labklasteryzacjaii

May 31, 2025

1 Volve G&G Dataset

The "Volve" oil field is an offshore oil and gas field located in the North Sea, approximately 190 kilometers west of Stavanger, Norway. It was discovered in 1993 and production began in 2008. The field is owned and operated by Equinor (formerly known as Statoil), which is one of the largest energy companies in the world. The field has estimated recoverable reserves of around 190 million barrels of oil equivalent, and produces mainly crude oil. Volve is a relatively small field compared to some of the other offshore fields in the North Sea, but it is still an important asset for Equinor and for Norway's overall oil and gas industry.

[1]:
```
from IPython.display import Image
Image(url="https://www.norskpetroleum.no/factpages/3420717.jpg", width=700)
```

[1]:
```
<IPython.core.display.Image object>
```

2 What kind of well-logs we will use?

- **NPHI** (neutron porosity): It is used to estimate the amount of fluids (usually hydrocarbons) contained in the rock formation by measuring the amount of neutron radiation that is emitted by the rock and reflected back to the sensor. This is important in the oil and gas industry to determine the potential productivity of a reservoir.
- **RHOB** (bulk density): It is used to determine the weight of the rock formation per unit volume. This is important for calculating the overall density of the rock formation and understanding its mechanical properties.
- **GR** (gamma ray): It is used to measure the amount of natural radiation that is emitted by the rock formation. This information can be used to identify certain types of rock formations, such as shale, and to estimate the amount of organic matter present in the formation.
- **RT** (resistivity): It is used to measure the electrical resistance of the rock formation to the flow of electric current. This information can be used to determine the presence and quality of fluids within the rock formation.
- **PEF** (photoelectric factor): It is used to measure the amount of X-ray radiation that is absorbed by the rock formation. This information can be used to identify certain types of rock formations, such as sandstone, and to estimate the amount of organic matter present in the formation.

- **CALI** (caliper): It is used to measure the diameter of the borehole. This information is important for determining the correct size of tools to be used for further measurements and for ensuring the stability of the borehole.
- **DT** (compressional travel time): It is used to measure the time it takes for a compressional (P-wave) sound wave to travel a known distance through the rock formation. This information can be used to determine the rock formation's mechanical properties, such as its porosity and permeability.

3 Step 1 - importing data from Excel file and plotting the logs

[2]:
```
import pandas as pd
```

3.1 Task 1

Rread Excel file into a pandas dataframe ans save it to the variable df using function pd.read_excel

exaple: variable = pd.read_excel('file_name.xlsx')

prinf df - what is the index of your data?

[5]:
```
df = pd.read_excel('well_subset.xlsx')
print(df.head())

print("Index of the DataFrame:")
print(df.index)
```

```
    DEPTH    NPHI    RHOB      GR      RT     PEF    CALI       DT
0  2800.0  0.1425  2.4629  3.2562  1.7704  8.0126  8.5782  68.2803
1  2800.1  0.1416  2.4583  3.2575  1.7734  8.0124  8.6250  68.4759
2  2800.2  0.1436  2.4548  2.8439  1.8059  8.0316  8.6250  68.6693
3  2800.3  0.1454  2.4504  2.4479  1.8467  8.0325  8.6249  68.7748
4  2800.4  0.1509  2.4438  3.0292  1.9006  7.9983  8.5781  68.8805
Index of the DataFrame:
RangeIndex(start=0, stop=8001, step=1)
```

3.2 Task 2

Set column DEPTH as the index using the following syntax

variable = variable.set_index('column_name')

Print the first 10 rows using the syntax

variable.head(number of rows)

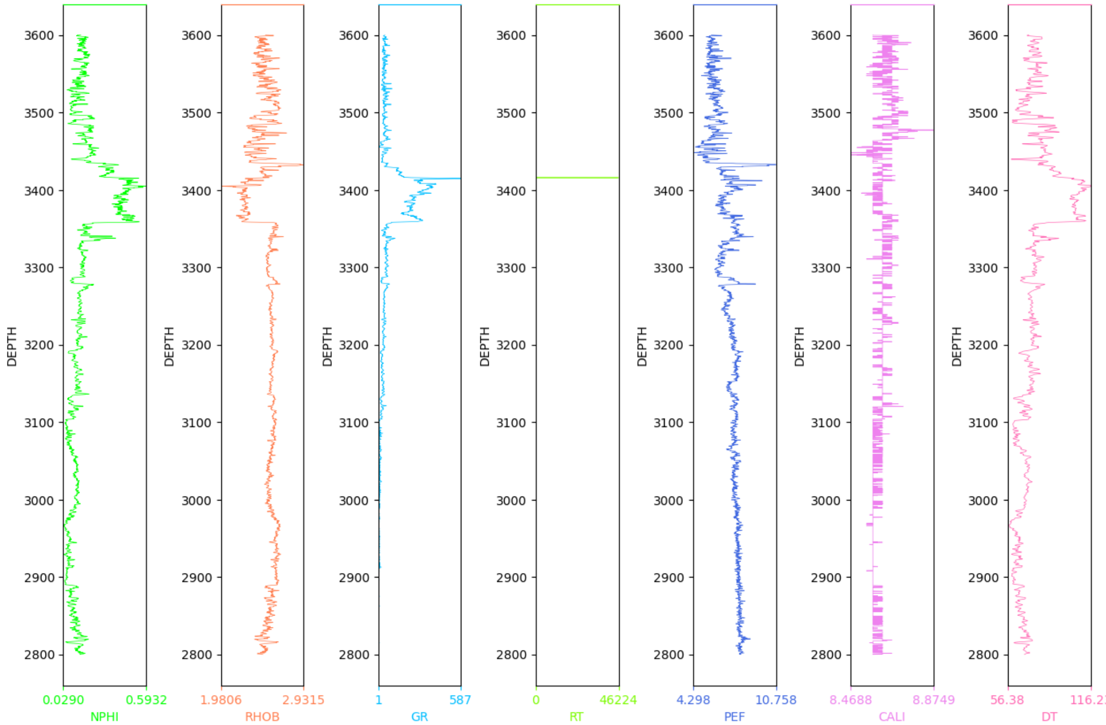
[6]:
```
df = df.set_index('DEPTH')
print(df.head(10))
```

```
          NPHI    RHOB      GR      RT     PEF    CALI       DT
DEPTH
```

```
2800.0  0.1425  2.4629  3.2562  1.7704  8.0126  8.5782  68.2803
2800.1  0.1416  2.4583  3.2575  1.7734  8.0124  8.6250  68.4759
2800.2  0.1436  2.4548  2.8439  1.8059  8.0316  8.6250  68.6693
2800.3  0.1454  2.4504  2.4479  1.8467  8.0325  8.6249  68.7748
2800.4  0.1509  2.4438  3.0292  1.9006  7.9983  8.5781  68.8805
2800.5  0.1549  2.4343  2.9266  1.9117  7.9443  8.5782  68.9858
2800.6  0.1573  2.4217  3.4017  1.8806  7.9051  8.6250  69.0042
2800.7  0.1632  2.4096  3.7842  1.8404  7.9249  8.6250  69.0204
2800.8  0.1679  2.4020  3.1949  1.8093  7.9677  8.6250  69.0371
2800.9  0.1703  2.4000  2.6821  1.7874  7.9796  8.6250  69.2040
```

3.3 Task 3 import library for ploting or install if needed. Specify the list of colors and plot the logs.

[7]:
```python
import matplotlib.pyplot as plt

# Set up the plot axes using the number of columns for this put ncols value␣
↪len(df.columns), set figsize to 15x10
fig, axs = plt.subplots(ncols=len(df.columns), figsize=(15,10),␣
↪gridspec_kw=dict(wspace=0.9))

# Define a list of 7 colors - see website https://matplotlib.org/stable/gallery/
↪color/named_colors.html for ccolor list
colors = ["lime", "coral",␣
↪"deepskyblue","lawngreen","royalblue","violet","hotpink"]

# Write a loop over the all columns in dataframe, specify linewidth to 0.5
for i, col in enumerate(df.columns):
    axs[i].plot(df.iloc[:,i], df.index, color=colors[i], linewidth=0.5)
    axs[i].set_xlabel(col)
    axs[i].xaxis.label.set_color(colors[i])
    axs[i].set_xlim(df.iloc[:,i].min(), df.iloc[:,i].max())
    axs[i].set_ylabel("DEPTH")
    axs[i].tick_params(axis='x', colors=colors[i])
    axs[i].spines["top"].set_edgecolor(colors[i])
    axs[i].title.set_color(colors[i])
    axs[i].set_xticks([df.iloc[:,i].min(), df.iloc[:,i].max()])
```



3.4 Task 4 Can you notice something strange? What with RT curve? write your comment

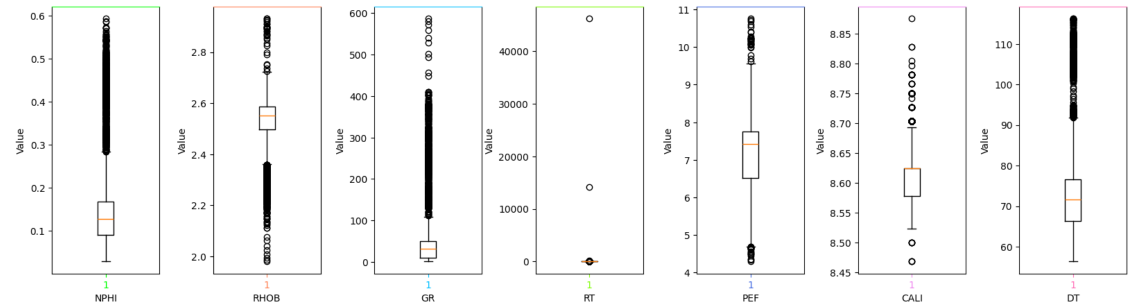
The RT curve (resistivity log) looks very unusual compared to the other logs. While other curves have continuous lines with clear variations along the depth, the RT curve seems to be almost flat, showing a single constant value for the majority of the well. This suggests that the resistivity data is either missing, constant, or not measured properly in this well interval.

3.5 Task 5 Check outliers - what is the outlier??

An outlier is a data point that is significantly different from the other values in a dataset.

[8]:
```
# Set up the plot axes similar as before but use the gig size 20,5
fig, axs = plt.subplots(ncols=len(df.columns), figsize=(20,5), gridspec_kw=dict(wspace=0.5))

# write a loop function similar as abowe

for i, col in enumerate(df.columns):
    axs[i].boxplot(df[col])
    axs[i].set_xlabel(col)
    axs[i].set_ylabel("Value")
    axs[i].tick_params(axis='x', colors=colors[i])
    axs[i].spines["top"].set_edgecolor(colors[i])
```

```
    axs[i].title.set_color(colors[i])
```



We need to remove weird high values of RT, but what to do with this observations? We can replace them using interpolation from the nearest samples

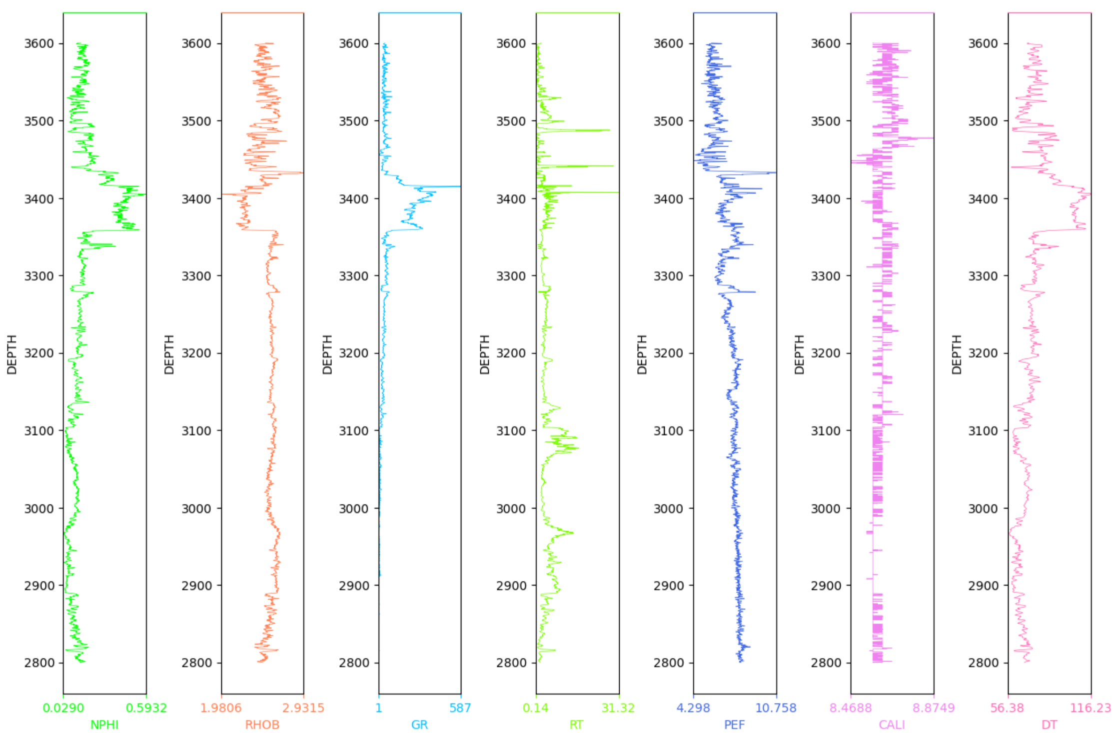
3.6 Task 6 Correct RT curve

[9]:
```
import numpy as np

# Replace RT log values greater than 100 with NaN
df.loc[df['RT'] > 100, 'RT'] = np.nan

# Interpolate NaN values based on nearest values
df['RT'] = df['RT'].interpolate(method='nearest')

# Print the resulting DataFrame
print(df)
```

```
          NPHI    RHOB       GR      RT     PEF    CALI       DT
DEPTH
2800.0  0.1425  2.4629   3.2562  1.7704  8.0126  8.5782  68.2803
2800.1  0.1416  2.4583   3.2575  1.7734  8.0124  8.6250  68.4759
2800.2  0.1436  2.4548   2.8439  1.8059  8.0316  8.6250  68.6693
2800.3  0.1454  2.4504   2.4479  1.8467  8.0325  8.6249  68.7748
2800.4  0.1509  2.4438   3.0292  1.9006  7.9983  8.5781  68.8805
…          …       …        …       …       …       …        …
3599.6  0.1289  2.5771  44.3674  2.3147  6.1787  8.5781  70.1850
3599.7  0.1259  2.5490  43.5794  2.3004  5.9839  8.5781  70.3162
3599.8  0.1312  2.5246  44.6774  2.2336  5.8875  8.5781  70.5137
3599.9  0.1365  2.5003  45.4844  2.1827  5.7913  8.5781  70.7711
3600.0  0.1470  2.4950  47.8596  2.1170  5.7226  8.5784  71.3462

[8001 rows x 7 columns]
```

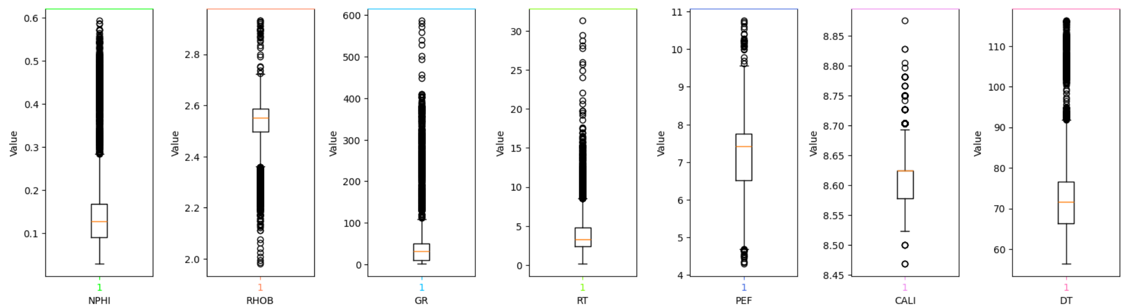
3.6.1 See if it helped! We will visualize data again

[10]:
```
fig, axs = plt.subplots(ncols=len(df.columns), figsize=(15,10),␣
 ↪gridspec_kw=dict(wspace=0.9))

colors = ["lime", "coral",␣
 ↪"deepskyblue","lawngreen","royalblue","violet","hotpink"]

for i, col in enumerate(df.columns):
    axs[i].plot(df.iloc[:,i], df.index, color=colors[i], linewidth=0.5)
    axs[i].set_xlabel(col)
    axs[i].xaxis.label.set_color(colors[i])
    axs[i].set_xlim(df.iloc[:,i].min(), df.iloc[:,i].max())
    axs[i].set_ylabel("DEPTH")
    axs[i].tick_params(axis='x', colors=colors[i])
    axs[i].spines["top"].set_edgecolor(colors[i])
    axs[i].title.set_color(colors[i])
    axs[i].set_xticks([df.iloc[:,i].min(), df.iloc[:,i].max()])
```



After correcting the RT (resistivity) curve by removing unrealistically high values and interpolating the missing data, the RT log now looks much more realistic and continuous. The extreme outliers have disappeared, and the curve shows natural variability along the depth, similar to the other logs. This correction makes the RT data usable for further geological interpretation and analysis.

[11]:
```python
fig, axs = plt.subplots(ncols=len(df.columns), figsize=(20,5),␣
↪gridspec_kw=dict(wspace=0.5))


for i, col in enumerate(df.columns):
    axs[i].boxplot(df[col])
    axs[i].set_xlabel(col)
    axs[i].set_ylabel("Value")
    axs[i].tick_params(axis='x', colors=colors[i])
    axs[i].spines["top"].set_edgecolor(colors[i])
    axs[i].title.set_color(colors[i])
```

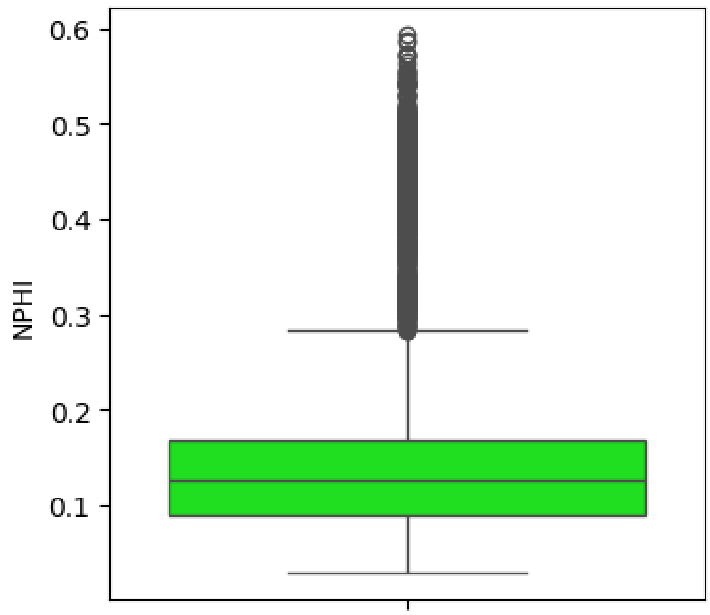


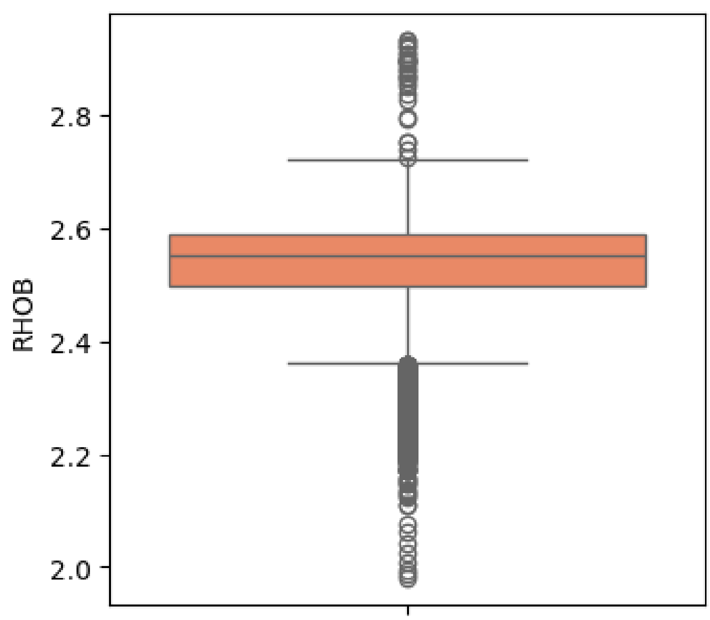
After correcting the RT curve and visualizing the data again with boxplots, we can see that the RT (resistivity) log no longer contains extreme outliers. All values now fall within a reasonable range, and the distribution of RT is much more similar to the other logs. The data looks consistent, with no abnormal spikes or unrealistically high values.

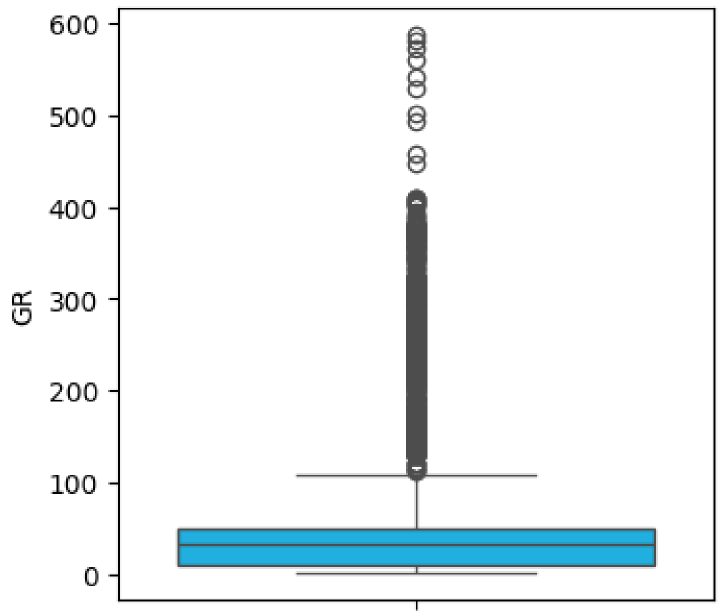
3.7 Task 7 check the outliers visualization using seaborn library

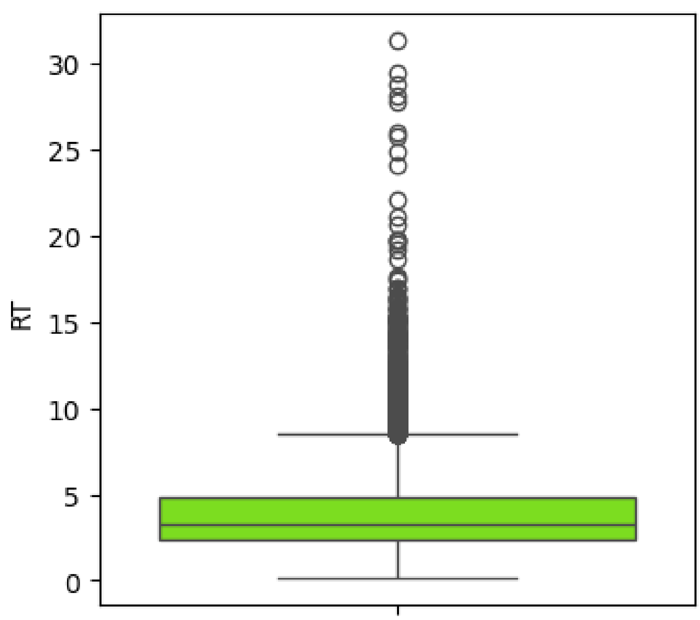
check the seaborn website: https://seaborn.pydata.org/examples/index.html write in the comment 3 visualizations that you can use in your daily tasks

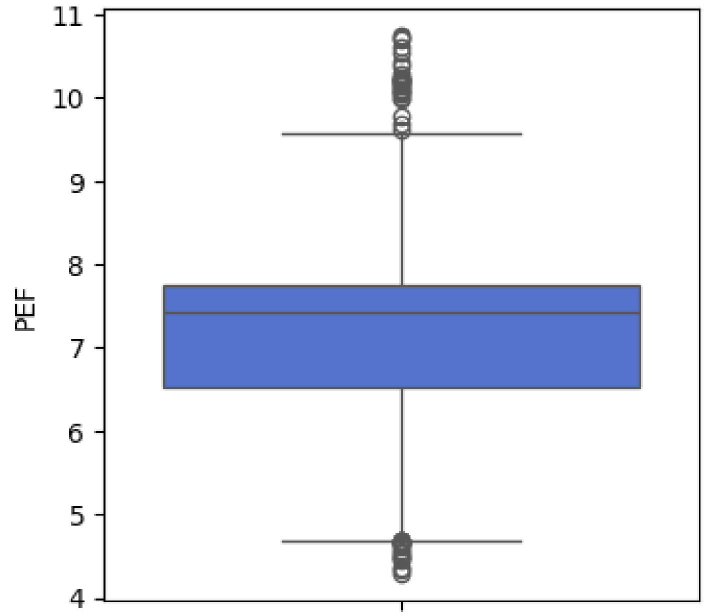
[12]:
```python
import seaborn as sns


for i, col in enumerate(df.columns):
    fig, ax = plt.subplots(figsize=(4, 4))
    sns.boxplot(y=df[col], color=colors[i], orient='v')
```

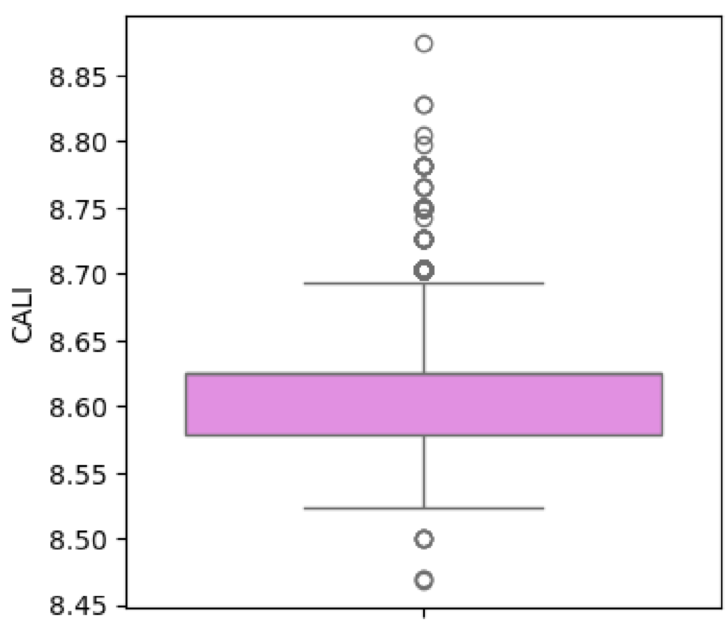

0.6
0.5
0.4
0.3
0.2
0.1
NPHI

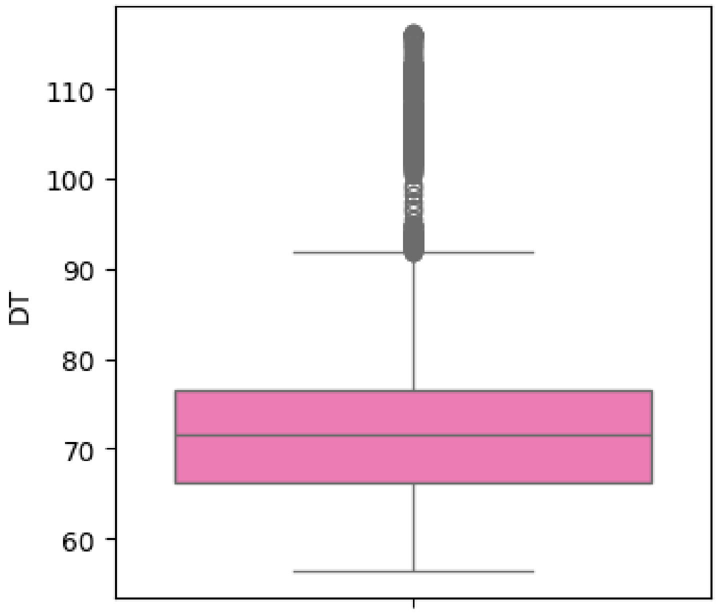

2.8
2.6
2.4
2.2
2.0
RHOB


600
500
400
300
200
100
0
GR


30
25
20
15
10
5
0
RT


11
10
9
8
7
6
5
4
PEF


8.85
8.80
8.75
8.70
8.65
8.60
8.55
8.50
8.45
CALI



Three Seaborn visualizations useful in daily data analysis tasks:

boxplot – for visualizing the distribution of data and identifying outliers.

heatmap – for visualizing correlations or missing data in a matrix format.

pairplot – for visualizing pairwise relationships in a dataset, useful for detecting trends and patterns across multiple variables.

3.8 Task 8 - Remove outliers that are 3 standard deviations from the mean in window of 100 samples

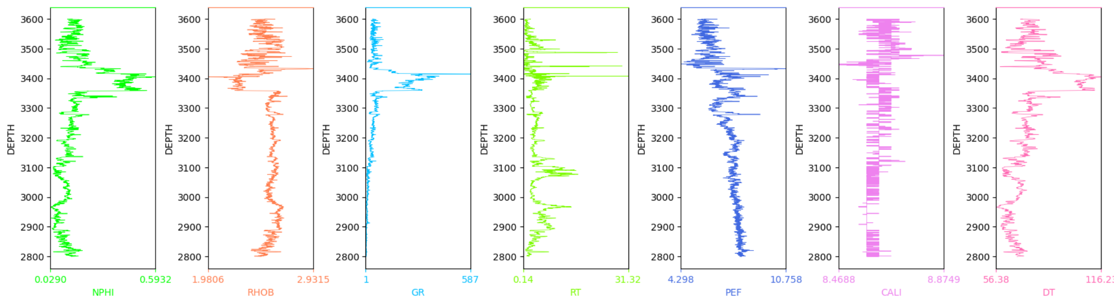
[13]:
```
# set window size to 100
window_size = 100

# calculate z-scores for each column using rolling window
z_scores = (df - df.rolling(window_size).mean()) / df.rolling(window_size).std()

# remove rows where any z-score is greater than 3
df_noout = df[(np.abs(z_scores) < 3).all(axis=1)]

# print cleaned dataframe
print(df_noout)
```

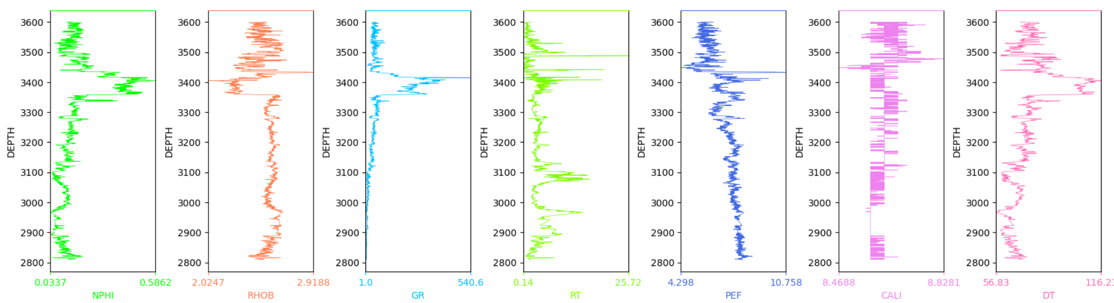
```
          NPHI    RHOB      GR      RT     PEF    CALI      DT
DEPTH
2809.9  0.1427  2.4589  4.2370  2.0991  7.9082  8.6249  70.0120
2810.0  0.1402  2.4615  4.1468  2.0325  7.9031  8.5782  69.9705
```

```
2810.1  0.1393  2.4696   4.2389  1.9359  7.9372  8.6015  69.8849
2810.2  0.1348  2.4811   3.6711  1.8266  7.9944  8.5781  69.7992
2810.3  0.1309  2.4924   2.8638  1.7334  8.0531  8.5781  69.7205
…          …      …       …        …       …       …       …
3599.0  0.1207  2.4931  41.2866  1.4664  5.8496  8.7030  71.5955
3599.1  0.1200  2.5050  42.4855  1.6046  6.0435  8.6718  71.1964
3599.2  0.1240  2.5236  42.9226  1.7066  6.2858  8.6252  70.8162
3599.9  0.1365  2.5003  45.4844  2.1827  5.7913  8.5781  70.7711
3600.0  0.1470  2.4950  47.8596  2.1170  5.7226  8.5784  71.3462

[6618 rows x 7 columns]
```

[14]:
```python
# Set up the plot axes
fig, axs = plt.subplots(ncols=len(df.columns), figsize=(20,5),
 ↪gridspec_kw=dict(wspace=0.5))


# make a for loop for visualization using df dataset

for i, col in enumerate(df.columns):
    axs[i].plot(df.iloc[:,i], df.index, color=colors[i], linewidth=0.5)
    axs[i].set_xlabel(col)
    axs[i].xaxis.label.set_color(colors[i])
    axs[i].set_xlim(df.iloc[:,i].min(), df.iloc[:,i].max())
    axs[i].set_ylabel("DEPTH")
    axs[i].tick_params(axis='x', colors=colors[i])
    axs[i].spines["top"].set_edgecolor(colors[i])
    axs[i].title.set_color(colors[i])
    axs[i].set_xticks([df.iloc[:,i].min(), df.iloc[:,i].max()])
plt.show()

# Set up the plot axes
fig, axs = plt.subplots(ncols=len(df.columns), figsize=(20,5),
 ↪gridspec_kw=dict(wspace=0.5))

# make a for loop for visualization using df_noout dataset
for i, col in enumerate(df_noout.columns):
    axs[i].plot(df_noout.iloc[:,i], df_noout.index, color=colors[i],
 ↪linewidth=0.5)
    axs[i].set_xlabel(col)
    axs[i].xaxis.label.set_color(colors[i])
    axs[i].set_xlim(df_noout.iloc[:,i].min(), df_noout.iloc[:,i].max())
    axs[i].set_ylabel("DEPTH")
    axs[i].tick_params(axis='x', colors=colors[i])
    axs[i].spines["top"].set_edgecolor(colors[i])
    axs[i].title.set_color(colors[i])
    axs[i].set_xticks([df_noout.iloc[:,i].min(), df_noout.iloc[:,i].max()])
```

```
plt.show()
```





4 Step 2 - Exploratory Data Analysis

EDA (Exploratory Data Analysis) is the process of analyzing and visualizing data in order to extract insights, patterns, and trends. It is typically one of the first steps in data analysis and is used to gain a better understanding of the data and its characteristics. EDA can help identify outliers, missing values, and any other issues with the data that may need to be addressed before further analysis. It can also help in selecting appropriate statistical methods and models for data analysis. EDA involves using a range of techniques such as histograms, scatter plots, box plots, and correlation matrices to explore the data visually and identify relationships between variables. EDA is an important part of data science and plays a crucial role in the data analysis process.

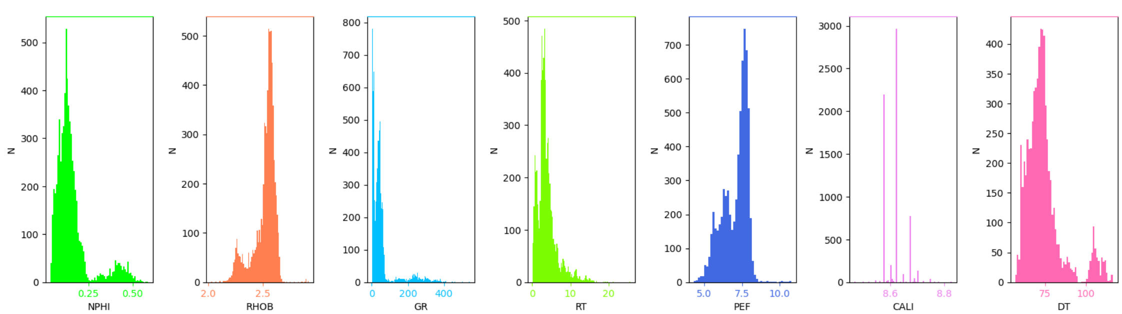
4.1 Histograms

Histograms are graphical representations of the distribution of data. They display the frequency distribution of a variable by creating a set of contiguous and non-overlapping intervals (or bins) along the range of the variable and then plotting the count or proportion of observations that fall within each bin. By examining a histogram, one can see the central tendency, variability, and shape of the distribution of the data. Additionally, it can also help identify any outliers or unusual patterns in the data. Overall, histograms are useful for understanding the distribution of a variable and gaining insight into the underlying patterns and characteristics of the data.

Optimal bin size

Freedman-Diaconis rule is a method for determining the bin width of a histogram in statistical data analysis. The rule uses the interquartile range (IQR) of the data and the total number of samples to calculate an appropriate bin width. The bin width is important because it determines the smoothness of the histogram and can affect the interpretation of the data. The Freedman-Diaconis rule aims to create a histogram with sufficient detail to reveal the underlying distribution of the data while avoiding oversmoothing or undersmoothing. It is considered a robust method for determining bin width because it is less sensitive to outliers than other methods such as Sturges' rule or Scott's rule. The Freedman-Diaconis rule has been widely adopted in various fields, including economics, environmental science, and medical research.

[15]:
```python
# Set up the plot axes as before, use size 20,5
fig, axs = plt.subplots(ncols=len(df.columns), figsize=(20,5), gridspec_kw=dict(wspace=0.5))

#write a loop
for i, col in enumerate(df_noout.columns):

    # Calculate the bin size using the Freedman-Diaconis rule
    # specify the quartiles
    q75, q25 = np.percentile(df_noout[col], [75, 25])
    # calculate iqr (interquartile range)
    iqr = q75 - q25
    n = len(df_noout[col])
    # calculate the width as h equals to double iqr divided by the cube root of the number of observation -> n = len(df[col])
    h =  2 * iqr / (n ** (1/3))
    bins = int((df_noout[col].max() - df_noout[col].min()) / h)

    axs[i].hist(df_noout[col], bins=bins, color=colors[i])
    axs[i].set_xlabel(col)
    axs[i].set_ylabel("N")
    axs[i].tick_params(axis='x', colors=colors[i])
    axs[i].spines["top"].set_edgecolor(colors[i])
    axs[i].title.set_color(colors[i])
```



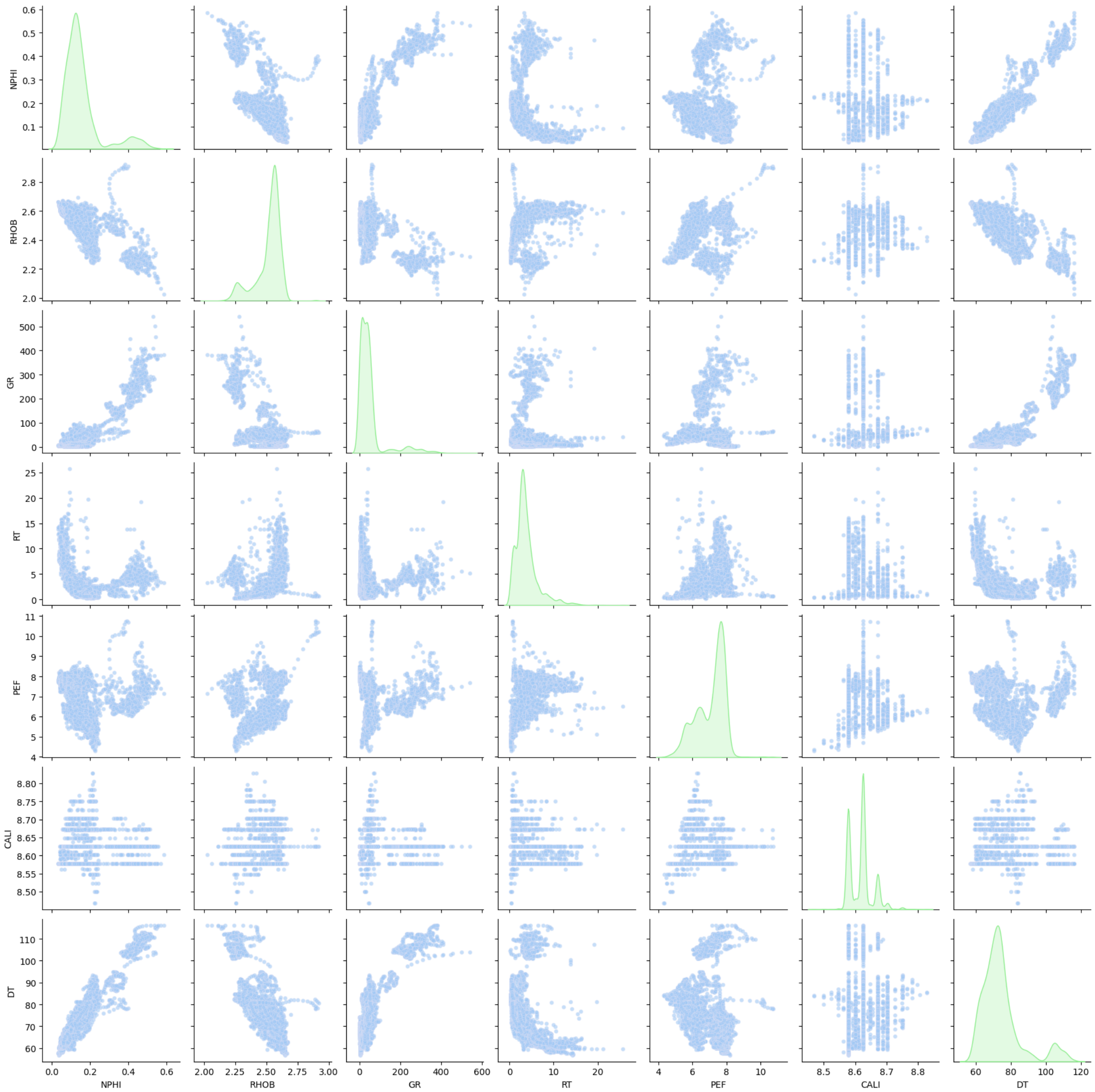
4.2 Pairplot

A pairplot is a graphical tool used in data analysis and visualization that creates pairwise scatterplots and histograms for a given dataset. It is a useful method for exploring the relationship between multiple variables in a dataset. Each scatterplot in the pairplot shows the relationship between two variables in the dataset, while the histograms show the distribution of each variable individually.

By examining the pairplot, we can gain insights into the relationships between variables in the dataset. We can identify variables that are strongly correlated, positively or negatively, as well as variables that are not correlated at all. We can also see the distribution of each variable, including whether they are normally distributed or skewed, and identify any outliers. Pairplots can be useful for identifying potential patterns or trends in the data, as well as for detecting any issues with the data, such as missing or erroneous values. Overall, pairplots are a useful tool for exploratory data analysis and for gaining a better understanding of the relationships between variables in a dataset.

[16]:
```
cols = ['NPHI', 'RHOB', 'GR', 'RT', 'PEF', 'CALI', 'DT']

# Define colors for histograms and scatters
hist_color = 'lightgreen'
scatter_color = 'lavender'

# Set the color palette
sns.set_palette("pastel")

# Create pairplot with different colors for histograms and scatters put as vars␣
↪your column list, and use diag_kind as 'kde', specify size s as 20
sns.pairplot(df_noout, vars=cols, diag_kind='kde',
             plot_kws = {'alpha': 0.6, 's': 20, 'edgecolor': scatter_color},
             diag_kws = {'color': hist_color})
```

[16]: <seaborn.axisgrid.PairGrid at 0x22d7eea6250>



Some variables are strongly correlated, e.g., NPHI and RHOB show a clear negative correlation, NPHI and DT show a clear positive correlation,GR and DT show a clear positive correlation

The pairplot helps to quickly see relationships, trends, and any remaining anomalies in the logs.

4.3 Correlation matrix

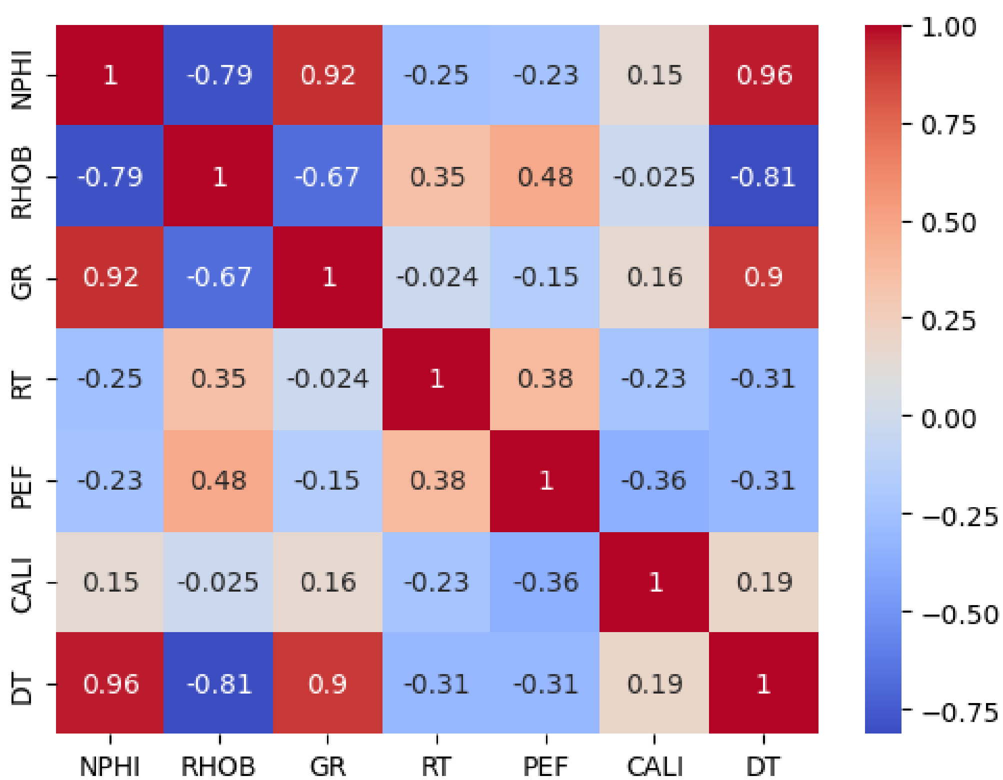
Correlation matrix is a table that displays the correlation coefficients between different variables in a dataset. It is commonly used in statistics and data analysis to identify patterns and relationships between variables.

The correlation coefficient is a statistical measure that ranges from -1 to 1, indicating the strength and direction of the linear relationship between two variables. A value of 1 indicates a perfect positive correlation, meaning that when one variable increases, the other variable increases proportionally. A value of -1 indicates a perfect negative correlation, meaning that when one variable increases, the other variable decreases proportionally. A value of 0 indicates no correlation, meaning that there is no relationship between the variables.

The correlation matrix is a square matrix where the diagonal contains the correlation coefficient between each variable and itself, which is always 1. The upper and lower triangles of the matrix contain the correlation coefficients between each pair of variables, with duplicates reflected across the diagonal. A correlation matrix can be visualized as a heat map, where the color of each cell represents the magnitude of the correlation coefficient. Correlation matrices are commonly used in data analysis, machine learning, and other applications to identify relationships between variables, detect multicollinearity, and perform feature selection.

[17]:
```
# Define the features to include in the correlation matrix
cols = ['NPHI', 'RHOB', 'GR', 'RT', 'PEF', 'CALI', 'DT']

# Calculate the correlation matrix using the following synthax␣
 ↪dataframe[columsn].corr()
corr_matrix = df_noout[cols].corr()

# Set the color palette
sns.set_palette("pastel")

# Create a heatmap of the correlation matrix, set annot to True
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
```

[17]: <Axes: >



There is a very strong positive correlation between NPHI and GR (0.92), as well as between NPHI and DT (0.96), and between GR and DT (0.90).

There is a strong negative correlation between NPHI and RHOB (-0.79), and also between DT and RHOB (-0.81), which is expected in many geological contexts.

RHOB shows a moderate positive correlation with RT (0.35) and PEF (0.48).

CALI (caliper) is only weakly correlated with other logs, suggesting it measures a different property (borehole size rather than rock properties).

5 Step 3 - data normalization

Data normalization, also known as feature scaling, is the process of transforming data into a common scale or range in order to facilitate data analysis and improve the performance of machine learning algorithms. Normalization is important because many machine learning algorithms are sensitive to the scale and distribution of input features, and may perform poorly or inaccurately if the features are not on a similar scale.

Normalization involves rescaling the features of a dataset to have a mean of 0 and a standard deviation of 1, or scaling the features to a range between 0 and 1. The normalization method used depends on the specific data and the requirements of the analysis or algorithm being used. Common methods of normalization include Min-Max scaling, Z-score normalization, and Log transformation.

Min-Max scaling involves scaling the features to a range between 0 and 1, where the minimum value of the feature is transformed to 0 and the maximum value is transformed to 1. Z-score normalization involves transforming the features so that they have a mean of 0 and a standard deviation of 1, which can be accomplished by subtracting the mean from each value and then dividing by the standard deviation. Log transformation is another normalization technique used for data that is highly skewed or has a wide range of values, and involves applying a logarithmic function to the data to transform it into a more normal distribution.

Overall, data normalization is an important preprocessing step in data analysis and machine learning, as it can help to improve the accuracy and performance of models and algorithms, reduce overfitting, and ensure that features are on a similar scale.

5.1 3.1 - Transform Resitivity Log to log scale

Resistivity data in well logs is typically measured in ohm-meters, and the values can span several orders of magnitude, making it difficult to visualize and analyze the data directly. To address this issue, resistivity data is often transformed using a logarithmic scale, which compresses the data into a more manageable range.

The logarithmic scale is a nonlinear scale that compresses large values into a smaller range, while expanding small values. This allows for a more accurate visualization of the data and makes it easier to identify patterns and trends. In particular, the use of logarithmic scales is useful for resistivity data because the range of resistivity values encountered in well logs can be very large, spanning several orders of magnitude.

[18]:
```
df_noout = df_noout.copy()
df_noout['RT_log'] = np.log10(df_noout['RT'])
```

[19]:
```
#check data
df_noout['RT_log']
```

[19]:
```
DEPTH
2809.9    0.322033
2810.0    0.308031
2810.1    0.286883
2810.2    0.261643
2810.3    0.238899
            …
3599.0    0.166252
3599.1    0.205367
3599.2    0.232132
3599.9    0.338994
3600.0    0.325721
Name: RT_log, Length: 6618, dtype: float64
```

5.2 3.2 - Transform data with skewed distribution

The power transform with Yeo-Johnson method is a data transformation technique used to normalize a dataset that has a skewed distribution. It is a variant of the Box-Cox transformation, which is used to normalize data that has a positive skew.

The Yeo-Johnson method is a modified version of the Box-Cox transformation that can be applied to both positively and negatively skewed data, as well as data that includes zero or negative values. It works by applying a power transformation to the data that varies based on the value of a lambda parameter, which is estimated from the data.

The Yeo-Johnson method is implemented in the PowerTransformer class of the scikit-learn library in Python. It can be used to transform a pandas DataFrame or numpy array to have a more normal distribution, which can be useful for machine learning models that assume a normal distribution of the data.

[20]:
```
from IPython.display import Image
Image(url="https://i.postimg.cc/sDRksm7T/2.png")
```

[20]: <IPython.core.display.Image object>

[21]:
```
#import necessary libraries
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import PowerTransformer
import pandas as pd

#define the columns to be transformed
```

```
numeric_features = ['NPHI', 'RHOB', 'GR', 'RT', 'PEF', 'CALI', 'DT']

#define the transformation pipeline using PowerTransformer with Yeo-Johnson␣
↪method and standardization
numeric_transformer = Pipeline(steps=[
('scaler', PowerTransformer(method='yeo-johnson', standardize=True))
])

#define the ColumnTransformer to apply the transformation pipeline to the␣
↪numeric features
preprocessor = ColumnTransformer(transformers=[
('num', numeric_transformer, numeric_features)
])

#fit and transform the data using the preprocessor
transformed_data = preprocessor.fit_transform(df_noout[numeric_features])

#convert the transformed data to a DataFrame with column names and add the␣
↪depth column from the original data
transformed_data = pd.DataFrame(transformed_data, columns=numeric_features)
transformed_data['DEPTH'] = df_noout.reset_index()['DEPTH']
transformed_data = transformed_data.set_index('DEPTH')
```

[22]:

```
transformed_data
```

[22]:

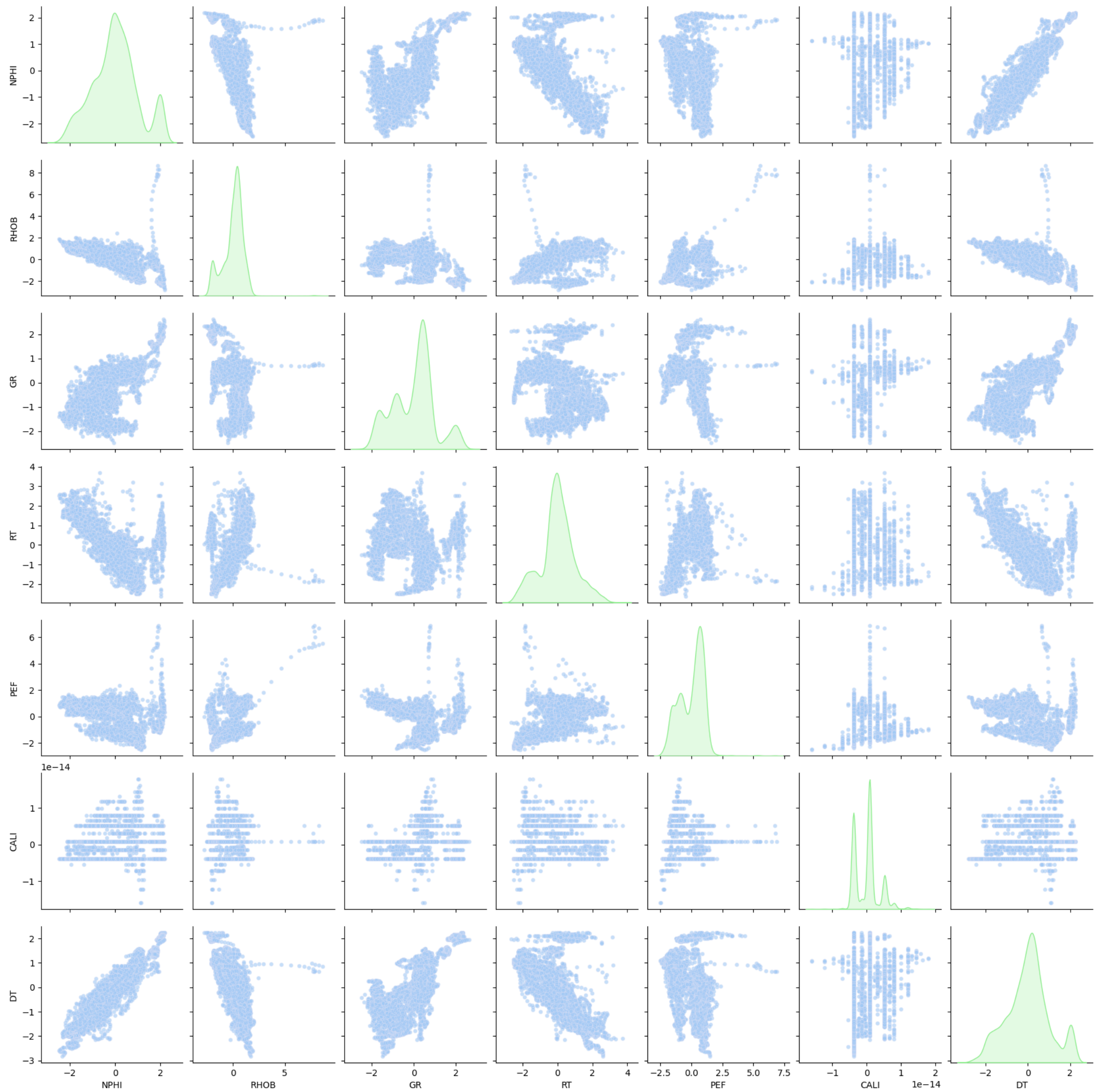
```
            NPHI      RHOB        GR        RT       PEF          CALI  \
DEPTH
2809.9  0.205930 -0.745286 -1.564246 -0.619959  1.055451  8.187895e-16
2810.0  0.167616 -0.721726 -1.581017 -0.663735  1.047754 -3.830269e-15
2810.1  0.153634 -0.647008 -1.563896 -0.728972  1.099386 -1.471046e-15
2810.2  0.082195 -0.537420 -1.674785 -0.805433  1.186874 -3.830269e-15
2810.3  0.018157 -0.425602 -1.859007 -0.873009  1.277809 -3.830269e-15
…            …         …         …         …         …             …
3599.0 -0.159180 -0.418538  0.391791 -1.080206 -1.395774  7.965850e-15
3599.1 -0.171897 -0.295911  0.417180 -0.970299 -1.216829  5.204170e-15
3599.2 -0.100195 -0.094388  0.426257 -0.892868 -0.979093  8.604228e-16
3599.9  0.109486 -0.344918  0.477658 -0.566325 -1.447658 -3.830269e-15
3600.0  0.270070 -0.399279  0.522775 -0.608354 -1.507675 -3.816392e-15

              DT
DEPTH
2809.9 -0.211915
2810.0 -0.217361
2810.1 -0.228633
2810.2 -0.239972
2810.3 -0.250432
…            …
```

```
3599.0 -0.013084
3599.1 -0.061596
3599.2 -0.108796
3599.9 -0.114460
3600.0 -0.043265

[6618 rows x 7 columns]
```

5.3 DRAW Pairplot using transformed data

[23]:
```
cols = ['NPHI', 'RHOB', 'GR', 'RT', 'PEF', 'CALI', 'DT']

# Define colors for histograms and scatters
hist_color = 'lightgreen'
scatter_color = 'lavender'

# Set the color palette
sns.set_palette("pastel")

# Create pairplot with different colors for histograms and scatters put as vars␣
↪your column list, and use diag_kind as 'kde', specify size s as 20
sns.pairplot(transformed_data, vars=cols, diag_kind='kde',
             plot_kws = {'alpha': 0.6, 's': 20, 'edgecolor': scatter_color},
             diag_kws = {'color': hist_color})
```

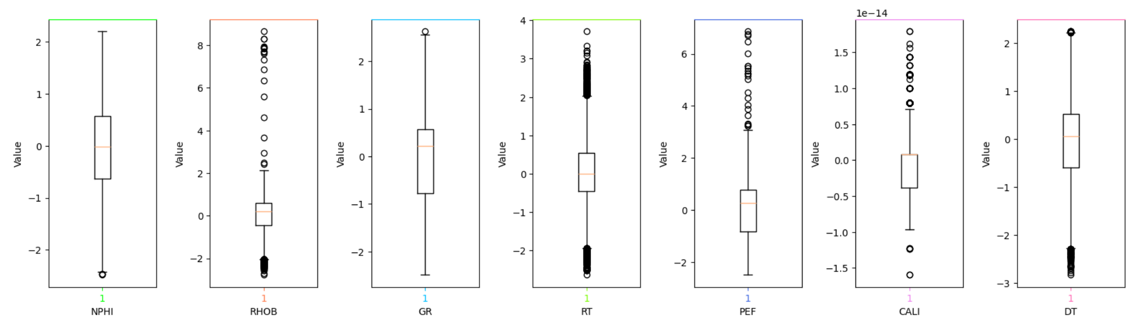
[23]:
```
<seaborn.axisgrid.PairGrid at 0x22d044a8710>
```



5.4 DRAW boxplots using transformed data

[24]:
```
# Set up the plot axes similar as before but use the gig size 20,5
fig, axs = plt.subplots(ncols=len(transformed_data.columns), figsize=(20,5),␣
 ↪gridspec_kw=dict(wspace=0.5))

# write a loop function similar as abowe

for i, col in enumerate(transformed_data.columns):
    axs[i].boxplot(transformed_data[col])
    axs[i].set_xlabel(col)
    axs[i].set_ylabel("Value")
```

```python
    axs[i].tick_params(axis='x', colors=colors[i])
    axs[i].spines["top"].set_edgecolor(colors[i])
    axs[i].title.set_color(colors[i])
```



The boxplots for the transformed data show that most variables are now centered around zero, and their spread is more balanced.

Outliers are still present, but their number is reduced compared to the original data.

The data is more symmetric, indicating that the transformation helped normalize the distributions.

6 Step 4 - final outlier removal using ML

The Isolation Forest algorithm is a type of computer program that helps find "outliers" in data. Outliers are data points that are very different from all the other data points. For example, imagine you have a list of test scores from your class, and one student got a score that is much higher or lower than everyone else. That student's score would be an outlier.

The Isolation Forest algorithm works by putting each data point in a "tree". Each tree has branches that divide the data into smaller and smaller groups. The algorithm keeps dividing the data until each point is in its own group, or until a certain number of groups have been made. This is like playing a game of "guess who" where you try to guess a character by asking yes-or-no questions, and keep dividing the characters into smaller groups until you know who the character is.

Once the data points are divided into groups, the algorithm looks at how many times each point was in a group with other points. If a point was in a group with other points many times, it is not an outlier. But if a point was in a group by itself many times, it is an outlier.

The Isolation Forest algorithm can be useful for finding outliers in data, which can be helpful in many different fields like finance, healthcare, and more.

[43]:
```python
from IPython.display import Image
Image(url="https://miro.medium.com/v2/resize:fit:1100/format:webp/
↪1*D78QLbcwXesymhquuofnOg.png")
```

[43]:
```
<IPython.core.display.Image object>
```

[26]:
```python
import pandas as pd
import numpy as np
from sklearn.ensemble import IsolationForest
```

[27]:
```python
# Define the numerical columns in your DataFrame
numeric_cols = ['NPHI', 'RHOB', 'GR', 'RT', 'PEF', 'CALI', 'DT']
```

[28]:
```python
# Instantiate the isolation forest for each column separately
isos = {}
for col in numeric_cols:
    iso = IsolationForest(n_estimators=100, max_samples='auto',␣
 ↪contamination=float(0.05), random_state=42)
    iso.fit(transformed_data[col].values.reshape(-1, 1))
    isos[col] = iso
```

[29]:
```python
# Replace the outliers with the mean of neighboring values for each column in a␣
 ↪single loop
for col in numeric_cols:
    outliers = isos[col].predict(transformed_data[col].values.reshape(-1, 1))␣
 ↪== -1
    values = transformed_data[col].values
    mean = np.mean(values[~outliers])
    for i in range(len(values)):
        if outliers[i]:
            # Replace outlier with the mean of neighboring values
            if i == 0:
                values[i] = values[i+1]
            elif i == len(values)-1:
                values[i] = values[i-1]
            else:
                values[i] = (values[i-1] + values[i+1])/2
    transformed_data[col] = values
```

[30]:
```python
# Fill any remaining NaN values with interpolated values
transformed_data[numeric_cols] = transformed_data[numeric_cols].interpolate()
```

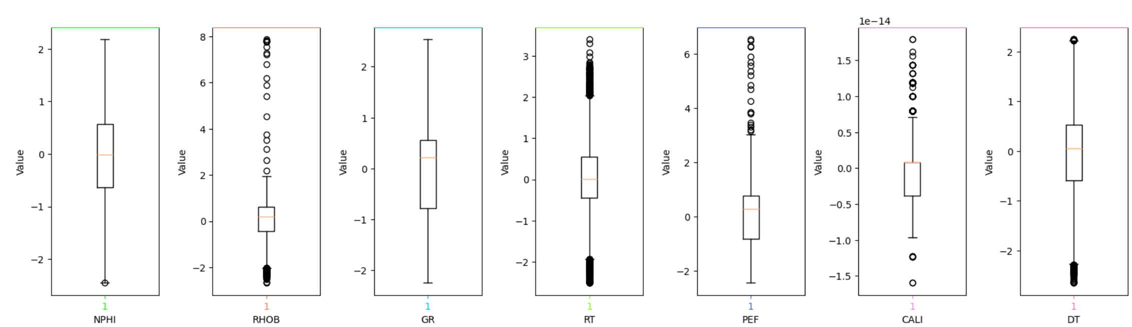
[31]:
```python
# Print the resulting DataFrame without outliers
print(transformed_data)
```

```
             NPHI      RHOB        GR        RT       PEF          CALI  \
DEPTH
2809.9   0.205930 -0.745286 -1.564246 -0.619959  1.055451  8.187895e-16
2810.0   0.167616 -0.721726 -1.581017 -0.663735  1.047754 -3.830269e-15
2810.1   0.153634 -0.647008 -1.563896 -0.728972  1.099386 -1.471046e-15
2810.2   0.082195 -0.537420 -1.674785 -0.805433  1.186874 -3.830269e-15
2810.3   0.018157 -0.425602 -1.859007 -0.873009  1.277809 -3.830269e-15
…             …         …         …         …         …             …
```

```
3599.0 -0.159180 -0.418538  0.391791 -1.080206 -1.395774  7.965850e-15
3599.1 -0.171897 -0.295911  0.417180 -0.970299 -1.216829  5.204170e-15
3599.2 -0.100195 -0.094388  0.426257 -0.892868 -0.979093  8.604228e-16
3599.9  0.109486 -0.344918  0.477658 -0.566325 -1.447658 -3.830269e-15
3600.0  0.270070 -0.399279  0.522775 -0.608354 -1.507675 -3.816392e-15

              DT
DEPTH
2809.9 -0.211915
2810.0 -0.217361
2810.1 -0.228633
2810.2 -0.239972
2810.3 -0.250432
…           …
3599.0 -0.013084
3599.1 -0.061596
3599.2 -0.108796
3599.9 -0.114460
3600.0 -0.043265

[6618 rows x 7 columns]
```

6.1 DRAW boxplots for transformed_data

[32]:
```python
# Set up the plot axes similar as before but use the gig size 20,5
fig, axs = plt.subplots(ncols=len(transformed_data.columns), figsize=(20,5),␣
 ↪gridspec_kw=dict(wspace=0.5))

# write a loop function similar as abowe

for i, col in enumerate(transformed_data.columns):
    axs[i].boxplot(transformed_data[col])
    axs[i].set_xlabel(col)
    axs[i].set_ylabel("Value")
    axs[i].tick_params(axis='x', colors=colors[i])
    axs[i].spines["top"].set_edgecolor(colors[i])
    axs[i].title.set_color(colors[i])
```



7 Step 5 - Facies analysis

Facies analysis from well logs is an important task in petroleum geology. It involves identifying and classifying the different rock types, or facies, encountered in a well. Traditionally, this has been done by geologists through visual inspection of the well logs. However, with the rise of machine learning techniques, it has become possible to automate this process.

One common approach to facies analysis is to use unsupervised clustering algorithms, such as K-means or hierarchical clustering, to group similar sections of the well log together. The goal is to identify natural groupings, or clusters, of log responses that correspond to different facies. Once the clusters have been identified, the geologist can assign each cluster to a specific facies based on their knowledge of the local geology.

To perform facies analysis using machine learning clustering, several well logs are typically collected and pre-processed. The logs are usually normalized and scaled to make them comparable, and any missing data is imputed or removed. Features, such as gamma-ray, resistivity, and porosity, are extracted from the logs and used as inputs to the clustering algorithm.

Once the features have been extracted, a clustering algorithm is applied to group the sections of the well log that have similar responses. The algorithm assigns each section to a cluster, which can be visualized on a plot. The plot can help identify any clear patterns or trends in the data and help the geologist make sense of the clustering results.

The next step is to assign each cluster to a specific facies. This is done by comparing the clustering results to the geological knowledge of the area. The geologist may use other data sources, such as core samples or outcrop data, to help identify the facies associated with each cluster.

Once the clusters have been assigned to facies, the results can be used to create a facies log. This log can be used to interpret the geology of the well and to help identify potential hydrocarbon reservoirs or other geological features.

[51]:
```
from IPython.display import Image
Image(url="https://interviewquery-cms-images.s3-us-west-1.amazonaws.com/
↪ac5da238-25ab-48ef-839a-407a7b76a167.jpg")
```

[51]:
```
<IPython.core.display.Image object>
```

7.1 K-means

is a type of unsupervised learning algorithm used for clustering data points into different groups or clusters based on the similarity of the data points. The algorithm works by iteratively assigning each data point to the nearest cluster center, and then computing the new cluster centers based on the mean of the assigned points. This process is repeated until the cluster centers no longer move significantly.

The algorithm requires the user to specify the number of clusters beforehand. The objective of the algorithm is to minimize the sum of squared distances between each data point and its assigned cluster center, which is also known as the Within-Cluster-Sum-of-Squares (WCSS) metric.

The k-means algorithm can be divided into three main steps:

1. **Initialization:** The algorithm randomly selects k data points to act as initial cluster centers.
2. **Assignment:** Each data point is assigned to the nearest cluster center based on the Euclidean distance between the point and the center.
3. **Update:** The mean of the data points assigned to each cluster is computed, and this value is used as the new cluster center.

These three steps are repeated iteratively until the cluster centers no longer move significantly, or a maximum number of iterations is reached. The final output of the algorithm is the cluster assignments of each data point.

One of the main advantages of the k-means algorithm is its simplicity and scalability, which allows it to handle large datasets efficiently. However, the algorithm is sensitive to the initial placement of the cluster centers, and may converge to suboptimal solutions. In addition, the algorithm is not effective when dealing with non-linearly separable data. Nonetheless, k-means is widely used in various applications such as image segmentation, market segmentation, and anomaly detection.

[52]:
```
from IPython.display import Image
Image(url="https://ds055uzetaobb.cloudfront.net/brioche/uploads/
 ↪y4KGN92h7r-screen-shot-2016-05-05-at-43007-pm.png?width=2000")
```

[52]: <IPython.core.display.Image object>

[33]:
```
from sklearn.cluster import KMeans
cols = ['NPHI', 'RHOB', 'GR', 'RT', 'PEF', 'CALI', 'DT']

# define features using cols
X = transformed_data[cols]
```

[34]:
```
#define model set n_clusters to 10

model = KMeans(n_clusters=10, random_state=42)
```

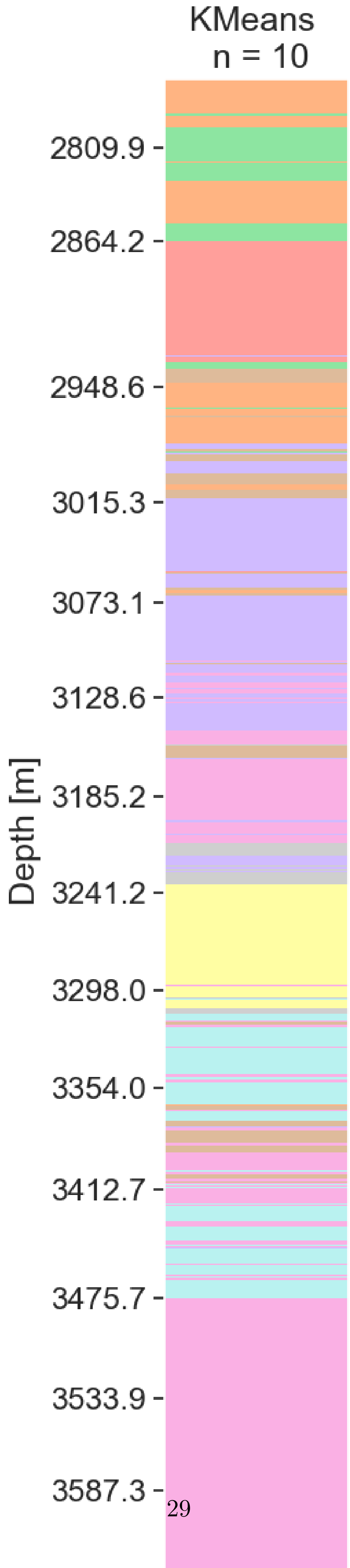
[35]:
```
# fit the model to data
y = model.fit_predict(X); y
```

[35]: array([4, 4, 4, …, 5, 5, 5])

[36]:
```
# save your data in numeric and string format
transformed_data['K10'] = y
transformed_data['K10_name'] = "Facies "+(transformed_data['K10']+1).
 ↪astype('str')
transformed_data = transformed_data.reset_index()
```

7.2 Print transformed_data what do you see?

[37]:
```
K5_graph = transformed_data.copy()
K5_graph['x'] = 1
sns.set_style("white")
sns.set(font_scale = 1.5)
plt.rcParams['xtick.major.size'] = 200
plt.rcParams['xtick.major.width'] = 40
plt.rcParams['xtick.bottom'] = True
plt.rcParams['ytick.left'] = True
plt.figure(figsize=(3,12))
ax = sns.scatterplot(data=K5_graph, x='x', y='DEPTH', hue='K10_name',␣
 ↪marker='s', s=50000, edgecolor='None', palette='pastel')
plt.tick_params(bottom=False, labelbottom=False)
plt.yticks(K5_graph['DEPTH'][::500])
plt.legend([],[], frameon=False)
plt.ylabel("Depth [m]")
plt.xlabel('')
plt.title("KMeans \n n = 10")
ax.invert_yaxis()
plt.tight_layout()
plt.rcParams.update({})
```



KMeans
n = 10
Depth [m]
2809.9
2864.2
2948.6
3015.3
3073.1
3128.6
3185.2
3241.2
3298.0
3354.0
3412.7
3475.7
3533.9
3587.3

The plot shows the clustering results from the KMeans algorithm (with n=10 clusters) applied to the well log data as a function of depth.

Each color represents a different cluster (facies), and the y-axis shows the depth in meters.

The well log has been divided into 10 segments, where each segment corresponds to a cluster with similar properties.

Some clusters (colors) cover thicker intervals, while others are more fragmented and appear only in thin layers.

[38]:
```
def score(n_clusters, data, cols):
    model = KMeans(n_clusters=n_clusters, max_iter=300, random_state=1234)
    X = transformed_data[cols]
    y = model.fit_predict(X)
    SSE = model.inertia_
    Silhouette = metrics.silhouette_score(X, y)
    CHS = metrics.calinski_harabasz_score(X, y)
    DBS = metrics.davies_bouldin_score(X, y)
    return {'SSE':SSE, 'Silhouette': Silhouette, 'Calinski_Harabasz': CHS,␣
 ↪'Davies_Bouldin':DBS, 'model':model}
```

[39]:
```
df_cluster_scorer = pd.DataFrame()
df_cluster_scorer['n_clusters'] = list(range(2, 21))
df_cluster_scorer
```

[39]:

	n_clusters
0	2
1	3
2	4
3	5
4	6
5	7
6	8
7	9
8	10
9	11
10	12
11	13
12	14
13	15
14	16
15	17
16	18
17	19
18	20

[40]:
```python
from sklearn import metrics



df_cluster_scorer['SSE'],df_cluster_scorer['Silhouette'],\
df_cluster_scorer['Calinski_Harabasz'], df_cluster_scorer['Davies_Bouldin'],\
df_cluster_scorer['model'] = zip(*df_cluster_scorer['n_clusters'].map(lambda␣
↪row: score(row, transformed_data, cols).values()))

df_cluster_scorer
```

[40]:

	n_clusters	SSE	Silhouette	Calinski_Harabasz	Davies_Bouldin	\
0	2	22230.216812	0.401670	5073.808019	1.008443	
1	3	16764.219576	0.334689	4441.972332	1.089447	
2	4	12413.770246	0.379482	4771.152971	0.939114	
3	5	9550.417545	0.397627	5146.157333	0.826371	
4	6	8381.828948	0.364676	4874.579682	0.945695	
5	7	7938.447980	0.358960	4349.901956	0.963785	
6	8	6921.304164	0.361978	4414.567504	0.872023	
7	9	6083.697134	0.358632	4507.791085	0.914323	
8	10	5443.355362	0.365196	4563.838448	0.885056	
9	11	5077.577685	0.340999	4450.259010	0.923962	
10	12	4828.615797	0.338450	4284.602147	0.917287	
11	13	4414.963018	0.333209	4346.463256	0.922982	
12	14	4258.998389	0.330302	4177.038749	0.988960	
13	15	3876.413331	0.345562	4307.392890	0.962555	
14	16	3773.995209	0.334039	4140.655879	0.992939	
15	17	3554.270053	0.329822	4146.691466	1.009258	
16	18	3419.100456	0.324435	4071.881770	1.008113	
17	19	3350.417576	0.324963	3931.389555	0.994934	
18	20	3192.980549	0.315202	3924.670354	1.013171	

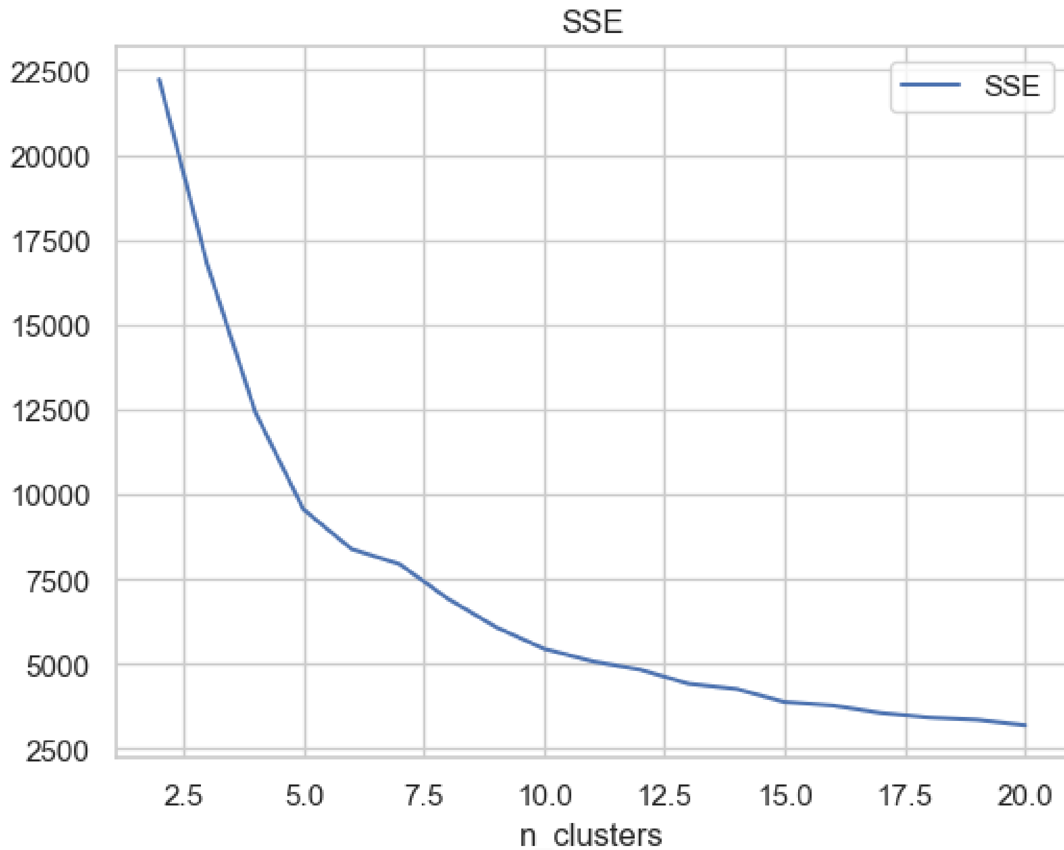
	model
0	KMeans(n_clusters=2, random_state=1234)
1	KMeans(n_clusters=3, random_state=1234)
2	KMeans(n_clusters=4, random_state=1234)
3	KMeans(n_clusters=5, random_state=1234)
4	KMeans(n_clusters=6, random_state=1234)
5	KMeans(n_clusters=7, random_state=1234)
6	KMeans(random_state=1234)
7	KMeans(n_clusters=9, random_state=1234)
8	KMeans(n_clusters=10, random_state=1234)
9	KMeans(n_clusters=11, random_state=1234)
10	KMeans(n_clusters=12, random_state=1234)
11	KMeans(n_clusters=13, random_state=1234)
12	KMeans(n_clusters=14, random_state=1234)
13	KMeans(n_clusters=15, random_state=1234)

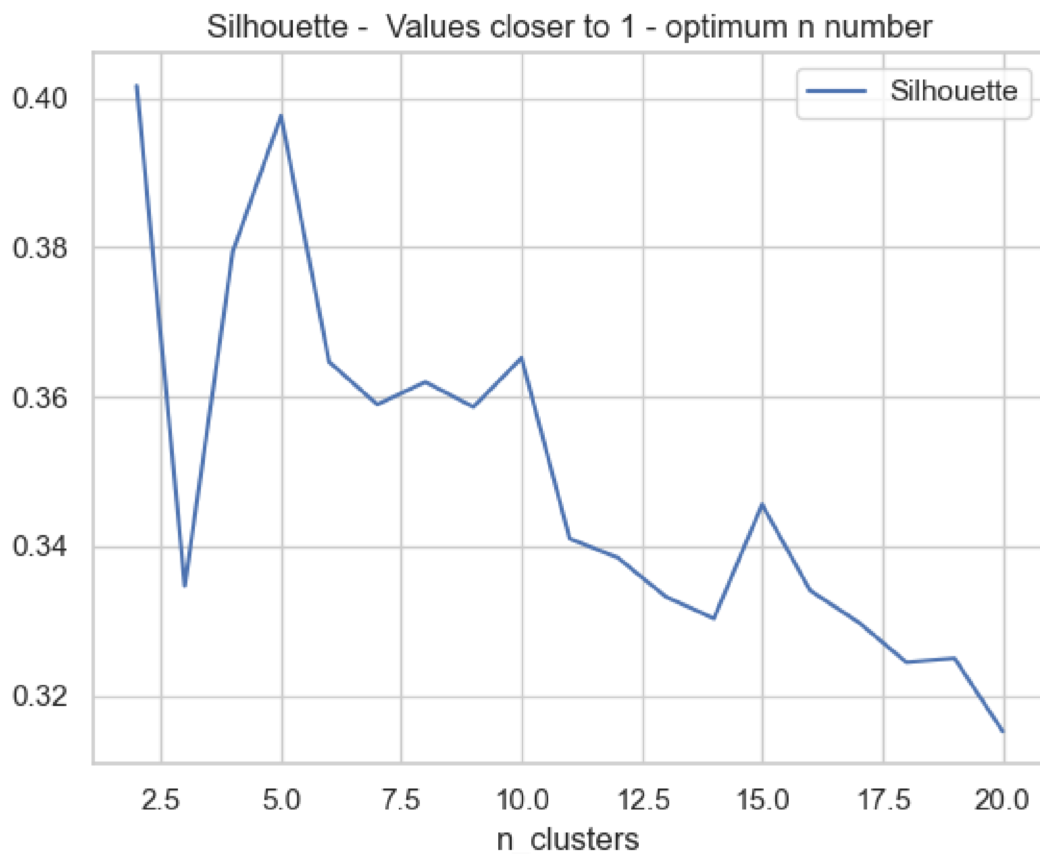
```
14  KMeans(n_clusters=16, random_state=1234)
15  KMeans(n_clusters=17, random_state=1234)
16  KMeans(n_clusters=18, random_state=1234)
17  KMeans(n_clusters=19, random_state=1234)
18  KMeans(n_clusters=20, random_state=1234)
```

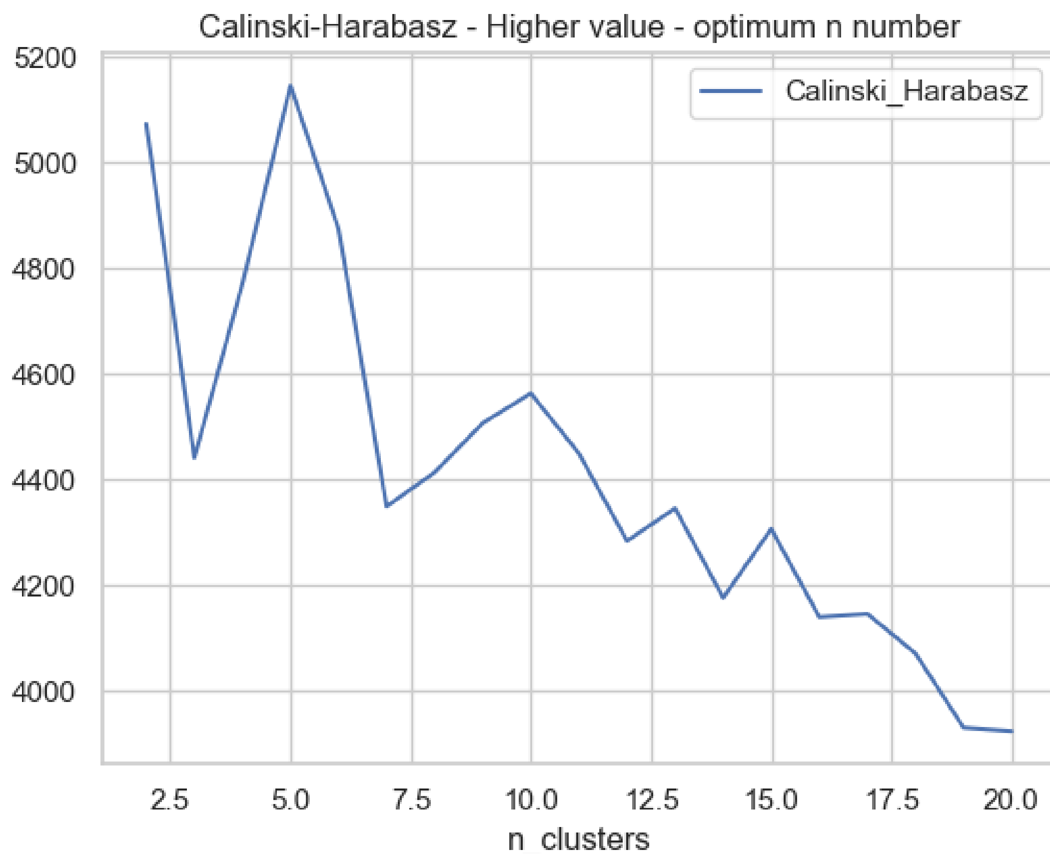
[41]:

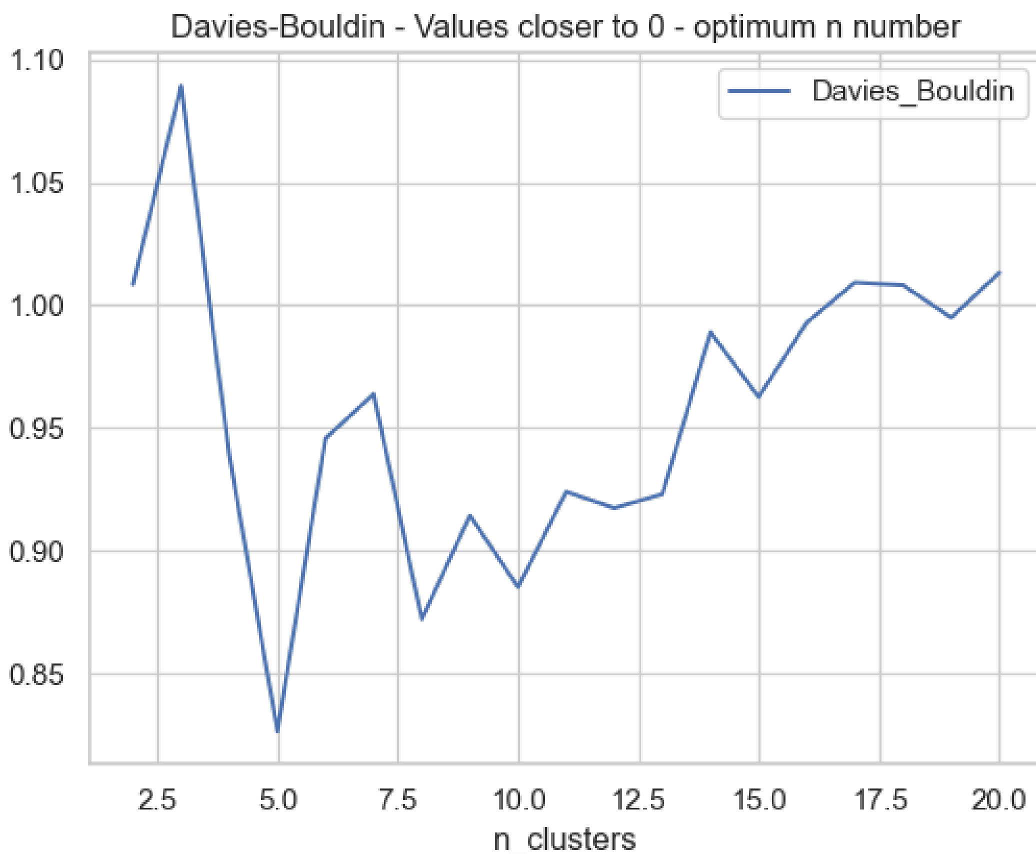
```
plt.rcParams.update({})
sns.set()
sns.set_style("whitegrid")


fig = plt.figure(figsize=(8, 6))


# Plot each graph on its own axis
df_cluster_scorer.plot.line(x='n_clusters', y='SSE', title="SSE")
df_cluster_scorer.plot.line(x='n_clusters', y='Silhouette', title="Silhouette -␣
↪ Values closer to 1 - optimum n number")
df_cluster_scorer.plot.line(x='n_clusters', y='Calinski_Harabasz',␣
↪title="Calinski-Harabasz - Higher value - optimum n number")
df_cluster_scorer.plot.line(x='n_clusters', y='Davies_Bouldin',␣
↪title="Davies-Bouldin - Values closer to 0 - optimum n number")

#plt.tight_layout()
plt.show()
```

```
<Figure size 800x600 with 0 Axes>
```


SSE
SSE
22500
20000
17500
15000
12500
10000
7500
5000
2500
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
n_clusters


Silhouette - Values closer to 1 - optimum n number
Silhouette
0.40
0.38
0.36
0.34
0.32
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
n_clusters


Calinski-Harabasz - Higher value - optimum n number
Calinski_Harabasz
5200
5000
4800
4600
4400
4200
4000
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
n_clusters



7.3 Choose optimum number of clusters and predict!

Read about metrics here: https://www.mdpi.com/1996-1073/16/1/493 and describe why did you pick this number

Based on all four metrics, 5 clusters is the optimal number:

The SSE curve forms an elbow at 5.

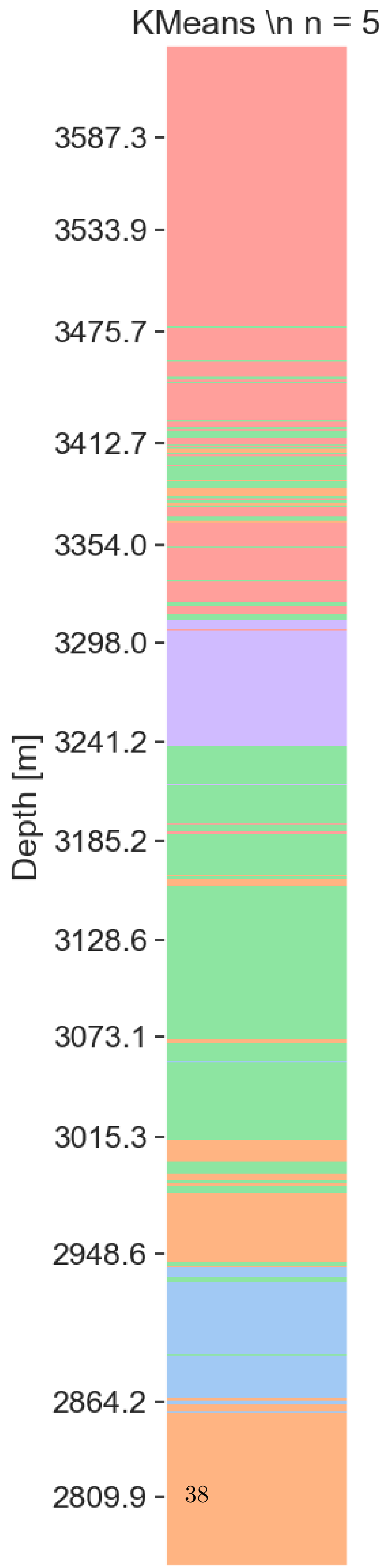
The silhouette score is still high at 5.

The Calinski-Harabasz index peaks at 5.

The Davies-Bouldin score is lowest at 5.

[42]:
```
cols = ['NPHI', 'RHOB', 'GR', 'RT', 'PEF', 'CALI', 'DT']
X = transformed_data[cols]

n_clusters = 5
model = KMeans(n_clusters=n_clusters, random_state=42)
y = model.fit_predict(X)
```

```
transformed_data['K5'] = y
transformed_data['K5_name'] = "Facies " + (transformed_data['K5']+1).
 ↪astype('str')
transformed_data = transformed_data.reset_index()

K5_graph = transformed_data.copy()
K5_graph['x'] = 1

import matplotlib.pyplot as plt
import seaborn as sns

sns.set(style="white")
sns.set(font_scale = 1.5)
plt.rcParams['xtick.major.size'] = 200
plt.rcParams['xtick.major.width'] = 40
plt.rcParams['xtick.bottom'] = True
plt.rcParams['ytick.left'] = True
plt.figure(figsize=(3,12))
ax = sns.scatterplot(
    data=K5_graph, x='x', y='DEPTH', hue='K5_name',
    marker='s', s=50000, edgecolor='None', palette='pastel'
)
plt.tick_params(bottom=False, labelbottom=False)
plt.yticks(K5_graph['DEPTH'][::500])
plt.legend([],[], frameon=False)
plt.xlabel('')
plt.ylabel('Depth [m]')
plt.title("KMeans \\n n = 5")
plt.tight_layout()
plt.show()
```



KMeans \n n = 5
Depth [m]
3587.3
3533.9
3475.7
3412.7
3354.0
3298.0
3241.2
3185.2
3128.6
3073.1
3015.3
2948.6
2864.2
2809.9

8 Compare your results with the stratigrapy

https://www.researchgate.net/publication/332441275_Estimation_of_Pore_Pressure_and_Fracture_Gradient_