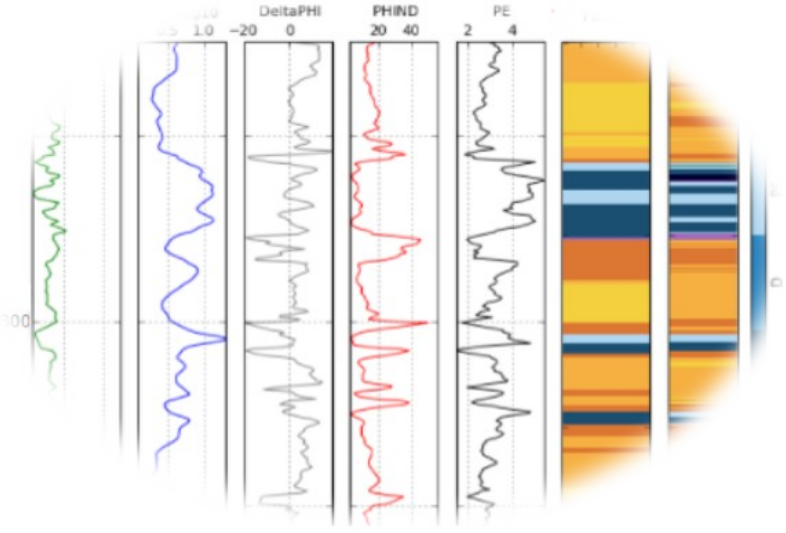
Next-Gen Well Logs: Data Standards, QC, and ML-Augmented Petrophysics

A Practical Guide for Subsurface Professionals

Author: Edy Irnandi Sudjana

Version: 1.0 (October 2025)
First Published: October 2025



Preface

"To the logging field engineers, vigilant data stewards, and all subsurface professionals: your tireless efforts in shaping, refining, and safeguarding the integrity of well log data form the quiet force behind clarity and insight. Through your devotion, the hidden truths of the earth are revealed, guiding reservoirs, wells, and the onward march of digital transformation. For this steadfast dedication, I extend my deepest gratitude."

Welcome to this e-book, crafted to connect field-proven logging practices with modern, data-driven workflows transforming subsurface operations. As data volumes and complexity continue to grow, mastering digital standards and AI-assisted techniques has become essential for achieving accuracy, efficiency, and insight in petrophysical analysis.

This e-book serves as a practical, technically grounded guide for early-career professionals in the upstream energy domain; logging engineers, geoscientists, petrophysicists, petroleum engineers, and subsurface data specialists. Its primary goal is to build a strong foundation in digital well log data standards (LAS, DLIS, LIS, BIT, and SEGY), introduce best practices for data conditioning and quality control, and provide step-by-step examples of machine learning (ML)-augmented petrophysical workflows.

Through real-world Python tools and use cases, readers will learn how to prepare, validate, and analyze well log data to support petrophysical interpretation and reservoir modeling in a modern, data-driven environment. By combining domain expertise with digital proficiency, this guide encourages readers to approach data with both technical rigor and creative insight.

Ultimately, this e-book aims to empower the next generation of subsurface professionals with the knowledge, tools, and confidence to deploy ML-enabled solutions that enhance quality, consistency, and interpretation across the petrophysical workflow. I wish you an insightful journey through these pages, and extend my sincere thanks for your dedication and curiosity.

Special thanks, with deepest gratitude to those who light my path: my beloved wife, Susie; our daughters, Aji, Aja, and JJ; and to all my parents, brothers and sisters, whose love and courageous hearts have been the quiet strength behind every step of this journey.

Code Repository

For all Python scripts and code examples featured in this e-book, please visit my GitHub portfolio: *github.com/edirnandi/petrophysical-qc*

This page intentionally left blank

Contents

A Practical Overview of Digital Well Log Data Standards

1. Introduction

Digital well log data forms the foundation for subsurface evaluation, reservoir characterization, and well performance analysis in the upstream oil and gas industry. Over the decades, several data standards have been developed to ensure consistency, interoperability, and reliability in the storage and exchange of logging data.

This guide provides an overview of the key digital well log data standards— **LAS 2.0**, **DLIS (API RP66)**, **LIS**, **BIT**, and **SEGY**. It is designed for junior geoscientists, petrophysicists, petroleum engineers and data specialists who need a clear understanding of how these formats are structured, their primary applications, and how they fit within modern data management platform and workflows.

---

2. LAS 2.0 Specification and Compliance

The **Log ASCII Standard (LAS)** is a widely adopted text-based format developed by the **Canadian Well Logging Society (CWLS)** for storing and exchanging digital well log data. Introduced in 1989, LAS 2.0 (released in 1992) addressed inconsistencies from earlier versions. While LAS 3.0 (1999) expanded storage capabilities, LAS 2.0 remains dominant for its simplicity and broad software support.

LAS 2.0 Structure

LAS 2.0 files are divided into structured sections:

1. **~VERSION INFORMATION** – Identifies the LAS version and wrapping type (WRAP/NO-WRAP).
    - VERS.: LAS version (e.g., 2.0)
    - WRAP.: Indicates line wrapping (YES or NO).
2. **~WELL INFORMATION** – Metadata about the well, including depth range and null values.
    - STRT.: Start depth

- STOP.: Stop depth
- STEP.: Step size
- NULL.: Missing value indicator

3. **~CURVE INFORMATION** – Describes the log curves with mnemonic, unit, and description.
   - Example: GR.API : Gamma Ray
4. **~PARAMETER INFORMATION** – Lists additional logging parameters.
5. **~ASCII LOG DATA** – Contains the actual numeric log values.

Compliance with LAS 2.0 ensures interoperability across diverse software systems used for petrophysical interpretation and data management.

LAS File Formats

- **WRAP (Multiple Lines per Depth Step)**: Allows more detailed data per interval.
- **NO-WRAP (One Line per Depth Step)**: Simpler format; each depth step is one record.
- **LAS 2.0 Sample**

```
~VERSION INFORMATION
VERS. 2.0 : CWLS LOG ASCII STANDARD - VERSION 2.0
WRAP. NO : ONE LINE PER DEPTH STEP

~WELL INFORMATION
#MNEM.UNIT      DATA              DESCRIPTION
STRT.M          100.0             START DEPTH
STOP.M          140.0             STOP DEPTH
NULL.          -999.25            NULL VALUE
WELL.          WELL-1              WELL NAME

~CURVE INFORMATION
#MNEM.UNIT      DESCRIPTION
DEPT.M          DEPTH
GR.API          GAMMA RAY
~A
100.0  45.5
110.0  46.0
120.0  46.7
130.0  96.7
140.0  47.3
```

Reference

1) Canadian Well Logging Society: LAS (Log ASCII Standard)
2) US Geological Survey: Log ASCII Standard (LAS) files for geophysical wireline well logs and their application to geologic cross sections through the central Appalachian basin

3. DLIS (Digital Log Interchange Standard / API RP66)

The **Digital Log Interchange Standard (DLIS)**, defined in **API Recommended Practice 66**, was introduced to overcome the limitations of earlier binary formats like LIS and BIT. It supports complex data structures, metadata organization, and dynamic sampling rates —making it ideal for modern wireline and logging-while-drilling data.

Key Features

- **Robust Data Identification**: Each data item is uniquely identified.
- **Complex Data Representation**: Handles arrays, strings, and multi-channel datasets.
- **Dynamic Channel Support**: Allows multiple frame types and sampling rates within a single file.

Data Organization

DLIS uses two hierarchical levels: - **Logical Format**: Organizes data into logical records and files, representing self-contained measurement datasets. - **Physical Format**: Defines how logical records are stored on physical media such as magnetic tape or disk.

DLIS ensures data can be accurately exchanged across systems, independent of acquisition equipment.

Reference: RP66/DLIS V1 Specification

4. LIS (Log Information Standard)

The **LIS** format, developed by **Schlumberger** in the early 1970s, was among the first attempts to standardize digital well log data. It groups data by type in logical records—**INFORMATION** (metadata) and **DATA** (measurements). Both are essential to reconstruct a complete digital service.

Characteristics

- **Dictionary-Controlled Mnemonics** define record attributes such as name, size, unit, and representation code.
- **Flexible Implementation**: Variations exist across contractors, which sometimes impairs decoding.
- **Limitations**: Short mnemonic length (max. 4 characters) and limited flexibility compared to DLIS.

Occasionally, LIS tapes converted from DLIS may exhibit inconsistencies (e.g., curve name truncation), requiring careful handling during import into modern databases.

Reference: LIS 79 Description Reference Manual

---

5. BIT (Basic Information Tape)

The **BIT** format, introduced by **Atlas Wireline Services** in the 1970s, was designed for recording basic log data on magnetic tape. Each tape consists of sequential, unblocked records separated by inter-record gaps (IRGs).

Structure

- **General Heading Record**: Contains well identification and processing parameters.
- **Data Records**: Store actual measurement values.

A BIT tape may include up to 20 curves, with each file beginning with a heading record followed by multiple data records. The format is simple but limited in scalability and metadata richness compared to LIS or DLIS.

---

6. SEGY (Standard for Exchange of Seismic Data)

The **SEGY** (or SEG Y) format, established by the **Society of Exploration Geophysicists (SEG)** in 1975 and updated in 2002 (Rev 1.0) and 2017 (Rev 2.0), is the industry standard for **seismic and borehole seismic (VSP)** data exchange.

Structure

1. **Textual Header (3200 bytes)** – Human-readable survey and acquisition information.
2. **Binary Header (400 bytes)** – File-level parameters like sample rate and format code.
3. **Trace Headers** – Contain metadata for each trace (e.g., depth, channel, coordinates).
4. **Trace Data** – Actual seismic samples (16-, 32-, or 64-bit values).

Applications

Commonly used for: - Checkshot and zero-offset VSPs
- Offset and walkaway VSPs
- Crosswell seismic data

Advantages

- Universally supported by seismic software.
- Flexible structure with extended trace headers (Rev 2.0).
- Integrates easily with LAS and DLIS for combined well-seismic analysis.

References: SEG-Y_r2.0 - SEG Technical Standards Committee 2017

7. Summary Comparison of Digital Log Data Standards

Format	Origin / Year	Data Type	Structure	Advantages	Limitations
LAS 2.0	CWLS, 1992	ASCII Well Logs	Sectioned text format (~VERSION, ~WELL, ~CURVE, ~DATA)	Simple, human-readable, widely supported	Limited metadata, less suited for complex datasets
DLIS	API RP66, 1991	Binary Well Logs	Logical and physical structures	Rich metadata, dynamic channels, standardized	Complex decoding, larger file size
LIS	Schlumberger, 1970s	Binary Well Logs	INFORMATION & DATA	Historical standard,	Proprietary variations,

Format	Origin / Year	Data Type	Structure	Advantages	Limitations
			records	flexible	short mnemonics
BIT	Atlas, 1970s	Binary Well Logs	Header + Data records	Simple and lightweight	Minimal metadata, legacy format
SEGY	SEG, 1975 (Rev. 2017)	Seismic / VSP Data	Header + Trace + Binary blocks	Universal seismic standard, integrates with logs	Large file size, limited log metadata

8. Conclusion

Understanding these digital log data standards is essential for any professional involved in subsurface data management or analysis and interpretation. **LAS** remains the most common for well logs, **DLIS** offers the most comprehensive data model, **LIS** and **BIT** are legacy but still encountered, and **SEGY** bridges the well and seismic domains through VSP and borehole seismic applications.

As data integration and digitalization accelerate in the energy sector, familiarity with these standards enables seamless interoperability, efficient QC, and more robust geoscience workflows-- key skills for the next generation of data-savvy geoscientists and engineers.

Digital Log Data Preparation and Editing: A Practical Guide for Petrophysical Analysis

1. Introduction

Digital well log data serve as the foundation for petrophysical interpretation, reservoir characterization, and data-driven analysis. The most common formats, LAS (Log ASCII Standard) and DLIS (Digital Log Interchange Standard), were developed under the Canadian Well Logging Society (CWLS) and the American Petroleum Institute (API) RP 66 respectively. Ensuring data quality and readiness is essential before loading and validating data in petrophysical software such as Techlog, or IP, or in modern data platforms like OSDU (Open Subsurface Data Universe). This includes verifying conformity to format specifications, using standardized null values and date formats, maintaining consistent depth intervals, and ensuring valid metadata. Studies show that poor data quality can increase project costs by 15–25% and consume up to 50% of project time resolving inefficiencies (Source: US Geological Survey [USGS] and Society of Petroleum Engineers [SPE]).

This document provides both automated and manual workflows using Python utilities and lightweight Hexadecimal (Hex) editor tools for practical, field-level data preparation.

2. Digital Log Validation and Quality Control

Validation ensures that well log files conform to their technical standards and contain the necessary sections for analysis. Python-based validators using the LASIO and DLISIO libraries provide an open-source, reproducible approach to QC.

2.1 LAS Validator

The LAS Validator checks for CWLS LAS 2.0 conformity. It verifies headers (~V, ~W, ~C, ~A), WRAP mode, null values, and consistency between declared START/STOP depths and actual data. The script supports multi-file batch validation and outputs a CSV summary.

```
# LASValidatorv2-free.py


import pandas as pd
import os
import lasio
from tkinter import Tk, filedialog
from datetime import datetime

# Step 1: Verify LAS 2.0 Conformity
def verify_las_file(las_file, tolerance=1e-3):
    try:
        las = lasio.read(las_file, ignore_header_errors=True)
        sections = [section.upper() for section in las.sections.keys()]
        errors = []

        # Check mandatory sections
        required_sections = ['VERSION', 'WELL', 'CURVES']
        for req in required_sections:
            if req not in sections:
                errors.append(f"Missing section: {req}")

        # Check version
        try:
            version = float(str(las.version['VERS'].value).strip())
            if version != 2.0:
                errors.append(f"Invalid version: {version} (Expected 2.0)")
        except Exception:
            errors.append("Missing or invalid VERSION information")

        # Check WRAP mode
        try:
            wrap_mode = str(las.version['WRAP'].value).strip().upper()
            if wrap_mode not in ['YES', 'NO']:
                errors.append(f"Invalid WRAP mode: {wrap_mode}")
        except Exception:
            errors.append("Missing WRAP mode in VERSION section")

        # Check first curve is DEPT, DEPTH, TIME, or INDEX
        try:
            first_curve = las.curves[0].mnemonic.strip().upper()
            if first_curve not in ['DEPT', 'DEPTH', 'TIME', 'INDEX']:
                errors.append(f"Invalid index curve: {first_curve}")
        except Exception:
            errors.append("Missing or invalid CURVE information")

        # Check NULL values
        if 'NULL' not in las.well:
            errors.append("Missing NULL value in WELL section")

        # Check WELL ID is present (UWI or WELL only)
        well_id_present = any(mnemonic.upper() in ['UWI', 'WELL'] for
mnemonic in las.well.keys())
        if not well_id_present:
```

```
            errors.append("Missing Well ID in WELL section (UWI or WELL)")

        # Check START and STOP consistency with tolerance
        try:
            # Possible keys to look for
            start_keys = ['STRT', 'START', 'STRT.M', 'START.M', 'STRT.F',
'START.F']
            stop_keys  = ['STOP', 'STOP.M', 'STOP.F']

            # Find actual keys present in LAS well section
            well_keys = {k.upper(): k for k in las.well.keys()}

            found_pairs = []
            for sk in start_keys:
                for ek in stop_keys:
                    # Match STRT with STOP (same suffix if present)
                    if sk.replace("START", "STOP") == ek or
sk.replace("STRT", "STOP") == ek:
                        if sk in well_keys and ek in well_keys:
                            found_pairs.append((well_keys[sk],
well_keys[ek]))

            if not found_pairs:
                errors.append("Missing START/STOP pair in WELL section")
            else:
                data_start = float(las.index[0])
                data_stop = float(las.index[-1])

                for sk, ek in found_pairs:
                    header_start = float(str(las.well[sk].value).strip())
                    header_stop = float(str(las.well[ek].value).strip())

                    if abs(header_start - data_start) > tolerance:
                        errors.append(
                            f"Mismatch START ({sk}): Header={header_start},
Data={data_start} "
                            f"(Diff={abs(header_start - data_start):.6f} >
Tolerance={tolerance})"
                        )
                    if abs(header_stop - data_stop) > tolerance:
                        errors.append(
                            f"Mismatch STOP ({ek}): Header={header_stop},
Data={data_stop} "
                            f"(Diff={abs(header_stop - data_stop):.6f} >
Tolerance={tolerance})"
                        )
        except Exception:
            errors.append("Error validating START/STOP consistency in WELL
section or data")

        return "Valid" if not errors else ", ".join(errors)


    except Exception as e:
```

```
        return f"Error reading file: {e}"


# Step 2: Interactive File Selection and Verification
def main():
    # Initialize file dialog
    Tk().withdraw()  # Hide the root window
    file_paths = filedialog.askopenfilenames(title="Select LAS Files",
filetypes=[("LAS files", "*.las")])

    if not file_paths:
        print("No files selected.")
        return

    # Verify each file
    results = []
    for file in file_paths:
        status = verify_las_file(file)
        results.append({"File": os.path.basename(file), "Status": status})
        print(f"{os.path.basename(file)}: {status}")

    # Save results to timestamped CSV
    output_df = pd.DataFrame(results)
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    output_file = f"las_verification_results_{timestamp}.csv"
    output_df.to_csv(output_file, index=False)
    print(f"Results saved to {output_file}")


if __name__ == "__main__":
    main()
```

This validator is a foundational QC tool that can be integrated into data pipelines or pre-ingestion steps in well log repositories.

2.2 DLIS Validator

The DLIS Validator validates binary DLIS files according to API RP 66 specification. It checks logical file structures, frames, and channels to confirm completeness. This script relies on the DLISIO library, which is widely adopted in digital subsurface workflows.

```
# DLISCheck-free.py
import os
import pandas as pd
from tkinter import Tk, filedialog
from dlisio import dlis
from pathlib import Path

def validate_dlis_file(dlis_file):
    """
    Validate a DLIS file for conformity to the DLIS/API RP66 standard using
both physical and logical file checks.
```

```
    Parameters:
    dlis_file (str): The file path of the DLIS file to be validated.
    Returns:
    str: A message indicating the validation result.
    """
    try:
        # Ensure the file exists
        if not os.path.isfile(dlis_file):
            return f"Error: {dlis_file} is not a valid file or does not
exist."

        # Load the DLIS file
        physical_file = dlis.load(dlis_file)
        if not physical_file:
            return "File is empty or not a valid DLIS file."

        # Describe the physical file
        description = physical_file.describe()
        print(description)

        # Logical file validation
        logical_file_issues = []
        for logical_file in physical_file:
            # Check logical file metadata
            if not logical_file.origins:
                logical_file_issues.append("Logical file missing origin
metadata.")

            # Validate channels
            for channel in logical_file.channels:
                if not channel.name:
                    logical_file_issues.append("Channel with missing name
found.")

            # Validate frames
            for frame in logical_file.frames:
                if not frame.name:
                    logical_file_issues.append("Frame with missing name
found.")

        if logical_file_issues:
            return "Logical file issues detected: " + ";
".join(logical_file_issues)

        return "DLIS file conforms to the standard."

    except dlis.DlisError as e:
        return f"DLIS-specific error: {e}"
    except Exception as e:
        return f"Error processing file: {e}"

def main():
    # Initialize Tkinter window (hidden)
    Tk().withdraw()
```

```
    # Select a folder containing DLIS files
    folder_path = filedialog.askdirectory(title="Select Folder Containing
DLIS Files")

    if not folder_path:
        print("No folder selected.")
        return

    folder_path = Path(folder_path).resolve()
    dlist_files = [str(folder_path / f) for f in os.listdir(folder_path) if
f.lower().endswith('.dlis')]

    if not dlist_files:
        print("No DLIS files found in the selected folder.")
        return

    # Verify each DLIS file
    results = []
    for file in dlist_files:
        status = validate_dlis_file(file)
        results.append({"File": os.path.basename(file), "Status": status})
        print(f"{os.path.basename(file)}: {status}")

    # Save results to CSV
    output_df = pd.DataFrame(results)
    output_file = "dlis_verification_results.csv"
    output_df.to_csv(output_file, index=False)
    print(f"Results saved to {output_file}")

if __name__ == "__main__":
    main()
```

3. Data Wrangling and Preparation

Data wrangling involves transforming, standardizing, and conditioning log datasets before analysis. The following Python utilities simplify key steps: format conversion, log header extraction, and null values standardization.

3.1 ASCII to LAS 2.0 Conversion (ascii2las)

Many legacy log datasets exist in tabular ASCII (CSV/TXT) form. Converting these to LAS 2.0 improves interoperability. This script generates a CWLS-compliant LAS with standardized headers and units.

```
# ascii2las.py - Convert ASCII logs into CWLS LAS 2.0

import pandas as pd
from tkinter import Tk, filedialog
import os
```

```
# --- Constants for LAS 2.0 header (Curve units taken  from SLB curve
mnemonic dictionary https://www.apps.slb.com/cmd/) ---
CURVE_INFO = [
    ("DEPT", "M", "Depth (m)"),
    ("GR", "GAPI", "Gamma Ray (API units)"),
    ("RES", "OHM.M", "Resistivity (ohm·m)"),
    ("RHOB", "G/CM3", "Bulk Density (g/cm³)"),
    ("NPHI", "NAPI", "Neutron Porosity (nAPI)"),
    ("DT", "US/FT", "Sonic Transit Time (µs/ft)")
]

# --- File dialog to pick CSV, TXT or Excel file ---
def select_file():
    root = Tk()
    root.withdraw()  # Hide the main window
    file_path = filedialog.askopenfilename(
        title="Select a CSV, TXT, or Excel File",
        filetypes=[("Data files", "*.csv *.txt *.xls *.xlsx")]
    )
    return file_path

# --- Read the file into a pandas DataFrame ---
def read_data(file_path):
    ext = os.path.splitext(file_path)[1].lower()
    if ext == ".csv":
        return pd.read_csv(file_path)
    elif ext == ".txt":
        return pd.read_csv(file_path, sep=None, engine="python")
    elif ext in [".xls", ".xlsx"]:
        return pd.read_excel(file_path)
    else:
        raise ValueError("Unsupported file format!")

# --- Generate LAS content ---
def generate_las(df, well_name):
    lines = []
    start_depth = df['Depth'].min()
    stop_depth = df['Depth'].max()
    step = df['Depth'].diff().dropna().mode()[0]  # most frequent step
    null_value = -999.25

    lines.append("~Version Information Section")
    lines.append("VERS.                  2.0           : CWLS LOG ASCII
STANDARD - VERSION 2.0")
    lines.append("WRAP.                  NO            : One line per depth
step\n")

    lines.append("~Well Information Section")
    lines.append("STRT.M                 {:.4f}         : START
DEPTH".format(start_depth))
    lines.append("STOP.M                 {:.4f}         : STOP
DEPTH".format(stop_depth))
    lines.append("STEP.M                 {:.4f}         : STEP".format(step))
```

```
    lines.append("NULL.                  {}           : NULL
VALUE".format(null_value))
    lines.append("COMP.                   UNKNOWN        : COMPANY")
    lines.append(f"WELL.                   {well_name}    : WELL NAME")
    lines.append("FLD.                    UNKNOWN        : FIELD")
    lines.append("LOC.                    UNKNOWN        : LOCATION")
    lines.append("PROV.                   UNKNOWN        : PROVINCE")
    lines.append("SRVC.                   UNKNOWN        : SERVICE COMPANY")
    lines.append("DATE.                   2025-06-14     : LOG DATE")
    lines.append("UWI.                    UNKNOWN        : UNIQUE WELL ID\n")

    lines.append("~Curve Information Section")
    lines.append("#MNEM.UNIT              API CODES    CURVE DESCRIPTION")
    for mnemonic, unit, desc in CURVE_INFO:
        lines.append(f"{mnemonic:<6}.{unit:<10}           : {desc}")
    lines.append("")

    lines.append("~ASCII Log Data")
    for _, row in df.iterrows():
        row_vals = [row.get(col, null_value) for col in ['Depth', 'GammaRay',
'Resistivity', 'Density', 'NeutronPorosity', 'SonicDT']]
        row_str = " ".join(f"{val:.4f}" if pd.notnull(val) else
f"{null_value:.2f}" for val in row_vals)
        lines.append(row_str)

    return "\n".join(lines)

# --- Save LAS file ---
def save_las_file(content, well_name):
    output_file = f"{well_name}.las"
    with open(output_file, "w") as f:
        f.write(content)
    print(f"LAS file saved as: {output_file}")


# --- Save LAS file to user-selected output folder ---
def save_las_file(content, well_name, output_folder):
    output_path = os.path.join(output_folder, f"{well_name}.las")
    with open(output_path, "w") as f:
        f.write(content)
    print(f"LAS file saved as: {output_path}")

# --- Main Process ---
def main():
    file_path = select_file()
    if not file_path:
        print("No file selected.")
        return

    output_folder = filedialog.askdirectory(title="Select Output Folder")
    if not output_folder:
        print("No output folder selected.")
        return
```

```
    df = read_data(file_path)

    # Handle column naming and filtering
    required_columns = ['WellName', 'Depth', 'GammaRay', 'Resistivity',
'Density', 'NeutronPorosity', 'SonicDT']
    for col in required_columns:
        if col not in df.columns:
            raise ValueError(f"Missing required column: {col}")

    for well_name, group_df in df.groupby("WellName"):
        group_df_sorted = group_df.sort_values("Depth")
        las_content = generate_las(group_df_sorted, well_name)
        save_las_file(las_content, well_name, output_folder)

if __name__ == "__main__":
    main()
```

3.2 LAS Header Extraction

In some QC workflows, only log header metadata are required. The following script extracts header sections (~V, ~W, ~C) and omits the ~A data block.

```
# extract_las_header.py
import os

for filename in os.listdir('.'):
    if filename.lower().endswith('.las'):
        with open(filename, 'r') as f:
            lines = f.readlines()

        # Open a new file to write the header
        with open(f"{filename}.header", 'w') as header_file:
            for line in lines:
                if '\176A' in line:  # Check for the marker \176A
                    break  # Stop when the data part starts
                header_file.write(line)
```

3.3 Standardizing LAS Null Values

Non-standard null representations (e.g., -9999 or -999.000) often cause issues during ingestion. This Python utility replaces them with -999.25 across multiple LAS files.

```
# ReplaceLASNull.py script to replace LAS Null values in multiple LAS files
e.g. -9999 into -999.25 in the Header and data_section ie. after ~A.
# take input CWLS LAS format from "curr_dir"  and save the edited in the
"output_dir"
#-----------------------------------------------------------------

import glob
import os
import ntpath
import tkinter as tk
```

```
from tkinter import filedialog
import re  # Import the regular expression module

# Set up Tkinter root window (it won't appear because we use the dialog box)
root = tk.Tk()
root.withdraw()  # Hide the main Tkinter window

# Prompt user to select the input directory (curr_dir)
curr_dir = filedialog.askdirectory(title="Select the Input Directory with LAS
files")
if not curr_dir:
    print("No input directory selected. Exiting.")
    exit()

# Prompt user to select the output directory (output_dir)
output_dir = filedialog.askdirectory(title="Select the Output Directory to
Save Edited LAS files")
if not output_dir:
    print("No output directory selected. Exiting.")
    exit()

# Ensure the output directory exists
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Regex pattern to match various forms of '-9999' and its decimal variants
pattern = r"-9999(\.0+)?(\.000+)?(\.0000+)?"

# Process each LAS file in the directory
for f in glob.glob(os.path.join(curr_dir, "*.las")):
    with open(f, 'r') as inputfile:
        # Create output file with the same name in the output directory
        output_file_path = os.path.join(output_dir, ntpath.basename(f))
        with open(output_file_path, 'w') as outputfile:
            is_data_section = False  # Flag to track the data section

            for line in inputfile:
                # If we encounter the data section (~A), we mark it
                if line.startswith("~A"):
                    is_data_section = True
                # If we encounter another section (~), we exit the data
section
                elif line.startswith("~") and is_data_section:
                    is_data_section = False

                # Replace matching values for all occurrences of '-9999' (in
header or data section)
                # Use Regex to replace all versions of '-9999' with '-999.25'
                line = re.sub(pattern, "-999.25", line)

                # Write the (possibly modified) line to the output file
                outputfile.write(line)

print("Processing complete. Edited files saved in:", output_dir)
```

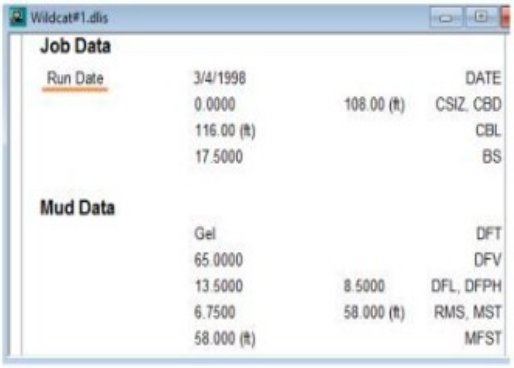
3.4 Standardizing Date Formats

Standardizing date formats is essential to maintain consistency across domains and disciplines. In well logging; whether Wireline or LWD/MWD and Cased-Hole— date and time are critical metadata for log data quality and traceability (Theys, 1999). A unified date format ensures that all specialists can correctly interpret and utilize the data without ambiguity.

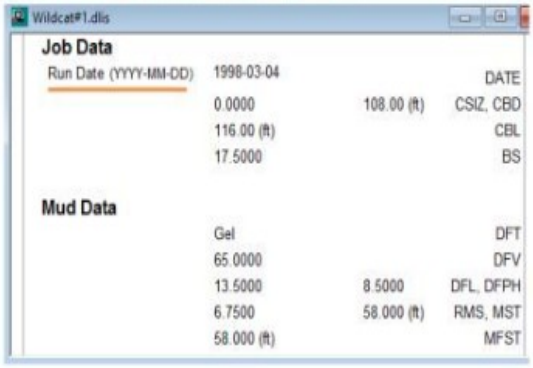
Adopting a standardized date format provides several benefits:

- **Prevents Misinterpretation:** Numerical date formats can be ambiguous (e.g., sample #1), whereas an ISO-compliant format (ISO 8601) ensures clarity (e.g., sample #2)

Sample# 1



Sample# 2 using ISO 8601 Date and Time format



- **Enhances Search and Sorting:** Standardized dates simplify querying and filtering data
- **Ensures System Consistency:** Uniform formatting across platforms reduces discrepancies when integrating multiple data sources.
- **Improves Data Exchange:** A consistent standard avoids errors in interpreting time and date when sharing data across systems and applications.

Best Practices for Standardizing Date Formats in Techlog

Normalize Upon Import.

Convert all incoming log timestamps to the **ISO 8601** standard format (e.g., YYYY-MM-DDThh:mm:ssZ) to ensure consistency across datasets. Apply this conversion to:

- **LAS headers** — DATE field
- **DLIS acquisition timestamps** — run or recording date
- **Drilling metadata** — bit depth vs. time records

Set a Clear Time Zone Policy

Use **UTC** as the master time reference for all log data. If local time is required for operational or reporting purposes, record the **local time offset** separately in metadata or header fields.

Validate During QC Checks

Leverage **Techlog QC modules** to automatically identify and correct issues such as:

- Missing or invalid timestamps
- Clock drift between tools or runs
- Overlapping log intervals
- Time/depth mismatch warnings

Document Within Metadata

Clearly record the adopted date and time standard within **job headers, log metadata, and data delivery specifications**. This ensures transparency and consistency across workflows, teams, and data platforms.

4. Manual Editing Techniques (DLIS Header Modification)

DLIS files, defined under API RP 66, are binary and not human-readable. When specialized software is unavailable, for example, during quick QC checks in the field, or when license-based applications are temporarily inaccessible. In these cases, a simple but powerful alternative is to use a hexadecimal editor like HexEdit 4.0. This lightweight utility allows users to directly view and adjust DLIS header parameters such as Well Name safely without needing a full petrophysical platform.

Step-by-Step Procedure

1. Download HexEdit 4.0 (free) software from the CNET Download website **https://download.cnet.com/HexEdit/3000-2352_4-10208432.html**

2. After extracting the HexEdit4_0 .zip file to your working folder, run the HexEd4_0.msi file and follow the instructions

3. Launch the HexEdit app, then open a DLIS file by clicking on the File menu and selecting Open, or by pressing **Ctrl-O**

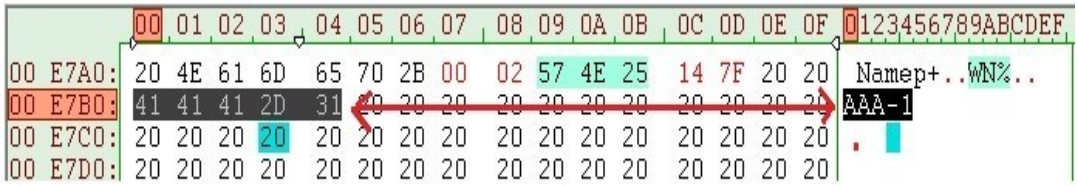
4. Navigate to the folder containing the DLIS files whose well header parameters you want to update. **Ensure you have backed up your DLIS files** before performing this task

5. Once a DLIS file is displayed in HexEdit, the left column—Hex column—shows the raw numbers, while the right column displays a textual representation of the DLIS file

6. To change the Well Name, use the Edit > Find menu or press Ctrl-F. Then, select the Text tab to search, enter 'WN%' in the Text find field, and click the Find Next button. (WN is the parameter mnemonic code for Well Name)

Text find/search result will looks like below:

7. Right after WN% character is existing Well name value, in given sample is AAA-1. Note that when you highlight AAA-1 characters, in the same time in left column its corresponding hexadecimal values (41 41 41 2D 31) is highlighted

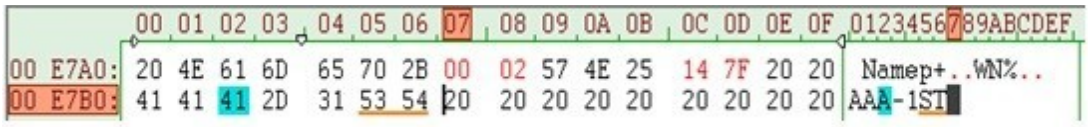


8. For instance, if you want to add 'ST' character (Hexadecimal: 5354)-- to indicate a side-track borehole, after AAA-1, go to left column and highlight hexadecimal number after the last hexadecimal number of Well name, which is 20

9. Put cursor in hexadecimal number 20, right click and clicking on 'Allow Changes', this needed to change Well name characters using allocated/existing internal storage. ⚠ **Do**

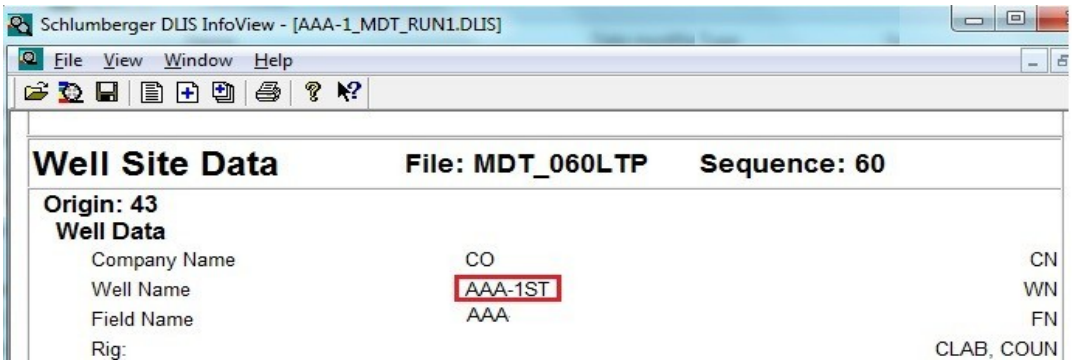
NOT choose **Insert** or **Alt-I** command when editing hexadecimal values-- this will corrupt your DLIS file. Always modify within the existing byte allocation.

Type 53 54, you will see immediately in right column new characters 'ST' is added



10. Clicking on Save toolbar or press **Ctrl-S** to apply the changes

11. Validate the changes by opening the updated DLIS file in any DLIS viewer app. In this example, we use DLIS InfoView, a freeware data utility from SLB. You should now see the Well Name changed to AAA-1ST, while the other DLIS header parameters and channels/curves data remain the same.



Tips and Cautions

⚠ Do NOT use the 'Insert' or 'Alt+I' command when editing hexadecimal values — this will corrupt your DLIS file. Always modify within the existing byte allocation. Always validate the modified file in a DLIS viewer before final delivery or further processing.

Editing DLIS headers using HexEdit is a simple yet effective technique, especially in environments where specialized tools are unavailable. It helps maintain data quality and consistency during time-critical operations. With careful attention, this approach ensures well log data remains intact, reliable, and ready for further analysis.

5. Conclusion

Consistent digital log data preparation underpins reliable petrophysical analysis and interpretation. Through automated validation, null values standardization, and structured file conversion, practitioners can ensure LAS and DLIS datasets are fit-for-purpose, reproducible, and petrophysical package and data platform-compliant. The workflows presented here complement formal data management standards and can be easily customized or integrated into corporate QC pipelines.

Data-Driven Petrophysical Applications: Practical Use Cases in Log QC, Synthetic Log Generation, and Multi-Well Rock Typing

1. Introduction

The upstream oil and gas industry is experiencing a significant digital transformation. Data-driven workflows are redefining how subsurface professionals acquire, process, and interpret well log data. Well logs, derived from wireline, LWD, and MWD tools and supported by core measurements, remain essential for petrophysical interpretation, reservoir characterization, optimal well placement, and informed production decision-making.

Despite technological advances, well log data often suffer from quality and completeness challenges. Issues such as tool calibration drift, borehole irregularities, missing log curves, or inconsistent data formats persist across projects.

These problems increase uncertainty in interpretation and slow decision-making.

Artificial Intelligence (AI), through the use of Machine Learning (ML) techniques, represents a paradigm shift by enabling systems to learn from data, make predictions, and improve performance autonomously, transforming traditional approaches to problem-solving. Instead of relying solely on manual QC or deterministic equations, petrophysicists can apply ML models to automate repetitive processes, discover hidden patterns, and augment interpretation accuracy. Data-driven techniques enable reproducibility, scalability, and objective validation of log-derived interpretations.

This guideline currently covers three practical and complementary ML-augmented petrophysical applications: **Outlier Detection for Log Quality Control**, **Synthetic Sonic Log Generation**, and **Multi-Well Rock Typing using Clustering** (two additional high-value use cases— Porosity **and Permeability Prediction from Conventional**

Logs and **Automated Flagging of Borehole Environment Effects** are planned for inclusion in the forthcoming updated edition of this book).

1. Outlier Detection for Log QC— detecting anomalous log readings due to measurement artifacts or borehole effects using algorithms such as Isolation Forest and DBSCAN.
2. Synthetic Sonic Log Generation— predicting missing Sonic (DT) log values using regression models like Random Forest and XGBoost.
3. Multi-Well Rock Typing with Clustering— grouping multi-well log data into electrofacies using unsupervised clustering methods such as K-Means or Gaussian Mixture.

Each use case demonstrates a data-centric workflow where ML complements domain expertise to ensure data integrity, improve interpretive consistency, and support integrated reservoir modeling.

2. Use Case 1: Outlier Detection for Log QC

2.1 Background

Well log anomalies or “spikes” often arise from sensor malfunction, tool sticking, borehole washouts, or transmission errors. If undetected, these artifacts distort computed properties such as porosity or water saturation. Traditional QC relies heavily on manual visual checks—time-consuming and subjective.

Unsupervised anomaly detection using ML can automate this process. The Isolation Forest (IF) algorithm isolates anomalous data points by randomly selecting features and split values to create decision trees. Outliers are identified by their short average path length in the tree structure, making the method efficient for large datasets.

2.2 Workflow Overview

1. Data Loading– Read LAS files using `lasio`, extract curves such as GR, RHOB, and NPHI.
2. Preprocessing– Handle null values, align depths, and standardize input features.
3. Anomaly Detection– Train Isolation Forest or DBSCAN to detect spikes.
4. Visualization & Review– Plot flagged data for inspection and validation.

5. Correction– Replace outliers via interpolation or smoothing filters.

LogsSpikeDetection_IsoForest.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest
import lasio
from tkinter import Tk, filedialog

# Function to load LAS files interactively
def load_las_files():
    Tk().withdraw()  # Hide the root window
    file_paths = filedialog.askopenfilenames(
        title="Select LAS File(s)",
        filetypes=[("LAS Files", "*.las")]
    )
    return file_paths

# Process each LAS file
def process_las_file(file_path, curve_name):
    las = lasio.read(file_path)
    if curve_name not in las.curves:
        raise ValueError(f"Curve '{curve_name}' not found in {file_path}")

    # Extract depth and specified curve
    depth = las["DEPT"]  # Assuming 'DEPT' is the depth curve name
    curve_data = las[curve_name]

    # Create a DataFrame
    data = pd.DataFrame({'Depth': depth, curve_name: curve_data})

    # Detect spikes using Isolation Forest
    iso_forest = IsolationForest(contamination=0.01, random_state=42)
    data['Anomaly_Score'] = iso_forest.fit_predict(data[[curve_name]])
    data['Anomaly'] = data['Anomaly_Score'] == -1

    # Plot the curve with anomalies highlighted
    plt.figure(figsize=(10, 6))
    plt.plot(data['Depth'], data[curve_name], label=curve_name, color='blue')
    plt.scatter(data['Depth'][data['Anomaly']], data[curve_name]
[data['Anomaly']],
                color='red', label='Detected Spikes', zorder=5)
    plt.xlabel('Depth (m)')
    plt.ylabel(f'{curve_name} (API)')
    plt.title(f'{curve_name} Log with Detected Spikes in {file_path}')
    plt.legend()
    plt.show()

# Main script
```

```
if __name__ == "__main__":
    print("Select LAS file(s) for processing...")
    las_files = load_las_files()

    if not las_files:
        print("No files selected. Exiting.")
    else:
        curve_name = input("Enter the curve name to process (e.g., GR for
Gamma Ray): ").strip()
        for file_path in las_files:
            try:
                print(f"Processing file: {file_path}")
                process_las_file(file_path, curve_name)
            except Exception as e:
                print(f"Error processing {file_path}: {e}")
```

2.3 Discussion

The Isolation Forest efficiently highlights outliers corresponding to spikes or sensor errors. Visual inspection confirms that flagged points typically coincide with acquisition noise or tool sticking. DBSCAN may also be used where anomalies form small, dense clusters.

This ML-based QC approach standardizes the process across wells, improves reproducibility, and significantly reduces manual QC time. It ensures a clean input dataset before performing reservoir property estimation or synthetic log prediction.

3. Use Case 2: Synthetic Sonic Log Generation

3.1 Background

Sonic travel-time (DT) logs are essential for porosity estimation, geomechanics, and seismic-well tie. However, missing DT data is common due to tool failure, environmental limitations, or economic constraints. Traditional imputation using empirical correlations (e.g., Gardner or Castagna) assumes fixed relationships that may not hold across lithologies.

Machine learning provides a flexible, data-driven solution. By training regression models on available logs— such as Gamma Ray (GR), Density (RHOB), Neutron Porosity (NPHI), and Resistivity— missing DT can be accurately predicted. Models like Random Forest or XGBoost capture nonlinear dependencies between features.

3.2 Workflow Steps

1. Data Preparation– Two wells are used: WELL_1 with full logs (training), WELL_2 with missing DT (prediction).
2. Feature Engineering– Normalize predictors and ensure consistent log naming.
3. Model Training– Train Random Forest using GR, RHOB, NPHI, and Resistivity as inputs.
4. Cross-Validation – Use 5-fold CV to measure model reliability via Mean Squared Error (MSE).
5. Prediction & Export– Apply trained model to WELL_2 to generate synthetic SonicDT and save as Techlog-ready CSV.

generate_SonicDT_log_x-val.py

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_score

# Load WELL_1 and WELL_2 data
data_well_1 = pd.read_csv("well_log_data_WELL_1.csv")  # Replace with your
actual file paths
data_well_2 = pd.read_csv("well_log_data_WELL_2.csv")

# Add WellName to distinguish between the two wells
data_well_1['WellName'] = 'WELL_1'
data_well_2['WellName'] = 'WELL_2'

# Combine the data into a single DataFrame
data = pd.concat([data_well_1, data_well_2], axis=0)

# Extract rows where Sonic DT is missing for WELL_2
missing_sonic_dt = data[(data['WellName'] == 'WELL_2') &
data['SonicDT'].isna()]

# Extract rows where Sonic DT is available for WELL_1 (training data)
training_data = data[data['WellName'] == 'WELL_1']

# Define features (excluding SonicDT)
features = ['GammaRay', 'Resistivity', 'Density', 'NeutronPorosity']
#features = ['Depth','GammaRay', 'Resistivity', 'Density', 'NeutronPorosity']

# Prepare the training data (features and target)
X = training_data[features]
y = training_data['SonicDT']
```

```
# Normalize the features (optional)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Initialize the RandomForestRegressor model
model = RandomForestRegressor(n_estimators=100, random_state=42)

# Perform cross-validation (e.g., 5-fold cross-validation)
cv_scores = cross_val_score(model, X_scaled, y, cv=5,
scoring='neg_mean_squared_error')

# The negative mean squared error needs to be converted to positive
cv_scores = -cv_scores

# Output the cross-validation results
print(f"Cross-Validation Mean Squared Errors: {cv_scores}")
print(f"Average Cross-Validation MSE: {cv_scores.mean()}")

# Train the model on the entire training data (since cross-validation is just
for evaluation)
model.fit(X_scaled, y)

# Prepare the missing data from WELL_2 for prediction
X_missing = missing_sonic_dt[features]
X_missing_scaled = scaler.transform(X_missing)

# Predict the missing Sonic DT values for WELL_2
predicted_sonic_dt = model.predict(X_missing_scaled)

# Fill in the missing values in the original data for WELL_2
data.loc[data['SonicDT'].isna() & (data['WellName'] == 'WELL_2'), 'SonicDT']
= predicted_sonic_dt

# Extract only the rows for WELL_2 with the filled Sonic DT values
well_2_filled = data[data['WellName'] == 'WELL_2']

# Save the updated WELL_2 data to a new CSV file
well_2_filled.to_csv("well_log_data_WELL_2_SonicDT_added.csv", index=False)

# Optional: You can also print out a message indicating the CSV file was
saved
print("CSV file with added Sonic DT values for WELL_2 has been saved as
'well_log_data_WELL_2_SonicDT_added.csv'")
```

3.3 Validation and Results

Validation ensures the predicted DT curve aligns with geological expectations. Crossplots (e.g., RHOB vs. NPHI) and depth-track comparisons between measured and synthetic DT help assess consistency. A good correlation ($R^2 > 0.9$) indicates reliability.

3.4 Discussion

Compared to empirical models, ML regression captures multi-dimensional relationships between logs, making it more adaptable across lithofacies. The approach enhances data completeness, enabling continuous velocity modeling, porosity estimation, and seismic interpretation. Synthetic log generation can be scaled to field-wide datasets once trained.

4. Use Case 3: Multi-Well Rock Typing with Clustering

4.1 Background

Multi-Well Rock typing links petrophysical data with geological facies and flow properties. Manual classification from core or thin sections is often limited in depth and coverage. By contrast, data-driven clustering uses continuous log data to classify electrofacies across multiple wells, improving consistency and scalability.

4.2 Workflow Steps

1. Data Preparation– Merge multi-well log datasets containing GR, RHOB, NPHI, and DT.
2. Standardization– Apply z-score normalization to prevent scale bias.
3. Clustering– Apply K-Means or Gaussian Mixture Models (GMM) to assign electrofacies clusters.
4. Facies Mapping– Relate cluster means to lithology using GR or core data.
5. Visualization– Generate crossplots and facies depth tracks for interpretation.

MultiWell_RockTyping_using_logs.py

```
#-- Multi-Well Rock Typing (Electrofacies Classification using #Well Logs)
# Objective: Automatically cluster multiple wells' log responses (GR, RHOB,
NPHI, DT) into consistent electrofacies.
# Use Case:

import pandas as pd
import numpy as np
import os
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
```

```
from tkinter import Tk, filedialog

# --- Step 1: Browse & Select Multiple Well Files ---
root = Tk()
root.withdraw()  # Hide main Tkinter window
file_paths = filedialog.askopenfilenames(
    title="Select Well Log CSV Files",
    filetypes=[("CSV files", "*.csv")]
)
root.update()

if not file_paths:
    raise FileNotFoundError("⚠ No CSV files selected. Please select one or
more well log files.")

dataframes = []
for file in file_paths:
    well_name = os.path.splitext(os.path.basename(file))
[0].replace("well_logs_", "")
    df = pd.read_csv(file)
    df["Well"] = well_name
    dataframes.append(df)

# Combine all selected wells
df_all = pd.concat(dataframes, ignore_index=True)
print(f"✅ Loaded {len(file_paths)} wells, total samples: {len(df_all)}")

# --- Step 2: Feature Selection & Scaling ---
features = ["GR", "RHOB", "NPHI", "DT"]
df_all = df_all.dropna(subset=features)  # Remove rows with missing key logs

X_scaled = StandardScaler().fit_transform(df_all[features])

# --- Step 3: K-Means Clustering (Global Model) ---
n_clusters = 4
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
df_all["Electrofacies"] = kmeans.fit_predict(X_scaled)

# --- Step 4: Facies Labeling (Based on Mean GR) ---
cluster_summary = df_all.groupby("Electrofacies")[["GR", "RHOB",
"NPHI"]].mean()
print("\nCluster Summary (All Wells):\n", cluster_summary)

facies_map = {}
gr_means = cluster_summary["GR"].sort_values()
for cluster in gr_means.index:
    if gr_means[cluster] < 80:
        facies_map[cluster] = "Sandstone"
    elif gr_means[cluster] < 100:
        facies_map[cluster] = "Siltstone"
    else:
        facies_map[cluster] = "Shale"

df_all["Facies_Label"] = df_all["Electrofacies"].map(facies_map)
```

```
# --- Step 5: Visualization Example (One Well) ---
plt.figure(figsize=(6, 5))
subset = df_all[df_all["Well"] == df_all["Well"].unique()[0]]
for label in subset["Facies_Label"].unique():
    part = subset[subset["Facies_Label"] == label]
    plt.scatter(part["GR"], part["RHOB"], label=label, s=40)
plt.xlabel("Gamma Ray (API)")
plt.ylabel("Bulk Density (g/cc)")
plt.title(f"Electrofacies Crossplot (Example Well:
{subset['Well'].iloc[0]})")
plt.legend()
plt.show()

# --- Step 6: Depth Track Visualization per Well ---
facies_colors = {"Sandstone": "gold", "Siltstone": "green", "Shale": "gray"}
for well in df_all["Well"].unique():
    wdf = df_all[df_all["Well"] == well]
    plt.figure(figsize=(3, 8))
    plt.scatter(wdf["Facies_Label"], wdf["Depth"],
c=wdf["Facies_Label"].map(facies_colors), s=25)
    plt.gca().invert_yaxis()
    plt.xlabel("Facies")
    plt.ylabel("Depth (m)")
    plt.title(f"Facies vs Depth Track: {well}")
    plt.show()

# --- Step 7: Save Combined Techlog-Ready Output ---
output_cols = ["Well", "Depth", "Electrofacies", "Facies_Label"]
output_file = "field_electrofacies_combined.csv"
df_all[output_cols].to_csv(output_file, index=False)

print(f"\n✅ Combined Techlog-ready electrofacies file saved as:
{output_file}")
print(f"Includes {len(df_all)} total samples from {len(file_paths)} wells.")
```

4.4 Interpretation and Discussion

Cluster analysis enables the identification of distinct lithologic groupings—for example, low gamma-ray (GR) values combined with high bulk density (RHOB) typically indicate sandstone units, whereas high GR and lower RHOB are commonly associated with shale intervals. When core-derived lithofacies are available, validating the cluster results against these ground-truth observations significantly improves interpretation confidence.

The resulting electrofacies can be visualized as color-coded log tracks alongside conventional curves or exported for incorporation into reservoir characterization and geomodeling workflows.

Clustering-based rock typing ensures consistency across wells, reduces interpreter bias, and supports scalable facies modeling. GMM can provide probabilistic facies assignment where transitions are gradual.

5. Discussion and Integration

The three workflows collectively demonstrate how ML strengthens the digital subsurface value chain. Outlier detection ensures data quality, synthetic log generation improves data completeness, and clustering supports consistent geological interpretation.

Integrating these steps within a unified data platform allows geoscientists to perform automated QC, feature engineering, and predictive modeling seamlessly. When combined with visualization and version control, this enables true Digital Petrophysics—efficient, traceable, and repeatable.

6. Conclusion

Machine learning is redefining how petrophysical data are processed and interpreted. By adopting ML-driven outlier detection, synthetic log generation, and clustering-based rock typing, practitioners achieve improved accuracy, speed, and reproducibility. These techniques complement traditional domain expertise rather than replace it. They empower geoscientists to focus on interpretation and decision-making, supported by data-driven, objective analyses. As digital maturity grows, such ML-augmented workflows will become a standard component of modern reservoir evaluation.

About the Author

Edy Irnandi Sudjana is a Certified AI Professional (CAIP) and Energy Data Practitioner with over 20 years of experience in Subsurface & Well Data Management, Petrophysical Analysis, and Well Log Processing & QC. He leverages digitalization and AI-driven solutions to optimize upstream operations and subsurface workflows. Edy graduated with distinction from the Oxford Artificial Intelligence Programme, Saïd Business School, University of Oxford.

License

This eBook is produced in **PDF/UA** (Universal Accessibility) compliant format to support inclusive access.