Notebook version: 0.1.0

Updated date: 10/10/2023

Time Series Features Demo

This notebook demonstrates feature store with time series features. It includes an end-2-end ML experiment cycle: feature creation, training and inference. It also demonstrate the interoperation between Feature Store and Model Registry.

It uses public NY taxi trip data to compute features. The public data can be downloaded from: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

In [ ]:
```
from snowflake.snowpark import Session
from snowflake.snowpark import functions as F, types as T
from snowflake.ml.feature_store import (
    FeatureStore,
    FeatureView,
    Entity,
    CreationMode
)
from snowflake.ml.utils.connection_params import SnowflakeLoginOptions
from snowflake.snowpark.types import TimestampType
from snowflake.ml._internal.utils import identifier
import datetime
```

Setup Snowflake connection

For detailed session connection config, please follow this tutorial.

In [ ]:
```
session = Session.builder.configs(SnowflakeLoginOptions()).create()
```

Prepare test data

Download Yellow Taxi Trip Records data (Jan. 2016) from https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page if you don't have it already. Rename `PARQUET_FILE_LOCAL_PATH` with your local file path. Rename `TEST_DATASET_DB` and `TEST_DATASET_SCHEMA` with your location.

Below code create a table with the test dataset.

In [ ]:
```
PARQUET_FILE_NAME = f"yellow_tripdata_2016-01.parquet"
PARQUET_FILE_LOCAL_PATH = f"file://~/Downloads/{PARQUET_FILE_NAME}"
TEST_DATASET_DB = "SNOWML_FEATURE_STORE_TEST_DB"
TEST_DATASET_SCHEMA = 'TEST_DATASET'
```

```python
def get_destination_table_name(original_file_name: str) -> str:
    return original_file_name.split(".")[0].replace("-", "_").upper()

table_name = get_destination_table_name(PARQUET_FILE_NAME)
session.file.put(PARQUET_FILE_LOCAL_PATH, session.get_session_stage())

df = session.read \
    .parquet(f"{session.get_session_stage()}/{PARQUET_FILE_NAME}")
for old_col_name in df.columns:
    df = df.with_column_renamed(
        old_col_name,
        identifier.get_unescaped_names(old_col_name)
    )

full_table_name = f"{TEST_DATASET_DB}.{TEST_DATASET_SCHEMA}.{table_name}"
df.write.mode("ignore").save_as_table(full_table_name)
rows_count = session.sql(
    f"SELECT COUNT(*) FROM {full_table_name}").collect()[0][0]

print(f"{full_table_name} has total {rows_count} rows.")
print("Script completes successfully.")
```

In [ ]:
```python
source_df = session.table(
    f"{TEST_DATASET_DB}.{TEST_DATASET_SCHEMA}.{table_name}")

source_df = source_df.select(
    [
        "TRIP_DISTANCE",
        "FARE_AMOUNT",
        "PASSENGER_COUNT",
        "PULOCATIONID",
        "DOLOCATIONID",
        F.cast(source_df.TPEP_PICKUP_DATETIME / 1000000, TimestampType())
            .alias("PICKUP_TS"),
        F.cast(source_df.TPEP_DROPOFF_DATETIME / 1000000, TimestampType())
            .alias("DROPOFF_TS"),
    ]).filter(
        """DROPOFF_TS >= '2016-01-01 00:00:00'
            AND DROPOFF_TS < '2016-01-03 00:00:00'
        """)
source_df.show()
```

Create FeatureStore Client

Let's first create a feature store client.

We can pass in an existing database name, or a new database will be created upon the feature store initialization. Replace `DEMO_DB` and `DEMO_SCHEMA` with your database and schema names.

In [ ]:
```python
DEMO_DB = "FS_TIME_SERIES_EXAMPLE"
DEMO_SCHEMA = "AWESOME_FS"
```

```
session.sql(f"DROP DATABASE IF EXISTS {DEMO_DB}").collect()
session.sql(f"CREATE DATABASE IF NOT EXISTS {DEMO_DB}").collect()
session.sql(f"""CREATE OR REPLACE WAREHOUSE PUBLIC WITH
                WAREHOUSE_SIZE='XSMALL'
            """).collect()

fs = FeatureStore(
    session=session,
    database=DEMO_DB,
    name=DEMO_SCHEMA,
    default_warehouse="PUBLIC",
    creation_mode=CreationMode.CREATE_IF_NOT_EXIST,
)
```

Create and register new Entities

Create entity by giving entity name and join keys. Then register it to feature store.

In [ ]:
```
trip_pickup = Entity(name="trip_pickup", join_keys=["PULOCATIONID"])
trip_dropoff = Entity(name="trip_dropoff", join_keys=["DOLOCATIONID"])
fs.register_entity(trip_pickup)
fs.register_entity(trip_dropoff)
fs.list_entities().to_pandas()
```

Define feature pipeline

We will compute a few time series features in the pipeline here. Before we have ***value based range between*** in SQL, we will use a work around to mimic the calculation (NOTE: the work around won't be very accurate on computing the time series value due to missing gap filling functionality, but it should be enough for a demo purpose)

We will define two feature groups:

1. pickup features
    - Mean fare amount over the past 2 and 5 hours
2. dropoff features
    - Count of trips over the past 2 and 5 hours

This is a UDF computing time window end

We will later turn these into built in functions for feature store

In [ ]:
```
DEMO_STAGE = "FS_STAGE"

session.sql(f"CREATE OR REPLACE STAGE {DEMO_STAGE}").collect()
```

In [ ]:
```
@F.pandas_udf(
    name="vec_window_end",
    is_permanent=True,
```

```
    stage_location=f'@{DEMO_STAGE}',
    packages=["numpy", "pandas", "pytimeparse"],
    replace=True,
    session=session,
)
def vec_window_end_compute(
    x: T.PandasSeries[datetime.datetime],
    interval: T.PandasSeries[str],
) -> T.PandasSeries[datetime.datetime]:
    import numpy as np
    import pandas as pd
    from pytimeparse.timeparse import timeparse

    time_slice = timeparse(interval[0])
    if time_slice is None:
        raise ValueError(f"Cannot parse interval {interval[0]}")
    time_slot = (x - np.datetime64('1970-01-01T00:00:00')) \
        // np.timedelta64(1, 's') \
        // time_slice * time_slice + time_slice
    return pd.to_datetime(time_slot, unit='s')
```

Define feature pipeline logics

In [ ]:

```
from snowflake.snowpark import Window
from snowflake.snowpark.functions import col

# NOTE: these time window calculations are approximates and are not handling

def pre_aggregate_fn(df, ts_col, group_by_cols):
    df = df.with_column("WINDOW_END",
            F.call_udf("vec_window_end", F.col(ts_col), "15m"))
    df = df.group_by(group_by_cols + ["WINDOW_END"]).agg(
            F.sum("FARE_AMOUNT").alias("FARE_SUM_1_hr"),
            F.count("*").alias("TRIP_COUNT_1_hr")
         )
    return df

def pickup_features_fn(df):
    df = pre_aggregate_fn(df, "PICKUP_TS", ["PULOCATIONID"])

    window1 = Window.partition_by("PULOCATIONID") \
        .order_by(col("WINDOW_END").desc()) \
        .rows_between(Window.CURRENT_ROW, 7)
    window2 = Window.partition_by("PULOCATIONID") \
        .order_by(col("WINDOW_END").desc()) \
        .rows_between(Window.CURRENT_ROW, 19)

    df = df.with_columns(
        [
            "SUM_FARE_2_hr",
            "COUNT_TRIP_2hr",
            "SUM_FARE_5_hr",
            "COUNT_TRIP_5hr",
        ],
        [
```

```
                F.sum("FARE_SUM_1_hr").over(window1),
                F.sum("TRIP_COUNT_1_hr").over(window1),
                F.sum("FARE_SUM_1_hr").over(window2),
                F.sum("TRIP_COUNT_1_hr").over(window2),
            ]
        ).select(
            [
                col("PULOCATIONID"),
                col("WINDOW_END").alias("TS"),
                (col("SUM_FARE_2_hr") / col("COUNT_TRIP_2hr"))
                    .alias("MEAN_FARE_2_hr"),
                (col("SUM_FARE_5_hr") / col("COUNT_TRIP_5hr"))
                    .alias("MEAN_FARE_5_hr"),
            ]
        )
    return df

def dropoff_features_fn(df):
    df = pre_aggregate_fn(df, "DROPOFF_TS", ["DOLOCATIONID"])
    window1 = Window.partition_by("DOLOCATIONID") \
        .order_by(col("WINDOW_END").desc()) \
        .rows_between(Window.CURRENT_ROW, 7)
    window2 = Window.partition_by("DOLOCATIONID") \
        .order_by(col("WINDOW_END").desc()) \
        .rows_between(Window.CURRENT_ROW, 19)

    df = df.select(
        [
            col("DOLOCATIONID"),
            col("WINDOW_END").alias("TS"),
            F.sum("TRIP_COUNT_1_hr").over(window1) \
                .alias("COUNT_TRIP_2_hr"),
            F.sum("TRIP_COUNT_1_hr").over(window2) \
                .alias("COUNT_TRIP_5_hr"),
        ]
    )
    return df

pickup_df = pickup_features_fn(source_df)
pickup_df.show()

dropoff_df = dropoff_features_fn(source_df)
dropoff_df.show()
```

Create FeatureViews and materialize

Once the FeatureView construction is done, we can materialize the FeatureView to the Snowflake backend and incremental maintenance will start.

In [ ]:

```
# NOTE:
# Due to a known issue on backend pipeline creation,
# if the source data is created right before the
# feature pipeline, there might be a chance for
# dataloss, so sleep for 60s for now.
```

```python
# This issue will be fixed soon in upcoming patch.

import time
time.sleep(60)
```

In [ ]:
```python
pickup_fv = FeatureView(
    name="trip_pickup_time_series_features",
    entities=[trip_pickup],
    feature_df=pickup_df,
    timestamp_col="ts"
)
pickup_fv = fs.register_feature_view(
    feature_view=pickup_fv,
    version="v1",
    refresh_freq="1 minute",
    block=True
)
```

In [ ]:
```python
dropoff_fv = FeatureView(
    name="trip_dropoff_time_series_features",
    entities=[trip_dropoff],
    feature_df=dropoff_df,
    timestamp_col="ts"
)
fs.register_feature_view(
    feature_view=dropoff_fv,
    version="v1",
    refresh_freq="1 minute",
    block=True
)
```

Explore FeatureViews

We can easily discover what are the materialized FeatureViews and the corresponding features with ***fs.list_feature_views()***.

We can also apply filters based on Entity name or FeatureView names.

In [ ]:
```python
fs.list_feature_views(entity_name="trip_pickup") \
    .select(["NAME", "VERSION", "ENTITIES", "FEATURE_DESC"]).show()
```

Generate training data and train a model

The training data generation will lookup **point-in-time correct** feature values and join with the spine dataframe. Optionally, you can also exclude columns in the generated dataset by providing `exclude_columns` argument.

In [ ]:
```python
spine_df = source_df.select([
    "PULOCATIONID",
    "DOLOCATIONID",
```

```
    "PICKUP_TS",
    "FARE_AMOUNT"])
training_data = fs.generate_dataset(
    spine_df=spine_df,
    features=[pickup_fv, dropoff_fv],
    materialized_table="yellow_tripdata_2016_01_training_data",
    spine_timestamp_col="PICKUP_TS",
    spine_label_cols = ["FARE_AMOUNT"]
)

training_data.df.show()
```

In [ ]:
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

training_pd = training_data.df.to_pandas()
X = training_pd.drop(["FARE_AMOUNT", "PICKUP_TS"], axis=1)
y = training_pd["FARE_AMOUNT"]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
X_train.head()
```

In [ ]:
```
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_squared_error

imp = SimpleImputer(missing_values=np.nan, strategy='mean')
estimator = make_pipeline(imp, LinearRegression())

reg = estimator.fit(X, y)
r2_score = reg.score(X_test, y_test)
print(r2_score * 100,'%')

y_pred = reg.predict(X_test)
print("Mean squared error: %.2f" % mean_squared_error(y_test, y_pred))
```

Log model with Model Registry

We can log the model along with its training dataset metadata with model registry.

In [ ]:
```
from snowflake.ml.registry import model_registry, artifact
import time

registry = model_registry.ModelRegistry(
    session=session,
    database_name="my_cool_registry",
    create_if_not_exists=True
)
```

In [ ]:
```
artifact_ref = registry.log_artifact(
    artifact_type=artifact.ArtifactType.DATASET,
    artifact_name="my_cool_dataset",
```

```python
    artifact_spec=training_data.to_json(),
    artifact_version="v1",
)
```

In [ ]:
```python
model_name = f"my_model_{time.time()}"

model_ref = registry.log_model(
    model_name=model_name,
    model_version="v1",
    model=estimator,
    artifacts=[artifact_ref],
)
```

Restore model and predict with latest features

Retrieve the training dataset from registry and construct dataframe of latest feature values. Then we restore the model from registry. Finally, we can predict with latest feature values.

In [ ]:
```python
# Prepare some source prediction data
pred_df = training_pd.sample(3, random_state=996)[
    ['PULOCATIONID', 'DOLOCATIONID', 'PICKUP_TS']]
pred_df = session.create_dataframe(pred_df)
pred_df = pred_df.select(
        'PULOCATIONID',
        'DOLOCATIONID',
        F.cast(pred_df.PICKUP_TS / 1000000, TimestampType()) \
    .alias('PICKUP_TS'))
```

In [ ]:
```python
# Enrich source prediction data with features
from snowflake.ml.dataset.dataset import Dataset

registered_artifact = registry.get_artifact(artifact_ref.id)

registered_dataset = Dataset.from_json(registered_artifact.spec, session)

enriched_df = fs.retrieve_feature_values(
    spine_df=pred_df,
    features=registered_dataset.load_features(),
    spine_timestamp_col='PICKUP_TS'
).drop(['PICKUP_TS']).to_pandas()
```

In [ ]:
```python
model_ref = model_registry.ModelReference(
    registry=registry,
    model_name=model_name,
    model_version="v1"
).load_model()

pred = model_ref.predict(enriched_df)

print(pred)
```