cleaned folder in data lake
==========

customers

loans

loan repayments

loan defaulters delinquent

loan defaulters public record, barkruptcies, enquiries

loan defaulter (any public record - only member id)

loan score
==========

higher the score, higher the chances of getting loan

based on 3 things this loan score is calculated.

=> payment history (loan repayments)

=> financial health (customers)

=> defaulters history (delinq, public records, backruptcies, enquiries)

- delinq
- additional columns (public records, public_records_bankruptcies, enquiries_6mnths)

```
loans_def_detail_records_enq_df = spark.sql("select member_id, pub_rec, pub_rec_bankruptcies, inq_last_6mths  from loan_defaulters")

loans_def_processed_df = loans_def_raw_df.withColumn("delinq_2yrs", col("delinq_2yrs").cast("integer")).fillna(0, subset=["delinq_2yrs"])

loans_def_p_pub_rec_df = loans_def_processed_df.withColumn("pub_rec", col("pub_rec").cast("integer")).fillna(0, subset=["pub_rec"])
```

```
loans_def_p_pub_rec_bankruptcies_df =
loans_def_p_pub_rec_df.withColumn("pub_rec_bankruptcies",
col("pub_rec_bankruptcies").cast("integer")).fillna(0,
subset=["pub_rec_bankruptcies"])

loans_def_p_inq_last_6mths_df =
loans_def_p_pub_rec_bankruptcies_df.withColumn("inq_last_6mths",
col("inq_last_6mths").cast("integer")).fillna(0, subset=["inq_last_6mths"])



loans_def_p_inq_last_6mths_df.createOrReplaceTempView("loan_defaulters"
)

loans_def_detail_records_enq_df = spark.sql("select member_id, pub_rec,
pub_rec_bankruptcies, inq_last_6mths  from loan_defaulters")

loans_def_detail_records_enq_df

loans_def_detail_records_enq_df.write \
.option("header",True) \
.format("csv") \
.mode("overwrite") \
.option("path","/public/trendytech/lendingclubproject/cleaned/loans_defaulters
_detail_records_enq_csv") \
.save()

loans_def_detail_records_enq_df.write \
.format("parquet") \
.mode("overwrite") \
.option("path","/public/trendytech/lendingclubproject/cleaned/loans_defaulters
_detail_records_enq_parquet") \
.save()
```

==========

customers - done

loans -

loan repayments

loan defaulters delinquent

loan defaulters public record, barkruptcies, enquiries

=> a few teams have to analyse the cleaned data. so we have to create permanent tables on top of cleaned data.

Dataframe

there are 2 kind of tables

table = data + metadata

1. managed tables - when we drop a managed table both the data and metadata is dropped.

2. external tables - when we drop a external table only metadata is dropped.

files
tables - drop

table - data + metadata

data - warehouse
metadata - metastore will be a hive metastore...

```
spark.sql("create database itv005857_lending_club")

spark.sql("""
    CREATE EXTERNAL TABLE
itv005857_lending_club.customers(member_id string,emp_title
string,emp_length int, home_ownership string,annual_income
float,address_state          string,address_zipcode   string,address_country
string,grade string,sub_grade    string,verification_status
string,total_high_credit_limit     float,application_type
string,join_annual_income float,verification_status_joint string,ingest_date
timestamp) stored as parquet
    LOCATION
'/public/trendytech/lendingclubproject/cleaned/customers_parquet'
""")

spark.sql("select * from itv005857_lending_club.customers")
```

==========

```
spark.sql("""
    CREATE EXTERNAL TABLE itv005857_lending_club.loans(loan_id
string,member_id string,loan_amount float,funded_amount
float,loan_term_years integer,interest_rate float,monthly_installment
```

```
float,issue_date string,loan_status string,loan_purpose string,loan_title
string,ingest_date timestamp) stored as parquet
    LOCATION '/public/trendytech/lendingclubproject/cleaned/loans_parquet'
""")

spark.sql("select * from itv005857_lending_club.loans")


==============

spark.sql("""
CREATE EXTERNAL TABLE
itv005857_lending_club.loans_repayments(loan_id
string,total_principal_received float,total_interest_received
float,total_late_fee_received float,total_payment_received
float,last_payment_amount float,last_payment_date string,next_payment_date
string,ingest_date timestamp) stored as parquet
    LOCATION
'/public/trendytech/lendingclubproject/cleaned/loans_repayments_parquet'
""")

spark.sql("select * from itv005857_lending_club.loans_repayments")

==============

spark.sql("""
CREATE EXTERNAL TABLE
itv005857_lending_club.loans_defaulters_delinq(member_id
string,delinq_2yrs integer,delinq_amnt float,mths_since_last_delinq integer)
stored as parquet
    LOCATION
'/public/trendytech/lendingclubproject/cleaned/loans_defaulters_delinq_parque
t'
""")

spark.sql("select * from itv005857_lending_club.loans_defaulters_delinq")

==============

spark.sql("""
CREATE EXTERNAL TABLE
itv005857_lending_club.loans_defaulters_detail_rec_enq(member_id string,
pub_rec integer, pub_rec_bankruptcies integer, inq_last_6mths integer) stored
as parquet
```

```
    LOCATION
'/public/trendytech/lendingclubproject/cleaned/loans_defaulters_detail_records
_enq_parquet'
""")

spark.sql("select * from
itv005857_lending_club.loans_defaulters_detail_rec_enq")

==============

spark.sql("drop table itv005857_lending_club.customers")


========================
```

A complete view of these 5 datasets

one single view

- need the most upto date data

join (5 tables)

24 hours

if we create a view on top of it...

underlying tables (24 hours)

- they really need quick access to this view data...

we have a weekly job that runs every 7 days one time.

the join of 5 tables is done & the results are put in a table...

even though the results are faster in this case but the data will be little older...

itv005857_lending_club.customers_loan_t - a managed table

quick access with little older data (max 7 day old)

slow access with newer data (max 1 day old)

==========

Loan score
==========

if loan score is high, higher the chances of loan getting approved.

1. loan repayment history (last payment, total payment received)

2. loan defaulters history (delinq 2 yrs, pub_rec, pub_rec_bankruptcies, inq_last_6mths)

3. financial health data (home ownership, loan status, funded amount, grade pts)

the tables that we have already created

customers - home ownership, grade pts, high credit limit

loans - monthly installment, loan status, funded amount

loans_repayments - last payment, total payment received

loans_defaulters_delinq - delinq 2 yrs

loans_defaulters_detail_rec_enq - pub_rec, pub_rec_bankruptcies, inq_last_6mths

payment_history = 20%
loan_default_history = 45%
financial health = 35%

customers - member_id
loans_defaulters_delinq - member_id
loans_defaulters_detail_rec_enq - member_id

==========

customers - 3157 (member_id) bad records

bad_data_loans_defaulters_delinq_df - 173

bad_data_loans_defaulters_detail_rec_enq_df - 3189

a consolidate file which has all the unique not repeating member ids from the above 3...

df1
df2 (union)
df3

distinct

I will store it on hdfs..

==========

1. loan repayment history (last payment, total payment received)

2. loan defaulters history (delinq 2 yrs, pub_rec, pub_rec_bankruptcies, inq_last_6mths)

3. financial health data (home ownership, loan status, funded amount, grade pts)

Notebooks - Exploration purpose

Visual Studio code

Pycharm

==========

Pyspark project

Macbook - python 3.10 installed (global version)

project-1
retailproject (global version)
pyspark 3.2.1
pytest

project-2
lendingclubproject (3.8)
pyspark 3.5
different version of pytest

python version installed on your laptop - python 3

Global python version = Python 3.10.6

pipenv = pip + venv

venv - you create a isolated virual environment for your project

5 projects in your system

virtual env1 for project1
- python 3.8
- pyspark 3.2.1

virtaul env2 for project2
- python 3.10
- pyspark 3.5

pip install pyspark

pip install pytest

python3 in your laptop
pip

pip - to install additional packages (package management)

venv - a specific env for each project (Virtual environment)

own version of python and own version of packages

global installation

pip install pipenv

old way
=======

- create a virual environment
- manually activate the env
- install the packages used in the project

pipenv install pyspark

/Users/trendytech/.local/share/virtualenvs/demoproject-A-e7zUHY

1. pipenv shell (to activate the environment)

2. python

=====

pipenv run python

pipenv install pytest --dev

pipenv uninstall pytest (to uninstall a package)

pipenv --rm (to get rid of the env)

pipenv install (to create a new env based on pipfile)

pyenv is to manage python versions...

==========

application.conf
==========

[LOCAL]
customers.file.path = data/customers.csv
orders.file.path = data/orders.csv

[TEST]
customers.file.path = data/customers.csv
orders.file.path = data/orders.csv

[PROD]
customers.file.path = data/customers.csv
orders.file.path = data/orders.csv

pyspark.conf
==========

[LOCAL]
spark.app.name = retail-local

[TEST]
spark.app.name = retail-test

```
spark.executor.instances = 3
spark.executor.cores = 5
spark.executor.memory = 15GB

[PROD]
spark.app.name = retail-prod
spark.executor.instances = 3
spark.executor.cores = 5
spark.executor.memory = 15GB
```

ConfigReader.py
==========

```
import configparser
from pyspark import SparkConf

# loading the application configs in python dictionary
def get_app_config(env):
    config = configparser.ConfigParser()
    config.read("configs/application.conf")
    app_conf = {}
    for (key, val) in config.items(env):
        app_conf[key] = val
    return app_conf

# loading the pyspark configs and creating a spark conf object
def get_pyspark_config(env):
    config = configparser.ConfigParser()
    config.read("configs/pyspark.conf")
    pyspark_conf = SparkConf()
    for (key, val) in config.items(env):
        pyspark_conf.set(key, val)
    return pyspark_conf
```

DataManipulation.py
==========

```
from pyspark.sql.functions import *

def filter_closed_orders(orders_df):
    return orders_df.filter("order_status = 'CLOSED'")

def join_orders_customers(orders_df, customers_df):
    return orders_df.join(customers_df, "customer_id")

def count_orders_state(joined_df):
```

```
    return joined_df.groupBy('state').count()
```

DataReader.py
==========

```
from lib import ConfigReader

#defining customers schema
def get_customers_schema():
    schema = "customer_id int,customer_fname string,customer_lname
string,username string,password string,address string,city string,state
string,pincode string"
    return schema

# creating customers dataframe
def read_customers(spark,env):
    conf = ConfigReader.get_app_config(env)
    customers_file_path = conf["customers.file.path"]
    return spark.read \
        .format("csv") \
        .option("header", "true") \
        .schema(get_customers_schema()) \
        .load(customers_file_path)

#defining orders schema
def get_orders_schema():
    schema = "order_id int,order_date string,customer_id int,order_status
string"
    return schema

#creating orders dataframe
def read_orders(spark,env):
    conf = ConfigReader.get_app_config(env)
    orders_file_path = conf["orders.file.path"]
    return spark.read \
        .format("csv") \
        .option("header", "true") \
        .schema(get_orders_schema()) \
        .load(orders_file_path)
```

Utils.py
=========

```
from pyspark.sql import SparkSession
```

```
from lib.ConfigReader import get_spark_conf

def get_spark_session(env):
    if env == "LOCAL":
        return SparkSession.builder \
            .config(conf=get_spark_conf(env)) \
            .master("local[2]") \
            .getOrCreate()
    else:
        return SparkSession.builder \
            .config(conf=get_spark_conf(env)) \
            .enableHiveSupport() \
            .getOrCreate()
```

application_main.py
==========

```
import sys
from lib import DataManipulation, DataReader, Utils
from pyspark.sql.functions import *

if __name__ == '__main__':

    if len(sys.argv) < 2:
        print("Please specify the environment")
        sys.exit(-1)

    job_run_env = sys.argv[1]

    print("Creating Spark Session")

    spark = Utils.get_spark_session(job_run_env)

    print("Created Spark Session")

    orders_df = DataReader.read_orders(spark,job_run_env)

    orders_filtered = DataManipulation.filter_closed_orders(orders_df)
```

```
    customers_df = DataReader.read_customers(spark,job_run_env)

    joined_df =
DataManipulation.join_orders_customers(orders_filtered,customers_df)

    aggregated_results = DataManipulation.count_orders_state(joined_df)

    aggregated_results.show()

    print("end of main")
```

Unit testing
==========

Testing small units of code

if we have written our code in a modular way, then we can test each function separately.

unittest

pytest (as part of best practises)

how do I install pytest?

pipenv install pytest

we want to identify which functions to test

=> read_customers_df - 12435
=> read_orders_df - 68883
=> filter_closed_orders - 7556
=> read_app_config

you have to create a new file where you can write the unit tests

the filename where you write your unit test cases should either start with test or end with test

test_retail_proj.py

to run the unit test cases we need

python -m pytest

/Users/trendytech/.local/share/virtualenvs/RetailAnalysis-xoFHaijo/bin/python -m pytest

/Users/trendytech/.local/share/virtualenvs/RetailAnalysis-xoFHaijo/bin/python -m pytest -v

setup should be done as part of fixture and should not be going in a test case..

fixture is to write the setup code

setup is done

unit test is run...

try writing your fixtures in a file names as

conftest.py

setup - fixture

do unit testing - define unit test

teardown - releasing the resources

```
@pytest.fixture
def spark():
    spark_session = get_spark_session("LOCAL")
    return spark_session

@pytest.fixture
def spark():
    spark_session = get_spark_session("LOCAL")
    yield spark_session
    spark_session.stop()
```

python -m pytest --fixtures

I want to write one more test case

I want to test count_orders_state

whether its doing aggregation and count properly or not...

count_orders_state(customers_df)

expected results

data -> test_result -> state_aggregate.csv

state_aggregate.csv
==========
AZ,213
SC,41
LA,63
MN,39
NJ,219
DC,42
OR,119
VA,136
RI,15
KY,35
MI,254
NV,103
WI,64
ID,9
CA,2012
CT,73
MT,7
NC,150
MD,164
DE,23
MO,92
IL,523
WA,72
ND,14
AL,3
IN,40
OH,276
TN,104
NM,73
IA,5
PA,261
NY,775
TX,635
WV,16
GA,169
MA,113
KS,29
CO,122
FL,374

AR,12
OK,19
PR,4771
UT,69
HI,87

markers
========

100 test cases

@pytest.mark.transformation

==========

```
@pytest.mark.latest()
def test_check_closed_count(spark):
    orders_df = read_orders(spark,"LOCAL")
    filtered_count = filter_orders_generic(orders_df,"CLOSED").count()
    assert filtered_count == 7556

@pytest.mark.latest()
def test_check_pendingpayment_count(spark):
    orders_df = read_orders(spark,"LOCAL")
    filtered_count =
filter_orders_generic(orders_df,"PENDING_PAYMENT").count()
    assert filtered_count == 15030

@pytest.mark.latest()
def test_check_complete_count(spark):
    orders_df = read_orders(spark,"LOCAL")
    filtered_count = filter_orders_generic(orders_df,"COMPLETE").count()
    assert filtered_count == 22899
```

Logging in apache spark
==========

till now we have seen print statements..

what is the issue with print statements?

1. you cannot set the priorities or logging level

info, warn, error, fatal etc...

2. you have written an application, 1000 print statements..

you have to manually comment all of those, or remove all of those..

3. print statements make your application slower..

so the best way to solve all of these issues is to implement a logging framework.

Log4j is a logging framework..

spark internally uses log4j for its logging, so we can reuse the same for our application level logs..

so we can get an instance of log4j object from spark session

Utils.py (adding one extra config while creating spark session)

log4j.properties (new file)

logger.py (new file)

application_main.py

==========

logging levels

debug < info <warn <error <fatal

if in your log4j.properties file if you have defined lets say logging level as warn

==========

target location

console
file

==========

message format

TRENDYTECH
UPLIFT YOUR CAREER!