- snowflake-ml-python version: 1.2.0
- Feature Store PrPr version: 0.4.0
- Updated date: 1/3/2024

Before getting started

Watch out object name case sensitivity

The Model Registry and Feature Store are not consistent with each other in the way they case names for databases, schemas, and other SQL objects. (Keep in mind that the objects in both APIs are Snowflake objects on the back end.) The model registry preserves the case of names for these objects, while the feature store converts names to uppercase unless you enclose them in double quotes. The way the feature store handles names is consistent with Snowflake's identifier requirements. We are working to make this more consistent. In the meantime, we suggest using uppercase names in both APIs to ensure correct interoperation between the feature store and the model registry.

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 [ ]:
```python
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 [ ]:
```python
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 [ ]:
```python
# 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"
# model registry database name.
MR_DEMO_DB = f"FEATURE_STORE_TIME_SERIES_FEATURE_NOTEBOOK_MR_DEMO"
# 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"DROP DATABASE IF EXISTS {FS_DEMO_DB}").collect()
session.sql(f"DROP DATABASE IF EXISTS {MR_DEMO_DB}").collect()
session.sql(f"CREATE DATABASE IF NOT EXISTS {FS_DEMO_DB}").collect()
session.sql(f"""CREATE SCHEMA IF NOT EXISTS
    {FS_DEMO_DB}.{TEST_DATASET_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.

We can pass in an existing database name, or a new database will be created upon the feature store initialization.

In [ ]:
```python
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.TPEP_PICKUP_DATETIME.alias("PICKUP_TS"),
# source_df.TPEP_DROPOFF_DATETIME.alias("DROPOFF_TS"),
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.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 [ ]:
```
@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 [ ]:
```
from snowflake.snowpark import Window
from snowflake.snowpark.functions import col

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

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

import time
time.sleep(60)
```

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", "FEATURE_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.

In [ ]:
```
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()
```

Train model with Snowpark ML

Now let's training a simple random forest model, and evaluate the prediction accuracy.

In [ ]:
```
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_data):
    training_df = training_data.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"))

    training_df = training_df.withColumn(
        "PICKUP_TS",
        unix_timestamp(col("PICKUP_TS")))

    train, test = training_df.random_split([0.8, 0.2], seed=42)
    excluded_columns = ["FARE_AMOUNT", "PICKUP_TS"]
    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="most_frequent")),
        ('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)
```

[Predict Option 1] With local model

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

In [ ]:
```
pred_df = training_data.df.sample(0.01).select(
    ['PULOCATIONID', 'DOLOCATIONID', 'PICKUP_TS'])

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

pred = estimator.predict(enriched_df)
print(pred)
```

[Predict Option 2] With Model Registry

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

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

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

Register the dataset into model registry with `log_artifact`. Artifact is a generalized concept of ML pipeline outputs that are needed for subsequent execution. Refer to https://docs.snowflake.com/LIMITEDACCESS/snowflake-ml-model-registry for more details about the API.

In [ ]:
```
DATASET_NAME = "MY_DATASET"
DATASET_VERSION = "V1"

my_dataset = registry.log_artifact(
    artifact=training_data,
    name=DATASET_NAME,
    version=DATASET_VERSION,
)
```

Now you can log the model together with the registered artifact (which is a dataset here).

In [ ]:
```
model_name = "MY_MODEL"

model_ref = registry.log_model(
    model_name=model_name,
    model_version="V1",
    model=estimator,
    artifacts=[my_dataset],
)
```

Step 2 : Restore model and predict with 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 [ ]:
```
# Enrich source prediction data with features
from snowflake.ml.dataset.dataset import Dataset

registered_dataset = registry.get_artifact(
    DATASET_NAME,
    DATASET_VERSION)

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 [ ]:
```
model_ref = model_registry.ModelReference(
    registry=registry,
    model_name=model_name,
    model_version="V1"
).load_model()

pred = model_ref.predict(enriched_df)

print(pred)
```

Cleanup notebook

Cleanup resources created in this notebook.

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