# Data explorations, event detection, and introduction to statistics

**1- Load the library welly in the conda environment DrillingAnalytics and launch Jupyter Labs**

conda activate DrillingAnalytics

conda list welly

pip install welly

conda list welly

cd ~/DataScienceComodoro2023

Jupyter-Lab

**2- Load welly library in notebook**

from welly import Well

from welly import Curve

**3- Load well log in welly object: 8267\_a0801\_1996\_comp.las**

well=Well.from\_las('Data/LAS/8267\_a0801\_1996\_comp.las')

well

**4- Plot well log data**

tracks =['MD','GR', ['NPHI','RHOB','DRHO'],'DT']

well.plot(tracks=tracks)

**5- How many GR values are in the log and how many are missing?**

well.data[‘GR']

**6- What is the average of GR values?**

gr=well.data['GR']

gr.describe()

**7- What is the average of GR values in a the sand?**

gr=well.data['GR']

gr[2600:3000].describe()

**8- Determine which curves have no data, which have gaps, flatlined and if GR is below 25**

import welly.quality as quality

test = {'Each':[quality.no\_flat,

quality.no\_gaps,

quality.not\_empty

],

'GR':[quality.all\_above(25)

]}

from IPython.display import HTML

data\_qc\_table=well.qc\_table\_html(test)

HTML(data\_qc\_table)

**9- What is the proportion of NaN values in each curve?**

test = {'Each':[quality.no\_flat,

quality.no\_gaps,

quality.not\_empty,

quality.fraction\_not\_nans

],

'GR':[quality.all\_above(25)

]}

data\_qc\_table=well.qc\_table\_html(test)

HTML(data\_qc\_table)

**10- Which curves have spikes?**

test = {'Each':[quality.no\_flat,

quality.no\_gaps,

quality.not\_empty,

quality.fraction\_not\_nans,

quality.no\_spikes(100),

quality.count\_spikes

],

'GR':[quality.all\_above(25)

]}

data\_qc\_table=well.qc\_table\_html(test)

HTML(data\_qc\_table)

**11- Load the gamma ray curve on to a dataframe**

gamma\_dataframe=gr.df

gamma\_dataframe

**12- Load all the curves on to a dataframe**

well\_dataframe=well.df()

well\_dataframe

**13- How many gamma ray readings are there in the dataframe? How many readings should there be for the entire well? If there is a difference, why would that be?**

len(gamma\_dataframe)

len(well\_dataframe)

**14- Define a Curve class to hold all the curves in a well log**

class Curve:

def \_\_init\_\_(self, well\_name, well\_location, curve\_name, data):

self.well\_name = well\_name

self.well\_location = well\_location

self.curve\_name = curve\_name

self.data = data

def get\_well\_name(self):

return self.well\_name

def get\_well\_location(self):

return self.well\_location

def get\_curve\_name(self):

return self.curve\_name

def get\_data(self, start=None, end=None):

if start is None and end is None:

return self.data

elif start is None:

return self.data[self.data.index <= end]

elif end is None:

return self.data[self.data.index >= start]

else:

return self.data[(self.data.index >= start) & (self.data.index <= end)]

**15- Instantiate the object with gamma ray data**

curve=Curve("well 1","cerro dragon","gamma ray”,gamma\_dataframe)

**16- Print the name of the well, the location, the curve names and data using attributes**

print(“Well name: "+curve.well\_name+"\nLocation: “+curve.well\_location+"\nCurve name: “+curve.curve\_name+"\nData:\n"+str(curve.data))

**17- Print the name of the well, the location, the curve names and data using methods**

print(“Well name: "+curve.get\_well\_name()+"\nLocation: “+curve.get\_well\_location()+"\nCurve name: “+curve.get\_curve\_name()+"\nData:\n"+str(curve.get\_data()))

**18- Print the name of the well, the location, the curve names and data from 30 to 3000 meters using methods**

print(“Well name: "+curve.get\_well\_name()+"\nLocation: “+curve.get\_well\_location()+"\nCurve name: “+curve.get\_curve\_name()+"\nData:\n"+str(curve.get\_data(30,3000)))

**19- Print the name of the well, the location, the curve names and data from 30 to 3000 meters using attributes**

print(“Well name: "+curve.well\_name+"\nLocation: “+curve.well\_location+"\nCurve name: “+curve.curve\_name+"\nData:\n"+str(curve.data[30:3000]))

**20- Load the las file using lasio and confirm the number of rows is correct: Data/LAS/8267\_a0801\_1996\_comp.las**

import lasio

las\_file\_path = 'Data/LAS/8267\_a0801\_1996\_comp.las'

well = lasio.read(las\_file\_path)

df=well.df()

df[30:3000]

**21- Load a csv file on to a dataframe, making sure the name of the columns are correct: Swell-1A\_AsciiDrillData\_183.0-5006.csv**

file\_path = 'Data//ASCII//Swell-1A\_AsciiDrillData\_183.0-5006.csv'

df = pd.read\_csv(file\_path)

df.head()

columns=df.columns

file\_path = 'Data//ASCII//Swell-1A\_AsciiDrillData\_183.0-5006.csv'

df = pd.read\_csv(file\_path, skiprows=1)

df.head()

df.columns=columns

df.head()

**22- Determine if there are spikes in the data**

df.plot(subplots=True, figsize=(15,35))

**23- Change all -999.25 readings to NaN**

import numpy as np

df.replace(-999.25, np.nan, inplace=True)

df.head()

**24- Determine how many NaN readings are there in each curve**

print(“Number of missing readings")

for column in df.columns:

count\_nan = df[column].isna().sum()

print(f"{column}: {count\_nan}”)

**25- Replace all NaN with the average of the before and after value. Compare the before and after**

import pandas as pd

import numpy as np

# Function to replace one NaN with the average of the surrounding values

def replace\_one\_nan\_with\_avg(df, column\_name):

for i in range(1, len(df[column\_name]) - 1):

if pd.isna(df.at[i, column\_name]):

if not pd.isna(df.at[i - 1, column\_name]) and not pd.isna(df.at[i + 1, column\_name]):

df.at[i, column\_name] = (df.at[i - 1, column\_name] + df.at[i + 1, column\_name]) / 2

else:

df.at[i, column\_name] = np.nan

return df

# Store the initial NaN count for each column in a dictionary

initial\_nan\_dic = df.isna().sum().to\_dict()

# Replace NaNs in specified columns

for column in columns: # 'columns' should be defined elsewhere with the list of columns

replace\_one\_nan\_with\_avg(df, column)

# Display the original and new NaN counts

for column in df.columns:

count\_nan = df[column].isna().sum()

print(f"Column: {column}: new NaN {count\_nan}, old NaN {initial\_nan\_dic[column]}")

**26- Replace all NaN with the average of the before and after value considering they could also be NaN values. Compare the before and after**

def replace\_many\_nan\_with\_avg(df, column\_name):

for i in range(len(df[column\_name])):

if pd.isna(df[column\_name][i]):

prev\_value = None

next\_value = None

for j in range(i-1, -1, -1):

if not pd.isna(df[column\_name][j]):

prev\_value = df[column\_name][j]

break

for j in range(i+1, len(df[column\_name])):

if not pd.isna(df[column\_name][j]):

next\_value = df[column\_name][j]

break

if prev\_value is None and next\_value is None:

df[column\_name][i] = np.nan

elif prev\_value is None:

df[column\_name][i] = next\_value

elif next\_value is None:

df[column\_name][i] = prev\_value

else:

df[column\_name][i] = (prev\_value + next\_value) / 2

return df

for column in columns:

replace\_many\_nan\_with\_avg(df,column)

for column in df.columns:

count\_nan = df[column].isna().sum()

print(f"Column: {column}: new NaN {count\_nan}, old NaN {initial\_nan\_dic[column]}")

df.head()

df.plot(subplots=True, figsize=(15,35))

**27- Show the use of the rolling method in pandas**

import pandas as pd

# create a sample dataframe

dft = pd.DataFrame({'values': [4.72, 3.89, 3.75, 4.96, 3.62, 4.68, 4.34, 3.63, 3.54, 6.71, 7.89, 3.58, 3.71, 4.19, 3.4, 3.01, 3.33, 4.36, 3.63, 3.61, 3.23, 4.7,

4.57, 2.84, 2.63, 2.86, 2.86, 2.66, 3.04, 3.06, 2.67, 2.57, 2.9, 2.55, 3.33, 2.58, 2.7, 2.38, 2.71, 2.25, 2.05, 3, 3.12, 2.36, 2.76,

3.52, 3.11, 2.94, 2.73, 2.57, 3.35, 3.07, 2.69, 2.95, 3.14, 3.8, 3.26, 2.91, 3.16, 3.23, 3.07, 3.32, 3.56, 3.35, 3.35, 5.31,7.21,

5.14, 4.11, 8.71, 6.28, 6.27, 4.13, 3.11, 4.46, 3.47, 3.33, 2.9, 3.3, 3.28, 3.3, 3.28]})

# define the window size

WS = 5

# calculate the rolling average

dft['rolling\_average'] = dft['values'].rolling(window=WS).mean()

dft.plot()

**28- Filter all curves using rolling average**

# define the window size

WS = 20

dfs=pd.DataFrame()

for column in columns:

dfs[column] = df[column].rolling(window=WS).mean()

dfs.plot(subplots=True, figsize=(15,35))

**29- Load time LAS files for an entire run**

def read\_las\_files(file\_list):

dfs=pd.DataFrame()

for file in file\_list:

well = lasio.read('Data/LAS/'+file)

df = well.df()

dfs=pd.concat([dfs, df], ignore\_index=True)

return dfs

file\_list=['210915\_IOA\_07\_BDC-2-04\_TD\_695.las','210916\_IOA\_08\_BDC-2-04\_TD\_695.las','210917\_IOA\_09\_BDC-2-04\_TD\_1171.las','210918\_IOA\_10\_BDC-2-04\_TD\_1551.las',

'210919\_IOA\_11\_BDC-2-04\_TD\_2047.las','210920\_IOA\_12\_BDC-2-04\_TD\_2261.las','210921\_IOA\_13\_BDC-2-04\_TD\_2327.las','210922\_IOA\_14\_BDC-2-04\_TD\_2462.las',

'210923\_IOA\_15\_BDC-2-04\_TD\_2516.las','210924\_IOA\_16\_BDC-2-04\_TD\_2516.las']

df=read\_las\_files(file\_list)

**30- Show the depth curve**

ax=df.plot(y='HDEP',use\_index=True)

ax.invert\_yaxis()

plt.show()

**31- Determine if the depth curve increases constantly**

df[‘HDEP'].is\_monotonic\_increasing

**32- Graph the depth curve where it does not increment constantly**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Sample DataFrame with 'HDEP' values (Hole Depth)

data = {

'Index': range(1, 11),

'HDEP': [1000, 1050, 1100, 1080, 1150, 1200, 1190, 1250, 1300, 1280]

}

# Create the DataFrame

df = pd.DataFrame(data)

df.set\_index('Index', inplace=True)

# Display the original DataFrame

print("Original DataFrame:")

print(df)

# Identify non-monotonic increasing points

# A point is non-monotonic if it is less than or equal to the previous point

non\_monotonic\_mask = df['HDEP'] <= df['HDEP'].shift(1)

# Display the mask

print("\nNon-Monotonic Mask:")

print(non\_monotonic\_mask)

# Plotting 'HDEP'

ax = df.plot(y='HDEP', use\_index=True, label='HDEP', figsize=(10, 6), marker='o')

# Highlight the non-monotonic points

ax.plot(df.index[non\_monotonic\_mask], df['HDEP'][non\_monotonic\_mask], 'ro', label='Non-Monotonic Points')

# Invert the y-axis to represent depth correctly

ax.invert\_yaxis()

# Add labels and title

ax.set\_ylabel('HDEP (m)')

ax.set\_xlabel('Run Index')

ax.set\_title('HDEP Plot with Non-Monotonic Points Highlighted')

# Show legend

ax.legend()

# Display the plot

plt.show()

**Identify Non-Monotonic Points**: Compare each 'HDEP' value with its previous value to determine where the sequence is not increasing.

**Create a Proper Boolean Mask**: Ensure the mask aligns with the DataFrame's index by using the shift() method.

**Plot the Data**: Plot the 'HDEP' values and highlight the non-monotonic points with distinct markers.

**Invert the Y-Axis**: Since 'HDEP' often represents depth, it's common to invert the y-axis for better visualization.

**33- What are the RPM ranges?**

1) Graph the RPM distribution:

df[‘RPMTOTAL'].plot(kind='kde')

2) Delete the distribution around zero:

df\_new = df.loc[df['RPMTOTAL'] > 0]

df\_new[‘RPMTOTAL'].plot(kind='kde')

3) Graph with library sklearn

from sklearn.neighbors import KernelDensity

# Extract the RPMTOTAL column

rpmtotal = df\_new['RPMTOTAL'].values.reshape(-1, 1)

# Fit the kernel density estimator

kde = KernelDensity(kernel='gaussian', bandwidth=8).fit(rpmtotal)

# Create a range of values to evaluate the estimator

x = np.linspace(rpmtotal.min(), rpmtotal.max(), 1000).reshape(-1, 1)

# Evaluate the estimator at the given values

y = np.exp(kde.score\_samples(x))

# Plot the KDE distribution

plt.plot(x, y)

plt.xlabel('RPMTOTAL')

plt.ylabel('Density')

plt.title('KDE Distribution of RPMTOTAL')

plt.show()

4) Compute normal values:

from sklearn.cluster import KMeans

# Fit the KMeans estimator

kmeans = KMeans(n\_clusters=5,n\_init=10).fit(rpmtotal)

# Get the cluster labels

labels = kmeans.labels\_

# Get the indices of the clusters

cluster\_indices = [np.where(labels == i)[0] for i in range(kmeans.n\_clusters)]

# Segregate the clusters into smaller arrays

clusters = [rpmtotal[indices] for indices in cluster\_indices]

# Plot the normal distributions for each array

for i, cluster in enumerate(clusters):

mu, std = cluster.mean(), cluster.std()

x = np.linspace(mu - 3 \* std, mu + 3 \* std, 1000)

y = np.exp(-0.5 \* ((x - mu) / std) \*\* 2) / (std \* np.sqrt(2 \* np.pi))

plt.plot(x, y, label=f'Cluster {i+1}')

plt.xlabel('RPMTOTAL')

plt.ylabel('Density')

plt.title('Normal Distributions of RPMTOTAL Clusters')

plt.legend()

plt.show()

**34- Graph the normal distribution and show 95% confidence level ranges**

from scipy import stats

def plot\_normal\_distribution(mean, std, ci):

x\_axis = np.arange(0, 260, 0.001)

pdf = stats.norm.pdf(x\_axis, mean, std)

fig, ax = plt.subplots()

fig.set\_figwidth(15) # Set the width of the figure to 10 inches

ax.plot(x\_axis, pdf)

std\_lim = stats.norm.ppf(1 - (1 - ci) / 2) # 95% CI

low = mean - std\_lim \* std

high = mean + std\_lim \* std

ax.fill\_between(x\_axis, pdf, where=(low < x\_axis) & (x\_axis < high))

ax.text(low, 0, f'{low:.2f}', ha='center',rotation=45)

ax.text(high, 0, f'{high:.2f}', ha='center',rotation=45)

plt.show()

plot\_normal\_distribution(150, 40, 0.90)

**35- Computer the statistical values for the normal values of RPM**

for i, cluster in enumerate(clusters):

mu, std = cluster.mean(), cluster.std()

print(f'Cluster {i+1}: Mean = {mu:.2f}, Standard Deviation = {std:.2f}')

**36- Graph the normal distributions for the normal values of RPM using a a range of confidence level of 95%**

for i, cluster in enumerate(clusters):

plot\_normal\_distribution(cluster.mean(), cluster.std(), 0.95)

**37- Graph all the normal values for RPM in one graph**

def plot\_normal\_distribution(mean, std, ci, color):

x\_axis = np.arange(0, 260, 0.001)

pdf = stats.norm.pdf(x\_axis, mean, std)

ax.plot(x\_axis, pdf, color=color)

std\_lim = stats.norm.ppf(1 - (1 - ci) / 2) # 95% CI

low = mean - std\_lim \* std

high = mean + std\_lim \* std

ax.fill\_between(x\_axis, pdf, where=(low < x\_axis) & (x\_axis < high))

ax.text(low, 0.015, f'{low:.2f}', ha='left', fontweight='bold', rotation=45)

ax.text(high, 0.015, f'{high:.2f}', ha='center', fontweight='bold', rotation=45)

fig, ax = plt.subplots(figsize=(25, 5))

colors = ['red', 'green', 'blue', 'orange', 'purple', 'brown', 'pink', 'gray', 'olive', 'cyan']

for i, cluster in enumerate(clusters):

plot\_normal\_distribution(cluster.mean(), cluster.std(), 0.95, colors[i % len(colors)])

plt.show()