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

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()
```

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

In [ ]:
```
# database name where test data, feature store and model 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"
# the schema model lives.
MODEL_DEMO_SCHEMA = "MODELS"
# warehouse name used in this notebook.
FS_DEMO_WH = "FEATURE_STORE_BASIC_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 WAREHOUSE IF NOT EXISTS {FS_DEMO_WH}").collect()
```

Create a new 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 connecte 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

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")
full_table_name = f"{FS_DEMO_DB}.{TEST_DATASET_SCHEMA}.WINE_DATA"
df.write.mode("overwrite").save_as_table(full_table_name)
```

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 [ ]:
```
wine_entity = Entity(name="WINE", join_keys=["WINE_ID"])
fs.register_entity(wine_entity)
fs.list_entities().show()
```

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 [ ]:
```
from snowflake.snowpark.window import Window

def addIdColumn(df, id_column_name):
    # Add id column to dataframe
    columns = df.columns
    new_df = df.withColumn(
        id_column_name,
        F.row_number().over(Window.order_by(F.col("quality"))))
    return new_df

source_df = session.table(full_table_name)
source_df = addIdColumn(source_df, "WINE_ID")
```

In [ ]:
```
source_df_rows_count = source_df.count()
print(f"Total number of rows in source df: {source_df_rows_count}")
source_df.show()
```

In [ ]:
```
def generate_new_feature(df):
    # Derive a new feature column
    new_df = df.withColumn(
```

```
        "MY_NEW_FEATURE", df["FIXED_ACIDITY"] * df["CITRIC_ACID"])
    return new_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 = generate_new_feature(source_df)
feature_df.show()
```

Create a new FeatureView and materialize the feature pipeline

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 [ ]:
```
draft_fv = FeatureView(
    name="WINE_FEATURES",
    entities=[wine_entity],
    feature_df=feature_df,
    refresh_freq="1 minute",
    desc="my wine features auto refreshed on a schedule"
)
wine_features = fs.register_feature_view(
    feature_view=draft_fv,
    version="1.0",
    block=True
)
```

We can examine the feature values in a feature view.

In [ ]:
```
fs.read_feature_view(wine_features).show()
```

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, we can create a new FeatureView for the additional features. Note `refresh_freq` of below Feature View is None. It means the Feature View is static and will not refresh on a schedule. You can still update the feature values by updating the data source (table `WINE_DATA`). On the backend it is a Snowflake view.

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

extra_draft_fv = FeatureView(
    name="EXTRA_WINE_FEATURES",
    entities=[wine_entity],
    feature_df=extra_feature_df,
    refresh_freq=None,
    desc="extra wine features"
)
extra_features = fs.register_feature_view(
    feature_view=extra_draft_fv,
    version="1.0",
    block=True
)
```

We can examine the status of all feature views.

In [ ]:
```
fs.list_feature_views(entity_name="WINE").show()
```

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 = source_df.select("WINE_ID", "QUALITY")
spine_df.show()
```

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 [ ]:
```
my_dataset = fs.generate_dataset(
    name="my_training_dataset",
    version="12",
    spine_df=spine_df,
    features=[
        wine_features.slice([
            "FIXED_ACIDITY", "VOLATILE_ACIDITY", "CITRIC_ACID"]),
```

```
        extra_features
    ],
    spine_timestamp_col=None,
    spine_label_cols=["QUALITY"],
    exclude_columns=['WINE_ID'],
    desc="my training dataset with EXTRA_WINE_FEATURES and WINE_FEATURES",
)
```

Convert dataset to a snowpark dataframe and examine all the features in it.

In [ ]:
```
training_data_df = my_dataset.read.to_snowpark_dataframe()
assert training_data_df.count() == source_df_rows_count
training_data_df.show()
```

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 [ ]:
```
from snowflake.ml.modeling.ensemble import RandomForestRegressor
from snowflake.ml.modeling import metrics as snowml_metrics
from snowflake.snowpark.functions import abs as sp_abs, mean, col

def train_model_using_snowpark_ml(training_data_df):
    train, test = training_data_df.random_split([0.8, 0.2], seed=42)
    feature_columns = \
        [col for col in training_data_df.columns if col != "QUALITY"]
    label_column = "QUALITY"

    rf = RandomForestRegressor(
        input_cols=feature_columns, label_cols=[label_column],
        max_depth=3, n_estimators=20, random_state=42
    )

    rf.fit(train)
    predictions = rf.predict(test)

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

    accuracy = 100 - snowml_metrics.mean_absolute_percentage_error(
        df=predictions,
        y_true_col_names=label_column,
        y_pred_col_names="OUTPUT_" + label_column
    )

    print(f"MSE: {mse}, Accuracy: {accuracy}")
```

```
    return rf

rf = train_model_using_snowpark_ml(training_data_df)
```

[Predict Optional 1] With local model

Now we can predict with a local model and the feature values retrieved from feature store.

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

# load back feature views from dataset
fvs = fs.load_feature_views_from_dataset(my_dataset)
enriched_df = fs.retrieve_feature_values(test_df, fvs)
enriched_df = enriched_df.drop('WINE_ID')
```

In [ ]:
```
pred = rf.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 into Model Registry

Firstly, we connect to a 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_RANDOM_FOREST_REGRESSOR_MODEL"

registry.log_model(
    model_name=model_name,
    version_name="V2",
    model=rf,
    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("V2")

# 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)
spine_df = spine_df.limit(3).select("WINE_ID")

enriched_df =fs.retrieve_feature_values(
    spine_df=spine_df,
    features=fvs,
    exclude_columns=["WINE_ID"]
)
```

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()
```