Data Mining: A Complete Analysis For Transformation, Validation and Evaluation Of An Applied Data Set

Mariana Ávalos Arce
Universidad Panamericana
Data Mining, Fall 2021

Abstract

The purpose of this document is to perform a thorough analysis of a data set used for a Binary Classification Problem, in order to make decisions on how to transform and validate/resample the data set before feeding it to a Machine Learning Problem, and after this is done, evaluate the resulting model with different metrics of performance: accuracy percentage, Cohen's Kappa coefficient and ROC AUC technique. The analysis includes some of the basics of data distribution and dispersion analysis of the said attributes. The document includes code snippets of the actions performed in order to achieve a better understanding of the results from the reader.

---

1 The Data Set

The data set used in the following pages is owned by UCI Machine Learning [2], and was found through Kaggle's data set browser. In simple words, this data set is used for a Supervised Classification Problem in which a Machine Learning algorithm learns to predict whether a female patient's breast tumor is malign or benign, given 10 attributes from said tumor. The features or attributes in the data set were extracted from digitized images of a breast cancer cell in different tumors, which describe the cell nuclei presented in the images. A brief description of each column or attribute is provided for clarity on further sections:

1. `id`: unique number of the patient.
2. `diagnosis`: M = malign or B = benign.
3. `radius_mean`: mean of distances from center to points on the perimeter.
4. `texture_mean`: standard deviation of gray-scale values found on the tumor's image.
5. `perimeter_mean`: mean size of the core tumor.
6. `area_mean`: mean area of the tumor.
7. `smoothness_mean`: mean of local variation in radius lengths that represent smoothness.
8. `compactness_mean`: mean of perimeter raised to the power of 2 over the area.
9. `concavity_mean`: mean of severity of concave portions of the tumor's contour.
10. `concave points`: mean for number of concave portions of the tumor's contour.
11. `symmetry_mean`: tumor's symmetry measure.
12. `fractal_dimension`: mean for *coastline approximation* - 1.

1.1 Understanding The Data set

The first approach to a data set should always involve a glimpse of the kind of instances that are registered in the table. We will visualize the first 5 instances that appear in said data set using Pandas' `head()` function, which outputs the first 5 instances of a data frame when no argument is sent [3].

id	diag	radius	texture	perimeter	area	smooth	compact	concavity	concave pts.	symmetry	fractal
842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	0.2419	0.07871
842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	0.1812	0.05667
843009	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	0.2069	0.05999
843483	B	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	0.2597	0.09744
843584	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	0.1809	0.05883

Table 1: First five instances in the data set.

After seeing what appears on Table 1, we can conclude that the data set has the target column of the binary classification under the column `diag`, which stands for *diagnosis*, and it contains inside what appears to be either B or M, when the tumor in the row is either Benign or Malign, but we will confirm this with another function. Nevertheless, we can confidently say that all the attributes related to the target are floating point numbers, some of which have a larger range of values, as in columns `area` and `perimeter`, especially in comparison to columns with a smaller range of values, which are the remaining columns. The column `id` is indeed numeric but let's not get fooled by the data type: this column brings no information regarding the prediction of the target column.

We can see that `diagnosis` column is the target column that contains the classification for each tumor row, but how to know for sure that this is a data set destined to predict for a Binary Classification Problem? Well, if we consider that a Binary Classification Problem consists in a set of attributes that can either belong to one class or another, it is necessary to have **only two** possible values in the target column. For confirming this, we will use the function `groupby()`:

```
df.groupby('diagnosis').size()
```

This snippet is definitely not the only code necessary, but the current document assumes the reader knows the missing previous steps of including libraries and reading a .csv file as a dataframe. Back to the point, we get the results presented in Table 2.

diagnosis	
Value	Count
B	357
M	212
dtype: Object	

Table 2: Results from *groupby*() function applied to *diagnosis* column.

As we were expecting, the Table 2 shows that the target column `diagnosis` contains only two possible values: B or M, which stand for Benign and Malign. Thus, each of the rows is a tumor and the column `diagnosis` describes the final classification of said tumor. Additionally, the output includes the `dtype` of the column, which is the data type(s) that the column contains. As described in Pandas Documentation [3], the Pandas library is based on Python's numeric library Numpy, and therefore, Pandas identifies any data type that is not numeric as `Object`, when in reality, the column is of `string` type. For further analysis, this column will be binarized in following sections, so that further Machine Learning models can use the column. Most Machine Learning algorithms for Binary Classification need that **all attributes are numeric**, which we can check by consulting the `dtypes` attribute of our data frame.

```
df.dtypes
```

The attribute `dtypes` of a data frame shows all the data types by column of said data frame [3]. The snippet above outputs the results presented in Table 3.

Column	Data Type
id	int64
diagnosis	object
radius	float64
texture	float64
perimeter	float64
area	float64
smoothness	float64
compactness	float64
concavity	float64
concave points	float64
symmetry	float64
fractal dimension	float64

Table 3: Results from *dtypes* attribute of the present data frame.

Considering Table 3, we can say that the data set contains all attributes in a numeric data type, more specifically, in a floating point number. Now, by checking the column name and its data type, each row is a collection of different measures taken from the tumor of each row and registered its measures in different units, which explains the difference in each column's ranges in value.

Coming back to the dimensions presented in Table 2, we can get an idea of the overall dimensions of the data set by consulting the data frame's `shape` attribute, which tells us the matrix dimensions of rows and columns of said data frame [3]. The code snippet is as simple as the one below.

```
df.shape
```

The output coming from this code is simply: `(569, 13)`. What is interesting is that even though the official explanation of the attributes determines that the data set contains 12 attributes, but the dimensions show 13. In fact, in the `head()` function output, there is a column at the end titled `Unnamed: 32` and is full of `NaN` instances. This column must have been an empty column stored as trash from the original table.

The final step to understanding the data set is to get a summary of the descriptive statistics of the columns, which is the purpose of function `describe()`. The output of the said instruction in presented in Table 4.

Measure	diag	radius	texture	perimeter	area	smooth	compact	concavity	concave pts.	symmetry	fractal
count	569.00	569.00	569.00	569.00	569.00	569.00	569.00	569.00	569.00	569.00	5.69e+02
mean	0.37	14.13	19.29	91.97	654.89	0.10	0.10	0.09	0.05	0.18	6.28e-02
std	0.48	3.52	4.30	24.30	351.91	0.01	0.05	0.08	0.04	0.03	7.06e-03
min	0.00	6.98	9.71	43.79	143.50	0.05	0.02	0.00	0.00	0.11	5.00e-02
25%	0.00	11.70	16.17	75.17	420.30	0.09	0.06	0.03	0.02	0.16	5.77e-02
50%	0.00	13.37	18.84	86.24	551.10	0.10	0.09	0.06	0.03	0.18	6.15e-02
75%	1.00	15.78	21.80	104.10	782.70	0.11	0.13	0.13	0.07	0.20	6.61e-02
max	1.00	28.11	39.28	188.50	2501.00	0.16	0.35	0.43	0.20	0.30	9.74e-02

Table 4: Summary of the descriptive statistics of the data frame using the *describe*() function.

What Table 4 presents to us is a quantile and central tendency calculation of all columns in the data set. By looking at 50% measure, which is the middle quantile or the *median* of each column, we can see that the center of the columns' distributions is quite similar in some attributes: radius and perimeter seem to have a median around the same level, and then smoothness, compactness, concavity and concave points also have a median but between the range of 0.00-0.10, whereas fractal dimension column appears to be the

smallest measure. This differences in the medians mean that their distributions are **centered in different points along its x axis**, which can indicate the need of a **standardization** transformation in order to center all these distributions, which will get discussed later in more detail. The *mean* of a column tends to be bigger than the *median* of said column, which suggests **a positive skewness in the mayority of the distributions**, and with this, **more outliers as part of the long right tails anticipated in the mayority of the distributions**. If we are anticipating long right tails in the distributions, this can be confirmed when we look at 75% measure, or the upper quantile, and we see that and **the upper quantile (75%) tends to be more separated from the middle quantile (median) in comparison to the lower quantile (25%)**, which explains the the positive skewness and longer right tails.

1.2 Data Visualization: Further Understanding

At the end of the previous section, we discussed mainly the quartile calculations of each column, where we suggested that there are indiactors of positive skewness along most of the attributes. However, all this assumptions where made from the analysis of numeric computations. When talking about distributions and its shape, the numeric central tendency measures do help, but it might be easier and visually confirmed by using plots.

In Figure 1, we made use of Seaborn's library function `distplot()`, which, as described in Seaborn's Documentation [4], is a combination of Matplotlib's `hist()` function that plots the frequency histogram of a numeric attribute, and a `kdeplot` that shows a distribution density estimate using a Gaussian Kernel.

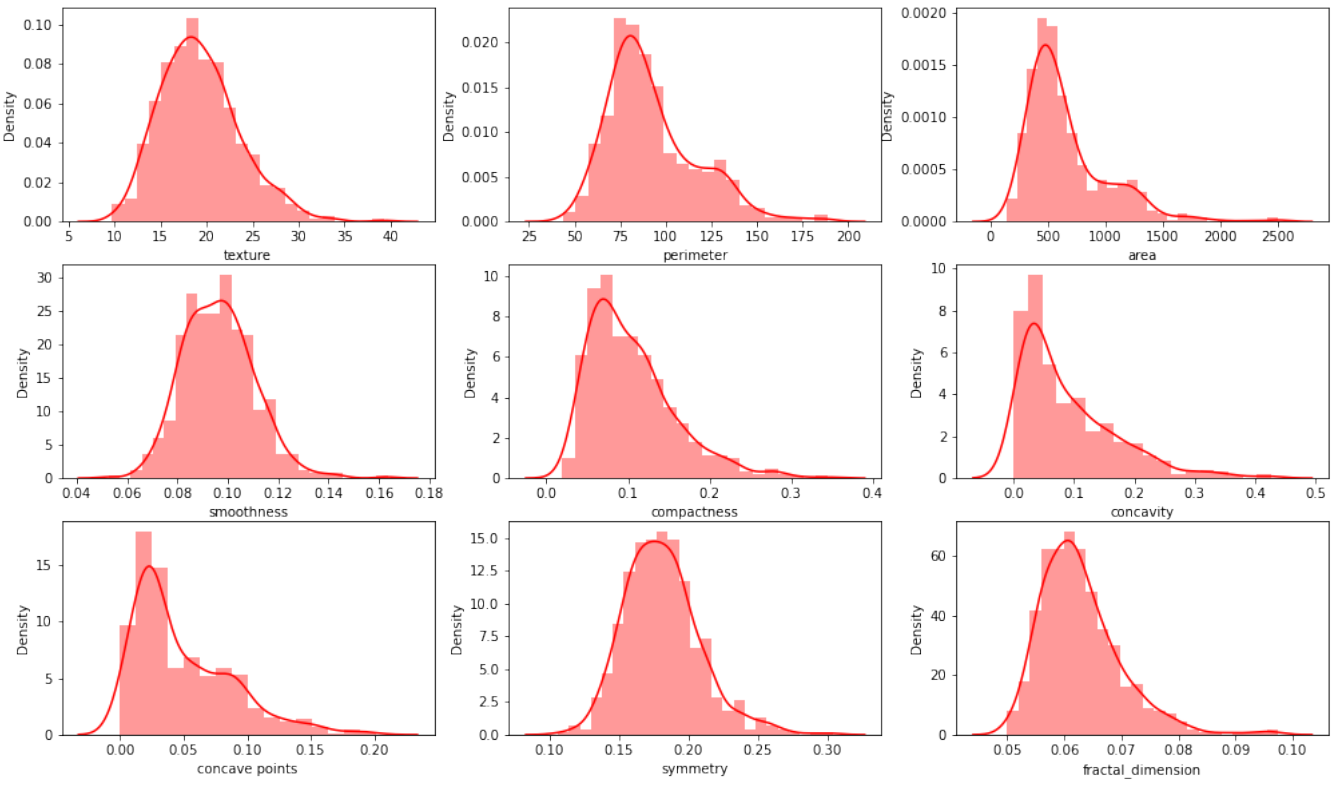

Figure 1: Distribution histogram and density plot of all attributes in the data set.

Thus, as implied by Figure 1, we confirm what we suggested at the end of the previous section, which was the **Positive Skewness** present in almost all attributes when the distribution is, instead of centered around the mode, concentrated towards the left side of the x axis, leaving a long tail in the left side. This

tendency of concentration towards the left side moves the mode and the median towards this side as well, whereas the mean will be carried towards the tail in the opposite side.

These conclusions invariably talk about where does the **mayority of data concentration is**, which describes the dispersion of the data found in each attribute. Now, conclusions about dispersion are greatly visualized with a **box plot** of the data, and for this we will use the function `plot()` with parameter `kind='box'`, which is a Pandas function that plots a box plot per attribute of the data frame. In Figure 2, the individual box plots of each attribute and a common-axis box plot of all the plots together is presented.

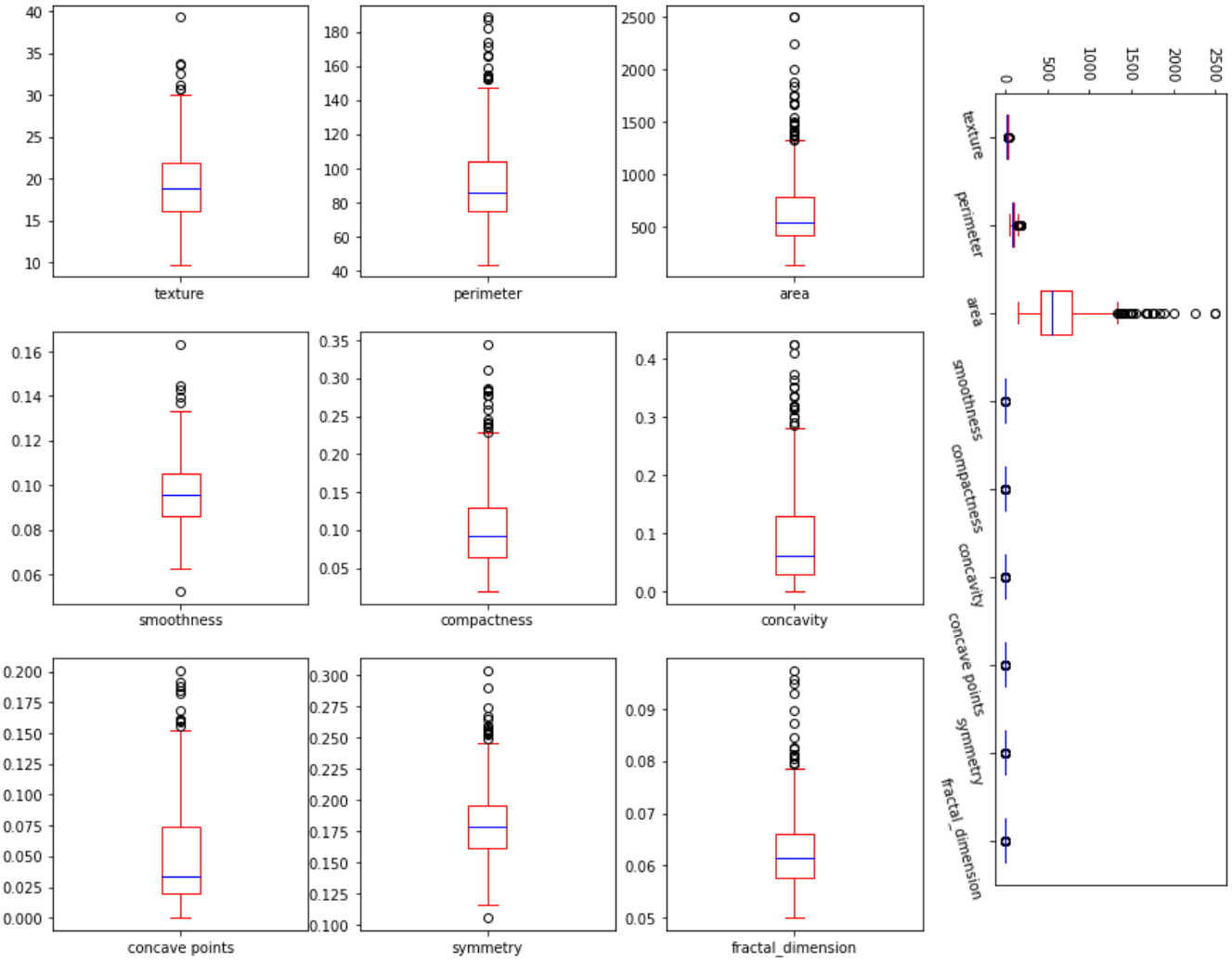

Figure 2: (Left) Box plot per attribute. (Right) All attributes' box plot in the same axis.

From Figure 2 we can see that **the concentration in the left side of the axis seen in the distribution density plots** in Figure 1 is now translated into a series of box plots that have **the box in the lower Y axis values, which represents this positive skewness** mentioned before. Also, in the right part of the figure a plot with all box plots is shown so that the dimensions of each attribute can be perceived visually: **the difference in the range of values of each attribute with one another is so large that most of the box plots are reduced to a line**, showing specifically how the area column differs considerably from all other ranges. This will definitely need a standardization transformation so that their means and medians are centered near the same values and thus ease the process of comparison. It is also evident that **the amount of outliers is significant**, which explains why the *mean* of the majority of the attributes presented in Table 4 is dragged towards the right-side tail or the upper Y axis of the box plot.

Now that we have understood single-variable relationships in the data set, the next step is to understand the two-variable relationships, and for that a **correlation matrix** is computed. A **correlation matrix** is a matrix that computes the correlation coefficient, which is the covariance of two variables divided by their individual standard deviations [1], for all possible permutations of two attributes among the data set, and then displayed in a matrix form for visual purposes. This correlation matrix is computed using Pearson method as a parameter to Pandas function `corr()` and then that matrix is sent to Seaborn's `heatmap()` plot function, as shown in the code snippet below.

```
correlation = df.corr(method='pearson')
plt.figure(figsize=(10,5))
ax = sea.heatmap(correlation, annot = True, cmap='plasma', vmin=-1, vmax=1)
# some axis and label formatting code [...]
plt.show()
```

The Seaborn's `heatmap()` plot function receives the matrix of correlation coefficients in order to give each matrix cell a color according to the value that is in the cell [5], so that a better visualization of the matrix data can be done using scaled colors. Figure 3 shows such heatmap for all correlations of all pairs of attributes in the data set.

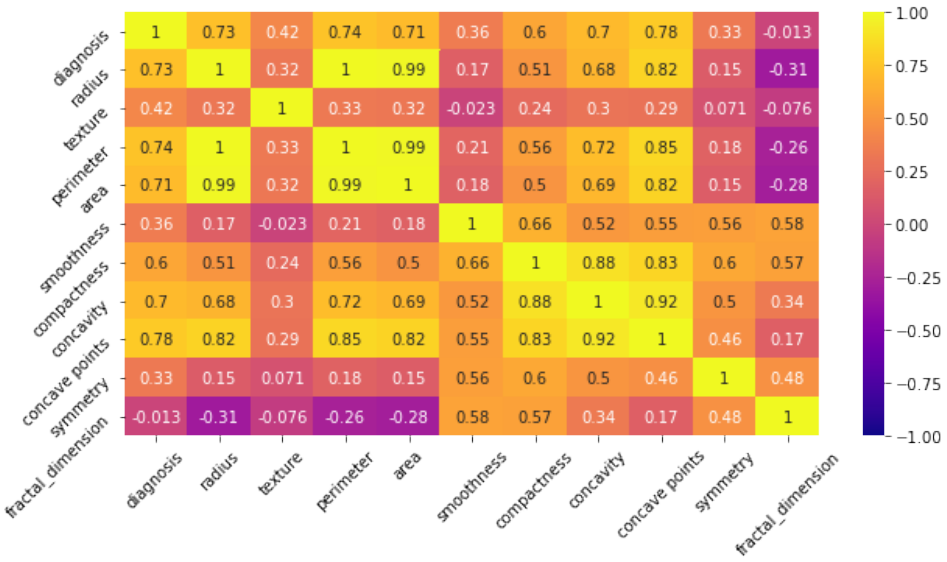

Figure 3: Correlation matrix for all the pairs of attributes in the data set.

2 Data Set Cleaning

2.1 Erasing Useless Columns

For the purposes of training a Logistic Regression algorithm, we need to binarize the `diagnosis` column, so that instead of `B` and `M` we have `0` and `1` as the possible values for this attribute, making it our class column. Additionally, the `id` attribute does not represent a numeric column per se, because its values do not intervene in the classification of a tumor. Thus, this column will better be dropped, as well as a column named `Unnamed: 32` that unexpectedly appears at the end of all the mentioned columns, and is filled with `NaN` entries. In this way, we are making our data set to have **only numeric columns**, and **one binary class column** that determines the classification attribute, which makes our problem **supervised**

binary classification. The following code snippet performs this data set operations.

```
import pandas as pd

url = 'db/data.csv'
bcancer = pd.read_csv(url, header=0)
bcancer['diagnosis'] = bcancer['diagnosis'].apply(lambda x: 1 if x == 'M' else 0)
del bcancer['Unnamed: 32']
del bcancer['id']
```

2.2 Dropping Off Based On Correlation

Now, after having our data set as only numeric columns, the next step is to look for **data redundancy**. An attribute (column or feature) of a data set is called **redundant** if it can be derived from any other attribute or set of attributes. This can be perfectly identified whenever a column presents a strong correlation to another attribute, which usually means a correlation close to 1. For a quick visualization of the correlations in the data set, a heat map is plotted using the calculated correlation with the *Pearson* method for all the possible combinations of column pairs, presented in Figure 4.

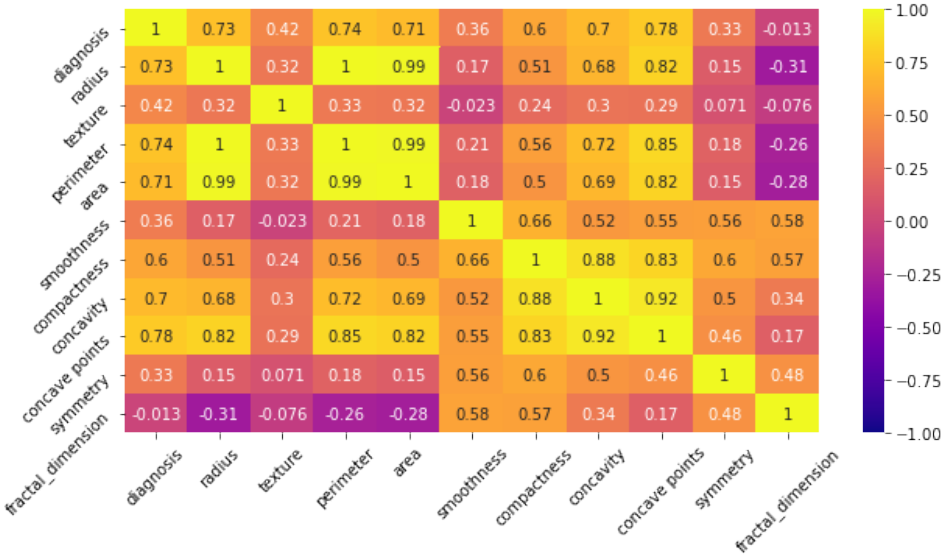

Figure 4: Correlation heat map plot of all the data set attributes.

Indeed there are some strongly correlated attributes, which show values close to 1, such as: `perimeter`, `area` and `radius`. The attributes `perimeter` and `area` present a correlation of 0.99 because of their calculation involving other similar measures, but they are completely different measures of the main properties of the tumor, so they were not dropped from the data set. Thus, the only pair remaining with a strong correlation value is `radius` and `perimeter`. Which one to delete? The column `perimeter` presents a correlation of 0.74 with the `diagnosis` class column, whereas the column `radius` presents a correlation of 0.73 with the `diagnosis`, therefore, `radius` can be considered as redundant, and thus is dropped from the data set. In other words, we delete the radius attribute due to its redundancy and lower relationship with the class. This operation is shown in the snippet below.

```
del df['radius_mean']
df.head()
```

class	texture	perimeter	area	smoothness	compactness	concavity	concave points	symmetry	fractal
1	10.38	122.80	1001.0	0.1184	0.2776	0.3001	0.1471	0.2419	0.07871
1	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667
1	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999
1	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	0.09744
1	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	0.05883

Table 5: Instances after cleaning operations (diagnosis is class).

At the end of these operations we can observe some instances that are left in the data set, shown in Table 5.

Thus, our data set seems ready to be plugged into the learning models. It is also important to mention that after performing the following,

```
benign = df.groupby('diagnosis').size()[0]
malign = df.groupby('diagnosis').size()[1]
total = malign + benign
benign_p = benign / (total) * 100
malign_p = malign / (total) * 100
print(f"benign: {benign_p:.2f}%")
print(f"malign: {malign_p:.2f}%")
```

we can see that 62.74% of the instances correspond to class 0 and 37.26% belongs to class 1. These percentages, although different, can say the data set is somehow balanced in terms of number of instances per class, to avoid an over-training of the model towards a specific class value, which would arise if the percentages differed much more.

3 Resampling Methods: Cross Validation

3.1 Raw Data

The first Resampling Method is the Cross Validation, in which we split the data into N parts and dedicate 1 to test the model and the N-1 remaining to train it, and iterate N times, each time changing the test part. We apply this method to **raw data**, which means that the **x** varibale containg a matrix of the data set values **without any transformation**, and the class column left out. The Logistic Regression needed in this case around **3500 iterations**. The **y** variable refers to our class column.

```
num_folds = 10
kfold = KFold(n_splits=num_folds, shuffle=True)
model = LogisticRegression(solver='lbfgs', max_iter=3500)
results = cross_val_score(model, x, y, cv=kfold)
acc = results.mean() * 100
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

As a result we get an **accuracy of 90.51% and a standard deviation of 0.0335** for this method applied with raw data. Requiring 3500 iterations for the ML modeling seems risky, since this is a method that can potentially involve repetitions, and thus cost a lot more time, computationally speaking. With further methods this will show up more clearly.

3.2 Standardized Data

Now we will transform the **x** data matrix using a transformation called Standardization. This operation on **x** involves that, for each column x_i, the following is applied to get every instance:

$$x_{scaled} = \frac{x_i - min(x_i)}{max(x_i) - min(x_i)} \tag{1}$$

This transformation moves the attribute distribution towards a mean of zero and a standard deviation of 1. If we plot the new standardized data columns, we can see such results as in Figure 5. This transformation only translates the distribution to a more centered range, in simple words; this means that the distribution of the attributes is not modified towards a complete Gaussian bell, because its purpose is to standardize the ranges of values. This standardized data set will be called now **rescaled x**.

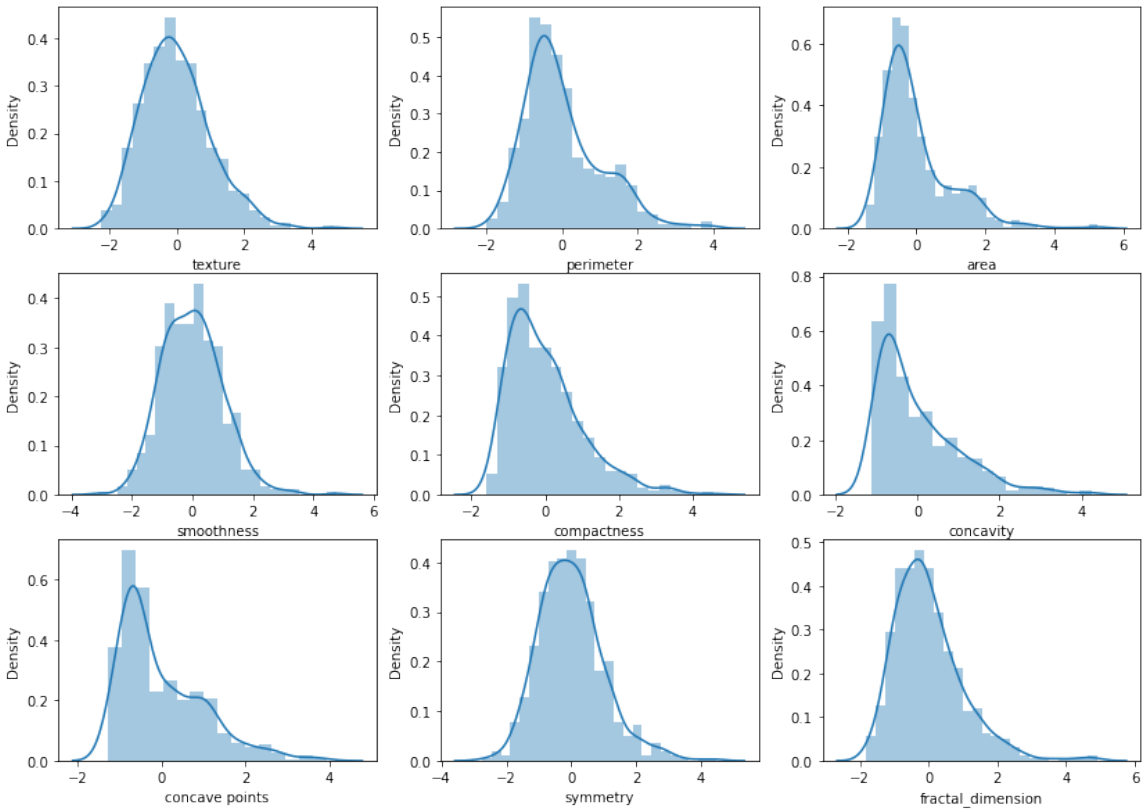

Figure 5: Rescaled x: Standardized data attributes.

```
num_folds = 10
kfold = KFold(n_splits=num_folds, shuffle=True)
model = LogisticRegression(solver='lbfgs', max_iter=100)
results = cross_val_score(model, rescaledX, y, cv=kfold)
acc = results.mean() * 100
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

The code snippet above outputs an **accuracy of 94.19% and a standard deviation of 0.024** for the method applied to standardized data. Compared to the method in raw data, the accuracy in the prediction of the class by the model increased significantly from 90.51% to 94.19%, but more importantly, the iterations needed were reduced so that the model can converge in just 100 iterations, **reducing the computation time and increasing the accuracy**.

3.3 Standardization and Yeo-Johnson Transformed Data

The **rescaled x** matrix of instances, as seen in Figure 5, is centered in a mean of zero and has a unit standard deviation, but it can suffer further transformations so that the column instances have a more Gaussian-like distribution. This transformation can be achieved by using either the *Box-cox method* or the *Yeo-Johnson method*, but in this case the *Yeo-Johnson method* is the only one of the two that accepts negative data values, such as the ones present in our data set. The method tries to fit the best parameters to achieve a Gaussian bell with each of the **rescaled x** attributes, producing an x matrix we will refer as **transformed x**, plotted in Figure 6.

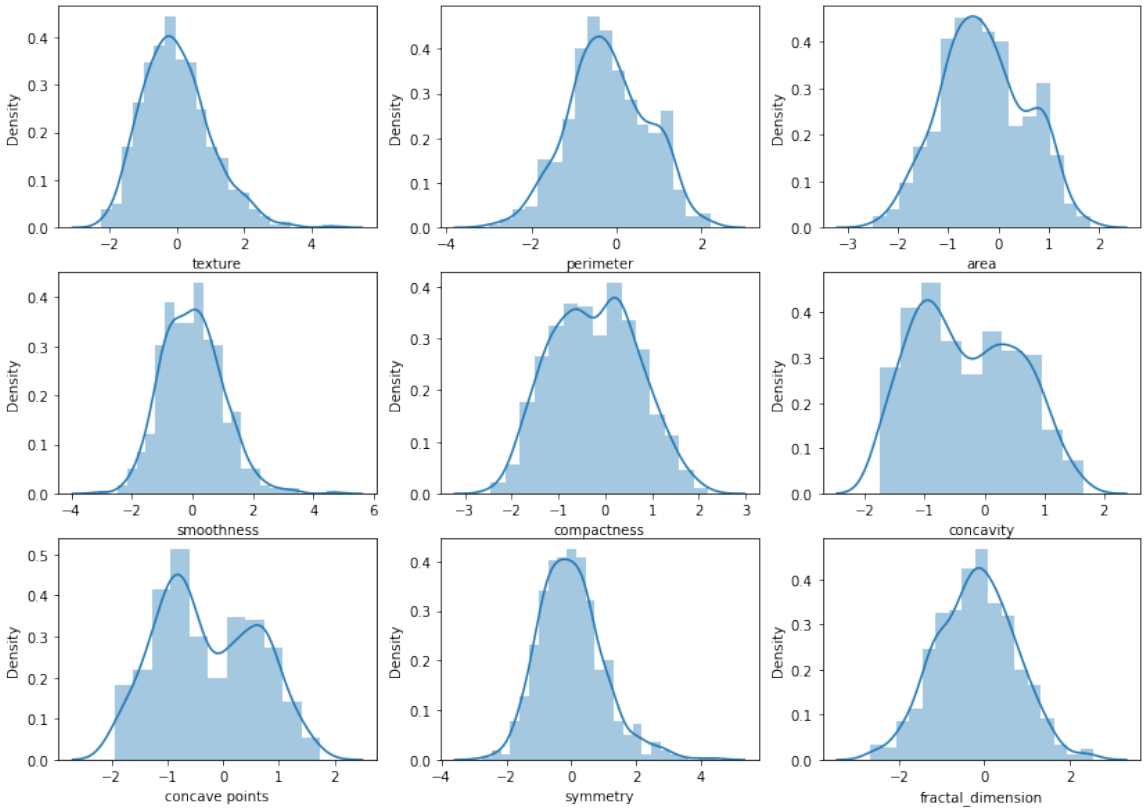

Figure 6: Transformed x: Standardized and Yeo-Johnson transformed data attributes.

If we compare Figure 5 and Figure 6, we can see that not every column was transformed by Yeo-Johnson's method, because not all attributes seemed to have a significant enough skewness to be transformed, such is the case of `texture`, `smoothness` and `symmetry` columns. This was conlcuded after analysing which box plots presented a strong skewness towards some side of the other, such the top plot in Figure 7. As a result, the bottom plot in Figure 7 shows also how the **number of atypical instances was reduced by Yeo-Johnson's method**, which most of the times reduces the solution search process in a Machine Learning algorithm.

With **transformed x** matrix ready, we compute now the Cross Validation to this data in the code snippet below.

```
num_folds = 10
kfold = KFold(n_splits=num_folds, shuffle=True)
model = LogisticRegression(solver='lbfgs', max_iter=100)
results = cross_val_score(model, transformedX, y, cv=kfold)
```

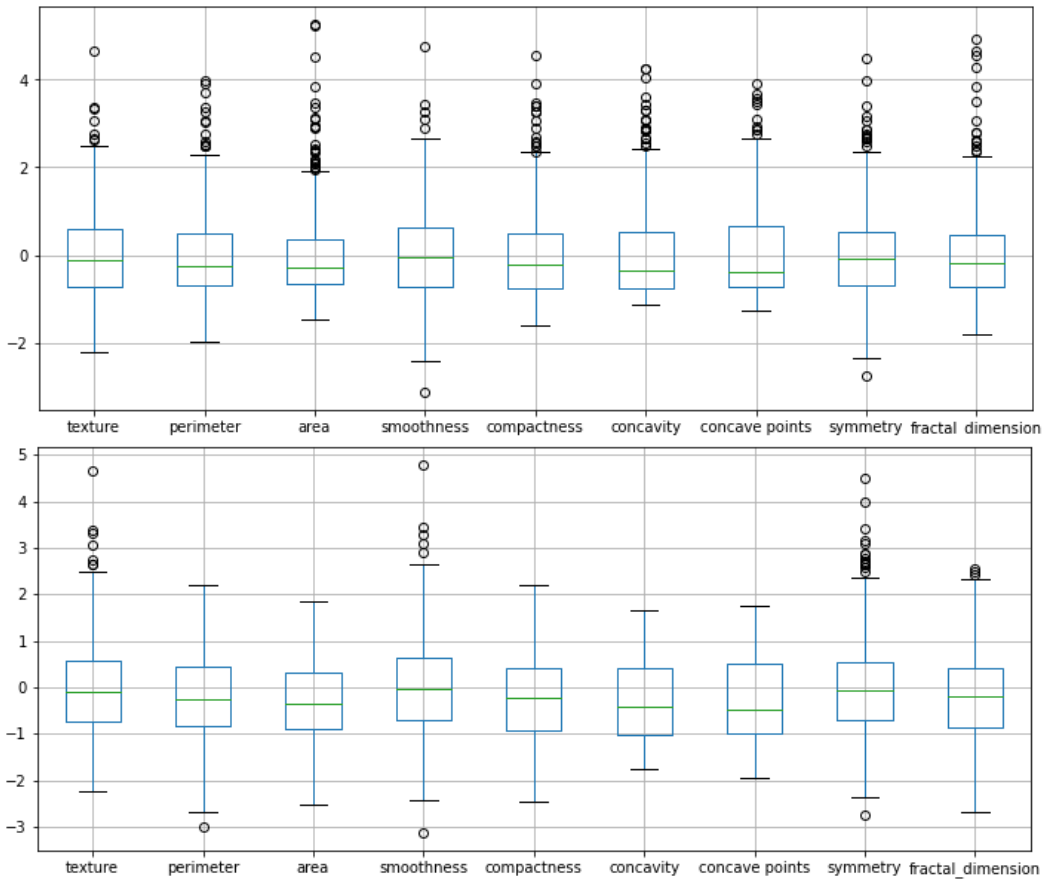

Figure 7: Rescaled x (top) Transformed x (bottom): box plots.

```
acc = results.mean() * 100 # percentage of accuracy
stdev = results.std() * 100
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}%")
```

Unfortunately, the **accuracy did not increase from last method, since now we get 91.56% with a stdev of 0.0271**, but it does increase with respect to the method applied in raw data. The algorithm also converges in around 100 iterations, so this method also involves a reduction from computation time when compared to raw data.

4 Resampling Methods: Cross Validation with Repetitions

4.1 Raw Data

Next up, the Cross Validation with Repetitions consists in, as its name implies, the performance of repeated Cross Validation methods. But, why would it help to repeat this method over and over? Because Cross Validation itself chooses the part that will be test **randomly**, and thus, by repeating it several times, we will reach further testing of our data with different combinations of N sets. We go back to applying this method to the raw data matrix **x**, as in the code snippet below.

```
num_folds = 10
num_repeated = 5
repeatedkfold = RepeatedKFold(n_splits=num_folds, n_repeats=num_repeated)
model = LogisticRegression(solver='lbfgs', max_iter=3500)
```

```
results = cross_val_score(model, x, y, cv=repeatedkfold)
acc = results.mean() * 100 # percentage of accuracy
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

Did it help? Well, the **accuracy in prediction was raised from 90.51% to 90.65% and the standard deviation from 0.0335 to 0.0370 with respect to Cross Validation with raw data**, which is somehow a growth in accuracy when raw data is used, but it is so small that it is likely to be insignificant and variate because of the randomness of the method. The fact that the standard deviation grew also a small bit is probably due to the fact that we simply have now more combinations of sets. Thus, at least with this data set, the repeated Cross Validation seems to be redundant and not significantly helpful when it comes to using raw data.

4.2 Standardized Data

Taking back the **rescaled x** data matrix, which is plotted in Figure 5, now it is time to see if the small increase in accuracy with raw data becomes a more significant increase in accuracy when applied to the standardized **x** matrix. We perform this Cross Validation with Repetitions below.

```
num_folds = 10
num_repeated = 5
repeatedkfold = RepeatedKFold(n_splits=num_folds, n_repeats=num_repeated)
model = LogisticRegression(solver='lbfgs', max_iter=100)
results = cross_val_score(model, rescaledX, y, cv=repeatedkfold)
acc = results.mean() * 100
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

The actual result was **the decrease to 93.85% prediction accuracy and 0.0326 standard deviation decrease as well**, and this decrease may mean the model without the repetitions was a bit over-trained and now with more sets of testing and training, we reduced this 94% accuracy to a 93% due to the bigger amount of combinations.

4.3 Standardization and Yeo-Johnson Transformed Data

Now that we know what and why use Yeo-Johnson method, we can go back to the **transformed x** matrix, which includes standardization and Yeo-Johnson transform the the raw **x**.

```
num_folds = 10
num_repeated = 5
repeatedkfold = RepeatedKFold(n_splits=num_folds, n_repeats=num_repeated)
model = LogisticRegression(solver='lbfgs', max_iter=100)
results = cross_val_score(model, transformedX, y, cv=repeatedkfold)
acc = results.mean() * 100
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

Interestingly, the same thing occurred when we compared the Cross Validation with raw data to Cross Validation with Standard and Yeo's method, **we go from 90% to 91.60% accuracy**, while the highest (93.85%) accuracy was reached with only the Standard Transformation. The fact that the same behaviour from the simple Cross Validation and Cross Validation with Repetitions happens tells us that maybe **the iterative process in still redundant and gives the same results with or without the repetitions**.

5 Resampling Methods: Division by Percentage

5.1 Raw Data

Now we will validate our Logistic Regression model with another technique called Division by Percentage, which consists in splitting the data set **x** into two parts only, but how big are these parts? The two parts will be divided following a given percentage, often 33% for the training and the remaining 67% for the testing or validation. This splitting is also random, but done once.

Let's use the raw data matrix **x** and apply this method to the trained model. Such code snippet is below.

```
test_size = 0.33
seed = 1
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = test_size,
                                                    random_state = seed)
model = LogisticRegression(solver='lbfgs', max_iter=3500)
model.fit(x_train, y_train)
results = model.score(x_test, y_test)
acc = results * 100
print(f"accuracy: {acc:.2f}%")
```

The output in this case was **87.3% of prediction accuracy**, which is quite lower compared to the two methods seen before, with or without raw data. This is probably due to the method itself: it is just splitting and testing **once**, whereas the previous two methods involve inner iterations in the method itself. This method **lowers the accuracy level but also decreases the computation time and complexity**.

5.2 Standardized Data

Let us locate ourselves again with the standardized x matrix, called **rescaled x** and plotted in Figure 5, and try to apply this method to see how the model is validated: let's see the accuracy with **rescaled x**.

```
test_size = 0.33
seed = 1
x_train, x_test, y_train, y_test = train_test_split(rescaledX, y, test_size = test_size,
                                                    random_state = seed)
model = LogisticRegression(solver='lbfgs', max_iter=100)
model.fit(x_train, y_train)
results = model.score(x_test, y_test)
acc = results * 100
print(f"accuracy: {acc:.2f}%")
```

The results here are very worth noticing. Compared to the method with raw data, this method with standardized data **increases the prediction accuracy from 87.3% to 93.1%, which is the biggest percentage increase so far**. With a reduced computation time such as the one needed for a Division by Percentage Validation, we can achieve similar accuracy results to the ones by Cross Validation and Cross Validation Repeated, which may be a crucial conclusion if our model needs large amounts of data and the computation time is an important issue.

5.3 Standardization and Yeo-Johnson Transformed Data

Changing our data back to **transformed x**, which is transformed by Standardization and Yeo-Johnson's Method, let's take a look at what the validation accuracy is with the code snippet below.

```
x_train, x_test, y_train, y_test = train_test_split(transformedX, y, test_size =
                                                    test_size, random_state = seed)
```

```
model = LogisticRegression(solver='lbfgs', max_iter=100)
model.fit(x_train, y_train)
results = model.score(x_test, y_test)
acc = results * 100
print(f"accuracy: {acc:.2f}%")
```

Once again, we see that by taking **transformed x** as the input data, **the prediction accuracy increases (91.49%), but not as much as with the standardized data**. This seems to be quite interesting, since the Yeo-Johnson transforms the set to be more similar to a Gauss bell, which is the data that a Machine Learning model such as Logistic Regression expects and works best with. This certainly will be addressed in the final conclusions of this document, as it seems to happen in all methods.

6 Resampling Methods: Division by Percentage with Repetitions

6.1 Raw Data

As mentioned above, the Division by Percentage method operates only one split and test over the input data, but takes the 33% randomly. Now, taking advantage of that randomness, there exists another variation of this method called Division by Percentage with Repetitions. This method validates our model by taking 33% for testing and 67% of our data as training, but several times, since the split is done randomly. This method looks similar to what a simple Cross Validation does, because it makes N iterations to shuffle the part taken as test around our data set. The code snippet below shows the implementation of a Division by Percentage with Repetitions method with raw data matrix **x**.

```
test_size = 0.33
n_splits = 10
kfold = ShuffleSplit(n_splits=n_splits, test_size = test_size)
model = LogisticRegression(solver='lbfgs', max_iter=3500)
results = cross_val_score(model, x, y, cv=kfold)
acc = results.mean() * 100
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

With raw data, we get **a prediction accuracy of 91.38% and a std deviation of 0.016**, which means that if we perform the same method of Division by Percentage over and over (10 times in this case), the accuracy of the model with respect to the correct and known **y** increases from **87% to around 91%** using raw data, which is the same 91% we get from Division by Percentage with **transformed x**, but by doing so simply with raw data **x**. Thus, we can say that Division by Percentage with Repetitions applied to raw data has the same output as a Simple Division by Percentage done with Standardized and Yeo-Johnson transformations.

6.2 Standardized Data

Taking back the Standradized matrix **rescaled x**, we can apply the Repeated Division by Percentage:

```
test_size = 0.33
n_splits = 10
kfold = ShuffleSplit(n_splits=n_splits, test_size = test_size)
model = LogisticRegression(solver='lbfgs', max_iter=100)
results = cross_val_score(model, rescaledX, y, cv=kfold)
acc = results.mean() * 100
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

Similarly, using the standard matrix we get **93.67% of prediction accuracy and a standard deviation of 0.0094**, which is the smallest deviation we have got in any previous method, showing that

the accuracy results in each repetition do not vary a lot throughout the 10 iterations. This can mean that the repetitions may be unnecessary, which is somehow shown also in the accuracy itself, because from raw data and standardized data we only increased around 2%, whereas other methods applied to standard data show a bigger increase.

6.3 Standardization and Yeo-Johnson Transformed Data

We can apply this method of validation now with **transformed x**, which includes Standard and Yeo-Johnson transfoms, to see if redundancy is also suggested:

```
test_size = 0.33
n_splits = 10
kfold = ShuffleSplit(n_splits=n_splits, test_size = test_size)
model = LogisticRegression(solver='lbfgs', max_iter=100)
results = cross_val_score(model, transformedX, y, cv=kfold)
acc = results.mean() * 100
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

Interestingly, we got **accuracy of 91.97% and standard deviation of 0.0018**, which is almost the same accuracy result we got with this same method but with raw data, but slightly bigger this time. This is a strange behaviour, since we were noticing that with **transformed x** the accuracy in prediction grew a bit more than just 0.59%. This might show that this method done over and over is not really being helped in accuracy by the transformations, but these just reduce the computing time from 3500 iteration to 100 iterations in the model convergence itself.

7 Resampling Methods: Leave One Out Cross Validation

7.1 Raw Data

Finally we have reached the last validation method in the scope of this document: Leave One Out Cross Validation. What this means is, in iteration 1, what it does is take just one column as test data and the remaining columns as training data, and so on. Depending on the number of columns (N), the number of iterations there will be. We basically take all data values as training except one, instead of a percentage, which goes to test. The standard deviation will be higher, because we will be comparing one data value against its predicted value for an error estimation. Let's see how this performs with raw data **x**:

```
loocv = LeaveOneOut()
model = LogisticRegression(solver='lbfgs', max_iter=3500)
results = cross_val_score(model, x, y, cv=loocv)
acc = results.mean() * 100
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

Surprisingly, this method took over 12 seconds to output the result, which is quite noticeable compared to all the other methods that take at best 1 second to output the calculations. Either way, we get **accuracy in prediction of 90.69% and a standard deviation of 0.29** which is similar to Cross Validation, Cross Validation Repeated and Division by Percentage Repeated, but done in much more time, whic might look as a disadvantage of this method, but quite logical due to the amount of iterations depending on the data set columns.

7.2 Standardized Data

Taking back the Standradized matrix **rescaled x** in Figure 5, we can apply the Leave One Out CV:

```
loocv = LeaveOneOut()
model = LogisticRegression(solver='lbfgs', max_iter=100)
results = cross_val_score(model, rescaledX, y, cv=loocv)
acc = results.mean() * 100
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

Not surprisingly, the **accuracy increased 94.20% and the standard deviation decreased to 0.23**, which is quite the same increase from raw to standard data we have seen in all the previous methods. But now, the computation time is not so large as this time it took around 2 seconds, which confirms in a more exaggerated way that **the standardization of data reduces computation time since it makes the Logistic Regression converge faster in less iterations**.

7.3 Standardization and Yeo-Johnson Transformed Data

Changing our data back to **transformed x**, which is transformed by Standardization and Yeo-Johnson's Method, let's take a look at what the validation accuracy is with the code snippet below and the Leave One out CV method.

```
loocv = LeaveOneOut()
model = LogisticRegression(solver='lbfgs', max_iter=100)
results = cross_val_score(model, transformedX, y, cv=loocv)
acc = results.mean() * 100
stdev = results.std()
print(f"accuracy: {acc:.2f}% std: {stdev:.2f}")
```

This result was almost exactly the same as the one with raw data **x**, where now the **accuracy is 91.74% with a standard deviation of 0.27**, but slightly increased and done also in 100 iterations (2 seconds), compared to the 3500 iterations that raw data required. Thus, the process done using **rescaled x** shows a bigger accuracy percentage, which has been happening with all methods.

8 Overall Conclusions

All the accuracy percentages we discussed might look at this point confusing and hard to synthesize, that is why all the accuracy throughout the different methods are plotted in Figure 8.

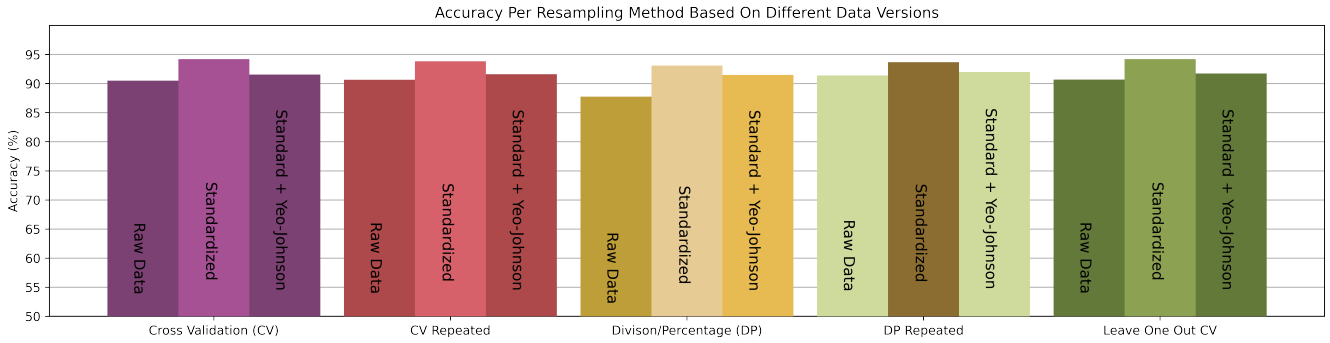

Figure 8: Prediction accuracy mean results by validation method

Now we can easily conclude that, throughout every validation method, the lowest accuracy was reached when using raw data, followed by when the method used Standardized and Yeo-Johnson transformed data, and the highest accuracy was reached when using only Standardized data. The fastest but lowest accuracy can be reached with Division by Percentage method, while the slowest method was Leave One Out Cross Validation, due to the iterations depending on the amount of columns of the data set.

Nevertheless, the accuracy in prediction using all these methods is quite high in general, surrounding the values of 87%-94%. This might look as too high even, and most probably will appear as an over-trained model when new input data is provided outside of the provided database and the accuracy decreases. I could be mistaken however, and the new input data set might as well result in similar accuracy levels, meaning that the model was not over-trained but actually extremely precise, which could be the case since Kaggle's data sets are very curated and maintained. And if this is true, the reason behind this model's precision (high accuracy) might rely on what we see in Figure 4: **almost all attributes present a correlation coefficient higher or equal to 0.5**, meaning attributes in **this data base are strongly positively linearly correlated, resulting in a Logistic Regression model that shows these levels of accuracy**.

The most notorious outcome of this method comparison seen in Figure 8 is how the **Standardization of raw data increased significantly the prediction accuracy of a model while reducing the convergence time**, compared to the model adjusted to raw data. This is due to the fact that raw data has **attribute ranges that differ greatly from other attributes'**, which **increases the searching space for the Machine Leraning model applied**, making the iterative method take longer to converge to the optimized solution. After Standardization of data, **the ranges of all attributes are similar, and thus the searching space is reduced, which makes the iterative method converge faster and all attributes have an equal influence on the model calculation**.

However, one thing that indeed surprises is the accuracy level with a Standard and Yeo-Johnson transformation: the purely Standardized data set decreases the computation time in all methods compared to the method applied with raw data, but **with a Standard and Yeo-Johnson transformation the accuracy is increased compared to raw data but decreased compared to only Standardized data**. This is surprising because, in theory, Machine Learning algorithms such as Logistic Regression actually expect data centered in $\mu = 0$ and $\sigma = 1$ (Standardized) and that resembles a Gaussian Bell distribution, which is exactly what Yeo-Johnson's method does to data. But why is the accuracy not the highest with this two transforms that deliver data as expected by Logistic Regression? Let's look at Figure 9.

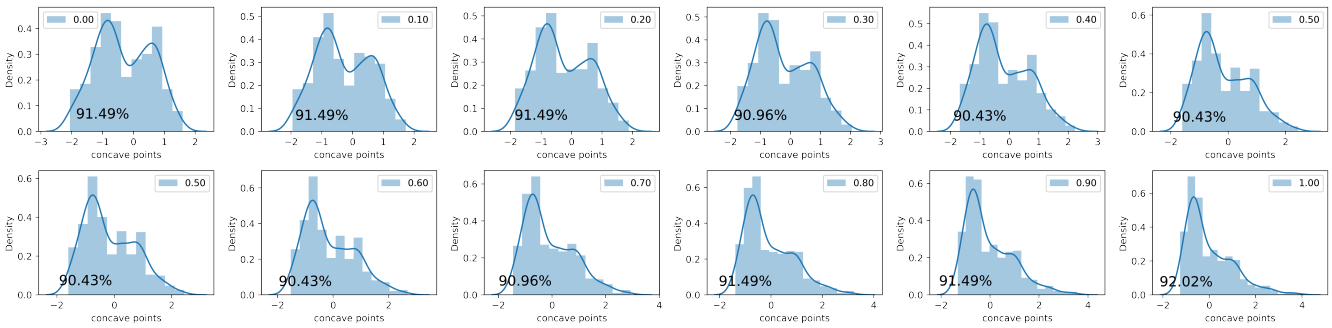

Figure 9: Lambda custom setting for Yeo-Johnson Transformation applied to one column.

The answer to this might rely on the *lambda* values that Yeo-Johnson's method fits to each column. By looking at Figure 9, where a sample column, in this case `concave points`, is taken to suffer a custom lambda setting each time in order to cover the range from 0 to 1 and then calculate the new accuracy of

the model with this transform, we can see that there is one value of this lambda that increase the accuracy percentage (percentage shown in the lower left corner of each plot in Figure 9) of the **overall** set of data columns, which is exactly a lambda of 1.00, which represent a no transform value that leaves the column with the same distribution shape as in Figure 5. If we set this column to a transformation with a lambda of 1.0, we increase the overall accuracy to 92.02%, whereas the fitted lambda value of the method (0.11) shows an overall accuracy of 91.49%. Thus, the conclusion might be that Yeo-Johnson method can increase the accuracy of the model, but some manual changes to the calculated lambdas are needed.

To finish, the conclusion for this data set specifically would be that the fastest method is Division by Percentage, with the data version being the Standardized data set, so that the model can converge in a much lesser amount of iterations. But Cross Validation seems to do a better job at the controlled but still random sets of data, producing the highest accuracy with Standardized data.

References

[1] Adam Hayes. *Correlation.* URL: `https://www.investopedia.com/terms/c/correlation.asp`.

[2] UCI Machine Learning. *Breast Cancer Wisconsin (Diagnostic) Data Set.* URL: `https://www.kaggle.com/uciml/breast-cancer-wisconsin-data`.

[3] The Pandas Development Team. *Pandas Documentation.* URL: `https://pandas.pydata.org/docs/`.

[4] Michael Waskom. *seaborn.distplot.* URL: `https://seaborn.pydata.org/generated/seaborn.distplot.html`.

[5] Michael Waskom. *seaborn.heatmap.* URL: `https://seaborn.pydata.org/generated/seaborn.heatmap.html?highlight=heatmap#seaborn.heatmap`.