Introduction to Drilling Data Analytics with Python

Peter Kowalchuk

Online Course. October 2024

Tuesday

Data Exploration, Event Detection and Basic Statistics

Making use of data

- Data exploration
  - Classes, attributes, methods - object oriented programing
- Event detection with Python
- Intro to Statistics

Advanced Data Management

Matrices and Dataframes

- Pandas
- Loading data
    - Data files CSV, LAS y WITSML
- Data fusion/concatenation
- Data grouping

Recap

- Development environments
- Conda environments
- IDE - Jupyter Notebooks
- Python variables
    - Numeric - integer and float
    - String
    - Lists
    - Dictionaries
- For Loop
- If

Two more structures

- If… else… elif
- While loop

If statement

If… else…

```
ROP = 35  # Current rate of penetration in meters/hour
threshold = 30  # Threshold for ROP

if ROP > threshold:
    print("ROP exceeds safe limit. Adjust drilling parameters.")
else:
    print("ROP is within safe limits.")
```

If statement

If… elif… else…

```python
ROP = 35  # Current rate of penetration in meters/hour
threshold_high = 40  # Upper threshold for ROP
threshold_low = 30   # Lower threshold for ROP

if ROP > threshold_high:
    print("ROP is too high! Adjust drilling parameters immediately.")
elif threshold_low <= ROP <= threshold_high:
    print("ROP is within the optimal range.")
else:
    print("ROP is too low. Increase drilling speed.")
```

While loop

```python
import pandas as pd

# Sample gamma ray values (measured in API units)
gamma_ray_values = [85, 90, 95, 105, 110, 108, 102, 99, 95, 90]

# Convert the list into a Pandas Series for better handling
gamma_ray_series = pd.Series(gamma_ray_values)

# Initialize variables
index = 0
above_100 = False

# Loop through the gamma ray series
while index < len(gamma_ray_series):
    value = gamma_ray_series[index]

    if not above_100 and value > 100:
        print(f"Gamma ray went above 100 at index {index}, value: {value}")
        above_100 = True
    elif above_100 and value < 100:
        print(f"Gamma ray dropped below 100 at index {index}, value: {value}")
        above_100 = False

    index += 1
```

One more type

Classes in Python

```python
import pandas as pd

class Well:
    def __init__(self, name, location, TD, reservoir):
        self.name = name                # Well name
        self.location = location        # Well location
        self.TD = TD                    # Total depth of the well
        self.reservoir = reservoir      # Reservoir type: oil or gas
        self.data = {}                  # Dictionary to store runs and drilling data

    # Method to write drilling data for a specific run
    def write_data(self, run, depth, WOB, RPM, TRQ, Gamma):
        # Create a DataFrame to store the run data
        run_data = pd.DataFrame({
            'Depth (m)': depth,
            'WOB (kN)': WOB,
            'RPM': RPM,
            'Torque (Nm)': TRQ,
            'Gamma Ray (API)': Gamma
        })
        # Store the run data in the dictionary using the run number as the key
        self.data[run] = run_data
```

```python
    # Method to read drilling data for a specific run
    def read_data(self, run):
        # Check if the run data exists
        if run in self.data:
            print(f"Data for run {run}:\n")
            print(self.data[run])
        else:
            print(f"No data available for run {run}")

    # Method to display well information
    def display_info(self):
        print(f"Well Name: {self.name}")
        print(f"Location: {self.location}")
        print(f"Total Depth (TD): {self.TD} meters")
        print(f"Reservoir Type: {self.reservoir}")
```

One more type

Classes in Python

```
well_1 = Well(name="Well A", location="Gulf of Mexico", TD=3500, reservoir="oil")

# Sample drilling data for run 100 (arrays for depth, WOB, RPM, TRQ, and Gamma Ray)
depth_100 = [100, 200, 300, 400, 500]
WOB_100 = [10, 12, 14, 13, 15]  # in kN
RPM_100 = [120, 130, 125, 135, 140]  # in rotations per minute
TRQ_100 = [1500, 1600, 1550, 1650, 1700]  # in Newton meters
Gamma_100 = [80, 85, 90, 110, 95]  # in API units

# Write the drilling data for run 100
well_1.write_data(run=100, depth=depth_100, WOB=WOB_100, RPM=RPM_100, TRQ=TRQ_100, Gamma=Gamma_100)

# Sample drilling data for run 200
depth_200 = [600, 700, 800, 900, 1000]
WOB_200 = [16, 17, 18, 17, 19]
RPM_200 = [150, 155, 160, 158, 162]
TRQ_200 = [1750, 1800, 1850, 1820, 1900]
Gamma_200 = [100, 110, 120, 115, 130]

# Write the drilling data for run 200
well_1.write_data(run=200, depth=depth_200, WOB=WOB_200, RPM=RPM_200, TRQ=TRQ_200, Gamma=Gamma_200)

# Display well information
well_1.display_info()

# Read and display the data for run 100
well_1.read_data(run=100)

# Read and display the data for run 200
well_1.read_data(run=200)
```

One more type

Classes in Python

```
type(object)
```

```
x = 42
print(type(x))  # Output: <class 'int'>

df = pd.DataFrame()  # Assuming pandas is imported
print(type(df))  # Output: <class 'pandas.core.frame.DataFrame'>
```

Classes and Objects

Object Oriented Programing

What is a Class in Python?

Class: A blueprint or template for creating objects (instances). It defines the attributes (data) and methods (functions) that the objects will have.

Attributes: Variables that store information about the object (e.g., name, location, depth in a well).

Methods: Functions that define the behavior or actions of the object (e.g., calculating, updating, or retrieving data).

Classes and Objects

Object Oriented Programing

Important Characteristics:

Instantiation:

Creating an object from a class. Each object is an instance of the class.

Constructor (__init__ method):

A special method that initializes an object with attributes when it's created.

Encapsulation:

Bundling of data (attributes) and methods into a single entity (class), allowing for organized code and data protection.

Classes and Objects

Object Oriented Programing

```python
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start(self):
        print(f"{self.make} {self.model} is starting...")

# Creating an instance of the Car class
my_car = Car('Jeep', 'Wrangler', 2018)
my_car.start()
```

Classes and Objects

Pandas library

```
import pandas as pd

# Creating a simple DataFrame using Pandas DataFrame class
data = {
    'Well Name': ['Well A', 'Well B', 'Well C'],
    'Depth (m)': [3500, 3200, 3000],
    'Pressure (psi)': [5000, 4800, 4600],
    'Reservoir': ['Oil', 'Gas', 'Oil']
}

# Create a DataFrame object (an instance of the DataFrame class)
df = pd.DataFrame(data)

# Display the DataFrame
print("DataFrame:")
print(df)

# Using methods from the DataFrame class

# Method 1: Get basic information about the DataFrame
print("\nInfo about the DataFrame:")
print(df.info())

# Method 2: Get statistics about numerical columns
print("\nStatistical Summary:")
print(df.describe())

# Method 3: Filter rows where Reservoir is 'Oil'
oil_wells = df[df['Reservoir'] == 'Oil']
print("\nWells with Oil Reservoir:")
print(oil_wells)
```

Classes and Objects

Pandas library

```
DataFrame:
  Well Name  Depth (m)  Pressure (psi) Reservoir
0    Well A       3500            5000       Oil
1    Well B       3200            4800       Gas
2    Well C       3000            4600       Oil

Info about the DataFrame:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 4 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Well Name       3 non-null      object
 1   Depth (m)       3 non-null      int64
 2   Pressure (psi)  3 non-null      int64
 3   Reservoir       3 non-null      object
dtypes: int64(2), object(2)
memory usage: 224.0+ bytes
```

```
Statistical Summary:
         Depth (m)  Pressure (psi)
count     3.000000        3.000000
mean   3233.333333     4800.000000
std     250.000000      200.000000
min    3000.000000     4600.000000
25%    3100.000000     4700.000000
50%    3200.000000     4800.000000
75%    3350.000000     4900.000000
max    3500.000000     5000.000000

Wells with Oil Reservoir:
  Well Name  Depth (m)  Pressure (psi) Reservoir
0    Well A       3500            5000       Oil
2    Well C       3000            4600       Oil
```

Classes and Objects

Lasio library

```python
import lasio

# Load the LAS file using lasio's LASFile class
las = lasio.read("example.las")

# Display basic information about the LAS file
print("Well Information:")
print(las.well)  # Access well information (metadata)

# Display curve information (data for each log)
print("\nCurve Information:")
print(las.curves)  # Access the curve information

# Display a portion of the data (log values)
print("\nLog Data (first 5 rows):")
print(las.data[:5])  # Print first 5 rows of log data

# Extract specific log data (e.g., Depth and Gamma Ray)
depth = las['DEPT']  # Access the depth log
gamma_ray = las['GR']  # Access the gamma ray log

# Display the first 5 values for Depth and Gamma Ray
print("\nDepth and Gamma Ray (first 5 values):")
for d, gr in zip(depth[:5], gamma_ray[:5]):
    print(f"Depth: {d:.2f} m, Gamma Ray: {gr:.2f} API")
```

```
Well Information:
Mnemonic    Value          Unit  Description
----------  -------------  ----  ------------------------------------------------
STRT        1000.0000      M     START DEPTH
STOP        2000.0000      M     STOP DEPTH
STEP        0.5000         M     STEP
NULL        -999.2500      M     NULL VALUE
WELL        EXAMPLE WELL         WELL NAME
...         ...            ...   ...

Curve Information:
Mnemonic  Unit  Value
--------  ----  -----------------------------------
DEPT      M     DEPTH
GR        API   GAMMA RAY
RES       OHMM  RESISTIVITY

Log Data (first 5 rows):
[[ 1.00000000e+03  4.50000000e+01  1.00000000e+02]
 [ 1.00050000e+03  4.80000000e+01  9.80000000e+01]
 [ 1.00100000e+03  5.10000000e+01  9.60000000e+01]
 [ 1.00150000e+03  5.30000000e+01  9.40000000e+01]
 [ 1.00200000e+03  5.60000000e+01  9.20000000e+01]]

Depth and Gamma Ray (first 5 values):
Depth: 1000.00 m, Gamma Ray: 45.00 API
Depth: 1000.50 m, Gamma Ray: 48.00 API
Depth: 1001.00 m, Gamma Ray: 51.00 API
Depth: 1001.50 m, Gamma Ray: 53.00 API
Depth: 1002.00 m, Gamma Ray: 56.00 API
```

Data Formats

Tidy Data

```
import pandas as pd

# Create a DataFrame with oil well drilling data in tidy form
data = {
    'Well Name': ['Well A', 'Well A', 'Well A', 'Well B', 'Well B', 'Well B'],
    'Section': ['Surface', 'Intermediate', 'Production', 'Surface', 'Intermediate', 'Production'],
    'Depth (m)': [1000, 2500, 3500, 900, 2400, 3400],
    'WOB (kN)': [50, 80, 90, 45, 85, 95],  # Weight on Bit
    'RPM': [120, 100, 90, 130, 110, 95],  # Revolutions per minute
    'Mud Weight (ppg)': [10, 11, 12, 9.8, 10.5, 11.8],  # Mud weight in pounds per gallon
    'Gamma Ray (API)': [50, 75, 100, 40, 78, 110]  # Gamma Ray in API units
}

# Convert the dictionary to a DataFrame
df = pd.DataFrame(data)

# Display the data in tidy form
print("Tidy DataFrame:")
print(df)

# Group the data by 'Well Name' and 'Section' to calculate statistics
grouped = df.groupby(['Well Name', 'Section']).agg({
    'Depth (m)': ['mean', 'std'],  # Mean and standard deviation of depth
    'WOB (kN)': ['mean', 'std'],  # Mean and standard deviation of WOB
    'RPM': ['mean', 'std'],  # Mean and standard deviation of RPM
    'Mud Weight (ppg)': ['mean', 'std'],  # Mean and standard deviation of mud weight
    'Gamma Ray (API)': ['mean', 'std']  # Mean and standard deviation of Gamma Ray
})
```

Data Formats

Tidy Data

Well Name	Section	Depth (m)	WOB (kN)	RPM	Mud Weight (ppg)	Gamma Ray (API)
Well A	Surface	1000	50	120	10	50
Well A	Intermediate	2500	80	100	11	75
Well A	Production	3500	90	90	12	100
Well B	Surface	900	45	130	9.8	40
Well B	Intermediate	2400	85	110	10.5	78
Well B	Production	3400	95	95	11.8	110

Data Formats

Tidy Data

Well Name	Section	Depth (m) Mean	Depth (m) Std	WOB (kN) Mean	WOB (kN) Std	RPM Mean	RPM Std	Mud Weight (ppg) Mean	Mud Weight (ppg) Std	Gamma Ray (API) Mean	Gamma Ray (API) Std
Well A	Surface	1000.00	NaN	50.00	NaN	120.00	NaN	10.00	NaN	50.00	NaN
Well A	Intermediate	2500.00	NaN	80.00	NaN	100.00	NaN	11.00	NaN	75.00	NaN
Well A	Production	3500.00	NaN	90.00	NaN	90.00	NaN	12.00	NaN	100.00	NaN
Well B	Surface	900.00	NaN	45.00	NaN	130.00	NaN	9.80	NaN	40.00	NaN
Well B	Intermediate	2400.00	NaN	85.00	NaN	110.00	NaN	10.50	NaN	78.00	NaN
Well B	Production	3400.00	NaN	95.00	NaN	95.00	NaN	11.80	NaN	110.00	NaN

First Normal Form?

Normalized Tables

RecordID	ClientName	ClientAddress	City	EnergyConsumed (kWh)	Month
001	ACME Corp	123 River St	Houston	500	January
002	Green Ltd	456 Maple Rd	Dallas	300	January
003	ACME Corp	123 River St	Houston	600	February

Second Normal Form

Normalized Tables

Clients Table:

ClientID	ClientName	ClientAddress	City
101	ACME Corp	123 River St	Houston
102	Green Ltd	456 Maple Rd	Dallas

EnergyConsumption Table:

RecordID	ClientID	EnergyConsumed (kWh)	Month
001	101	500	January
002	102	300	January
003	101	600	February

First Normal Form?

Normalized Tables

OrderID	CustomerName	CustomerAddress	ProductName	Quantity	ProductPrice	TotalOrderValue
001	John Smith	123 Elm St	T-shirt	2	$15.00	$30.00
002	Jane Doe	456 Oak St	Jeans	1	$40.00	$40.00
003	John Smith	123 Elm St	Hat	1	$10.00	$10.00

Second Normal Form

Normalized Tables

Orders Table:

OrderID	CustomerID	TotalOrderValue	ProductName	Quantity	ProductPrice
001	101	$30.00	T-shirt	2	$15.00
002	102	$40.00	Jeans	1	$40.00
003	101	$10.00	Hat	1	$10.00

Customers Table:

CustomerID	CustomerName	CustomerAddress
101	John Smith	123 Elm St
102	Jane Doe	456 Oak St

Third Normal Form

Normalized Tables

Customers Table:

CustomerID	CustomerName	CustomerAddress
101	John Smith	123 Elm St
102	Jane Doe	456 Oak St

Products Table:

ProductName	ProductPrice
T-shirt	$15.00
Jeans	$40.00
Hat	$10.00

Final OrderDetails Table:

OrderID	ProductName	Quantity
001	T-shirt	2
002	Jeans	1
003	Hat	1

Summary

Normalized Tables

- 1NF: Remove repeating groups, ensure atomic values.
- 2NF: Remove partial dependency (non-key attributes depend on the whole primary key).
- 3NF: Remove transitive dependency (attributes depend only on the primary key).

Numpy and Pandas

Data libraries

- Numpy
  - Matrices (XD) for numerical computing
- Pandas
  - Built on top of Numpy
  - Series (1D) and Dataframes (2D) for data manipulation and analysis

Numpy

Multi dimensional matrix

```python
import numpy as np

# Creating an array
arr = np.array([1, 2, 3])

# Performing element-wise addition
arr2 = arr + 10  # Output: array([11, 12, 13])

# Matrix multiplication
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = np.dot(A, B)  # Matrix multiplication result
```

Pandas Sequence

Multiple data types

```python
import pandas as pd

# Create a Pandas Series with different data types
data = [10, 3.14, 'Hello', True, None]
index = ['A', 'B', 'C', 'D', 'E']
series = pd.Series(data, index=index)

# Display the Series
print(series)
```

Pandas Sequence

Multiple numerical calculations

```
import pandas as pd
import numpy as np

# Create a Pandas Series with drilling depths (in meters)
drilling_depths_meters = pd.Series([1500, 2000, 2500, 3000, 3500], index=['Well A', 'Well B', 'Well C', 'Well D', 'Well E'])

# Conversion factor: 1 meter = 3.28084 feet
conversion_factor = 3.28084

# Apply NumPy multiplication to convert meters to feet
drilling_depths_feet = drilling_depths_meters * conversion_factor

# Display the original and converted Series
print("Original Drilling Depths (meters):")
print(drilling_depths_meters)
print("\nConverted Drilling Depths (feet):")
print(drilling_depths_feet)
```

Pandas Sequence

Multiple numerical calculations

```
import pandas as pd
import numpy as np

# Create a Pandas Series with drilling depths (in meters)
well_depths = pd.Series([1500, 2000, 2500, 3000, 3500], index=['Well A', 'Well B', 'Well C', 'Well D', 'Well E'])

# Define constants
overburden_density = 2300  # kg/m^3 (average overburden density)
g = 9.81  # gravity in m/s^2

# Calculate overburden pressure using the formula: Pressure = Density * Depth * g
overburden_pressure = overburden_density * well_depths * g

# Display the calculated overburden pressures (in Pascals)
print("Overburden Pressure (Pa) for each well:")
print(overburden_pressure)
```

What is a Pandas DataFrame?

2-dimensional labeled data structure, similar to a table in a database or an Excel spreadsheet

- Key Features:
    - Rows and columns: Data is organized in labeled rows and columns.
    - Heterogeneous data: It can hold different data types (e.g., integers, strings, floats).
    - Flexible operations: Supports operations like filtering, merging, sorting, and grouping.
    - Indexed: Both rows and columns have labels (index).

What is a Pandas DataFrame?

Column 1	Column 2	Column 3
Row 0	Data	Data
Row 1	Data	Data
Row 2	Data	Data

Pandas Dataframe

Multiple numerical calculations

```
import pandas as pd

# Create a DataFrame to represent well sections and their parameters
data = {
    'Well Section': ['Surface', 'Intermediate', 'Production'],
    'Depth (meters)': [0, 2000, 4000],
    'Mud Weight (ppg)': [8.5, 10.0, 12.0],  # pounds per gallon
    'Pore Pressure (psi)': [0, 8000, 15000],  # in psi
}

# Create DataFrame
well_df = pd.DataFrame(data)

# Convert mud weight from ppg to psi (1 ppg ≈ 0.4335 psi per foot of depth)
# Approximate depth is used to calculate mud weight in psi
well_df['Mud Weight (psi)'] = well_df['Mud Weight (ppg)'] * 0.4335 * well_df['Depth (meters)']

# Calculate the difference between mud weight and pore pressure
well_df['Pressure Control (psi)'] = well_df['Mud Weight (psi)'] - well_df['Pore Pressure (psi)']

# Display the DataFrame
print(well_df)
```

Pandas Dataframe

Multiple numerical calculations

```
import pandas as pd

# Create a DataFrame to represent well sections and their parameters
data = {
    'Well Section': ['Surface', 'Intermediate', 'Production', 'Deep Production', 'Final Section'],
    'Depth (meters)': [0, 2000, 4000, 6000, 8000],
    'Mud Weight (ppg)': [8.5, 10.0, 12.0, 14.0, 16.0],  # pounds per gallon
    'Pore Pressure (psi)': [0, 8000, 15000, 25000, 35000],  # in psi
}

# Create DataFrame
well_df = pd.DataFrame(data)

# Convert mud weight from ppg to psi (1 ppg ≈ 0.4335 psi per foot of depth)
# Approximate depth is used to calculate mud weight in psi
well_df['Mud Weight (psi)'] = well_df['Mud Weight (ppg)'] * 0.4335 * well_df['Depth (meters)']

# Calculate the difference between mud weight and pore pressure
well_df['Pressure Control (psi)'] = well_df['Mud Weight (psi)'] - well_df['Pore Pressure (psi)']

# Identify sections at risk of blowout
well_df['Blowout Risk'] = well_df['Pressure Control (psi)'] < 0

# Display the DataFrame
print(well_df)

# Find sections at risk of blowout
blowout_risk_sections = well_df[well_df['Blowout Risk']]
print("\nSections at Risk of Blowout:")
print(blowout_risk_sections[['Well Section', 'Depth (meters)', 'Pressure Control (psi)']])
```

```
iimport pandas as pd

# Create a DataFrame comparing average drilling parameters for two wells
data = {
    'Well Section': ['Surface', 'Intermediate', 'Production', 'Final Section'],
    'Well A: Average Depth (meters)': [0, 2000, 4000, 6000],
    'Well A: Average Rate of Penetration (m/h)': [10, 12, 8, 6],  # meters per hour
    'Well A: Average Weight on Bit (kN)': [50, 80, 120, 100],  # kilonewtons
    'Well A: Average Mud Weight (ppg)': [8.5, 10.0, 12.0, 14.0],  # pounds per gallon
    'Well A: Average Torque (Nm)': [1000, 1200, 1500, 1300],  # Newton meters
    'Well A: Average Pressure (psi)': [0, 8000, 15000, 25000],  # pounds per square inch

    'Well B: Average Depth (meters)': [0, 2100, 3900, 5800],
    'Well B: Average Rate of Penetration (m/h)': [9, 14, 7, 5],  # meters per hour
    'Well B: Average Weight on Bit (kN)': [55, 85, 110, 90],  # kilonewtons
    'Well B: Average Mud Weight (ppg)': [8.0, 11.0, 13.0, 15.0],  # pounds per gallon
    'Well B: Average Torque (Nm)': [950, 1250, 1400, 1200],  # Newton meters
    'Well B: Average Pressure (psi)': [0, 8200, 14800, 24000],  # pounds per square inch
}

# Create DataFrame
comparison_df = pd.DataFrame(data)

# Convert average mud weights from ppg to kg/m^3 for additional analysis
comparison_df['Well A: Average Mud Weight (kg/m^3)'] = comparison_df['Well A: Average Mud Weight (ppg)'] * 119.826
comparison_df['Well B: Average Mud Weight (kg/m^3)'] = comparison_df['Well B: Average Mud Weight (ppg)'] * 119.826

# Calculate differences between Well A and Well B
comparison_df['Depth Difference (meters)'] = comparison_df['Well A: Average Depth (meters)'] - comparison_df['Well B: Average Depth (meters)']
comparison_df['ROP Difference (m/h)'] = comparison_df['Well A: Average Rate of Penetration (m/h)'] - comparison_df['Well B: Average Rate of Penetration (m/h)']
comparison_df['WOB Difference (kN)'] = comparison_df['Well A: Average Weight on Bit (kN)'] - comparison_df['Well B: Average Weight on Bit (kN)']
comparison_df['Mud Weight Difference (ppg)'] = comparison_df['Well A: Average Mud Weight (ppg)'] - comparison_df['Well B: Average Mud Weight (ppg)']
comparison_df['Torque Difference (Nm)'] = comparison_df['Well A: Average Torque (Nm)'] - comparison_df['Well B: Average Torque (Nm)']
comparison_df['Pressure Difference (psi)'] = comparison_df['Well A: Average Pressure (psi)'] - comparison_df['Well B: Average Pressure (psi)']

# Display the DataFrame
print(comparison_df)
```