Version: 0.1.0

Updated date: 10/10/2023

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 [ ]:
```python
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

For detailed session connection config, please follow this tutorial.

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

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

SOURCE_DB = session.get_current_database()
SOURCE_SCHEMA = session.get_current_schema()

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 [ ]:
```
DEMO_DB = "FS_DEMO_DB"
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 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"{SOURCE_DB}.{SOURCE_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", "FEATU
```

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="v1")
```

Generate Training Data

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

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).

In [ ]:
```
spine_df = session.table(f"{SOURCE_DB}.{SOURCE_SCHEMA}.wine_data")
spine_df = addIdColumn(source_df, "wine_id")
spine_df = spine_df.select("wine_id", "quality")
spine_df.to_pandas()
```

In [ ]:
```
training_dataset_full_path = f"{DEMO_DB}.{DEMO_SCHEMA}.wine_training_data_ta
session.sql(f"DROP TABLE IF EXISTS {training_dataset_full_path}").collect()
training_data = fs.generate_dataset(
    spine_df=spine_df,
    features=[full_fv],
    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)
```

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

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},
)
```

Restore model and predict with latest 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.id)

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

# test_pdf = training_pd.sample(3, random_state=996)[['WINE_ID']]
test_df = spine_df.limit(3).select("WINE_ID")
# test_df = session.create_dataframe(test_pdf)

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)
```