⭐ Rules Based Enginee

Here's a full Python project structure to implement the enhanced PySpark code, handle runtime configuration, and include unit tests.

Project Structure

```
feature_derivation/
├── config/
│   └── example_config.json  # Sample configuration file
├── src/
│   ├── __init__.py
│   ├── feature_derivation_engine.py  # Main logic for feature derivation
│   └── main.py  # Entry point for spark-submit
├── tests/
│   ├── __init__.py
│   └── test_feature_derivation_engine.py  # Unit test cases
├── requirements.txt  # Dependencies
└── README.md  # Project documentation

```

`example_config.json` (Configuration File)

```
{
  "features": [
    {
      "output_column": "category",
      "type": "complex",
      "rule": {
        "conditions": [
          {
            "condition": "age < 18",
            "sub_conditions": [
              {"condition": "score < 10", "value": "child with score less than 10"},
              {"condition": "score >= 10 and score < 20", "value": "child with score between 10 and 20"},
              {"condition": "score >= 20", "value": "child with score more than 20"}
            ]
          },
          {
            "condition": "age >= 18 and age < 60",
            "value": "adult"
          },
          {
            "condition": "age >= 60",
            "value": "senior"
          }
        ],
        "default": "unknown"
      }
    },
    {
      "output_column": "score_bucket",
      "type": "transformation",
      "rule": {
        "expression": "CASE WHEN score < 10 THEN 'low' WHEN score BETWEEN 10 AND 20 THEN 'medium' ELSE 'high' END"
```

```
      }
    },
    {
      "output_column": "age_squared",
      "type": "arithmetic",
      "rule": {
        "expression": "age * age"
      }
    }
  ]
}

```

`feature_derivation_engine.py` **(Engine Logic)**

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

class FeatureDerivationEngine:
    def __init__(self, config):
        self.config = config

    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):
        for feature in self.config["features"]:
            output_col = feature["output_column"]
            rule_type = feature["type"]
            rule = feature["rule"]

            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)
```

```
        return df

```

`main.py` (Entry Point)

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

def main(config_path):
    # Initialize Spark
    spark = SparkSession.builder.appName("FeatureDerivation").getOrCreate()

    # Load Configuration
    with open(config_path, "r") as f:
        config = json.load(f)

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

    # Show Results
    result_df.show()

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` (Unit 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_apply_rules(self):
        # Mock Configuration
        config = {
            "features": [
                {
                    "output_column": "category",
                    "type": "complex",
                    "rule": {
                        "conditions": [
                            {"condition": "age < 18", "value": "minor"},
                            {"condition": "age >= 18", "value": "adult"}
                        ],
                        "default": "unknown"
                    }
                }
            ]
        }

        # Sample Data
        data = [(1, 15), (2, 25), (3, None)]
        columns = ["id", "age"]
        df = self.spark.createDataFrame(data, columns)

        # Apply Feature Derivation
        engine = FeatureDerivationEngine(config)
        result_df = engine.apply_rules(df)

        # Expected Results
        expected_data = [(1, 15, "minor"), (2, 25, "adult"), (3, None, "unknown")]
        expected_df = self.spark.createDataFrame(expected_data, columns + ["category"])

        # Assert Equality
        self.assertEqual(sorted(result_df.collect()), sorted(expected_df.collect()))

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

```

`requirements.txt`

```
pyspark==3.4.1
unittest2==1.1.0

```

Running the Project

1. **Run the PySpark Job**:

```
spark-submit src/main.py config/example_config.json

```

2. **Run Unit Tests**:

```
python -m unittest discover -s tests

```

This setup provides a production-ready project that supports runtime configuration and unit testing, ensuring maintainability and scalability.