23   ,  2.2407 ,  2.2513 ,
2.2618 ,  2.2721 ,  2.2824 ,  2.2925 ,  2.3026 ])
```

[ ]:
```
instance_to_drop = np.where(np.isinf(real_feature))[0][0]
print(instance_to_drop)
```

100

Because we have no information given about these features we cannot infer what -inf represents. If this was clear, we could find an appropiate solution to deal with it, e.g. imputation. Moreover, there is just one instance with -inf value for this feature, thus dropping it from the dataset makes more sense since it will not have a significative impact on our final results.

Nevertheless, we have also tried mean imputation to handle this value during our experiments with the Complex Matrices' approach. Results and comparisons can be found in Annex: Complex Matrices.

1.0.3 Replace fourth feature by new features and delete instance with infinity

[ ]:
```
# delete original feature column
X = np.delete(X_potential_drivers, 3, 1)
# add new feature columns
n = len(Y_snow)
X = np.hstack((X[:,[0,1,2]],
               np.array(real_feature).reshape(n,1),
               np.array(imag_feature).reshape(n,1),
               np.array(abs_feature).reshape(n,1),
               X[:,3].reshape(n,1)))

# drop the feature with -inf
X = np.delete(X,instance_to_drop,axis=0)
X = X.astype('float64')

X[:3]
```

[ ]:
```
array([[ 11.728     ,  -0.64836   ,  -5.        ,   2.3026    ,
           3.1416    ,   3.89507604, -23.        ],
       [ 11.771     ,  -0.51455   ,  -4.95      ,   2.2925    ,
           3.1416    ,   3.88911388, -22.7       ],
       [ 11.817     ,  -0.39388   ,  -4.9       ,   2.2824    ,
           3.1416    ,   3.88316885, -22.4       ]])
```

[ ]:
```
# we also need to delete the instance in the target array
y = np.delete(Y_snow, instance_to_drop)
```

[ ]:
```
# number of instances
n = X.shape[0]
n
```

[ ]: 200

[ ]:
```
# number of features:
p = X.shape[1]
p
```

[ ]: 7

1.0.4 Data Inspection

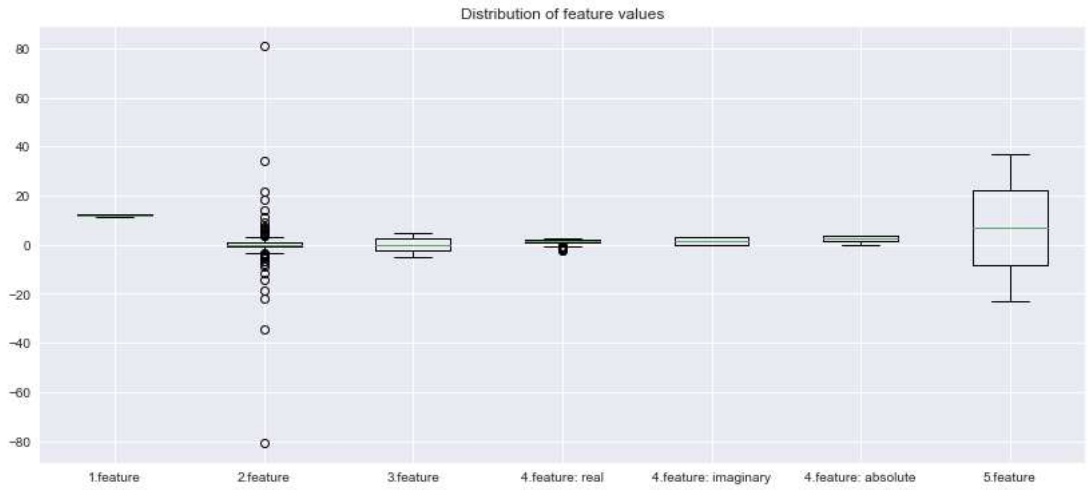
[ ]:
```
feature_names = ['1.feature','2.feature','3.feature','4.feature: real','4.
↪feature: imaginary','4.feature: absolute','5.feature']
```

[ ]:
```
df_X = pd.DataFrame(X, columns= feature_names)

print("- First 5 realisations:")
print(df_X.head())
print("\n- Type:")
print(df_X.dtypes)
print("\n- Main statistics:")
print(df_X.describe())
```

```
- First 5 realisations:
   1.feature  2.feature  3.feature  4.feature: real  4.feature: imaginary  \
0     11.728   -0.64836      -5.00           2.3026                3.1416
1     11.771   -0.51455      -4.95           2.2925                3.1416
2     11.817   -0.39388      -4.90           2.2824                3.1416
3     11.864   -0.28239      -4.85           2.2721                3.1416
4     11.913   -0.17704      -4.80           2.2618                3.1416

   4.feature: absolute  5.feature
0             3.895076      -23.0
1             3.889114      -22.7
2             3.883169      -22.4
3             3.877124      -22.1
4             3.871097      -21.8

- Type:
1.feature               float64
2.feature               float64
3.feature               float64
```

```
4.feature: real         float64
4.feature: imaginary    float64
4.feature: absolute     float64
5.feature               float64
dtype: object

- Main statistics:
        1.feature     2.feature   3.feature  4.feature: real  \
count  200.000000  2.000000e+02  200.000000       200.000000
mean    12.000000 -7.105427e-17    0.000000         1.334810
std      0.346815  9.682931e+00    2.915691         0.925720
min     11.500000 -8.071300e+01   -5.000000        -2.302600
25%     11.667250 -8.931725e-01   -2.512500         0.945705
50%     12.000000  0.000000e+00    0.000000         1.619300
75%     12.332750  8.931725e-01    2.512500         2.018200
max     12.500000  8.071300e+01    5.000000         2.302600

       4.feature: imaginary  4.feature: absolute   5.feature
count            200.000000           200.000000  200.000000
mean               1.570800             2.510633    7.000000
std                1.574742             1.127963   17.494148
min                0.000000             0.000000  -23.000000
25%                0.000000             1.643825   -8.075000
50%                1.570800             2.722100    7.000000
75%                3.141600             3.541173   22.075000
max                3.141600             3.895076   37.000000
```

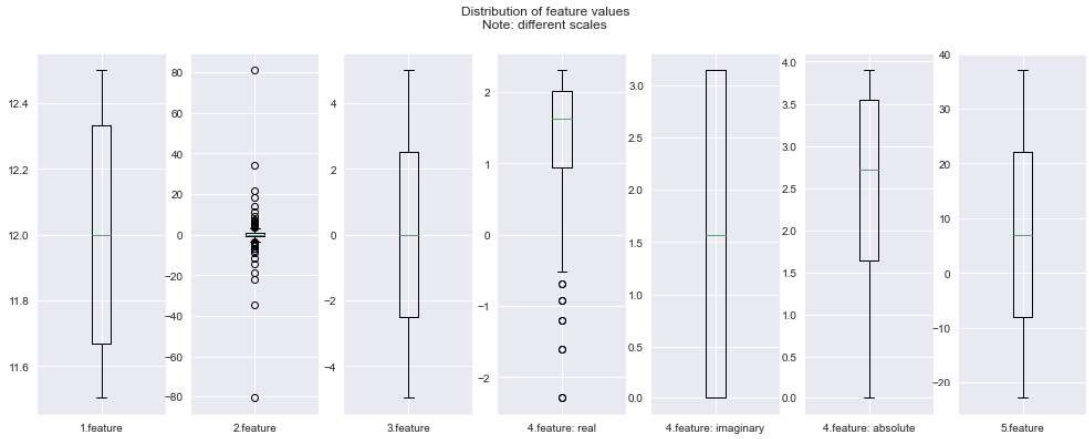
[ ]:
```
fig, ax = plt.subplots(figsize=(14,6))
ax.boxplot(X)
ax.set_title('Distribution of feature values')
ax.set_xticklabels(feature_names)
plt.show()
```



In order to get a better impression of the feature values we create individual boxplots for each feature using **optimal yaxis scales for each feature**.

Note: In the above plot, we have drawn all the features together to compare their distributions using the same scale value for the y-axis, so we can have a general idea about the differences between their value ranges. On the other hand, the next plot just wants to assess the singularities of each distribution individually (i.e. kind of distribution, skeweness, mean,...). Thus, we need a suitable scale value for each of them in order to illustrate these characteristics properly.

[ ]:
```
fig, ax = plt.subplots(1,p,figsize=(17,6))
for i in range(p):
    ax[i].boxplot(X[:,i])
    name = str(i+1)+'.feature'
    ax[i].set_xticklabels([feature_names[i]])
plt.suptitle('Distribution of feature values\nNote: different scales')
plt.show()
```



As you can see, each feature has a quite unique distribution. The first feature has a mean of approximately 12, the third feature of approximately 0. Some features have values in a quite small range, e.g. the 4.feature, while others have a huge range, e.g. the 5.feature.

In the following we use the following schema: * triangles: original 4.feature has an imaginary part * dots: original 4.feature has no imaginary part

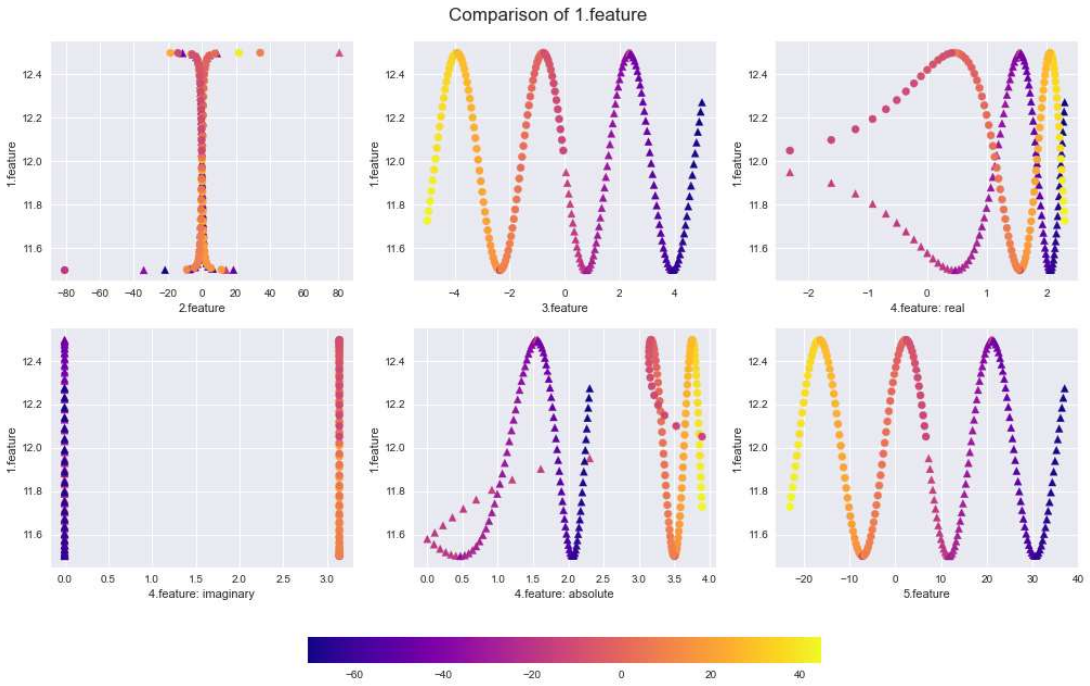
and the color of the traingles/dots represents the y value

[ ]:
```
# resource: https://stackoverflow.com/questions/55271471/
 ↪plotting-some-third-variable-against-x-and-y-in-matplotlib-scatter
import matplotlib.colors as colors
import matplotlib.cm as cmx
```

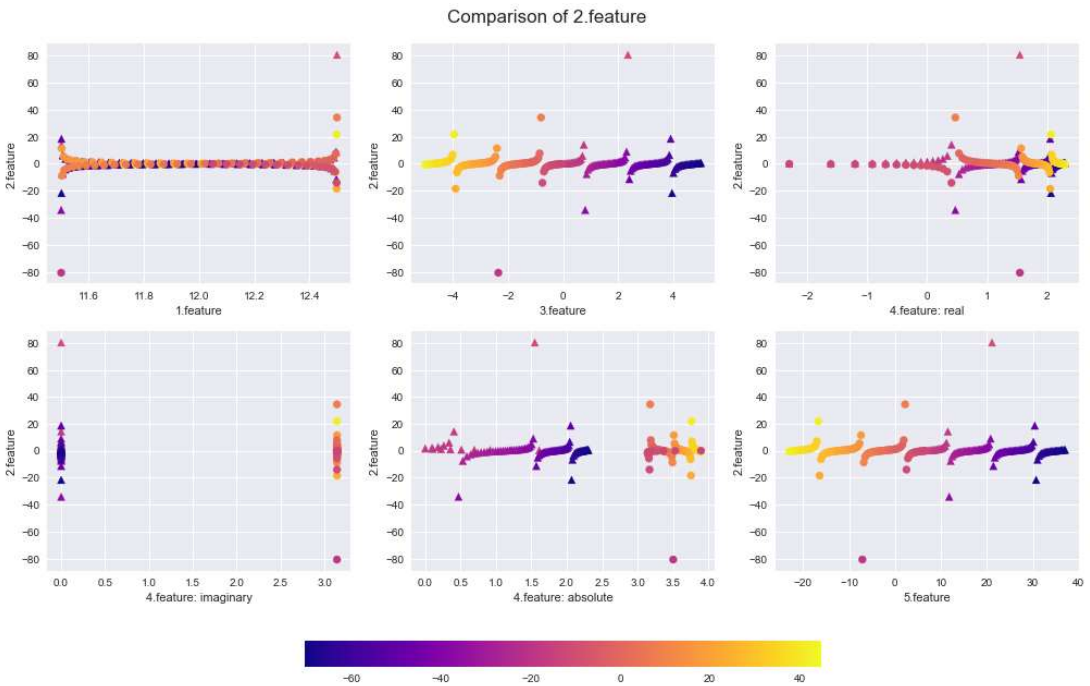
[ ]:
```
def compare_specific_feature(index_feature, title, feature_names):
    fig, axis = plt.subplots(2, 3, figsize=(17,11))
    cmap = plt.get_cmap('plasma')
    cNorm  = colors.Normalize(vmin=min(y), vmax=max(y))
    scalarMap = cmx.ScalarMappable(norm=cNorm, cmap=cmap)
    fig.subplots_adjust(top=0.94)
    f = 0

    for i,ax in enumerate(axis.flatten()):
        index_to_compare = i
        if i>=index_feature:
            index_to_compare = i+1
        else:
            index_to_compare = i
        ax.scatter(X[100:,index_to_compare],X[100:,index_feature], c=scalarMap.
 ↪to_rgba(y[100:]), marker='^', cmap=cmx.plasma)
        ax.scatter(X[:100,index_to_compare],X[:100,index_feature], c=scalarMap.
 ↪to_rgba(y[:100]), marker='o', cmap=cmx.plasma)
        ax.set_xlabel(feature_names[index_to_compare])
        ax.set_ylabel(feature_names[index_feature])

    plt.suptitle(title, fontsize=17, y=0.98)
    fig.colorbar(scalarMap,ax=axis[:, :],location='bottom', shrink=0.5,pad=0.1)
    plt.show()
```

[ ]:
```
feature_names = ['1.feature','2.feature','3.feature','4.feature: real','4.
 ↪feature: imaginary','4.feature: absolute','5.feature']
```

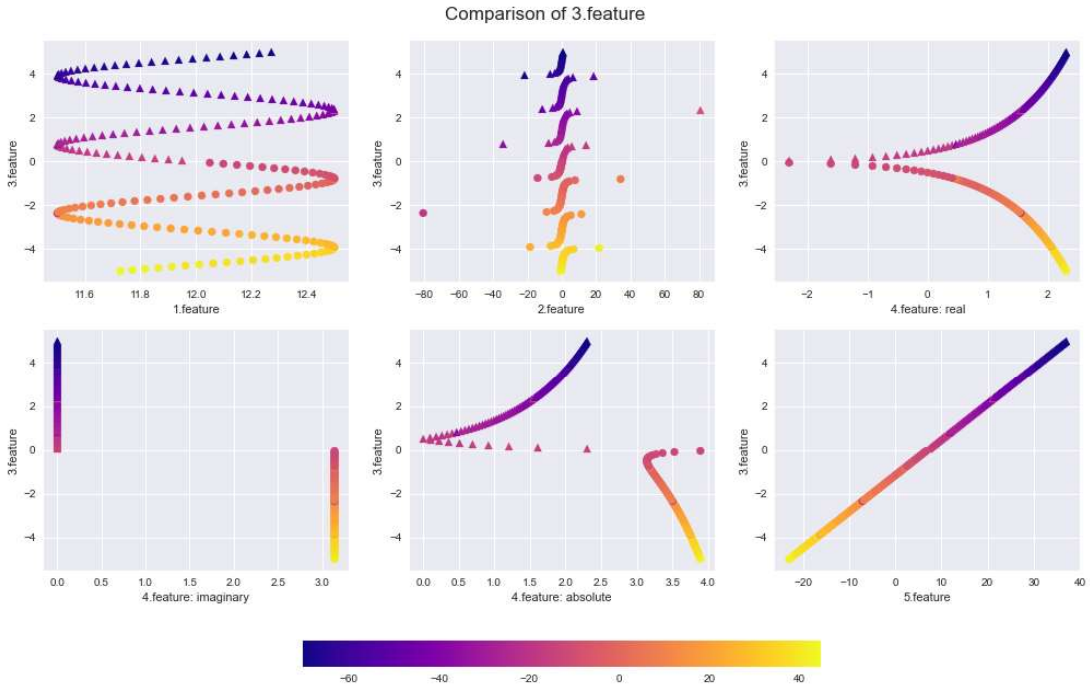
[ ]:
```
compare_specific_feature(index_feature=0, title='Comparison of 1.feature',␣
 ↪feature_names=feature_names)
```



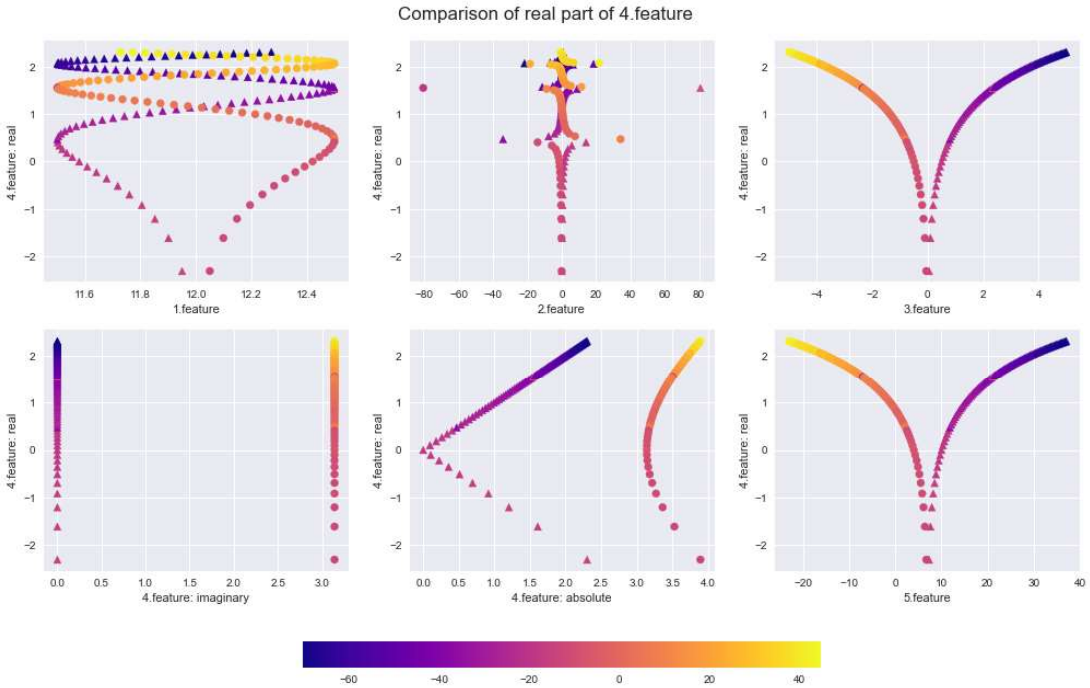
[ ]:
```
compare_specific_feature(index_feature=1, title='Comparison of 2.feature',␣
↪feature_names=feature_names)
```



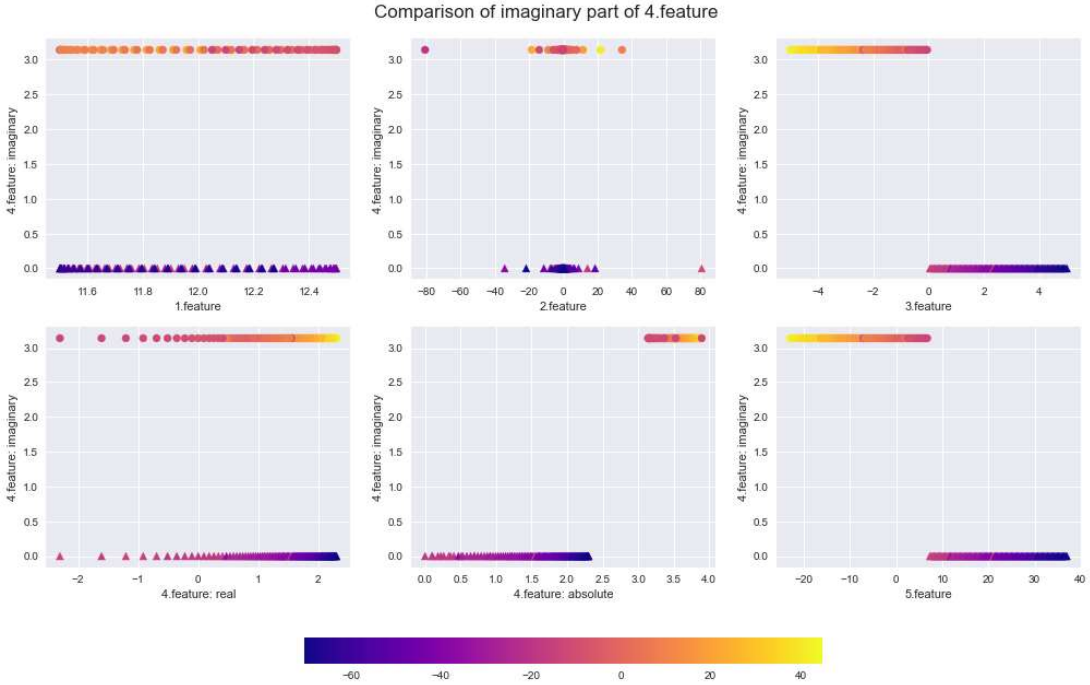
[ ]:
```
compare_specific_feature(index_feature=2, title='Comparison of 3.feature',␣
↪feature_names=feature_names)
```



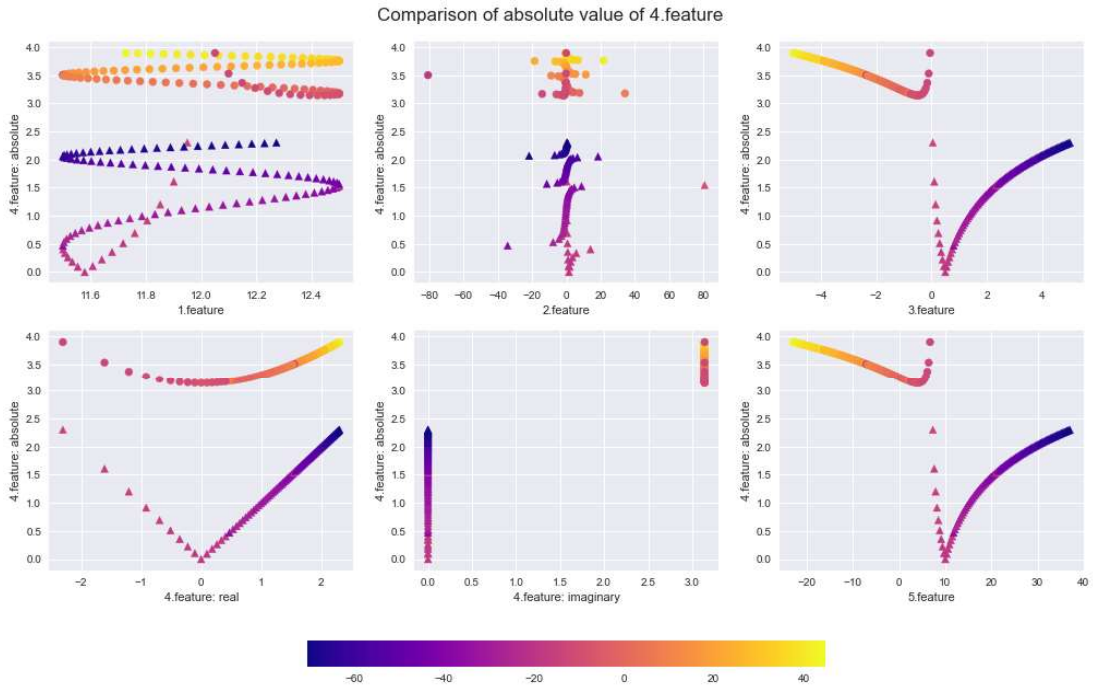
[ ]:
```
compare_specific_feature(index_feature=3, title='Comparison of real part of 4.
↪feature', feature_names=feature_names)
```



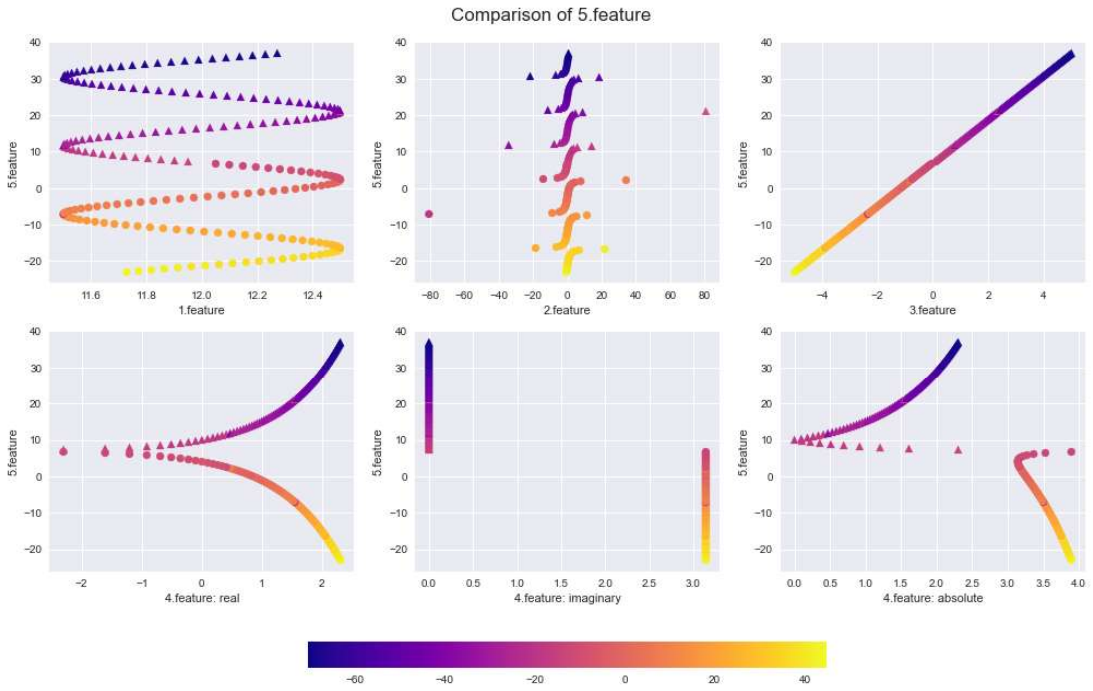
[ ]:
```
compare_specific_feature(index_feature=4, title='Comparison of imaginary part of␣
↪4.feature', feature_names=feature_names)
```



[ ]:
```
compare_specific_feature(index_feature=5, title='Comparison of absolute value of␣
↪4.feature', feature_names=feature_names)
```



[ ]:
```
compare_specific_feature(index_feature=6, title='Comparison of 5.feature',␣
↪feature_names=feature_names)
```



One important observation can be done in these plots: The 3. and the 5.feature seem correlated. Let's compute the Pearson correlation coefficient:

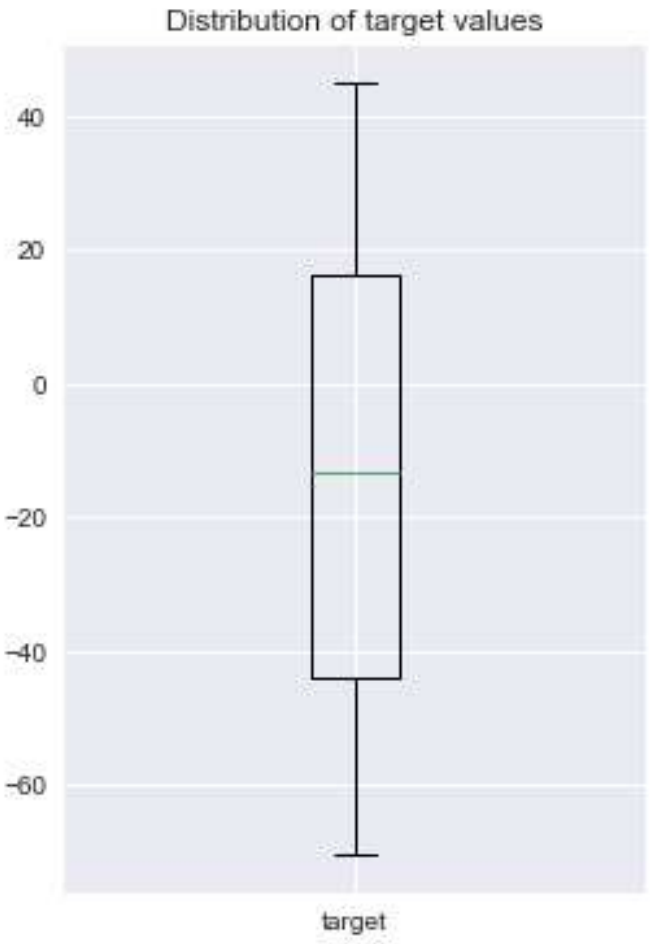
[ ]:
```
from scipy.stats import pearsonr
corr, _ = pearsonr(X[:,2],X[:,6])
print('Pearsons correlation: %.3f' % corr)
```

```
Pearsons correlation: 1.000
```

1 is the highest possible Pearsons correlation score and indicates that the 3. and the 7.feature are perfectely linear correlated.

Target

[ ]:
```
fig, ax = plt.subplots(figsize=(4,6))
ax.boxplot(y)
ax.set_title('Distribution of target values')
ax.set_xticklabels(['target'])
plt.show()
```



1.0.5 Should we extend the feature matrix for an additional column of 1's to allow a bias?

[ ]:
```
p = X.shape[1]
r = np.linalg.matrix_rank(X)
print('Number of features:', n)
print('Column rank of matrix:', r)
```

```
Number of features: 200
Column rank of matrix: 7
```

[ ]:
```
X_extended = np.hstack((np.ones((n,1)),X))
r_extended = np.linalg.matrix_rank(X_extended)
print('Number of columns in matrix:', p+1)
print('Column rank of matrix:', r_extended)
```

```
Number of columns in matrix: 8
Column rank of matrix: 7
```

If X is not of full column rank, $X^T X$ will also not be of full column rank, i.e. be a singular matrix. This means, it is not invertible (Using the python function linalg.inv(.) we will probably be able to compute an output, but it is questionable how valuable this is.) This is probably due to the correlation of the 3. and the 5.feature. In the following we will thus use two types of matrixes. See **3.2 Second approach: Select features based on corrected coefficient of determination** for more details on this.

2 2 Prepare functions for computation and evaluation of model

2.0.1 Define the function that can compute the LS-estimator

[ ]:
```
def beta_hat(X,y):
    return inv(X.T@X)@X.T@y
```

2.0.2 Coefficient of Determination

[ ]:
```
def r_sq(true, pred):
    mean_true = np.mean(true)
    nominator = np.sum(np.power(true-pred,2))
    denominator = np.sum(np.power(true-mean_true,2))
    return 1 - (nominator/denominator)
```

2.0.3 The corrected coefficient of determination

Motivation for corrected/adjusted coefficient of determination:

In contrast to the regular coefficient of determination this statistics takes the number of features p into account. The coefficient of determination has the characteristics that increasing the number of features will **never** decrease it, it can only increase. Since we want to compare different feature combinations we don't want that the number of features itself to affect our evaluation. The corrected/adjusted coefficient of determination is supposed to provide exactly this.

[ ]:
```
def corr_r_sq(true, pred, p):
    n = len(true)
    return 1-((n-1)/(n-p-1))*(1-r_sq(true, pred))
```

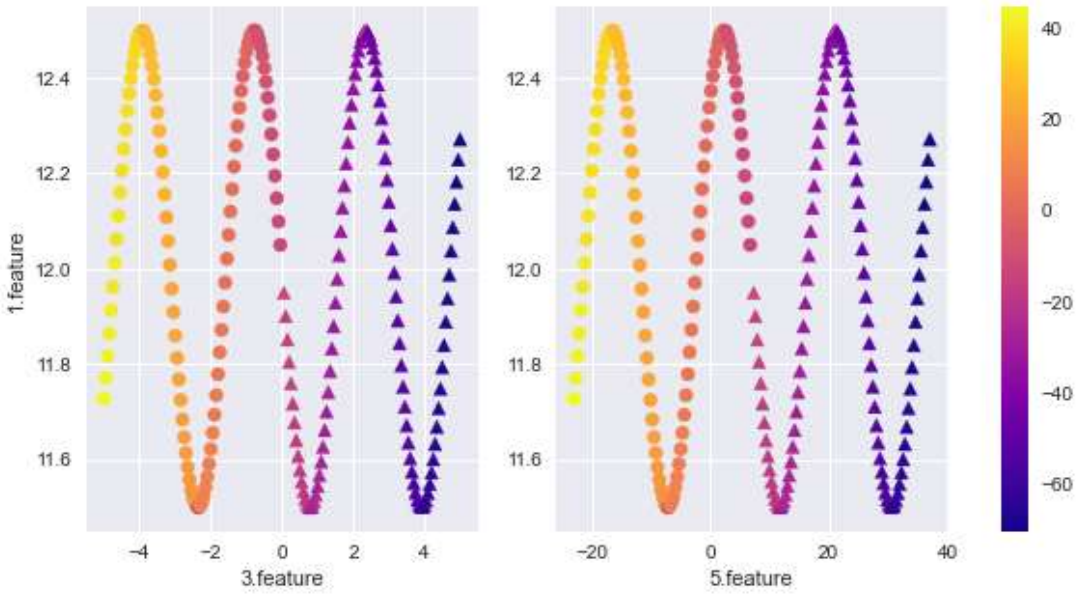
3 3 Feature Selection

3.1 3.1 First approach: Select features based on observations in plots

During the data analysis we noticed dependencies of our labels y to the 3. and 5.feature:

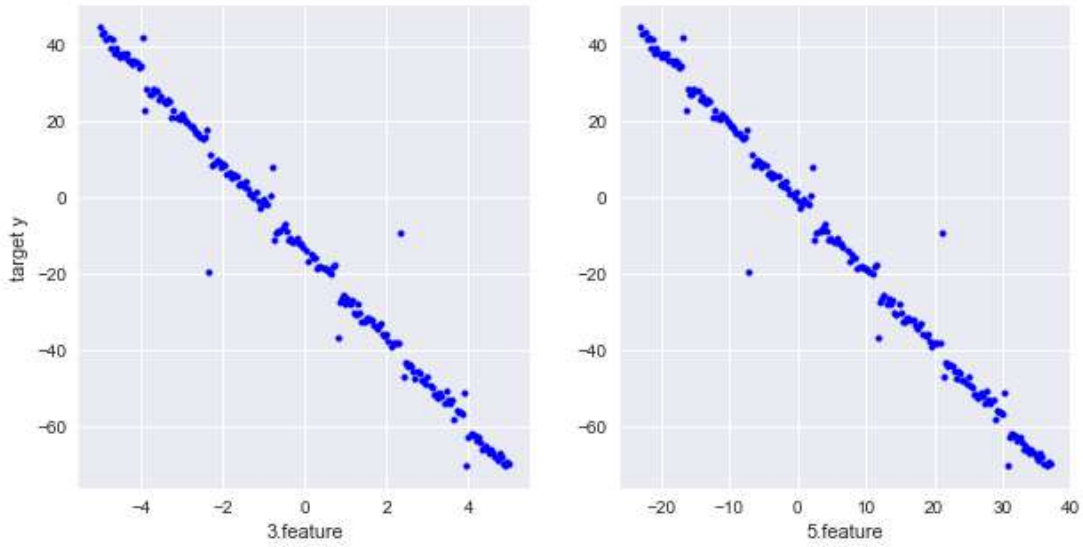
[ ]:
```
fig, ax = plt.subplots(1, 2, figsize=(10,5))
cmap = plt.get_cmap('plasma')
cNorm  = colors.Normalize(vmin=min(y), vmax=max(y))
scalarMap = cmx.ScalarMappable(norm=cNorm, cmap=cmap)
ax[0].scatter(X[100:,2],X[100:,0], c=scalarMap.to_rgba(y[100:]), marker='^',␣
↪cmap=cmx.plasma)
```

```
ax[0].scatter(X[:100,2],X[:100,0], c=scalarMap.to_rgba(y[:100]), marker='o',␣
 ↪cmap=cmx.plasma)
ax[0].set_xlabel('3.feature')
ax[0].set_ylabel('1.feature')
ax[1].scatter(X[100:,6],X[100:,0], c=scalarMap.to_rgba(y[100:]), marker='^',␣
 ↪cmap=cmx.plasma)
ax[1].scatter(X[:100,6],X[:100,0], c=scalarMap.to_rgba(y[:100]), marker='o',␣
 ↪cmap=cmx.plasma)
ax[1].set_xlabel('5.feature')
#ax[1].set_ylabel('1.feature')
fig.colorbar(scalarMap,ax=ax[:])
plt.show()
```



These two plots, that already appeared in our data analysis part, show that the target (represented by the color) can be described pretty well both by the 3. and the 5.feature. In the following two plots this relationship gets even more obvious:

[ ]:
```
fig, ax = plt.subplots(1, 2, figsize=(10,5))
ax[0].scatter(X[:,2],y, s=12, facecolors='blue')
ax[0].set_xlabel('3.feature')
ax[0].set_ylabel('target y')
ax[1].scatter(X[:,6],y, s=12, facecolors='blue')
ax[1].set_xlabel('5.feature')
plt.show()
```



Based on this observation, it seems realistic that we can already get a good model just using the 3. or the 5.feature. Here we can include a bias.

[ ]:
```
# Check the rank of the matrix we will use for our first model:
np.linalg.matrix_rank(X_extended[:,[0,3]]) == 2
```

[ ]:
```
True
```

[ ]:
```
# Check the rank of the matrix we will use for our second model:
np.linalg.matrix_rank(X_extended[:,[0,7]]) == 2
```

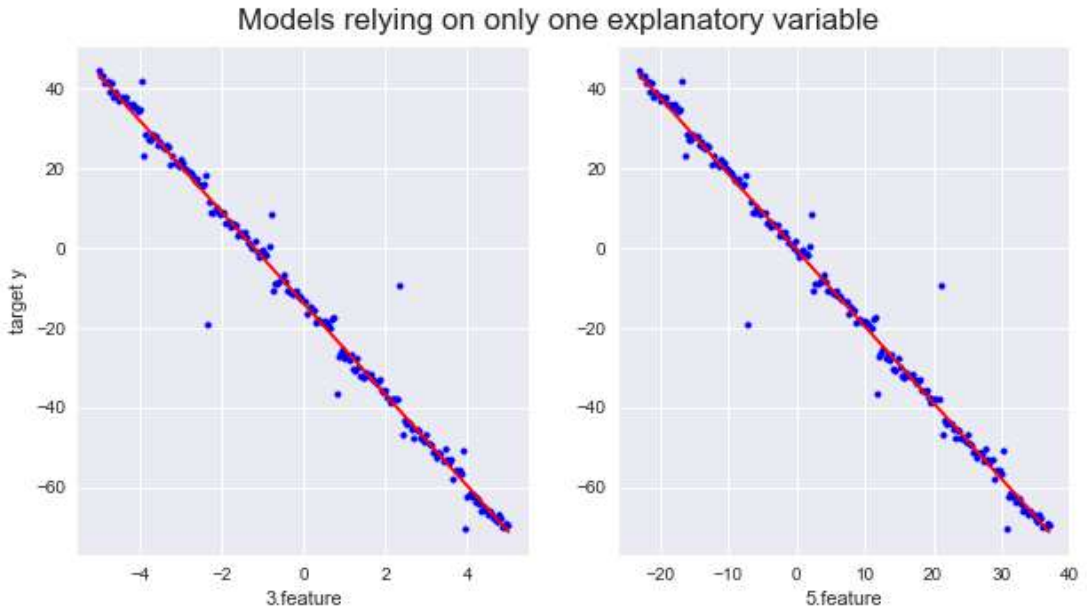
[ ]:
```
True
```

[ ]:
```
beta1 = beta_hat(X_extended[:,[0,3]],y)
pred1 = X_extended[:,[0,3]]@beta1
r_sq1 = r_sq(y, pred1)
corr_r_sq1 = corr_r_sq(y, pred1, 2)
print('Results of model using 3.feature and bias:')
print('- coefficient of determination: {:.6f}'.format(r_sq1))
print('- corrected coefficient of determination: {:.6f}'.format(corr_r_sq1))

beta2 = beta_hat(X_extended[:,[0,7]],y)
pred2 = X_extended[:,[0,7]]@beta2
r_sq2 = r_sq(y, pred2)
corr_r_sq2 = corr_r_sq(y, pred2, 2)
print('\nResults of model using 5.feature and bias:')
print('- coefficient of determination: {:.6f}'.format(r_sq2))
print('- corrected coefficient of determination: {:.6f}'.format(corr_r_sq2))
```

```
Results of model using 3.feature and bias:
- coefficient of determination: 0.985808
- corrected coefficient of determination: 0.985664

Results of model using 5.feature and bias:
- coefficient of determination: 0.985808
- corrected coefficient of determination: 0.985664
```

[ ]:
```
x_plot_1model = np.linspace(X[:,2].min(),X[:,2].max(),5)
x_plot_2model = np.linspace(X[:,6].min(),X[:,6].max(),5)
```

[ ]:
```
fig, ax = plt.subplots(1, 2, figsize=(10,5))
fig.subplots_adjust(top=0.92)
ax[0].scatter(X[:,2],y, s=12, facecolors='blue')
ax[0].set_xlabel('3.feature')
ax[0].set_ylabel('target y')
ax[0].plot(x_plot_1model,beta1[0] + beta1[1]*x_plot_1model, color='red')
ax[1].scatter(X[:,6],y, s=12, facecolors='blue')
ax[1].set_xlabel('5.feature')
ax[1].plot(x_plot_2model,beta2[0] + beta2[1]*x_plot_2model, color='red')
plt.suptitle('Models relying on only one explanatory variable', fontsize=17, y=0.
↪98)
plt.show()
```



Both our models performed pretty good and achieved a high (corrected) coefficient of determination. Since the 3. and 5.feature are correlated, their scores are equal and no model is better than the other.

In the following we will try to further improve our model.

3.2 3.2 Second approach: Select features based on corrected coefficient of determination

Note: From now on we will use the expressions "1.,2., ..., 7.feature" to refer to the features. Otherwise we will explicitely use the term "original feature(s)".

example: * original 3.feature -> 3.feature * original 5.feature -> 7.feature

In the following we have used the corrected coefficient of determination to determine which features to use to build a good model. This is subdivided into two parts: 1. part: no bias, using all features 2. part: bias included, leaving out the 3.feature to ensure full column rank

Looking back on this work, the 2.part is the really important one. However, we also wanted to document here how our work progressed over time and we already made some useful observations durin the 1.part, so we decided to not drop it.

3.2.1 In the following we want to try each possible feature combination. Let's compute the powerset of our feature indexes:

[ ]:
```
# the following function is copied from:
# https://stackoverflow.com/questions/1482308/
 ↪how-to-get-all-subsets-of-a-set-powerset
def powerset(s):
    result = []
    x = len(s)
    masks = [1 << i for i in range(x)]
    for i in range(1, 1 << x):
        yield [ss for mask, ss in zip(masks, s) if i & mask]
```

[ ]:
```
feature_comb = list(powerset(range(p)))
print(feature_comb[:5])
print(feature_comb[-5:])
```

```
[[0], [1], [0, 1], [2], [0, 2]]
[[0, 1, 3, 4, 5, 6], [2, 3, 4, 5, 6], [0, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6],
[0, 1, 2, 3, 4, 5, 6]]
```

The list feature_comb describes our features using the index (starting at 0), that's useful for the computation part. However for interpretation it is useful to start the counting at 1 for the 1.feature and so on. Therefore we create a new list:

[ ]:
```
feature_comb_interp = []
for f in feature_comb:
    feature_comb_interp.append([i+1 for i in f])
```

[ ]:
```
print(feature_comb_interp[:5])
print(feature_comb_interp[-5:])
```

```
[[1], [2], [1, 2], [3], [1, 3]]
[[1, 2, 4, 5, 6, 7], [3, 4, 5, 6, 7], [1, 3, 4, 5, 6, 7], [2, 3, 4, 5, 6, 7],
[1, 2, 3, 4, 5, 6, 7]]
```

3.2.2 1. Part: No bias, using all features

[ ]:
```
r_sq_results = []
corr_r_sq_results = []
for f in feature_comb:
    beta = beta_hat(X[:,f],y)
    pred = X[:,f]@beta
    r_sq_results.append(r_sq(y, pred))
    corr_r_sq_results.append(corr_r_sq(y, pred, len(f)))
```

The list r_sq_results and corr_r_sq_results contain the coefficient of determination R^2 and the corrected coefficient of determination for every feature combination model. The models are trained on the training data and evaluated on the test data.

We combine the results in a data frame. For this we use the notation where we start the counting of the features at 1 in order to fit to the plots created at the beginning.

[ ]:
```
results = pd.DataFrame(data = np.vstack((np.array(r_sq_results), np.
↪array(corr_r_sq_results))).T,
                       index = [str(x) for x in feature_comb_interp],
                       columns=['R^2', 'corr R^2'])
```

Let's take a look at the feature combinations that performed the worst and at the feature combinations that performed the best.

1. Sorted according to the coefficient of determination
2. Sorted according to the corrected coefficient of determination

Worst feature combinations according to regular coefficient of determination:

[ ]:
```
results.sort_values('R^2').head(10)
```

[ ]:
```
             R^2  corr R^2
[2]    -0.165994 -0.171883
[6]    -0.163031 -0.168904
[2, 6] -0.158094 -0.169851
[5]    -0.071306 -0.076717
[2, 5] -0.064315 -0.075120
[4]    -0.055286 -0.060616
[2, 4] -0.050126 -0.060788
[1]    -0.005058 -0.010134
[1, 4] -0.004969 -0.015171
[1, 2]  0.000372 -0.009776
```

Worst feature combinations according to corrected coefficient of determination:

[ ]:
```
results.sort_values('corr R^2').head(10)
```

[ ]:

	R^2	corr R^2
[2]	-0.165994	-0.171883
[2, 6]	-0.158094	-0.169851
[6]	-0.163031	-0.168904
[5]	-0.071306	-0.076717
[2, 5]	-0.064315	-0.075120
[2, 4]	-0.050126	-0.060788
[4]	-0.055286	-0.060616
[1, 4]	-0.004969	-0.015171
[1, 2, 4]	0.000452	-0.014847
[1]	-0.005058	-0.010134

Best feature combinations according to regular coefficient of determination:

[ ]:
```
results.sort_values('R^2').tail(10)
```

Best feature combinations according to corrected coefficient of determination:

[ ]:
```
results.sort_values('corr R^2').tail(10)
```

[ ]:

	R^2	corr R^2
[1, 2, 6, 7]	0.999187	0.999170
[2, 3, 4, 7]	0.999188	0.999171
[1, 2, 3, 5, 7]	0.999192	0.999171
[1, 2, 3, 4, 7]	0.999192	0.999171
[2, 3, 5, 7]	0.999188	0.999171
[1, 2, 3, 6, 7]	0.999192	0.999172
[2, 3, 6, 7]	0.999189	0.999172
[1, 2, 7]	0.999186	0.999174
[2, 3, 7]	0.999188	0.999175
[1, 2, 3, 7]	0.999192	0.999175

[ ]:
```
np.sum(results['R^2']>=0.9)
```

[ ]: 94

[ ]:
```
np.sum(results['corr R^2']>=0.9)
```

[ ]: 94

Which feature combinations got a corrected coefficient of determination of at least 0.999 ?

[ ]:
```
results[results['corr R^2']>=0.999].sort_values('corr R^2')
```

[ ]:

	R^2	corr R^2
[2, 7]	0.999034	0.999024
[2, 4, 7]	0.999136	0.999123

[2, 4, 5, 6, 7]	0.999169	0.999148
[2, 5, 7]	0.999163	0.999150
[2, 5, 6, 7]	0.999168	0.999151
[2, 4, 6, 7]	0.999168	0.999151
[2, 4, 5, 7]	0.999169	0.999152
[2, 6, 7]	0.999168	0.999155
[1, 2, 4, 5, 6, 7]	0.999190	0.999165
[1, 2, 4, 5, 7]	0.999186	0.999165
[1, 2, 3, 4, 5, 6, 7]	0.999195	0.999166
[1, 2, 5, 6, 7]	0.999187	0.999167
[1, 2, 4, 6, 7]	0.999188	0.999167
[2, 3, 4, 5, 6, 7]	0.999192	0.999167
[1, 2, 3, 4, 5, 7]	0.999192	0.999167
[2, 3, 4, 5, 7]	0.999188	0.999167
[1, 2, 3, 4, 6, 7]	0.999193	0.999168
[1, 2, 3, 5, 6, 7]	0.999193	0.999168
[2, 3, 5, 6, 7]	0.999189	0.999169
[2, 3, 4, 6, 7]	0.999190	0.999169
[1, 2, 5, 7]	0.999186	0.999170
[1, 2, 4, 7]	0.999186	0.999170
[1, 2, 6, 7]	0.999187	0.999170
[2, 3, 4, 7]	0.999188	0.999171
[1, 2, 3, 5, 7]	0.999192	0.999171
[1, 2, 3, 4, 7]	0.999192	0.999171
[2, 3, 5, 7]	0.999188	0.999171
[1, 2, 3, 6, 7]	0.999192	0.999172
[2, 3, 6, 7]	0.999189	0.999172
[1, 2, 7]	0.999186	0.999174
[2, 3, 7]	0.999188	0.999175
[1, 2, 3, 7]	0.999192	0.999175

3.2.3 Discussion/Interpretation:

In order to simplify the explanations, we will sometimes use the word "metrics" if we refer to both the coefficient of determination and the adjusted coefficient of determination.

- As expected the models that used only one or two features didn't perform well.
- From 127 feature combinations 94 have a coefficient of determination and an adjusted coefficient of determination of at least 0.9. Note: the best possible score is 1. We can interpret this as: Many feature combinations already include a lot of valuable information to build a good model. It is not necessary to use all features.
- The top "worst" feature combinations are quite similar for both metrics.
- In the top "best" feature combinations the results look different for the two metrics. The coefficient of determination is the highest when we use all features. The adjusted coefficient of determination suggests to use either the feature [1, 2, 3, 7] or [2, 3, 7] (i.e. without 1).
- This different observations for the two metrics fit their characteristics described above: The coefficient of determination is affected by the number of features used. By adding new features, the score will never decrease. The adjusted coefficient of determination is not pe-

nalizing a more complex model (i.e. a model using more features) and is thus more suitable for this kind of model comparison.

- If we take a closer look at the top 10 list (based on the adjusted coefficient of determination), we can see that the 2. and 7.feature (the 7.feature is in the original dataset the 5.feature) appear in all feature combinations. We can see in the last table that those two features are the only pair that got a corrected coefficient of determination above 0.999. Thus, using those features is for sure a good choice.
- To further improve the model it might be worth to include at least one additional feature, e.g. feature 1 or 3. (The combinations [1, 2, 7] and [2, 3, 7] can be found quite low in the results tables).
- It is interesting to see that even if the 3.feature and the 7.feature (original 5.feature) are correlated, the 7.feature appears more often than the 3.feature. A reason might be that the 7.feature has a bigger range of values than the 3.feature.

3.2.4 2. Part: Including bias, excluding 3.feature

We have seen above that our matrix including the extra column of 1's is not of full column rank. This can be explained by the correlation of the 3. and 7. feature. Leaving out either one of them and including the bias we get a matrix with full column rank.

[ ]:
```
# This matrix doesn't have full column rank!
np.linalg.matrix_rank(X_extended[:,[0,3,7]])
```

[ ]: 2

[ ]:
```
# leave out the 3.feature -> full column rank.
np.linalg.matrix_rank(X_extended[:,[0,1,2,4,5,6,7]])
```

[ ]: 7

[ ]:
```
# leave out the 7.feature -> full column rank.
np.linalg.matrix_rank(X_extended[:,[0,1,2,3,4,5,6]])
```

[ ]: 7

Since we have seen above that the feature 7 was quite valuable for our models, we will now drop the 3.feature, add the column of 1's and repeat the evaluation done above:

[ ]:
```
X_new = X_extended[:,[0,1,2,4,5,6,7]]
```

Again, we create a new list that corresponds to feature_comb (index based) but which will allows for easier interpreation of the result table.

[ ]:
```
feature_comb_interp_new = []
for f in feature_comb:
    feature_comb_interp_new.append([i if i<3 else i+1 for i in f])
```

[ ]:
```
r_sq_results_new = []
corr_r_sq_results_new = []
for f in feature_comb:
    beta = beta_hat(X_new[:,f],y)
    pred = X_new[:,f]@beta
    r_sq_results_new.append(r_sq(y, pred))
    corr_r_sq_results_new.append(corr_r_sq(y, pred, len(f)))
```

[ ]:
```
results_new = pd.DataFrame(data = np.vstack((np.array(r_sq_results_new), np.
↪array(corr_r_sq_results_new))).T,
                        index = [str(x) for x in feature_comb_interp_new],
                        columns=['R^2', 'corr R^2'])
```

The table uses the same index as the first results table. 0 indicates that a bias was used, the digits from 1 to 7 indicate the feature (1. -> first feature).

[ ]:
```
results_new.sort_values('corr R^2').tail(10)
```

[ ]:

	R^2	corr R^2
[1, 2, 6, 7]	0.999187	0.999170
[0, 2, 4, 7]	0.999188	0.999171
[0, 1, 2, 5, 7]	0.999192	0.999171
[0, 1, 2, 4, 7]	0.999192	0.999171
[0, 2, 5, 7]	0.999188	0.999171
[0, 1, 2, 6, 7]	0.999192	0.999172
[0, 2, 6, 7]	0.999189	0.999172
[1, 2, 7]	0.999186	0.999174
[0, 2, 7]	0.999188	0.999175
[0, 1, 2, 7]	0.999192	0.999175

3.2.5 Discussion/Interpretation:

Similar to our first results the 2. and 7.feature appear in each feature combination in the top 10. Most interesting is probably the feature combination [0,2,7] (same score as [0,1,2,7]). This model is based on just the two features 2 and 7 plus a bias. To use it at application time we would need to collect only two values.

Because of the correlation of the 3. and the 7.feature, we can expect that replacing the 7. by the 3.feature (and remaining a bias!) should result in an equally good model:

[ ]:
```
beta = beta_hat(X_extended[:,[0,2,3]],y)
pred = X_extended[:,[0,2,3]]@beta
print('Model: 2. and 3. feature including bias:')
print(r_sq(y, pred))
print(corr_r_sq(y, pred, 3))

beta = beta_hat(X_extended[:,[0,1,2,3]],y)
pred = X_extended[:,[0,1,2,3]]@beta
```

```
print('\nModel: 1., 2. and 3. feature including bias:')
print(r_sq(y, pred))
print(corr_r_sq(y, pred, 4))
```

```
Model: 2. and 3. feature including bias:
0.9991878139546391
0.9991753825355775

Model: 1., 2. and 3. feature including bias:
0.999192060398189
0.9991754872781519
```

As expected we get the same results for the (adjusted) coefficient of determination if we do the replacement.

3.3 3.3 Third approach: Hypothesis Testing

In the following we will apply our test to the model trained on all features except for the 3.feature and we will use a bias term.

References:

http://reliawiki.org/index.php/Multiple_Linear_Regression_Analysis

https://online.stat.psu.edu/stat462/node/131/

Given our multiple linear regression model trained with all the available features (without the bias term) we might ask our selves: "Is the i-th feature contributing much to y? Or is it just correcting minor noise?"

To test it, we can use hypothesis testing on the individual factors $\hat{\beta}_i$ with the following test:

$$H_0 : \hat{\beta}_i = 0 H_A : \hat{\beta}_i \neq 0$$

The go-to test statisitc for this setup (one sample, normally distributed errors and "big" sample size of 200 points) is the **one-sample Student's t-test**. The history of this name is interesting and involves Guinness, so check it if you like beer.

The general formulation of the t-test statistic is:

$$t = \frac{\bar{X} - \mu}{\sigma / \sqrt{n}}$$

For our case we don't have samples of β_i, but only one single occurrence and it's variance. And we have $\mu = 0$, so we can reformulate the test statistic as follows:

$$t_i = \frac{\hat{\beta}_i}{se(\hat{\beta}_i)}$$

Where $se(\hat{\beta}_i)$ is the standard error of the factor $\hat{\beta}_i$. This is the square root of the i-th diagonal element of the covariance matrix C:

$$C = \hat{\sigma}(X^T X)^{-1}$$

Finally $\hat{\sigma}$ is the variance of the predicitons, $\hat{\sigma} = \frac{MSE}{(n-p)}$. Note that we divide by $(n-p)$ and not $(n-p-1)$, because our p is here defined as the number of columns in our input matrix (including the column of 1's) and not by the number of features.

With this, we can proceed to test which features are significant. We work with the standard significance level of $\alpha = 0.05$

Step 1: Define constants

[ ]:
```
n = X_new.shape[0]
p = X_new.shape[1] # <- p is not defined by the number of features, but by the␣
↪number of columns
# significance level
alpha = 0.05
```

Step 2: Train model

[ ]:
```
beta = beta_hat(X_new, y)
pred = X_new@beta
print(beta)
```

```
[-2.73818683  0.18198005  0.4016157  -0.06941017 -0.14394245  0.17572647
 -1.92418342]
```

Step 3: Estimate the variance and compute the variance-covariance matrix of the estimated regression coefficients

[ ]:
```
est_var = np.sum((y - pred)**2) / (n - p)
# variance-covariance matrix of the estimated regression coefficients
C = est_var*inv(X_new.T@X_new)
```

Step 4: Define t-test function for our scenario

[ ]:
```
from scipy.stats import t

def t_test(beta, se_beta, alpha, df) -> str:

    t_stat = beta / se_beta
    t_bound = t.ppf(1. - alpha, df=df)

    if abs(t_stat) > t_bound:
        print(f"""    t-test result:
                REJECT null hypothesis (mean 0) with t-value {t_stat}
        """)
        return "REJECT"
    else:
        print(f"""    t-test result:
        FAIL TO REJECT null hypothesis (mean 0) with t-value {t_stat}
        """)
        return "FAIL TO REJECT"
```

```
[ ]: for i in range(0,7):
         if i>=3:
             f = i+1
         else:
             f = i
         if i != 0:
             print('Test of ' + str(f) + '.feature:')
             t_test(beta[i], np.sqrt(C[i,i]), alpha, (n - p))
```

```
Test of 1.feature:
    t-test result:
        FAIL TO REJECT null hypothesis (mean 0) with t-value 0.8599102751778964

Test of 2.feature:
    t-test result:
                REJECT null hypothesis (mean 0) with t-value 55.53469904349027

Test of 4.feature:
    t-test result:
        FAIL TO REJECT null hypothesis (mean 0) with t-value -0.6712261591596251

Test of 5.feature:
    t-test result:
        FAIL TO REJECT null hypothesis (mean 0) with t-value -0.7227015207511889

Test of 6.feature:
    t-test result:
        FAIL TO REJECT null hypothesis (mean 0) with t-value 0.8202076221553543

Test of 7.feature:
    t-test result:
                REJECT null hypothesis (mean 0) with t-value -213.8776281826563
```

So we would only reject the $H_0 : \hat{\beta}_i = 0$ for feature 2 and 7. Therefore, the regression model should take into acount at least these features and might be able to omit the rest. (Note: the 3.feature and the bias were excluded from this analysis)

$$y = \hat{\beta_0} + \hat{\beta_2} X_2 + \hat{\beta_7} X_7$$

We can go one step further and look at the (absolute) value of the t-statistic. The meaning of it is: "the lower, the less significant the feature". So we can say that the 4.feature is the least significant, closely followed by the 5.feature.

4 4 Conclusion

We have seen in our first approach that both the 3. and the original 5.feature can already be used to build a good model. The second approach has showed that we should continue either with the 3. or the 5.feature and combine it with the 2.feature and add a bias. During hypothesis testing we have seen that the only regression coefficients for which we could reject the null hypothesis that these coefficients are zero were those of the 2. and the 5. feature. (3.feature was not included during test).

All these findings support our final conclusion that the best model would be based on the 2. and 5. feature (or alternatively the 3.feature) combined with bias. The respective corrected coefficient of correlation is 0.999175.

We would like to point out that the feature selection cannot be based solely on these results. What these features actually are should also be taken into account. For example assume that it is expensive or time intensive to measure the 2.feature. Then, it might better to just use the 3. or 5. feature like we have seen in our 1.approach and to tolerate a model that is slightly worse than the best possible one.

4.1 Annex: Complex Matrices

As commented at the very beginning of this notebook, there are other takes on how to deal with the complex feature. Worth of special mention is that in which all features are casted into complex format and the Least Squares estimation is performed on the Complex Matrix. See below a summary of the reasoning, the differences to simple Least Squares and the resulting coefficients of determination. To see the full implementation please refer to the individual submissions from Max Serra and Ignacio Llorca.

4.1.1 Reasoning

Let the module of the imaginary number $z = x + yi$ be $|z| = \sqrt{x^2 + y^2}$, and its principal argument $arg(z) = tan^{-1}(\frac{y}{x})$.

In our regression with complex values $Y_i = \beta_0 + \beta_1 X_{i,1} + \beta_2 X_{i,2} + \beta_3 X_{i,3} + \beta_4 X_{i,4} + \beta_5 X_{i,5}$ multiplication by β_p will rescale the independent variable $X_{i,p}$ by the modulus $|\beta_p|$ and rotate it around the origin by $arg(\beta_p)$.

The LS-estimator $\hat{\beta}$ that we want to find can be calculated as in the simple-number case with $\hat{\beta} = (X^T X)^{-1} X^T y$, only that the transpose operation X^T is now replaced with the conjugate transpose $X^* = \bar{X}^T$, where conjugating means inverting the sign of the imaginary part of the complex number, i.e: $\bar{z} = x - yi$. Thus, $\hat{\beta} = (X^* X)^{-1} X^* y$

4.1.2 Data preprocessing

After dealing with outliers and making necessary transformations to cast into complex format, the feature matrix looks like this:

	0	1	2	3	4
0	11.728000+0.000000j	-0.648360+0.000000j	-5.000000+0.000000j	2.302600+3.141600j	-23.000000+0.000000j
1	11.771000+0.000000j	-0.514550+0.000000j	-4.950000+0.000000j	2.292500+3.141600j	-22.700000+0.000000j
2	11.817000+0.000000j	-0.393880+0.000000j	-4.900000+0.000000j	2.282400+3.141600j	-22.400000+0.000000j
3	11.864000+0.000000j	-0.282390+0.000000j	-4.850000+0.000000j	2.272100+3.141600j	-22.100000+0.000000j
4	11.913000+0.000000j	-0.177040+0.000000j	-4.800000+0.000000j	2.261800+3.141600j	-21.800000+0.000000j
...	...	...	...	...	...
196	12.087000+0.000000j	0.177040+0.000000j	4.800000+0.000000j	2.261800+0.000000j	35.800000+0.000000j
197	12.136000+0.000000j	0.282390+0.000000j	4.850000+0.000000j	2.272100+0.000000j	36.100000+0.000000j
198	12.183000+0.000000j	0.393880+0.000000j	4.900000+0.000000j	2.282400+0.000000j	36.400000+0.000000j
199	12.229000+0.000000j	0.514550+0.000000j	4.950000+0.000000j	2.292500+0.000000j	36.700000+0.000000j
200	12.272000+0.000000j	0.648360+0.000000j	5.000000+0.000000j	2.302600+0.000000j	37.000000+0.000000j

201 rows × 5 columns

With conjugate transpose looking like:

	0	1	2	3	4	5	6	
0	11.728000-0.000000j	11.771000-0.000000j	11.817000-0.000000j	11.864000-0.000000j	11.913000-0.000000j	11.962000-0.000000j	12.012000-0.000000j	12.062(
1	-0.648360-0.000000j	-0.514550-0.000000j	-0.393880-0.000000j	-0.282390-0.000000j	-0.177040-0.000000j	-0.075364-0.000000j	0.024783-0.000000j	0.1254
2	-5.000000-0.000000j	-4.950000-0.000000j	-4.900000-0.000000j	-4.850000-0.000000j	-4.800000-0.000000j	-4.750000-0.000000j	-4.700000-0.000000j	-4.650(
3	2.302600-3.141600j	2.292500-3.141600j	2.282400-3.141600j	2.272100-3.141600j	2.261800-3.141600j	2.251300-3.141600j	2.240700-3.141600j	2.230(
4	-23.000000-0.000000j	-22.700000-0.000000j	-22.400000-0.000000j	-22.100000-0.000000j	-21.800000-0.000000j	-21.500000-0.000000j	-21.200000-0.000000j	-20.900(

5 rows × 201 columns

4.1.3 Least Squares Estimator

As explained above, the function to obtain β now looks like:

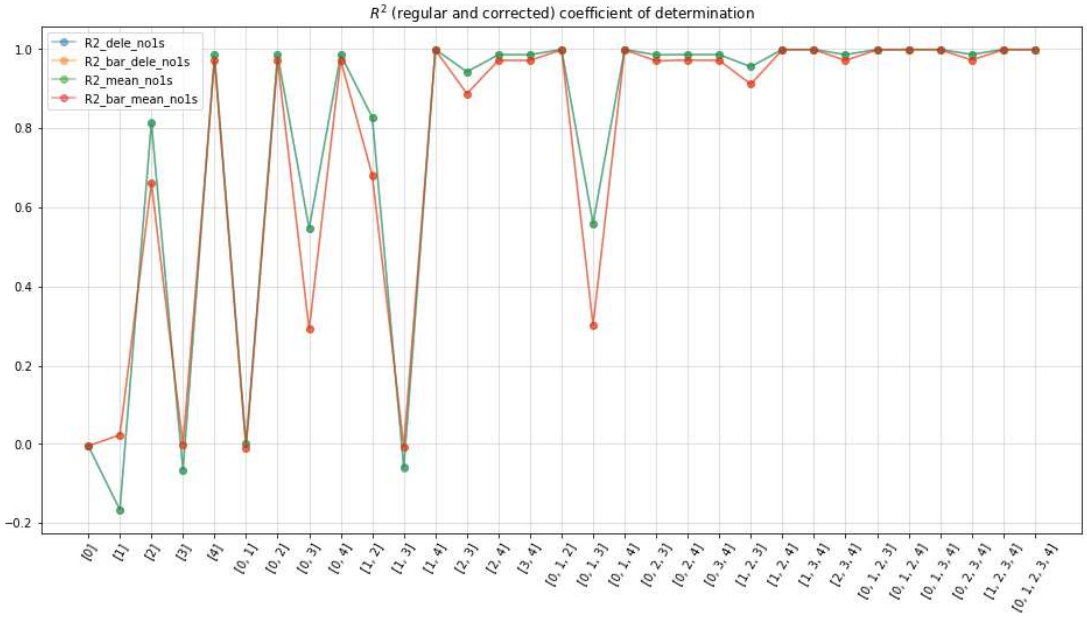
[ ]:
```
def least_squares_complex(X,y):
    # np.linalg.inv outputs the inverse of a matrix
    # .H outputs the conjugate transpose as explained above = .conj().T
    return np.linalg.inv(X.H @ X) @ X.H @ y
```

4.1.4 Results

The feature combinations and the coefficients of determination R^2 and its corrected counterpart $\bar{R}^2$ can be calculated just the same as in the simple-matrix method.

Let "deletion" and "mean" denote the technique used to deal with the outlier in row 100 from the feature matrix . The coefficients of determination shed the following results:

	R2_bar_deletion	R2_bar_mean	R2_deletion	R2_mean
[0]	-0.005025	-0.004999	-0.005058	-0.005059
[1]	0.022643	0.022990	-0.165994	-0.166959
[2]	0.661964	0.660387	0.814655	0.813686
[3]	-0.000708	-0.000487	-0.065730	-0.067200
[4]	0.971270	0.971255	0.985603	0.985596
[0, 1]	-0.010152	-0.010101	0.000372	0.000370
[0, 2]	0.970253	0.970247	0.985166	0.985162
[0, 3]	0.291900	0.291933	0.546824	0.546822
[0, 4]	0.971495	0.971489	0.985790	0.985786
[1, 2]	0.682449	0.680846	0.828034	0.827066
[1, 3]	-0.006591	-0.006361	-0.059372	-0.060851
[1, 4]	0.998049	0.998034	0.999034	0.999026
[2, 3]	0.887344	0.887185	0.942590	0.942504
[2, 4]	0.971532	0.971527	0.985808	0.985805
[3, 4]	0.971486	0.971479	0.985785	0.985781
[0, 1, 2]	0.997985	0.997978	0.999007	0.999003
[0, 1, 3]	0.302013	0.302064	0.559049	0.559046
[0, 1, 4]	0.998348	0.998341	0.999186	0.999183
[0, 2, 3]	0.970109	0.970104	0.985170	0.985166
[0, 2, 4]	0.972322	0.972317	0.986275	0.986272
[0, 3, 4]	0.971368	0.971363	0.985799	0.985795
[1, 2, 3]	0.911435	0.911273	0.955390	0.955303
[1, 2, 4]	0.998351	0.998344	0.999188	0.999184
[1, 3, 4]	0.998282	0.998274	0.999154	0.999149
[2, 3, 4]	0.971394	0.971389	0.985812	0.985808
[0, 1, 2, 3]	0.997979	0.997972	0.999009	0.999006
[0, 1, 2, 4]	0.998352	0.998345	0.999192	0.999189
[0, 1, 3, 4]	0.998340	0.998333	0.999186	0.999183
[0, 2, 3, 4]	0.972224	0.972220	0.986297	0.986294
[1, 2, 3, 4]	0.998343	0.998336	0.999188	0.999184
[0, 1, 2, 3, 4]	0.998343	0.998336	0.999192	0.999189



4.1.5 Conclusions

The above results and plot can be used to infer the most meaningful combinations of features, which seem to be aligned to what has been concluded in the simple-matrix method. We can see, for instance, how the 5th feature alone (index 4) or the combination of the 2nd and 5th (index 1 and 5), reach coefficients as good as some the 4 and 5-feature combinations.

[ ]: