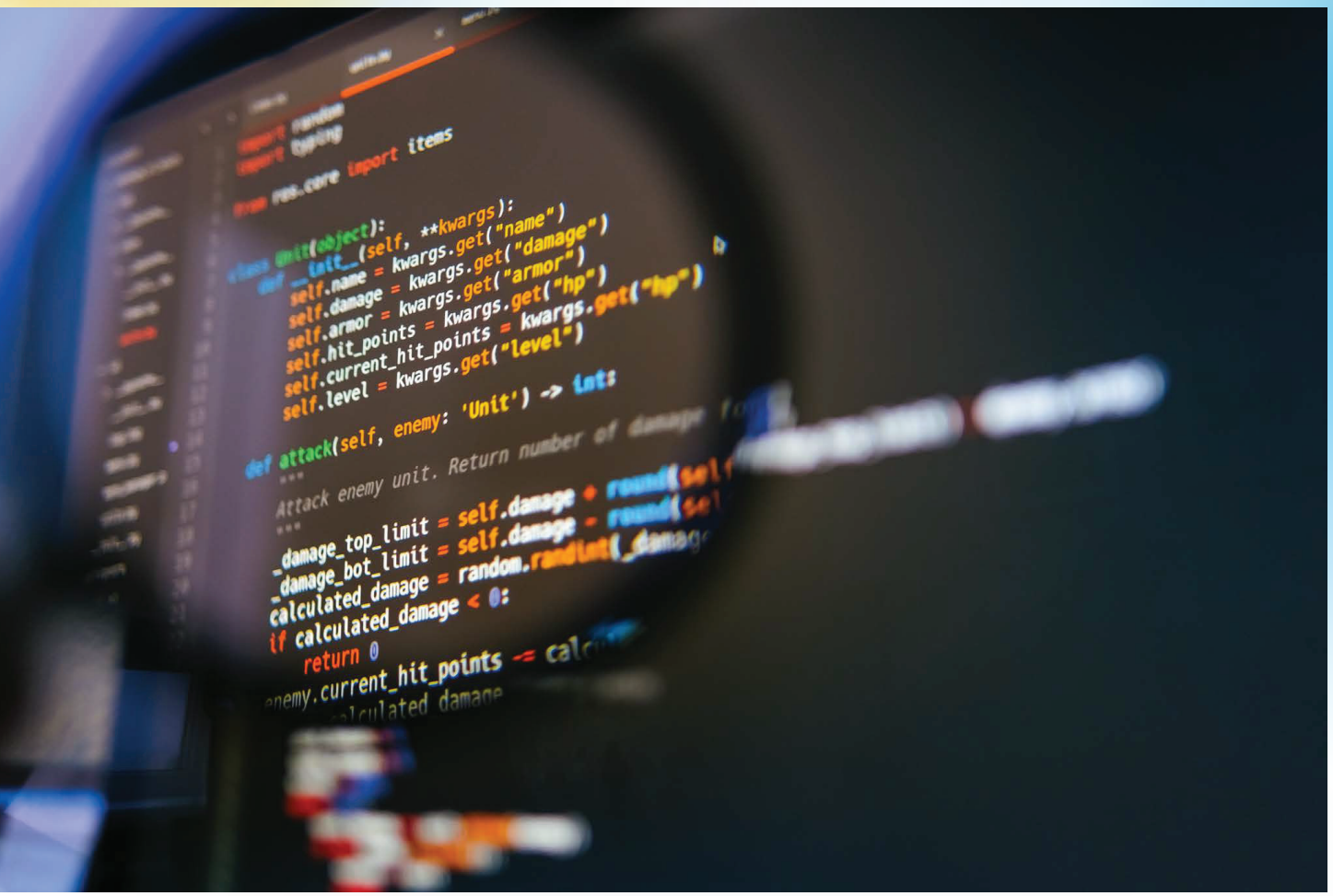
LTIMindtree



Point of View

Python-Stored Procedures

Making developers limitless in their approach

by **Srinivasaraghavan Sundar**

Why Python Stored Procedure?

Ever since its inception, SQL-based paradigms have been hampered by their inability to solve procedural logic problems. Even though PL SQL was widely used, it was often clunky when compared to more modern procedural coding languages. Snowflake solved this problem by introducing Stored Procedures which enabled you to write procedural code that executes SQL. Since you would write this code in Javascript, you could implement branching, looping and variable assignments far more effortlessly than on SQL. While this innovation proved to elevate the customer experience, Snowflake did not stop there. Instead, they improved the selection of languages available to write Stored Procedures. Today, we can write Stored Procedures in Javascript, Scala, Java, SQL and Python.

With Python Stored Procedures, Snowflake enables the user to write Procedural Logic in what is undoubtedly the most popular coding language in the world. Moreover, it facilitates solving complex data engineering problems right where the data resides. All this while also taking advantage of the famed scalable compute provided by Snowflake.

With access to hundreds of open source libraries, deployment of data science models can be done on Snowflake with ease!

Python SP vs. Python UDF

Now, for those of you who have followed our previous post regarding Python UDFs, you may be wondering what is the difference between Python UDF and Python Stored Procedure?

Well, both allow you to write and execute Python on Snowflake. Stored Procedures, in addition, can execute SQL queries. And therein lies the crucial difference. Python Stored Procedures make use of the Snowpark library in order to execute SQL queries. This provides an added benefit of being able to execute SQL either using the Dataframe API or the usual SQL queries.

"Well, why can't I just run the code on my local machine using Snowpark?" I hear you say.

That certainly is an option, and while it depends on the use case, by using Python Stored Procedure, 100% of your code is being run on Snowflake by the Virtual Warehouse. In contrast to this, when you use Snowpark, all the code which makes use of the Snowpark APIs run on Snowflake and other operations run on your local machine. There may be instances where you require Snowflake to do all the heavy lifting when it comes to the compute. Moreover, if you want to schedule a procedure to run by making use of tasks, you would not be able to do so on Snowpark.

Now then, let us jump into a demo to see what Python Stored Procedure can do.

Classification using Python Stored Procedure

Problem description

The dataset we are making use of today can be found here:
https://www.kaggle.com/datasets/johnsmith88/heart-disease-dataset

Given a number of features and parameters regarding a patient's heart condition, our objective is to predict if they have a heart disease. In order to do so, we are making use of Logistic Regression. For a given table containing the necessary parameters, a new table "RESULTS" is constructed comprising all the predicted values.

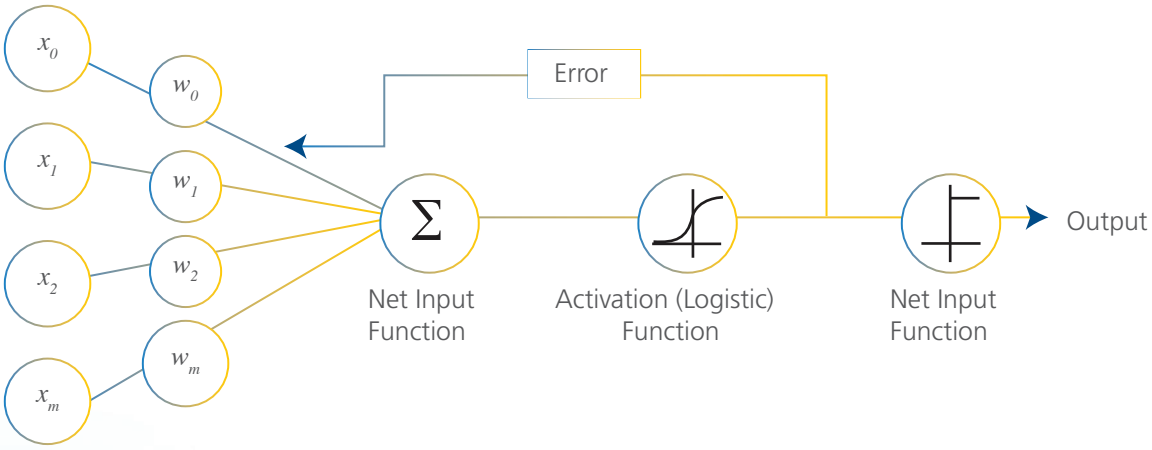


Fig 1: Logistic Regression

Dataflow of the Solution

Our goal is to train once and deploy multiple times. This means that there must be an isolation between training and prediction modules. Therefore, we build two Stored Procedures—One to train and store the model and another to load the saved model and perform predictions. Once the predictions are obtained hey are written into a new table. As and when data drift/model drift occurs, the model can be retrained.

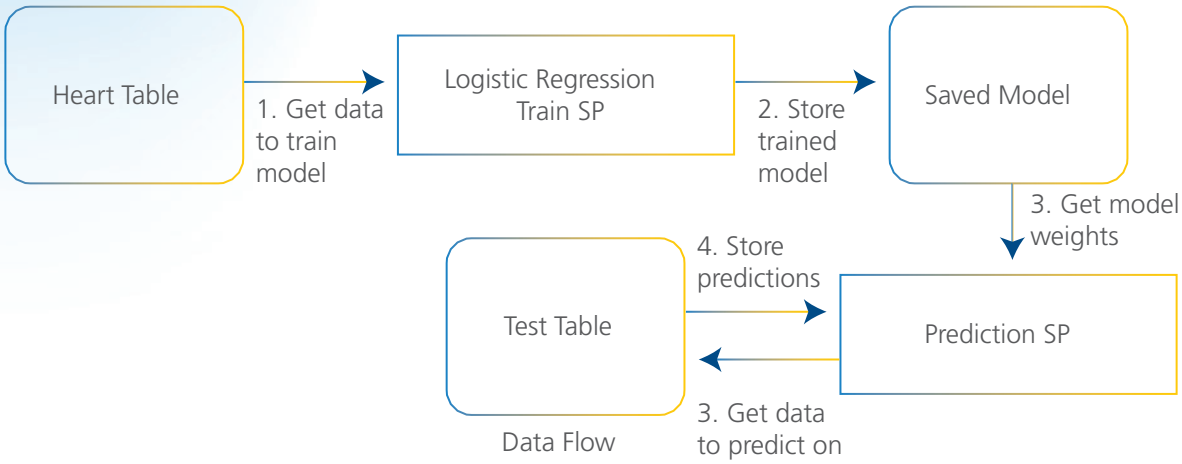


Fig 2: Python SP Dataflow

Dataset Description

SEX	CP	TRESTBPS	CHOL	...	FBS	RESTECG	THALACH	EXANG	OLDPEAK	SLOPE	CA	THAL	PREDICTION
1	0	125	212		0	1	168	0	1	2	2	3	0
1	0	140	203		1	0	155	1	3.1	0	0	3	0
1	0	145	174		0	1	125	1	2.6	0	0	3	0
1	0	148	203		0	1	161	0	0	2	1	3	1
0	0	138	294		1	1	106	0	1.9	1	3	2	0
0	0	100	248		0	0	122	0	1	1	0	2	1
1	0	114	318		0	2	140	0	4.4	0	3	1	0
1	0	160	289		0	0	145	1	0.8	1	1	3	0

Fig 3: Sample data

As we can see, there are 12 different features for us to work with in order to obtain a prediction. None of these features are categorical therefore one hot encoding is not needed. However, not all these features are useful for us to make a prediction. Moreover, some scaling must be done across the table.

Training the model

```
CREATE OR REPLACE PROCEDURE HEART_DISEASE_TRAINER(TABLE_NAME STRING)
RETURNS string
LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
PACKAGES = ('snowflake-snowpark-python','pandas==1.2.3','numpy==1.19.2','scikit-learn')
HANDLER = 'heart_disease_train'
AS
$$
import pickle
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
import pandas as pd
from snowflake.snowpark.functions import *
#Creating handler function
def heart_disease_train(session, TABLE_NAME):
    table_name=TABLE_NAME
#defining snowflake dataframe
    df=session.table(table_name)
#converting snowflake dataframe to pandas dataframe
    dataset=df.to_pandas()
    s_sc = StandardScaler()
#selecting columns to scale
    col_to_scale = ['AGE', 'TRESTBPS', 'CHOL', 'THALACH', 'OLDPEAK']
    dataset[col_to_scale] = s_sc.fit_transform(dataset[col_to_scale])
```

```
    X = dataset.drop('TARGET', axis=1)
    y = dataset.TARGET
#obtaining train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
#using logistic regression and declaring parameters
                lr_clf = LogisticRegression(penalty='elasticnet', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='saga', max_iter=1000, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=0.3)
#fitting the logistic regression model
    lr_clf.fit(X_train, y_train)
    test_score = str(accuracy_score(y_test, lr_clf.predict(X_test)) * 100)+'%'
#converting the trained model into serialized binary string using pickle
    serialized_model=pickle.dumps(lr_clf)
#storing the binary string containing the trained model along with test score in a snowflake dataframe  df_model=session.create_dataframe([[serialized_model,test_score]],schema=["model_serialized","test_score"])
#writing snowflake dataframe into a new table.
    df_model.write.mode("overwrite").save_as_table("MODELS")
    return "success"
$$;
```

The procedure can be called by the following query:

```
call HEART_DISEASE_TRAINER('HEART');
```

The above procedure trains the logistic regression model and stores it in a table in the form of a binary string. The stored model is called using the following procedure which performs prediction on a given table.

Perform Predictions

```
CREATE OR REPLACE PROCEDURE HEART_DISEASE_PRED(TABLE_NAME STRING)
RETURNS string
LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
```

```
PACKAGES = ('snowflake-snowpark-python','pandas==1.2.3','numpy==1.19.2','scikit-learn')
HANDLER = 'heart_disease_predict'
AS
$$
import numpy as np
import pickle
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
import pandas as pd
from snowflake.snowpark.functions import *
#create handler function to perform prediction
def heart_disease_predict(session, TABLE_NAME):
#read the stored model from the table and convert it to a pandas dataframe
    df=session.table("MODELS")
    dfm=df.to_pandas()
#load binary string into a model using pickle
    model=dfm["model_serialized"].iloc[0]
    s_sc = StandardScaler()
    lr_clf=pickle.loads(model)
    df2=session.table(TABLE_NAME)
#read data that is to be predicted on
    data_test=df2.to_pandas()
    col_to_scale = ['AGE', 'TRESTBPS', 'CHOL', 'THALACH', 'OLDPEAK']
    data_test[col_to_scale] = s_sc.fit_transform(data_test[col_to_scale])
    X_pred=data_test
#predict using model
    prediction=lr_clf.predict(X_pred)
    data_test["PREDICTION"]=prediction
```

```
    data_test[col_to_scale] = s_sc.inverse_transform(data_test[col_to_scale])
#in order to be more human readable, we convert numerical 1 to “MALE” and 0 to “FEMALE”
    data_test.loc[data_test["SEX"] == 1, "SEX"] = "MALE"
    data_test.loc[data_test["SEX"] == 0, "SEX"] = "FEMALE"
#write results into a new table.
    session.write_pandas(data_test, "RESULTS",auto_create_table=True)
    return "success"
$$;
```

The procedure can be called by the following query:

```
call HEART_DISEASE_PRED('HEART_TEST');
```

Upon running the above command, we identify people with a potential heart disease. And thanks to Snowflake Python Stored Procedure, they can be notified quickly!

In Conclusion

Hopefully, this tutorial has helped you to get a grip on what can be done using Python Stored Procedures. It is quite remarkable that we can perform these complex machine learning algorithms right where the data resides.

Snowflake does not seem to stop with just the extra mile. They push further to enable developers to become limitless. Today, you can write Stored Procedures in any one of 5 different languages based on your preference. Mine just happens to be Python, what about you?

About the Author

Srinivasaraghavan Sundar

Senior Data Engineer, Snowflake Center of Excellence, LTIMindtree

Srinivas is currently a part of LTIMindtree's Snowflake COE in the capacity of a Senior Data Engineer with a natural penchant for Data Science. Srinivas usually spends his time either picking up new skills or honing and deep diving into his areas of interest in Data Engineering & Data science. He channels his creativity into finding unique solutions for technical problems and writing content to help our readers.