- Required snowflake-ml-python version **1.5.0** or higher
- Required snowflake version **8.17** or higher
- Updated on: 5/5/2024

# 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 and database

For detailed session connection config, please follow this tutorial.

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

Below cell creates temporary database, schema and warehouse for this notebook. All temporary resources will be deleted at the end of this notebook. You can rename with your own name if needed.

In [ ]:
```
# database name where test data and feature store lives.
FS_DEMO_DB = f"FEATURE_STORE_TIME_SERIES_FEATURE_NOTEBOOK_DEMO"
# schema where test data lives.
TEST_DATASET_SCHEMA = 'TEST_DATASET'
# feature store name.
FS_DEMO_SCHEMA = "AWESOME_FS_TIME_SERIES_FEATURES"
# the schema model lives.
MODEL_DEMO_SCHEMA = "MODELS"
# stages for UDF.
FS_DEMO_STAGE = "FEATURE_STORE_TIME_SERIES_FEATURE_NOTEBOOK_STAGE_DEMO"
FS_DEMO_STAGE_FULL_PATH = \
    f"{FS_DEMO_DB}.{TEST_DATASET_SCHEMA}.{FS_DEMO_STAGE}"
# warehouse name used in this notebook.
FS_DEMO_WH = "FEATURE_STORE_TIME_SERIES_FEATURE_NOTEBOOK_DEMO"

session.sql(f"CREATE OR REPLACE DATABASE {FS_DEMO_DB}").collect()
session.sql(f"""
    CREATE OR REPLACE SCHEMA {FS_DEMO_DB}.{TEST_DATASET_SCHEMA}
""").collect()
session.sql(f"""
    CREATE OR REPLACE SCHEMA {FS_DEMO_DB}.{MODEL_DEMO_SCHEMA}
""").collect()
session.sql(f"""
    CREATE OR REPLACE STAGE {FS_DEMO_STAGE_FULL_PATH}
""").collect()
session.sql(f"CREATE WAREHOUSE IF NOT EXISTS {FS_DEMO_WH}").collect()
```

## Create FeatureStore Client

Let's first create a feature store client. With `CREATE_IF_NOT_EXIST` mode, it will try to create schema and all necessary feature store metadata if it doesn't exist already. It is required for the first time to setup a Feature Store. Afterwards, you can use `FAIL_IF_NOT_EXIST` mode to connect to an existing Feature Store.

Note database must already exist. Feature Store will **NOT** try to create the database even in `CREATE_IF_NOT_EXIST` mode.

In [ ]:
```
fs = FeatureStore(
    session=session,
    database=FS_DEMO_DB,
    name=FS_DEMO_SCHEMA,
    default_warehouse=FS_DEMO_WH,
    creation_mode=CreationMode.CREATE_IF_NOT_EXIST,
)
```

## 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. 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}"

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"{FS_DEMO_DB}.{TEST_DATASET_SCHEMA}.{table_name}"
df.write.mode("overwrite").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.")
```

In [ ]:
```
source_df = session.table(full_table_name)
source_df = source_df.select(
    [
        "TRIP_DISTANCE",
        "FARE_AMOUNT",
        "PASSENGER_COUNT",
        "PULOCATIONID",
        "DOLOCATIONID",
        F.cast(F.col("TPEP_PICKUP_DATETIME") / 1000000, TimestampType())
            .alias("PICKUP_TS"),
        F.cast(F.col("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_row_count = source_df.count()
source_df.show()
```

## 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().show()
```

## 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 [ ]:
```python
@F.pandas_udf(
    name="vec_window_end",
    is_permanent=True,
    stage_location=f'@{FS_DEMO_STAGE_FULL_PATH}',
    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 [ ]:
```python
from snowflake.snowpark import Window
from snowflake.snowpark.functions import col

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

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
```

```python
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

Now we construct a Feature View with above DataFrame. We firstly create a draft feature view. We set the `refresh_freq` to 1 minute, so it will be refreshed every 1 minute. On the backend, it creates a Snowflake dynamic table. At this point, the draft feature view will not take effect because it is not registered yet. Then we register the feature view by via `register_feature_view`. It will materialize to Snowflake backend. Incremental maintenance will start if the query is supported.

In [ ]:
```
pickup_fv = FeatureView(
    name="TRIP_PICKUP_TIME_SERIES_FEATURES",
    entities=[trip_pickup],
    feature_df=pickup_df,
    timestamp_col="TS",
    refresh_freq="1 minute",
)
pickup_fv = fs.register_feature_view(
    feature_view=pickup_fv,
    version="V1",
    block=True
)
```

In [ ]:
```
dropoff_fv = FeatureView(
    name="TRIP_DROPOFF_TIME_SERIES_FEATURES",
    entities=[trip_dropoff],
    feature_df=dropoff_df,
    timestamp_col="TS",
    refresh_freq="1 minute",
)
dropoff_fv = fs.register_feature_view(
    feature_view=dropoff_fv,
    version="V1",
    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 [ ]:
```
fs.list_feature_views(entity_name="TRIP_PICKUP") \
    .select(["NAME", "VERSION", "ENTITIES", "DESC"]).show()
```

## Generate training data

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. Under the hood, `generate_dataset` uses ASOF join to join spine_df and feature views on timstamp column.

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

In [ ]:
```python
my_dataset = fs.generate_dataset(
    name="my_training_data",
    version="2",
    spine_df=spine_df,
    features=[pickup_fv, dropoff_fv],
    spine_timestamp_col="PICKUP_TS",
    spine_label_cols=["FARE_AMOUNT"],
    exclude_columns=["PICKUP_TS"],
    desc="my training dataset with pickup and dropoff features",
)

training_data_df = my_dataset.read.to_snowpark_dataframe()
assert training_data_df.count() == source_df_row_count
```

## Train model with Snowpark ML

Now let's training a simple random forest model, and evaluate the prediction accuracy. When you call `fit()` on a DataFrame that converted from Feature Store Dataset, The linkage between model and dataset is automatically wired up. Later, you can easily retrieve the dataset from this model, or you can query the lineage about the dataset and model. This is work-in-progress and will be ready soon.

In [ ]:
```python
from snowflake.ml.modeling.pipeline import Pipeline
from snowflake.ml.modeling.linear_model import LinearRegression
from snowflake.ml.modeling.impute import SimpleImputer
from snowflake.ml.modeling import metrics as snowml_metrics
from snowflake.snowpark.functions import col, unix_timestamp

def train_model_using_snowpark_ml(training_df):
    # preprocess the data
    for col_name in ["DOLOCATIONID",
                     "PULOCATIONID",
                     "COUNT_TRIP_2_HR",
                     "COUNT_TRIP_5_HR"]:
```

```
        training_df = training_df.withColumn(col_name, col(col_name)
                                             .cast("float"))

    train, test = training_df.random_split([0.8, 0.2], seed=42)
    excluded_columns = ["FARE_AMOUNT"]
    feature_columns = [col for col in training_df.columns
                       if col not in excluded_columns]
    label_column = "FARE_AMOUNT"

    # Create the pipeline
    steps = [
        ('imputer', SimpleImputer(
            input_cols=feature_columns,
            output_cols=feature_columns,
            drop_input_cols=True,
            strategy="mean")),
        ('linear_regression', LinearRegression(
            input_cols=feature_columns,
            label_cols=[label_column]))
    ]
    pipeline = Pipeline(steps)

    model = pipeline.fit(train)
    predictions = model.predict(test)

    mse = snowml_metrics.mean_squared_error(
        df=predictions,
        y_true_col_names=label_column,
        y_pred_col_names="OUTPUT_" + label_column
    )

    r2 = snowml_metrics.r2_score(
        df=predictions,
        y_true_col_name=label_column,
        y_pred_col_name="OUTPUT_" + label_column
    )

    # Display the metrics
    print(f"Mean squared error: {mse}, R² score: {r2}")

    return model

estimator = train_model_using_snowpark_ml(training_data_df)
```

## [Predict Option 1] With local model

Now let's predict with the model and the feature values retrieved from feature store.

In [ ]:
```
pred_df = source_df.sample(0.00001).select(
    ['PULOCATIONID', 'DOLOCATIONID', 'PICKUP_TS']
)

# load back feature views from dataset
fvs = fs.load_feature_views_from_dataset(my_dataset)
```

```
# retrieve latest feature values from feature store
enriched_df = fs.retrieve_feature_values(
    spine_df=pred_df,
    features=fvs,
    spine_timestamp_col='PICKUP_TS'
).drop(['PICKUP_TS'])
```

In [ ]:
```
pred = estimator.predict(enriched_df.to_pandas())
print(pred)
```

## [Predict Option 2] With Model Registry

We can also predict with models in Model Registry.

### Step 1 : Log the model along with its training dataset metadata into Model Registry

In [ ]:
```
from snowflake.ml.registry import Registry

registry = Registry(
    session=session,
    database_name=FS_DEMO_DB,
    schema_name=MODEL_DEMO_SCHEMA,
)
```

Then we log the model to model registry. Later, we can get it back with same model name and version.

In [ ]:
```
model_name = "MY_LINEAR_REGRESSION_MODEL"
model_version = 'V7'

registry.log_model(
    model_name=model_name,
    version_name=model_version,
    model=estimator,
    comment="log my model trained with dataset",
)
```

### Step 2 : Restore model and predict with features

We read the model back from model registry. We get the features from the dataset, and retrieve latest values for these features from Feature Store. We will same features that the model previously trained on for future inference.

We are working on retrieving dataset from a model directly. For now, we just use previously created dataset object.

In [ ]:
```
model = registry.get_model(model_name).version(model_version)
```

```
# We are working on loading dataset back from a model.
# For now, we use previously created dataset.
fvs = fs.load_feature_views_from_dataset(my_dataset)

# retrieve latest feature values from feature store
enriched_df =fs.retrieve_feature_values(
    spine_df=pred_df,
    features=fvs,
    spine_timestamp_col='PICKUP_TS',
).drop(['PICKUP_TS'])
```

Now we predict on the model and latest feature values.

In [ ]:
```
restored_prediction = model.run(
    enriched_df.to_pandas(), function_name="predict")

print(restored_prediction)
```

## Cleanup notebook

Cleanup resources created in this notebook.

In [ ]:
```
session.sql(f"DROP DATABASE IF EXISTS {FS_DEMO_DB}").collect()
session.sql(f"DROP WAREHOUSE IF EXISTS {FS_DEMO_WH}").collect()
```