Rules Based Enginee-2

Here is the updated code, including a utility for JSON validation and logging to avoid runtime failures caused by incorrect JSON values.

`feature_derivation_engine.py` (Updated with Logging and Validation)

```
import logging
from pyspark.sql import functions as F

class FeatureDerivationEngine:
    def __init__(self, config):
        self.config = config
        self.logger = logging.getLogger("FeatureDerivationEngine")

    def validate_config(self):
        """
        Validates the configuration JSON to ensure it adheres to the expected schema.
        """
        required_keys = ["features"]
        for key in required_keys:
            if key not in self.config:
                raise ValueError(f"Missing required key '{key}' in configuration.")

        for feature in self.config["features"]:
            if "output_column" not in feature or "type" not in feature or "rule" not in feature:
                raise ValueError(f"Each feature must have 'output_column', 'type', and 'rule'.")
            if feature["type"] == "complex" and "conditions" not in feature["rule"]:
                raise ValueError(f"Complex type feature '{feature['output_column']}' must include 'conditions'.")
            if feature["type"] in ["transformation", "arithmetic"] and "expression" not in feature["rule"]:
                raise ValueError(f"Transformation type feature '{feature['output_column']}' must include 'expression'.")

        self.logger.info("Configuration validation successful.")

    def apply_complex_condition(self, df, conditions, default_value):
        exprs = None
        for condition in conditions:
            parent_condition = F.expr(condition["condition"])
            if "sub_conditions" in condition:
                sub_exprs = F.when(F.expr(condition["sub_conditions"][0]["condition"]), condition["sub_conditions"][0]["value"])
                for sub in condition["sub_conditions"][1:]:
                    sub_exprs = sub_exprs.when(F.expr(sub["condition"]), sub["value"])
                sub_exprs = sub_exprs.otherwise(default_value)
                parent_expr = F.when(parent_condition, sub_exprs)
            else:
                parent_expr = F.when(parent_condition, condition["value"])
            exprs = parent_expr if exprs is None else exprs.otherwise(parent_expr)
        exprs = exprs.otherwise(default_value)
        return exprs

    def apply_transformation(self, df, rule, output_col):
        expression = rule["expression"]
        return df.withColumn(output_col, F.expr(expression))

    def apply_rules(self, df):
        self.logger.info("Starting feature derivation process.")
        for feature in self.config["features"]:
            output_col = feature["output_column"]
            rule_type = feature["type"]
            rule = feature["rule"]

            self.logger.debug(f"Processing feature '{output_col}' of type '{rule_type}'.")

            if rule_type == "complex":
                conditions = rule["conditions"]
                default_value = rule.get("default", "unknown")
                complex_expr = self.apply_complex_condition(df, conditions, default_value)
                df = df.withColumn(output_col, complex_expr)

            elif rule_type in ["transformation", "arithmetic", "string"]:
                df = self.apply_transformation(df, rule, output_col)

        self.logger.info("Feature derivation process completed.")
        return df

```

`main.py` (Updated with JSON Validation and Logging)

```
import json
import sys
import logging
from pyspark.sql import SparkSession
from src.feature_derivation_engine import FeatureDerivationEngine

def configure_logging():
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
        handlers=[
            logging.StreamHandler(sys.stdout),
        ],
    )

def main(config_path):
    # Configure logging
    configure_logging()
    logger = logging.getLogger("Main")

    # Initialize Spark
    spark = SparkSession.builder.appName("FeatureDerivation").getOrCreate()
    logger.info("Spark session initialized.")

    try:
        # Load Configuration
        logger.info("Loading configuration.")
        with open(config_path, "r") as f:
            config = json.load(f)

        # Initialize Engine and Validate Config
        engine = FeatureDerivationEngine(config)
        logger.info("Validating configuration.")
        engine.validate_config()

        # Sample Data
        data = [
            (1, 15, 5), (2, 15, 15), (3, 15, 25), (4, 30, 50), (5, 70, 50)
        ]
        columns = ["id", "age", "score"]
        df = spark.createDataFrame(data, columns)

        # Apply Feature Derivation
        logger.info("Applying feature derivation.")
        result_df = engine.apply_rules(df)

        # Show Results
        logger.info("Feature derivation completed. Showing results:")
        result_df.show()

    except Exception as e:
        logger.error(f"An error occurred: {e}")
        sys.exit(1)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: spark-submit main.py <config_file>")
        sys.exit(-1)

    config_path = sys.argv[1]
    main(config_path)

```

`test_feature_derivation_engine.py` (Updated with Validation Tests)

```
import unittest
from pyspark.sql import SparkSession
from src.feature_derivation_engine import FeatureDerivationEngine

class FeatureDerivationTest(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.spark = SparkSession.builder.appName("UnitTest").getOrCreate()

    @classmethod
    def tearDownClass(cls):
        cls.spark.stop()

    def test_valid_config(self):
        config = {
            "features": [
                {"output_column": "category", "type": "complex", "rule": {"conditions": [{"condition": "age < 18", "value": "minor"}]}}
            ]
        }
        engine = FeatureDerivationEngine(config)
        try:
            engine.validate_config()
        except Exception as e:
            self.fail(f"Validation failed unexpectedly: {e}")

    def test_invalid_config_missing_keys(self):
        config = {"features": [{"type": "complex", "rule": {"conditions": [{"condition": "age < 18", "value": "minor"}]}}]}
        engine = FeatureDerivationEngine(config)
        with self.assertRaises(ValueError):
            engine.validate_config()

    def test_apply_rules(self):
        config = {
            "features": [
                {"output_column": "category", "type": "complex", "rule": {"conditions": [{"condition": "age < 18", "value": "minor"}]}}
            ]
        }
        data = [(1, 15), (2, 25)]
        columns = ["id", "age"]
        df = self.spark.createDataFrame(data, columns)

        engine = FeatureDerivationEngine(config)
        engine.validate_config()
        result_df = engine.apply_rules(df)

        expected_data = [(1, 15, "minor"), (2, 25, "unknown")]
        expected_df = self.spark.createDataFrame(expected_data, columns + ["category"])
        self.assertEqual(sorted(result_df.collect()), sorted(expected_df.collect()))

if __name__ == "__main__":
    unittest.main()

```

Key Enhancements

1. **Validation Utility**:
   - Ensures the JSON configuration adheres to the expected schema before runtime.
   - Catches configuration errors early.
2. **Logging**:
   - Adds detailed logging to track the execution flow and debug issues.
   - Logs initialization, validation, and rule application steps.

These enhancements make the application robust and easy to debug during failures or misconfigurations.

Here is the full Python project structure for the updated PySpark code, including instructions on how to compile and package it into a deployable package.

Project Directory Structure

```
feature-derivation-project/
```

```
├── config/
│   └── sample_config.json          # Sample JSON configuration file
├── src/
│   ├── __init__.py
│   └── feature_derivation_engine.py # Feature Derivation Engine implementation
├── tests/
│   ├── __init__.py
│   └── test_feature_derivation_engine.py # Unit tests
├── main.py                         # Entry point for the PySpark job
├── requirements.txt                # Python dependencies
├── setup.py                        # Packaging script
└── README.md                       # Documentation
```

File Contents

1. `config/sample_config.json`

```
{
  "features": [
    {
      "output_column": "category",
      "type": "complex",
      "rule": {
        "conditions": [
          { "condition": "age < 18", "value": "child" },
          { "condition": "age >= 18 AND age < 60", "value": "adult" },
          { "condition": "age >= 60", "value": "senior" }
        ],
        "default": "unknown"
      }
    }
  ]
}
```

2. `src/__init__.py`

```
# Empty init file to mark this as a package
```

3. `src/feature_derivation_engine.py`

(Refer to the updated implementation provided earlier.)

4. `tests/__init__.py`

```
# Empty init file to mark this as a package
```

5. `tests/test_feature_derivation_engine.py`

(Refer to the updated test implementation provided earlier.)

6. `main.py`

(Refer to the updated implementation provided earlier.)

7. `requirements.txt`

```
pyspark==3.4.0

```

8. `setup.py`

```python
from setuptools import setup, find_packages

setup(
    name="feature_derivation_project",
    version="1.0.0",
    description="Feature Derivation Engine for PySpark",
    author="Your Name",
    packages=find_packages(),
    install_requires=[
        "pyspark==3.4.0",
    ],
    entry_points={
        "console_scripts": [
            "feature-derivation=main:main"
        ]
    },
)

```

9. `README.md`

~~~markdown
# Feature Derivation Project

This project implements a feature derivation engine in PySpark, supporting configurable and rule-based transformations.

## How to Use

### Configuration
Provide a JSON configuration file specifying the features and rules. Example:

```json
{
  "features": [
    {
      "output_column": "category",
      "type": "complex",
      "rule": {
        "conditions": [
          { "condition": "age < 18", "value": "child" },
          { "condition": "age >= 18 AND age < 60", "value": "adult" },
          { "condition": "age >= 60", "value": "senior" }
        ],
        "default": "unknown"
      }
    }
  ]
}

~~~

Run the Application

Run the project using `spark-submit`:

```
spark-submit main.py config/sample_config.json

```

Testing

Run unit tests with `unittest`:

```
python -m unittest discover -s tests

```

Packaging

To build a deployable package:

```
python setup.py sdist

```

Deploy the generated `.tar.gz` file as needed.

~~~

How to Compile and Package

1. **Install Dependencies**
Set up a virtual environment and install dependencies:

```bash
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt

~~~

### 2. Run Tests

Run the unit tests to ensure everything works:

```
python -m unittest discover -s tests

```

### 3. Package the Project

Package the project into a `.tar.gz` file:

```
python setup.py sdist

```

This will generate a `dist/` directory containing the package file, e.g., `feature_derivation_project-1.0.0.tar.gz`.

### 4. Deploy and Run

Use `spark-submit` to run the packaged project:

```
spark-submit --py-files dist/feature_derivation_project-1.0.0.tar.gz main.py config/sample_config.json

```

### Logging Notes

The logging information is displayed on the console and can be configured to write to a file if needed. Update the `logging.basicConfig()` in `main.py` for file logging.

This setup ensures your project is modular, testable, and deployable!