Version: 0.1.1

Updated date: 10/11/2023

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.

Basic Feature Demo

This notebook demonstrates feature store with simple 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.

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

Setup Snowflake connection and database

For detailed session connection config, please follow this tutorial.

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

Following sections will use below database and schema name to store test data and feature store objects. 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_BASIC_FEATURE_NOTEBOOK_DEMO"
# schema where test data lives.
```

```
TEST_DATASET_SCHEMA = 'TEST_DATASET'
# feature store name.
FS_DEMO_SCHEMA = "AWESOME_FS_BASIC_FEATURES"
# Model registry database name.
MR_DEMO_DB = f"FEATURE_STORE_BASIC_FEATURE_NOTEBOOK_MR_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()
```

Prepare test data

We will use wine quality dataset for this demo. Download the public dataset from kaggle if you dont have it already: https://www.kaggle.com/datasets/uciml/red-wine-quality-cortez-et-al-2009. Replace `TEST_CSV_FILE_PATH` with your local file path.

In [ ]:
```
TEST_CSV_FILE_PATH = 'winequality-red.csv'
session.file.put(
    f"file://{TEST_CSV_FILE_PATH}",session.get_session_stage())

from snowflake.snowpark.types import (
    StructType,
    StructField,
    IntegerType,
    FloatType
)
input_schema = StructType(
    [
        StructField("fixed_acidity", FloatType()),
        StructField("volatile_acidity", FloatType()),
        StructField("citric_acid", FloatType()),
        StructField("residual_sugar", FloatType()),
        StructField("chlorides", FloatType()),
        StructField("free_sulfur_dioxide", IntegerType()),
        StructField("total_sulfur_dioxide", IntegerType()),
        StructField("density", FloatType()),
        StructField("pH", FloatType()),
        StructField("sulphates", FloatType()),
        StructField("alcohol", FloatType()),
        StructField("quality", IntegerType())
    ]
)
df = session.read.options({"field_delimiter": ";", "skip_header": 1}) \
    .schema(input_schema) \
    .csv(f"{session.get_session_stage()}/winequality-red.csv")
df.write.mode("overwrite").save_as_table("WINE_DATA")
```

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.

In [ ]:
```
session.sql(f"""CREATE OR REPLACE WAREHOUSE PUBLIC WITH
                WAREHOUSE_SIZE='XSMALL'
            """).collect()

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

Create and register a new Entity

We will create an Entity called *wine* and register it with the feature store.

You can retrieve the active Entities in the feature store with list_entities() API.

In [ ]:
```
entity = Entity(name="WINE", join_keys=["WINE_ID"])
fs.register_entity(entity)
fs.list_entities().to_pandas()
```

Load source data and do some simple feature engineering

Then we will load from the source table and conduct some simple feature engineerings.

Here we are just doing two simple data manipulation (but more complex ones are carried out the same way):

1. Assign a WINE_ID column to the source
2. Derive a new column by multipying two existing feature columns

In [ ]:
```
source_df = session.table(f"{FS_DEMO_DB}.{TEST_DATASET_SCHEMA}.WINE_DATA")
source_df.to_pandas()
```

In [ ]:
```
def addIdColumn(df, id_column_name):
    # Add id column to dataframe
    columns = df.columns
    new_df = df.withColumn(id_column_name, F.monotonically_increasing_id())
    return new_df[[id_column_name] + columns]

def generate_new_feature(df):
```

```
    # Derive a new feature column
    return df.withColumn(
        "MY_NEW_FEATURE", df["FIXED_ACIDITY"] * df["CITRIC_ACID"])

df = addIdColumn(source_df, "WINE_ID")
feature_df = generate_new_feature(df)
feature_df = feature_df.select(
    [
        'WINE_ID',
        'FIXED_ACIDITY',
        'VOLATILE_ACIDITY',
        'CITRIC_ACID',
        'RESIDUAL_SUGAR',
        'CHLORIDES',
        'FREE_SULFUR_DIOXIDE',
        'TOTAL_SULFUR_DIOXIDE',
        'DENSITY',
        'PH',
        'MY_NEW_FEATURE',
    ]
)
feature_df.to_pandas()
```

Create a new FeatureView and materialize the feature pipeline

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 [ ]:
```
fv = FeatureView(
    name="WINE_FEATURES",
    entities=[entity],
    feature_df=feature_df,
    desc="wine features"
)
fs.register_feature_view(
    feature_view=fv,
    version="V1",
    refresh_freq="1 minute",
    block=True
)
```

In [ ]:
```
# Examine the FeatureView content
```

```
fs.read_feature_view(fv).to_pandas()
```

Explore additional features

Now I have my FeatureView created with a collection of features, but what if I want to explore additional features on top?

Since a materialized FeatureView is immutable (due to singe DDL for the backend dynamic table), we will need to create a new FeatureView for the additional features and then merge them.

In [ ]:
```
extra_feature_df = df.select(
    [
        'WINE_ID',
        'SULPHATES',
        'ALCOHOL',
    ]
)

new_fv = FeatureView(
    name="EXTRA_WINE_FEATURES",
    entities=[entity],
    feature_df=extra_feature_df,
    desc="extra wine features"
)
fs.register_feature_view(
    feature_view=new_fv,
    version="V1",
    refresh_freq="1 minute",
    block=True
)
```

In [ ]:
```
# We can easily retrieve all FeatureViews for a given Entity.
fs.list_feature_views(entity_name="WINE"). \
    select(["NAME", "ENTITIES", "FEATURE_DESC"]).show()
```

Create new feature view with combined feature results [Optional]

Now we have two FeatureViews ready, we can choose to create a new one by merging the two (it's just like a join and we provide a handy function for that). The new FeatureView won't incur the cost of feature pipelines but only the table join cost.

Obviously we can also just work with two separate FeatureViews (most of our APIs support multiple FeatureViews), the capability of merging is just to make the features better organized and easier to share.

In [ ]:
```
full_fv = fs.merge_features(
    features=[fv, new_fv], name="FULL_WINE_FEATURES")
```

```
fs.register_feature_view(feature_view=full_fv, version="V2")
```

Generate Training Data

After our feature pipelines are fully setup, we can start using them to generate training data and later do model prediction.

In [ ]:
```
spine_df = session.table(f"{FS_DEMO_DB}.{TEST_DATASET_SCHEMA}.WINE_DATA")
spine_df = addIdColumn(source_df, "WINE_ID")
spine_df = spine_df.select("WINE_ID", "QUALITY")
spine_df.to_pandas()
```

Generate training data is easy since materialized FeatureViews already carry most of the metadata like join keys, timestamp for point-in-time lookup, etc. We just need to provide the spine data (it's called spine because we are essentially enriching the data by joining features with it). We can also generate dataset with a subset of features in the feature view by `slice`.

In [ ]:
```
training_dataset_full_path = \
    f"{FS_DEMO_DB}.{FS_DEMO_SCHEMA}.WINE_TRAINING_DATA_TABLE"
session.sql(f"DROP TABLE IF EXISTS {training_dataset_full_path}") \
    .collect()
training_data = fs.generate_dataset(
    spine_df=spine_df,
    features=[
        full_fv.slice([
            "FIXED_ACIDITY",
            "VOLATILE_ACIDITY",
            "CITRIC_ACID"
        ])
    ],
    materialized_table="WINE_TRAINING_DATA_TABLE",
    spine_timestamp_col=None,
    spine_label_cols=["QUALITY"],
    save_mode="merge",
    exclude_columns=['WINE_ID']
)

training_data.df.show()
```

Train a model

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

In [ ]:
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
```

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

In [ ]:
```
def train_model(X_train, X_test, y_train, y_test):
    rf = RandomForestRegressor(
        max_depth=3, n_estimators=20, random_state=42)
    rf.fit(X_train, y_train)
    y_pred = rf.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    accuracy = round(100*(1-np.mean(
        np.abs((y_test - y_pred) / np.abs(y_test)))))
    print(f"MSE: {mse}, Accuracy: {accuracy}")
    return rf

rf = train_model(X_train, X_test, y_train, y_test)
print(rf)
```

[Optional 1] Predict with local model

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

In [ ]:
```
test_df = spine_df.limit(3).select("WINE_ID")

enriched_df = fs.retrieve_feature_values(
    test_df, training_data.load_features())
enriched_df = enriched_df.drop('WINE_ID')
```

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

print(pred)
```

[Option 2.1] Log model with Model Registry

We can also log the model along with its training dataset metadata into Model Registry.

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

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

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

```
    artifact_spec=training_data.to_json(),
    artifact_version="v3",
)
```

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

model_ref = registry.log_model(
    model_name=model_name,
    model_version="V2",
    model=rf,
    tags={"author": "my_rf_with_training_data"},
    artifacts=[artifact_ref],
    options={"embed_local_ml_library": True},
)
```

[Optional 2.2] Restore model and predict with features

We retrieve the training dataset from registry then construct dataframe of latest feature values. Then we restore the model from registry. At last, we can predict with latest feature values.

In [ ]:
```
from snowflake.ml.dataset.dataset import Dataset

registered_artifact = registry.get_artifact(
    artifact_ref.name,
    artifact_ref.version)
registered_dataset = Dataset.from_json(registered_artifact.spec, session)
test_df = spine_df.limit(3).select("WINE_ID")

enriched_df = fs.retrieve_feature_values(
    test_df, registered_dataset.load_features())
enriched_df = enriched_df.drop('WINE_ID')
```

In [ ]:
```
model_ref = model_registry.ModelReference(
    registry=registry,
    model_name=model_name,
    model_version="V2"
)
restored_model = model_ref.load_model()
restored_prediction = restored_model.predict(enriched_df.to_pandas())

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 DATABASE IF EXISTS {MR_DEMO_DB}").collect()
```