Maciej Zagórski
Computer Science, 3rd year

PROJECT REPORT
Evaluating the performance of regression models

1. Object of the project

The object of the project was to assess how the machine learning regression models work:

- linear regression model,
- second-degree polynomial regression model (second-degree polynomial fitting),
- third-degree polynomial regression model.

Additionally, an assessment was made on the performance of the classification performed using logistic regression (which is precisely a classification algorithm (a classifier)).

The above was done on the basis of the sample datasets "product.csv" and "product2.csv", which contain values of "x" and "y" coefficients (features, parameters). The use of regression models in this context makes it possible to predict (to determine) what the value of the "y" coefficient should be for a product with a given "x" coefficient. On the other hand, the use of a logistic regression model in this context serves to qualify samples with given parameters into a specific category.

2. Project implementation

2.1. Analysis and initial preparation of datasets

2.1.1. General comments

Each of the above-mentioned datasets – "product.csv" and "product2.csv" – contains information on five hundred products, as one might assume from the filenames. This information consists of the "x" and "y" parameter values for each product (record). The "product2.csv" collection, in addition to the same information as the "product.csv" dataset, contains information about the class label of each record – i.e. in the "good" column it contains information about the classification of each product into group "0" or "1". There are fewer products classified as group "0" (184) in the dataset than products assigned to group "1" (314).

It is not clear from the data contained in the datasets to what they relate (e.g. what type of products), by which characteristics they are described, or in which units the values given are expressed. The "product2.csv" dataset does not also provide an answer as to what it means that some products are classified in group "0" and some in group "1". One can only guess that the value "1" in the "good" column should be read as *true*, i.e. that the product in question is a "good" (proper) product; nevertheless, this is of little help without knowing which products and which classification are involved.

The "product.csv" dataset will be used to evaluate the performance of the machine learning regression models – linear regression and polynomial regression – while the "product2.csv" dataset will be used to check the classification performed using logistic regression – as this collection contains information about the class labels of the individual records, and logistic regression is a classification algorithm.

Both datasets contain ordinal data – in the form of a first, unnamed column containing the ordinal numbers of individual records. To ensure that such data did not inappropriately affect the effects of the application of the algorithms, it was removed (after the datasets were imported into the programme using the *pandas* library).

```
data_product = pd.read_csv('../data/report_3_regression_models_product.csv')

data_product.drop([data_product.columns[0]], axis=1, inplace=True)
```

```
data_product_2 = pd.read_csv('../data/report_3_regression_models_product2.csv')

data_product_2.drop([data_product_2.columns[0]], axis=1, inplace=True)
```

2.1.2. Assessment of the completeness and accuracy of the data

The data contained in the "product.csv" and "product2.csv" datasets are complete, i.e:

- for none of the products is the value of the attribute "x" or "y" empty – it is not a blank, unfilled field,
- in the "product2.csv" set, each product is classified in only one of the two "good" categories, i.e. "1" or "0" – there is no product that is not classified on one of these categories or that is classified in another category, e.g. "2".

It turns out, however, that samples with the same "x" and "y" values may occur several times in the datasets – usually two or three times, although in one case the same values characterise up to nine different records. There are as many as 257 such "repeated" rows (duplicates), so they account for more than half of each (500-element) set.

Furthermore, in the "product2.csv" set, of these 257 duplicate records, in 101 cases the same "x" and "y" values are present, but with different class labels in the "good" column. For example, in the case of two duplicates, one indicates a class of "0" and the other a class of "1"; in the case of four duplicates, two may be classified in group "0" and two in group "1", etc. In this sense, the data appear to be incorrect – one would expect that the records with the same "x" and "y" parameters would be classified in the same "good" category.

2.2. Linear and polynomial regression models

2.2.1. Standardisation

Firstly, the data from the "product.csv" dataset was standardised by using the *StandardScaler* function from the *sklearn.preprocessing* module of the *scikit-learn* library. It should not matter whether the standardisation was performed for each feature ("x" and "y") individually or for the entire imported dataset – according to the *scikit-learn library* documentation, the standardisation activities are performed independently for each feature in the dataset[1].

```
stdsc = StandardScaler()

data_product_std = stdsc.fit_transform(data_product)
```

[1] *"Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set"* ([n.a.], *scikit-learn: machine learning in Python – scikit-learn 1.2.2 documentation*, https://scikit-learn.org/stable/, [accessed: 23.05.2023].

2.2.2. Removal of outliers

Outliers were then removed from such a "standardized" dataset using a method based on the interquartile range – samples were removed from the dataset for which the "y" value was:

- greater than or equal to the third quartile (quantile of 0.75) for the entire set of "y" values, plus the value of the interquartile range multiplied by a factor of 1.5; or
- less than or equal to the first quartile (quantile of 0.25) for the entire set of "y" values, minus the value of the interquartile range multiplied by a factor of 1.5.

For this purpose, the function *remove_outliers* was defined, returning a dataset with the outliers removed (the "non-outlier" dataset – the inliers dataset) and a dataset containing only the removed outliers.

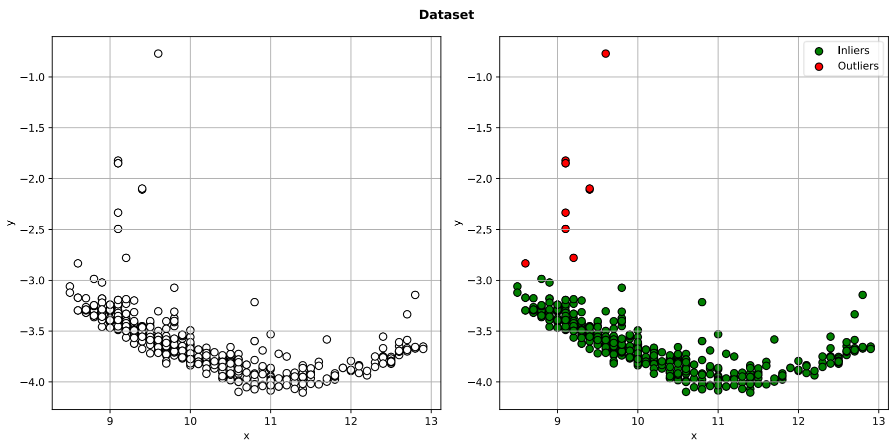
```
def remove_outliers(dataset):
    q1 = np.quantile(dataset[:, 1], 0.25)
    q3 = np.quantile(dataset[:, 1], 0.75)
    iqr = q3 - q1

    print("Q1: %.3f\nQ2: %.3f\nIQR: %.3f" % (q1, q3, iqr))

    inliers = dataset[dataset[:, 1] < q3 + 1.5 * iqr]
    inliers = inliers[inliers[:, 1] > q1 - 1.5 * iqr]

    outliers = dataset[dataset[:, 1] >= q3 + 1.5 * iqr]
    outliers = np.vstack((outliers, dataset[dataset[:, 1] <= q1 - 1.5 * iqr]))

    return inliers, outliers
```

```
Q1: -0.623
Q2: 0.541
IQR: 1.164
```

The "product.csv" dataset is visualised below after the standardisation and the removal of outliers. The *matplotlib.pyplot* module from the *matplotlib* library was used for this purpose. Each dot on the plot is a separate sample from the set. In order to express the "x" and "y" characteristics in real values – to "invert" the standardisation of these data – the *inverse_transform* method was used.

```
f, (plt1, plt2) = plt.subplots(1, 2, figsize=(12, 6))

plt1.scatter(data_product.x, data_product.y, c='white', marker='o',
             edgecolor='black', s=50)

plt1.set(xlabel="x", ylabel="y")
plt1.grid()

plt2.scatter(stdsc.inverse_transform(data_product_stand_inliers)[:, 0],
             stdsc.inverse_transform(data_product_stand_inliers)[:, 1], c='green',
             marker='o', edgecolor='black', s=50, label="Inliers")
plt2.scatter(stdsc.inverse_transform(data_product_stand_outliers)[:, 0],
             stdsc.inverse_transform(data_product_stand_outliers)[:, 1], c='red',
             marker='o', edgecolor='black', s=50, label="Outliers")

plt2.legend(scatterpoints=1)
plt2.set(xlabel="x", ylabel="y")
plt2.grid()
```

```
f.suptitle("Dataset", weight='bold')
plt.tight_layout()
```



2.2.3. Splitting into subsets and data matching

The data was then split into two subsets, i.e.:

- the subset containing the values of the parameter "x" for each of the samples and
- subset containing the corresponding values of the parameter "y" (also for each of the samples).

As further functions from the *scikit-learn* library to be used will require the use of two-dimensional arrays, a new dimension was added to each subset (using the *newaxis* function from the *numpy* library).

```
X_std = data_product_std[:, 0, np.newaxis]
y_std = data_product_std[:, 1, np.newaxis]
```

At this stage, a set with "x" values was also prepared to be used to adequately represent the graphs of the polynomial functions – in the case of second- and third-degree polynomial regression models. For this purpose, using the *arrange* function from the *numpy* library, uniformly distributed values were generated within a given interval – here: based on the minimum and maximum "x" values occurring in the set and an interval equal to 0.1.

```
X_fit = np.arange(X_std.min(), X_std.max(), 0.1)[:, np.newaxis]

print(X_fit)
```

```
[[-1.48185406]
 [-1.38185406]
 [-1.28185406]
 [-1.18185406]
 [-1.08185406]
# [...]
```

A similar result can be achieved based on the actual "x" values provided they have been sorted beforehand. If the set "X_std" was used without such sorting, the function drawing the graph (see p. 3.1.2 below) would be “jumping” between successive values of the parameter “x”, plotting in this way a “zigzag” or (with 450 samples) a painted shape resembling a “banana”.

```
X_fit = np.sort(X_std.flatten())[:, np.newaxis]
```

2.2.4. Division of data into training and test data

The data was then randomly split into a subset of the training data (70%) and a subset of the test data (30%) – using the *train_test_split* function from the *sklearn.model_selection* module (from the *scikit-learn* library).

```
X_train, X_test, y_train, y_test = train_test_split(X_std, y_std, test_size=0.30,
                                                    random_state=0)
```

As the data from the “product.csv” collection does not have information on the classification of the individual samples (class labels), the above function was used without the *stratify* parameter (cf. p. 2.3.3 below).

2.2.5. Application of regression models

The machine learning regression models indicated in p. **Error! Reference source not found.** were then applied to the subsets of data prepared according to the above notes:

- The linear regression model, implemented in the *scikit-learn* library in the form of the *LinearRegression* function from the *sklearn.linear_model* module:

```
lr = LinearRegression()
lr.fit(X_train, y_train)
```

- The linear regression model using the *RANdom SAmple Consensus (RANSAC)* algorithm, which makes the model more robust to outlier samples; the *RANSACRegressor* function from the *sklearn.linear_model* module of the *scikit-learn* library was used to apply it:

```
ransac = RANSACRegressor(estimator=LinearRegression(), max_trials=100,
                         loss='absolute_error', random_state=0)
ransac.fit(X_train, y_train)
```

- The second-degree polynomial regression model; in this case, the *PolynomialFeatures* transform function from the *sklearn.preprocessing* module of the *scikit-learn* library – with parameter *degree=2*, for second-degree polynomial – was used first, followed by a linear regression model:

```
quadratic = PolynomialFeatures(degree=2)
X_train_quad = quadratic.fit_transform(X_train)
X_test_quad = quadratic.fit_transform(X_test)

lr_quadratic = LinearRegression()
lr_quadratic.fit(X_train_quad, y_train)
```

The *PolynomialFeatures* transform function transforms the “x” values so that they are adjusted for polynomial regression; such a transformation was performed above for both the training (*X_train*) and test (*X_test*) data, which will be used in the evaluation of the applied models (cf. p. 2.3.1 below).

- The Polynomial regression model – third-degree polynomial fitting – analogous to above, *PolynomialFeatures* function – with parameter *degree=3* – and the linear regression model were used:

```
cubic = PolynomialFeatures(degree=3)
X_train_cubic = cubic.fit_transform(X_train)
X_test_cubic = cubic.fit_transform(X_test)

lr_cubic = LinearRegression()
lr_cubic.fit(X_train_cubic, y_train)
```

2.2.6. Evaluation of the applied models

In the next step, the applied models were evaluated by calculating for each model (for the results of its application):

- *Mean Squared Error* (MSE) values – the averaged values of the SSE cost function, i.e. the *Sum* of *Squared Errors*. This is the sum of the squares of the differences between the actual and predicted values, divided by the number of samples. Thus, the smaller the MSE value (0 means no errors), the more accurate the applied model is – the more accurate the predictions it makes.
- Determination coefficient R^2 – it is a standardised version of the MSE method (coefficient), calculated based on the quotient of SSE and SST. The SST is the *Sum of Squared Total* – the sum of the squares of the differences between the actual values and the mean value. For the training dataset, the R^2 factor takes on values from 0 to 1 (where 1 indicates the best fit – no errors, so MSE equal to 0), while for the test data it can also be a negative value[2] .

In doing so, the above coefficients were calculated separately for the subset containing the training data and the subset containing the test data.

The *mean_squared_error* and *r2_score* functions from the *sklearn.metrics* module of the *scikit-learn* library, as well as the defined *score* function, were used to calculate these coefficients.

```
def score(X, y, estimator, estimator_name):
    y_pred = estimator.predict(X)
    print(estimator_name + '\nMSE: %.3f,\nR^2: %.3f\n' % (mean_squared_error(y, y_pred),
                                                           r2_score(y, y_pred)))


score(X_train, y_train, lr, "LR: training dataset:")
score(X_test, y_test, lr, "LR: test dataset:")

score(X_train, y_train, ransac, "RANSAC: training dataset:")
score(X_test, y_test, ransac, "RANSAC: test dataset:")

score(X_train_quad, y_train, lr_quadratic, "^2: training dataset:")
score(X_test_quad, y_test, lr_quadratic, "^2: test dataset:")

score(X_train_cubic, y_train, lr_cubic, "^3: training dataset:")
score(X_test_cubic, y_test, lr_cubic, "^3: test dataset:")
```

[2] S. Raschka and V. Mirjalili, *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2,* 3rd ed., Gliwice 2021, p. 326.

2.3. Logistic regression model

2.3.1. General comments

As part of the project, the performance of classification by logistic regression was also tested. Contrary to what the name might suggest, logistic regression is not a regression model, but precisely a classification model. It is used to solve linear and binary problems (it works especially well for linearly separable classes)[3].

The logistic regression model allows not only to predict the classification (class labels) of individual samples from datasets, but also to indicate with what probability a particular sample belongs to a given class.

2.3.2. Dataset breakdown and standardisation

In order to apply the logistic regression model, it was first decided to prepare two datasets – the "data_product_2" dataset, which was a simple import of data from the "product2.csv" dataset (with the column with the ordinal numbers of the individual records removed), and the "data_product_2_clean" dataset, from which those duplicate records that had the same "x" and "y" feature values, but different classifications within the "good" column, were removed – treating such records as invalid data (for comments in this regard, see p. 2.1.2 above). These sets will be referred to hereafter as (respectively) "unmodified data" and "cleaned data".

In order to prepare such datasets, the *pandas* library and its functions, including *duplicated, drop_duplicates* and *merge,* were used. Firstly, a set of records containing all duplicates, i.e. records with the same "x" and "y" parameters, was obtained (257 records). Then, from such a set, those records that also have the same class in the "good" column ("good duplicates") were removed, obtaining as a result a set of records having the same "x" and "y" values, but different "good" values ("bad duplicates"). Then, records with the same "x" and "y" parameters as in the obtained set were removed from the copy of the "data_product_2" set – obtaining a "cleaned" set "data_product_2_clean", containing 399 records (101 records considered to be invalid were removed).

```
data_product_2 = pd.read_csv('../data/report_3_regression_models_product2.csv')

data_product_2.drop([data_product_2.columns[0]], axis=1, inplace=True)

data_product_2_temp = data_product_2.copy()
data_product_2_temp = data_product_2_temp[data_product_2_temp.duplicated(subset=
                                                                          ['x', 'y'],
                                                                          keep=False)]
data_product_2_temp.drop_duplicates(keep='first', inplace=True)
data_product_2_temp = data_product_2_temp[data_product_2_temp.duplicated(subset=
                                                                          ['x', 'y'],
                                                                          keep=False)]

data_product_2_clean = data_product_2.merge(data_product_2_temp, how="outer",
                                            indicator=True)
data_product_2_clean = data_product_2_clean.loc[data_product_2_clean["_merge"]
                                                == "left_only"].drop("_merge", axis=1)

print(data_product_2)
print(data_product_2_clean)
```

[3] S. Raschka, V. Mirjalili, op. cit., p. 82.

```
        x         y  good
0     8.8 -3.339879     1
1     9.5 -3.535988     1
2    10.1 -3.681798     1
3     9.9 -3.536677     1
4     9.9 -3.536677     1
..    ...       ...   ...
495  12.5 -3.807757     1
496   8.6 -3.296743     0
497  12.5 -3.800713     1
498  10.0 -3.775396     0
499   8.6 -3.296743     0

[500 rows x 3 columns]
        x         y  good
0     8.8 -3.339879     1
1     8.8 -3.339879     1
2     9.5 -3.535988     1
3     9.5 -3.535988     1
4    10.1 -3.681798     1
..    ...       ...   ...
495  10.7 -3.907848     1
496  12.5 -3.807757     1
497   8.6 -3.296743     0
498   8.6 -3.296743     0
499  12.5 -3.800713     1

[399 rows x 3 columns]
```

Each of these sets was then divided into two distinct subsets:

- subset „X" ("X_clean" containing information on the "x" and "y" parameters for individual records,
- subset "y" ("y_clean") containing information on the class labels of the samples.

```
X = data_product_2.drop('good', axis=1)
X_clean = data_product_2_clean.drop('good', axis=1)

y = data_product_2['good']
y_clean = data_product_2_clean['good']
```

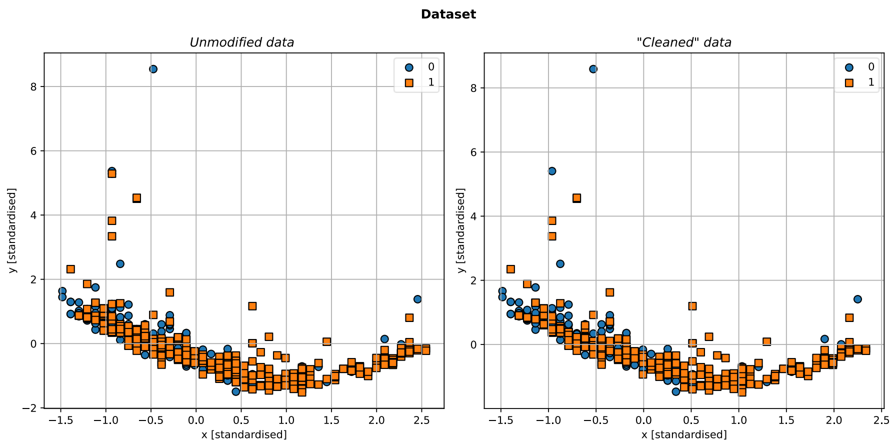
Standardisation of the data contained in the "X" and "X_clean" subsets was also performed (only, as the "y" and "y_clean" subsets contain information on the records class labels); more comments on the standardisation are provided in sec. 2.2.1 above.

```
stdsc = StandardScaler()

X_std = stdsc.fit_transform(X)
X_clean_std = stdsc.fit_transform(X_clean)
```

Below, the "product2.csv" dataset is visualised after the above steps and standardisation (separately for the unmodified data and the "cleaned" data). The *matplotlib.pyplot* module from the *matplotlib* library was used for this purpose. Each dot on the plot is a separate sample from the set.

```
f, (plt1, plt2) = plt.subplots(1, 2, figsize=(12, 6))

plt1.scatter(ma.array(X_std[:, 0], mask=y), ma.array(X_std[:, 1], mask=y),
             c='#1f77b4', marker='o', edgecolor='black', s=50, label="0")
plt1.set(xlabel="x [standardised]", ylabel="y [standardised]")
```

```
plt1.scatter(ma.array(X_std[:, 0], mask=np.logical_not(y)),
             ma.array(X_std[:, 1], mask=np.logical_not(y)),
             c='#ff7f0e', marker='s', edgecolor='black', s=50, label="1")
plt1.set(xlabel="x [standardised]", ylabel="y [standardised]")

plt1.legend(scatterpoints=1)
plt1.set_title("Unmodified data", style="italic")
plt1.grid()

plt2.scatter(ma.array(X_clean_std[:, 0], mask=y_clean),
             ma.array(X_clean_std[:, 1], mask=y_clean),
             c='#1f77b4', marker='o', edgecolor='black', s=50, label="0")
plt2.set(xlabel="x [standardised]", ylabel="y [standardised]")
plt2.scatter(ma.array(X_clean_std[:, 0], mask=np.logical_not(y_clean)),
             ma.array(X_clean_std[:, 1], mask=np.logical_not(y_clean)),
             c='#ff7f0e', marker='s', edgecolor='black', s=50, label="1")
plt2.set(xlabel="x [standardised]", ylabel="y [standardised]")

plt2.legend(scatterpoints=1)
plt2.set_title("\"Cleaned\" data", style="italic")
plt2.grid()

f.suptitle("Dataset", weight='bold')
plt.tight_layout()
```



2.3.3. Splitting data into training and test data

In the next step, the data were randomly split into a subset of training data (70%) and a subset of test data (30%). This splitting was performed separately for the unmodified data ("X_std" and "y") and the "cleaned" data ("X_clean_std" and "y_clean"), using the *train_test_split* function with the parameter *stratify=y* (*stratify=y_clean*). By using this parameter, the subsets returned by the function retain the same proportions of class labels as the input set.

```
X_train, X_test, y_train, y_test = train_test_split(X_std, y, test_size=0.30,
                                                     random_state=0, stratify=y)

X_clean_train, X_clean_test, y_clean_train, y_clean_test = train_test_split(X_clean_std,
                                                                             y_clean,
                                                                             test_size
                                                                             =0.30,
                                                                             random_state
                                                                             =0,
                                                                             stratify=
                                                                             y_clean)
```

2.3.4. Application of the logistic regression model

The logistic regression model was applied to the data subsets prepared in this way, using the *LogisticRegression* function from the *sklearn.linear_model* module of the *scikit-learn* library. The indicated function was applied using, among others, the following parameters:

- *C=0.1*, which means that a high regularisation of the model was applied (the lower the parameter C, the higher the regularisation),
- *multi_class='ovr'*, because the classification of the individual samples in the case under consideration is of binary character (each sample belongs to either category "0" or "1").

```
logreg = LogisticRegression(C=0.1, random_state=0, solver='lbfgs', multi_class='ovr')

logreg.fit(X_train, y_train)

# [...]

logreg.fit(X_clean_train, y_clean_train)
```

2.3.1. Evaluation of the applied model

To evaluate the results of the application of the logistic regressionmodel, the evaluation methods suitable for classification algorithms seem to be more appropriate. Therefore, the percentage accuracy of the applied model was determined using the *score* method (from the *scikit-learn* library) and the defined *score_logreg* function.

```
def score_logreg(X, y, estimator, estimator_name):
    y_pred = estimator.predict(X)
    print(estimator_name + "Score: %.3f" % (logreg.score(X, y)))
```

In doing so, the above value was calculated separately for the training and test data, for each of the datasets – i.e. the unmodified data and the "cleaned" data.

In the next step, an assessment was made for each of these datasets, but removing the columns containing information about the value of the "y" parameter – wanting to assess whether the model would perform worse if only the column containing information about the value of "x" was taken into account in the datasets.

```
logreg.fit(X_train, y_train)

score_logreg(X_train, y_train, logreg, "LogReg: unmodified data: training dataset: ")
score_logreg(X_test, y_test, logreg, "LogReg: unmodified data: test dataset: ")


logreg.fit(X_clean_train, y_clean_train)

score_logreg(X_clean_train, y_clean_train, logreg, "LogReg: \"cleaned\" data: training "
                                                   "dataset: ")
score_logreg(X_clean_test, y_clean_test, logreg, "LogReg: \"cleaned\" data: test "
                                                 "dataset: ")

# [...]
```

```
X_train = np.delete(X_train, 0, 1)
X_test = np.delete(X_test, 0, 1)
logreg.fit(X_train, y_train)
score_logreg(X_train, y_train, logreg, "LogReg: unmodified data (without \"y\"): "
                                       "training dataset: ")
score_logreg(X_test, y_test, logreg, "LogReg: unmodified data (without \"y\"): "
                                     "test dataset: ")

X_clean_train = np.delete(X_clean_train, 0, 1)
X_clean_test = np.delete(X_clean_test, 0, 1)
logreg.fit(X_clean_train, y_clean_train)

score_logreg(X_clean_train, y_clean_train, logreg, "LogReg: \"cleaned\" data: (without "
                                                   "\"y\"): training dataset:")
score_logreg(X_clean_test, y_clean_test, logreg, "LogReg: \"cleaned\" data: (without "
                                                 "\"y\"): test dataset: ")
```

3. Presentation of results

3.1. Linear and polynomial regression models

3.1.1. The obtained values of the MSE and R^2

The values of the MSE and R^2 coefficients (rounded to three decimal places) obtained for the applied linear and polynomial regression models are presented in the table below – separately for each regression model and subset of data (training and test).

	Regression model	**Data subset**	**MSE**	**R^2**
1.	Linear	Training	0.296	0.465
2.		Test	0.212	0.541
3.	Linear (RANSAC)	Training	0.414	0.252
4.		Test	0.319	0.311
5.	Polynomial of the 2nd degree	Training	0.117	0.788
6.		Test	0.086	0.813
7.	Polynomial of the 3rd degree	Training	0.107	0.806
8.		Test	0.080	0.827

3.1.2. Visualising the effects of the application of the regression models

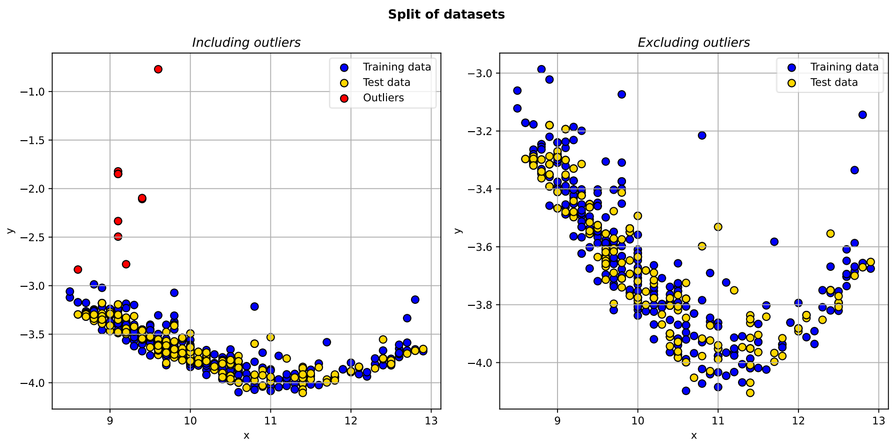
The effects of the application of the regression models are visualised below as a scatter plot – using the *matplotlib.pyplot* module from the *matplotlib* library. Each dot on the plot is a separate sample from the dataset. In doing so, the regression lines were plotted against a prepared "X_fit" set of the fitted data (see sec. 2.2.3 above).

```
X_fit = np.arange(X_std.min(), X_std.max(), 0.1)[:, np.newaxis]
# X_fit = np.sort(X_std.flatten())[:, np.newaxis]
# [...]

y_lr = lr.predict(X_fit)
y_ransac = ransac.predict(X_fit)
```

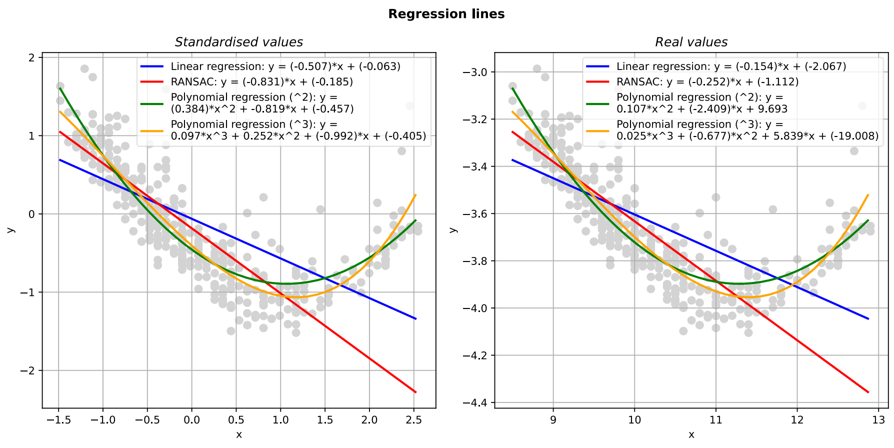
```
y_quadratic = lr_quadratic.predict(quadratic.fit_transform(X_fit))
y_cubic = lr_cubic.predict(cubic.fit_transform(X_fit))
```

Furthermore, in order to express the "x" and "y" characteristics in real values – to "invert" the standardisation of these data – the *inverse_ transform* method was used (see p. 2.2.2 above).

For the sake of legibility, a separate plot shows the breakdown of the data subsets, i.e. the individual records divided into the training subset and the test subset.



The next pair of plots shows the regression lines for each of the regression models, together with the obtained equation of the models – with the "x" and "y" values and coefficients (weighting and y-axis intercept) expressed in standardised values, as well as in real values.



3.2. Logistic regression model

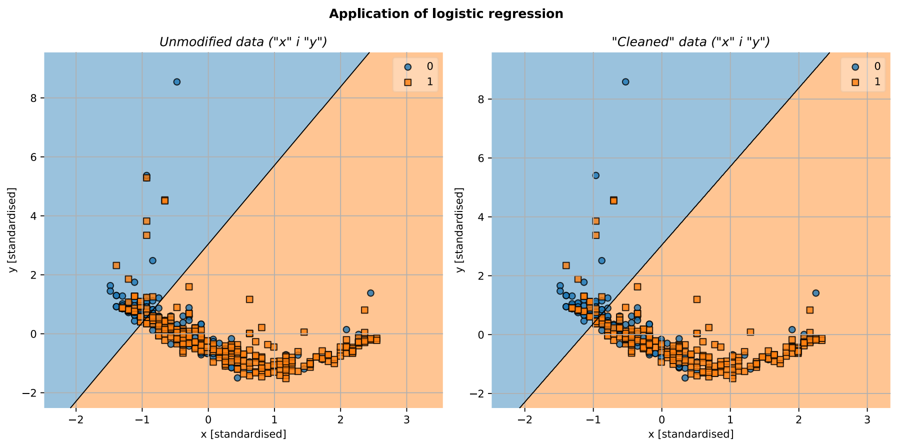
3.2.1. Percentage accuracy of the model used

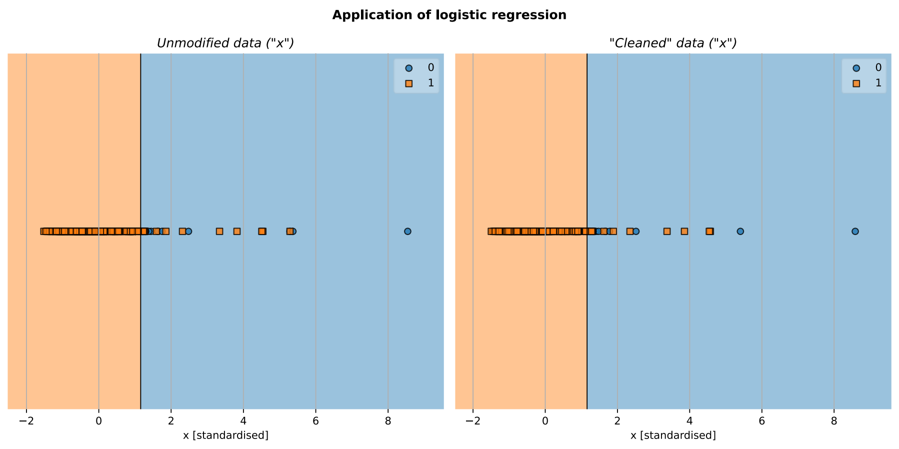
The obtained values of the percentage accuracy of the applied logistic regression model are shown in the table below – separately for each dataset (the unmodified one and the "cleaned"), with and without the "y" parameter in the datasets, as well as separately for the training data and the test data.

	Dataset	Data subset	***Score***
1.	Unmodified	Training	0.666
2.		Test	0.667
3.	„Cleared”	Training	0.685
4.		Test	0.783
5.	Unmodified without „y” value	Training	0.629
6.		Test	0.620
7.	“Cleared” without the “y” value	Training	0.652
8.		Test	0.633

3.2.2. Visualisation of the effects of the application of the logistic regression model

The effects of the application of the logistic regression model are also visualised below in the form of a scatter plot – using the *plot_decision_regions* function from the *mlxtend.plotting* module from the *mlxtend* library and the *matplotlib.pyplot* module from the *matplotlib* library. Each dot on the plot is a separate sample from the corresponding dataset.



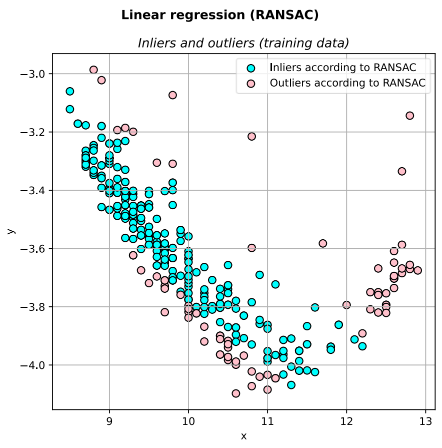


4. Comments and conclusions

4.1. Linear and polynomial regression models

Based on the work carried out and the results obtained with regard to the linear and polynomial regression models used, it can be indicated that:

- **The best results were achieved as a result of the application of the third-degree polynomial regression model –** in this case:
  - the MSE coefficient approached 0 (0.107 for the training data, 0.080 for the test data; a value of 0 would mean no errors),
  - The R^2 factor was above 0.8 (0.806 for training data, 0.827 for test data; a value of 1 would indicate the best fit – no errors).
- In contrast, **the worst results were achieved as a result of the application of the linear regression model using the RANSAC algorithm**: MSE with values of 0.414 for the training data and 0.319 for the test data (values approximately four times higher than the ones in best solution) and R^2 with values of 0.252 for the training data and 0.311 for the test data (values approximately three times lower than ones in the best solution).
- The above results are not surprising in the sense that **already after visualising the data in the form of the scatter plot (see sec. 2.2.2 above), it can be seen that the samples are arranged in the form of an "arc"** – which means that polynomial regression models will fit them better than models using linear regression.
- **The lack of fit of linear regression models to the data stacked in such a way is illustrated by the use of a model using the RANSAC algorithm.** This model (as indicated in sec. **Error! Reference source not found.**) is more robust to outliers in the sense that it already extracts outliers as part of its operation. Despite the fact that such a separation has been made as a part of the project – outlier samples have been removed from the datasets (using the method based on the interquartile range – see sec. 2.2.2 above) – the applied algorithm considered further samples as outliers and did not take them into account when applying the linear regression model. The scatter plot below illustrates these "extra" samples considered as outliers – as can be seen, the algorithm considered as such the samples on the right side, after the "bend" of the curve (as if there was no such bend).



- The results of the regression models show that **no overtraining occurred for any of the models –** in each case, the MSE and R^2 coefficients obtained for the test data are no worse than those calculated for the training data.
- **The lack of more complete information on to what the analysed datasets refer makes it difficult to assess their quality –** how to treat, for example, the records appearing in the datasets with the same "x" and "y" parameters (see in this respect the comments in sec. 2.1.2 above). And such duplicate records may affect, for example, the extraction of outliers – using a method based on the interquartile range for this – and consequently the results of applying specific models.

4.2. Logistic regression model

Based on the activities carried out and the results obtained with regard to the applied logistic regression model, it can be indicated that:

- **The best results were achieved as a result of applying logistic regression to the "cleaned" data –** in this case, the percentage accuracy of the applied model used was 0.685 for the training data and 0.783 for the test data.
- In contrast, **the worst results were achieved as a result of applying this regression to datasets from which the "y" values were removed.** Nevertheless, it should be noted that in the case of the unmodified data, these differences were not significant (the unmodified data, training – 0.666, test – 0.667; the unmodified data without "y" values, training – 0.629, test – 0.620).
- In the case of removing the "y" value from the datasets, better results of the model application were possible for the "cleaned" data by decreasing the regularisation of the model (increasing the value of the C parameter in the algorithm used). On this basis, it could be considered that **the fewer the number of features on which the model operates, the lower the regularisation should be** – in such a case it is not necessary, and it may result in over-fitting the model (overtraining it). However, in the case of the unmodified data, increasing parameter C (decreasing regularisation) resulted in worse results.

```
logreg.set_params(C=10)

X_train = np.delete(X_train, 0, 1)
X_test = np.delete(X_test, 0, 1)
logreg.fit(X_train, y_train)

score_logreg(X_train, y_train, logreg, "LogReg: unmodified data (without \"y\"): "
                                       "training dataset: ")
score_logreg(X_test, y_test, logreg, "LogReg: unmodified data (without \"y\"): "
                                     "test dataset: ")


X_clean_train = np.delete(X_clean_train, 0, 1)
X_clean_test = np.delete(X_clean_test, 0, 1)
logreg.fit(X_clean_train, y_clean_train)

score_logreg(X_clean_train, y_clean_train, logreg, "LogReg: \"cleaned\" data: (without "
                                                   "\"y\"): training dataset: ")
score_logreg(X_clean_test, y_clean_test, logreg, "LogReg: \"cleaned\" data: (without "
                                                 "\"y\"): test dataset: ")
```

```
LogReg: unmodified data (without "y"): training dataset: Score: 0.634
LogReg: unmodified data (without "y"): test dataset: Score: 0.607
LogReg: "cleaned" data: (without "y"): training dataset: Score: 0.695
LogReg: "cleaned" data: (without "y"): test dataset: Score: 0.725
```

- From the results obtained from the application of the logistic regression with parameter C equal to 0.1 (high regularisation), it is apparent that **overtraining of the model has occurred** for the datasets stripped of "y" values – for the unmodified data as well as the "cleaned" data the results obtained from the application of the model are better for the training data than for the test data.
- As in the case of the linear and polynomial regression models, the lack of more complete information about the datasets made it difficult to assess their quality – and to process them appropriately in such a way as to increase the effectiveness of the model used without simultaneously removing relevant information from the sets. At the same time, it can be pointed out that the applied model **achieved better results in the case of the "cleaned" data,** i.e. the data that did not contain "contradictory" information (different class labels with the same value of parameters "x" and "y").

5. Sources

[n.a.], *Matplotlib – Visualization with Python*, https://matplotlib.org/, [accessed: 26.05.2023 r.]

[n.a.], *mlxtend*, https://rasbt.github.io/mlxtend/, [accessed: 26.05.2023 r.]

[n.a.], *pandas – Python Data Analysis Library*, https://pandas.pydata.org/, [accessed: 26.05.2023 r.]

[n.a.], *scikit-learn: machine learning in Python – scikit-learn 1.2.2 documentation*, https://scikit-learn.org/stable/, [accessed: 26.05.2023 r.]

S. Raschka, V. Mirjalili, *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2,* 3rd ed., Gliwice 2021.