devUDF: Increasing UDF development efficiency through IDE Integration. It works like a PyCharm!

Mark Raasveldt
CWI
Amsterdam
m.raasveldt@cwi.nl

Pedro Holanda
CWI
Amsterdam
holanda@cwi.nl

Stefan Manegold
CWI
Amsterdam
manegold@cwi.nl

ABSTRACT

User-defined functions (UDFs) facilitate the execution of analytics pipelines inside the database. They provide many advantages over traditional methods, such as close-to-data execution and automatic parallelization. However, the standard workflow for developing and debugging UDFs does not allow developers to use their regular toolchains and Integrated Development Environments (IDEs). As a result, writing functional UDFs is challenging. In this demo, we present the devUDF, a plugin to the PyCharm IDE that allows developers to develop and debug their MonetDB/Python UDFs directly from within the IDE.

1 INTRODUCTION

To perform data analysis, data scientists frequently use scripting languages, such as R and Python. These languages have a huge ecosystem of existing machine-learning and classification libraries (e.g., TensorFlow [1] or Sci-Kit Learn [6]). Using these languages in conjunction with a relational database management system (RDBMS) has many advantages, as the RDBMS can offer robust storage of data and handle common data wrangling operations. The traditional method of combining a RDBMS with these scripting languages is to connect to a RDBMS using a client protocol. The data is then transferred from the database to the analytical tool. However, this is not efficient when a large amount of data needs to be retrieved [8].

This issue can be solved by in-database analytics. By performing the analytics inside the database, the data transfer overhead is mitigated [7]. The primary way of performing in-database analytics is through the use of UDFs. To facilitate this, most RDBMS vendors support UDFs in at least one scripting language frequently used for analysis [3].

The development workflow for UDFs differs depending on the DBMS that is used. Certain databases provide their own custom tools for developing UDFs, such as pgAdmin [9] for Postgres and ODS [4] for DB2 and Oracle. However, these tools have a number of limitations. They are not database agnostic, only work for developing UDFs written in PL/SQL and require developers to learn how to use complex tools designed for DBAs.

The generic workflow for developing a UDF is to write a function using a simplistic text editor. The function can then be created inside the RDBMS through a SQL command, and used by calling it within a SQL query. If there are bugs or problems within the UDF, the function has to be recreated and the SQL query has to be rerun. This process has to be repeated until the problem is fixed.

This workflow is problematic when developing complex UDFs, as advanced IDE features and modern debugging techniques cannot be used. Using these IDE features is not easily doable because the developer has to manually perform code transformations to convert the Python code to a SQL command that creates the UDF. As seen in Table 1 [2], IDEs are heavily preferred for development over simplistic text editors due to their development features. Therefore, we argue that offering support for the usage of these features in the development workflow of UDFs will make developing UDFs more attractive, faster and easier for many developers.

Name	Market Share	Type
Eclipse	25.2%	**IDE**
Visual Studio	19.5%	**IDE**
Android Studio	9.5%	**IDE**
Vim	7.9%	Text Editor
XCode	5.2%	**IDE**
IntelliJ	4.8%	**IDE**
NetBeans	4.0%	**IDE**
Xamarin	3.8%	**IDE**
Komodo	3.4%	**IDE**
Sublime Text	3.3%	Text Editor
Visual Studio Code	3.3%	Text Editor
PyCharm	2.3%	**IDE**

Table 1: Most Popular Development Environments.

IDEs are also attractive because they facilitate the usage of sophisticated interactive debugging techniques, such as stepping through the code line by line and pausing code execution. However, these techniques cannot be used in conjunction with UDFs because the RDBMS must be in control of the code flow while the UDF is being executed. Instead, developers have to resort to inefficient debugging strategies (e.g., print debugging) to make their code work [3].

Another issue with the standard UDF workflow is that UDFs are stored within the database server. As a result, version control systems (VCSs) such as Git [5] cannot be easily integrated to keep track of changes to UDFs. Without a VCS, cooperative development is challenging and the development history is not stored.

In this demo we showcase devUDF, a plugin for the popular IDE PyCharm that facilitates developing and debugging MonetDB/Python UDFs [7] directly from within the IDE. Using our plugin, advanced debugging features can be used while refining and refactoring UDFs.

2 THE DEVUDF PLUGIN

The devUDF plugin is developed for the PyCharm IDE that facilitates the usage of advanced IDE features for development of MonetDB/Python UDFs. It allows developers to create, modify and test UDFs without leaving their IDE environment. All

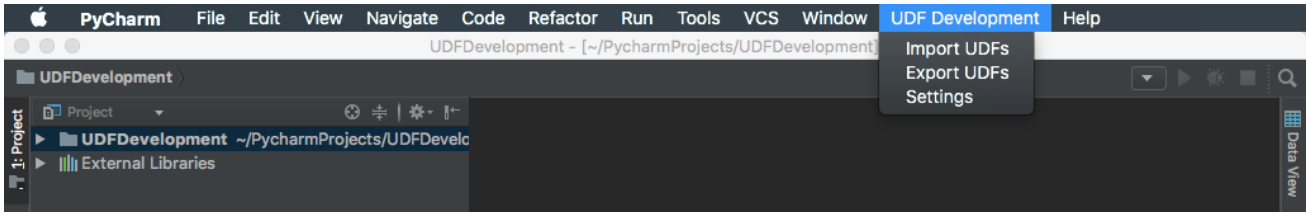


Figure 1: PyCharm Main Menu.

features of the IDE can be used to develop UDFs, including the sophisticated interactive debugger and VCS support.

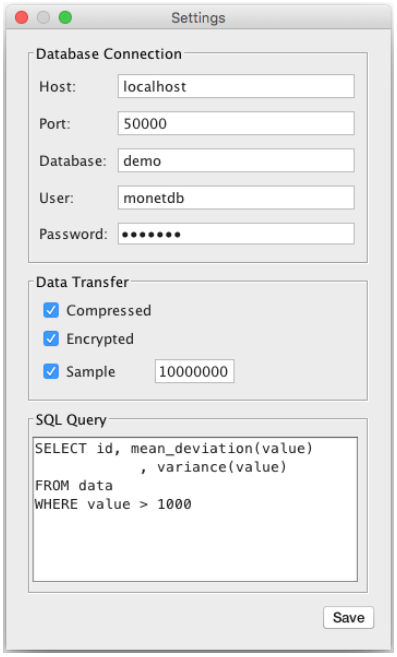


Figure 2: Settings.

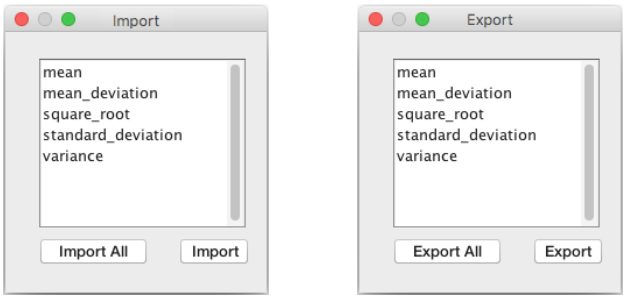
2.1 Usage

The devUDF plugin can be accessed through the main menu of the IDE (See Figure 1). In this menu, a submenu labeled "UDF Development" contains the three main aspects of the plugin.

Initially, devUDF must be configured so it can connect to an existing database server. This can be done through the settings window shown in Figure 2. The parameters required are the usual database client connection parameters (i.e., host, port, database, user and password).

After the devUDF plugin has been configured to connect to a running database server, the development process begins by importing the existing UDFs within the server into the development environment. This is done through the "Import UDFs" window, shown in Figure 3(a). The developer has the option to select the functions that he wishes to import, or he can choose to import all functions stored within the database server.

After the UDFs are imported, the code of the UDFs is exported from the database and imported into the IDE as a set of files in the current project. The developer can then modify the code of the UDFs in these files, use version control to keep track of changes to the UDFs and export the UDFs back to the database server for execution through the "Export UDFs" window (see Figure 3(b)).



(a) Import (b) Export

Figure 3: Importing and Exporting UDFs from the Database.

The developer can also run any of the imported UDFs with the IDEs interactive debugger by running the project as they would run a normal PyCharm project (using the "Debug" command). Since a UDF is never executed in isolation, but always within the context of a SQL query, the user must provide a SQL query which executes the to-be-debugged UDF. This SQL query must be specified in the Settings menu (see Figure 2).

Running the UDF in the interactive debugger will execute the function locally on the developers' machine instead of remotely inside the database server. As the UDF requires data from the database (as its input parameters), the data must be transferred from the database server to the developers machine. For this data transfer, the developer can configure another set of options. As the data can be large, we offer a method of compressing the data during the transfer, leading to faster transfer times. In addition, the developer can choose to execute the UDF using a uniform random sample of the input data instead of the full set of input data. This will alleviate the data transfer overhead.

Since the data contained inside the database server might be sensitive, and it must be exported for debugging purposes, we also offer an optional encryption feature that can be used to safely transfer the sensitive data.

2.2 Implementation

The devUDF plugin works by connecting to the database using a JDBC connection. It then extracts the source code of the UDF together with its input parameters from the database by querying the databases' meta tables. An example of how MonetDB stores the source code of a Python function is shown in Listing 1. In order to be able to execute the UDF locally a set of code transformations has to be applied to this code, as the database only contains the function body. We need to create the header of the function using the function name and its parameters. To then run the created function, we need to obtain the input data from

the database. In the generated code, we load the input data from a binary blob using the `pickle` library and pass it as a parameter to the function. The final transformed code is shown in Listing 2. When the user wants to export the UDF back to the database, these transformations are reversed and only the function body is committed.

When the user wants to debug the UDF locally using the interactive debugger, the input data of the function has to be extracted from the database. To obtain the input data, we take the user-submitted SQL query containing the call to the UDF, and we replace the call to the UDF with a predefined extract function that transfers the input data back to the client instead of executing the UDF inside the server. We then run the transformed SQL query inside the database server to obtain the input data, store it on the developers machine and run the code of the transformed UDF.

The extract function used changes depending on the data transfer options selected by the user. If encryption is requested, the data is encrypted by the extract function before being transferred using the password of the database user as a key. The client then reverses the encryption to obtain the actual input data. The compression option works in a similar fashion. If the sample option is enabled, a uniform random sample of a size specified by the user is taken before extracting the data from the database server.

```
+----------------+-----------------------------------+
| name           | func                              |
+================+===================================+
| train_rnforest | {                                 |
:                : import pickle                     :
:                : from sklearn.ensemble             :
:                :     import RandomForestClassifier :
:                :                                   :
:                : clf = RandomForestClassifier(n)   :
:                : clf.fit(data, classes)            :
:                : return {'clf': pickle.dumps(clf), :
:                :      'estimators':n }             :
:                : };                                :
+----------------+-----------------------------------+
```

Listing 1: MonetDB UDF Example.

2.3 Nested UDFs

Loopback queries inside UDFs are supported by MonetDB/Python. They allow users to query the database directly from within the UDF. The results of the query are converted to the host language of the UDFs. In MonetDB/Python UDFs, loopback queries can be can issued through the _conn object that is passed to every UDF. They are useful because they can bypass cardinality restrictions of the relational querying model.

The loopback queries can also contain UDFs themselves. An example of a nested UDF is shown in Listing 3. This UDF calls the function depicted in Listing 2 with a set of different parameters in order to find the best classifier and its parameters. In order to provide support for extracting and debugging these nested UDFs, we must execute the same transformation steps on the nested UDFs as we did for the main UDF being executed. With an additional transformation rule on the _conn object to the correct function call. After being transformed, we can execute the nested UDFs locally by transferring their input data in conjunction with the main UDF data, Finally they can be executed from within the IDE.

```
import pickle

def train_rnforest(data, classes, n_estimators):
    import pickle
    from sklearn.ensemble
        import RandomForestClassifier
    clf = RandomForestClassifier(n_estimators)
    clf.fit(data, classes)
    dict = {'classifier': pickle.dumps(clf),
            'estimators':n_estimators }
    return dict;

input_parameters =
        pickle.load(open('./input.bin','rb'))

train_rnforest(input_parameters['data'],
        input_parameters['classes'],
        input_parameters['n_estimators'])
```

Listing 2: Exported UDF Code Example.

```
CREATE FUNCTION find_best_classifier(esttest INT)
RETURNS DOUBLE LANGUAGE PYTHON {
import pickle
(tdata, tlabels) = _conn.execute("""SELECT data,
    labels FROM testingset""");
best_classifier = None
best_classifier_answers = -1
best_estimator = -1
for estimator in esttest:
    res = _conn.execute
        ("""
    SELECT *
    FROM train_rnforest(
        (SELECT data, labels
        FROM trainingset), %d);
    """ % estimator)
    classifier = pickle.loads(res['clf'])
    predictions = classifier.predict(tdata)
    correct_pred = predictions == tlabels
    correct_ans = numpy.sum(correct_predictions)
    if correct_ans > best_classifier_answers:
        best_classifier = classifier
        best_classifier_answers = correct_ans
        best_estimator = estimator
return {'clf': best_classifier,
        'n_estimators': best_estimator}
};
```

Listing 3: Nested UDF Example.

2.4 Extensions

Extending to Other Databases. Our solution is implemented for MonetDB. However, our plugin can be easily extended to work with other RDBMSes, as the same implementation strategy can be used. However, the processing model of the respective database needs to be taken into consideration. MonetDB uses the operator-at-a-time processing model, which means the UDFs

are only called once with the entire columns as input. Row-store databases (e.g., Postgres or MySQL) use the tuple-at-a-time processing model, under which the UDFs are called many times with only individual rows as input. As this changes the way UDFs are called, the execution of the UDF must be adapted to these differing processing models. The tuple-at-a-time execution method can be simulated by issuing a loop over the input tuples.

Extending to Other UDF Languages. Our solution is implemented for Python/PyCharm. However the plugin is fully developed in Java and compatible with all the other JetBrains IDEs. In order to extend our plugin for other UDF languages, the code transformations for the new language must be added into the plugin. Additional care must be taken when dealing with compiled languages. Our model assumes that the RDBMS stores the source code of the UDF. If the database stores only a compiled blob of the UDF, the code transformations cannot be applied and an alternate solution must be used. In addition, when dealing with compiled languages some additional work must be performed on compiling and linking the code prior to execution.

2.5 Demo Outline

In the general outline for our interactive demo, we will introduce the typical setup for UDF Development. The general presentation for all the scenarios is as follows:

(1) We introduce the typical setup for UDF Development: Developers write code in their text editor of choice, insert the UDF into the database through a SQL command, repeating this process if the UDF has any bugs.
(2) We show common pitfalls in developing a UDF. Foremost, we focus on issues related to debugging.
(3) We show how bugs can be located using simplistic debugging strategies like print debugging.
(4) Finally, we repeat the same process but using devUDF to facilitate the development workflow. Showcasing how easy, fast and secure it is to use in the UDF development workflow.

In our demonstration we will ingest several CSV files, located in one directory, with one column of integers, our final goal is to create a UDF that calculates the mean deviation of said column, as a reference we compare the results with a correct version of the function. As common pitfalls, we will showcase the following scenarios:

Scenario A. In this scenario we present a UDF that calculates the median of a column with a bug, depicted in Listing 4. In line 9, the regular difference is calculated instead of the absolute difference which produces a semantic error, that is syntactically correct but logically incorrect.

Scenario B. Now, we use a correct version of the *mean_deviation* function. However, we introduce a bug in our data loader. Creating a data dependent error depicted in listing 5. In line 5 we introduce the bug that skips one of the CSV files in a given directory because it considers that *range* is right side inclusive.

3 SUMMARY

When it comes to assessing the potential impact of devUDF, we point out two current trends: First, the use of UDFs to perform In-Database Analytics is gaining popularity with support of many languages in major DBMSs. Especially in data science environments when the data is already stored inside a DBMS. Second, IDEs like Eclipse, IntelliJ and PyCharm have been gaining popularity over more simplistic text editors. Looking at both

```
1  CREATE FUNCTION mean_deviation(column INTEGER)
2  RETURNS DOUBLE LANGUAGE PYTHON {
3      mean = 0
4      for i in range (0, len(column)):
5              mean += column[i]
6      mean = mean / len(column)
7      distance = 0
8      for i in range (0, len(column)):
9              distance += column[i] - mean
10     deviation = distance/len(column)
11     return deviation;
12 };
```

Listing 4: Wrong mean deviation.

```
1  CREATE FUNCTION loadNumbers(path STRING)
2  RETURNS TABLE(i INTEGER)
3  LANGUAGE PYTHON {
4  files = os.listdir(path)
5  result = []
6  for i in range (0,len(files)-1):
7          file = open(files[i],"r")
8          for line in file:
9              result.append(int(line))
10 return result
11 };
```

Listing 5: Wrong data loader.

trends, we see a growing market for tools like devUDF, especially considering the void it fills in UDF development workflow.

4 ACKNOWLEDGMENTS

This work was funded by the Netherlands Organisation for Scientific Research (NWO), projects “Data mining on high volume simulation output” (Holanda) and “Process mining for multi-objective online control” (Raasveldt).

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).
[2] Pierre Carbonnelle. 2018. Top IDE index. https://pypl.github.io/IDE.html
[3] Pedro Holanda, Mark Raasveldt, and Martin Kersten. 2017. Don’t Hold My UDFs Hostage - Exporting UDFs For Debugging Purposes. In *SSBD, Brazil.*
[4] IBM. 2018. Optim Development Studio. https://www.ibm.com/support/knowledgecenter/en/SSZ6WM_3.0.0/com.ibm.dbapp.doc/cloud-tools/cods_overview.html
[5] Jon Loeliger and Matthew McCullough. 2012. *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O’Reilly Media, Inc.".
[6] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.
[7] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *SSDBM.*
[8] Mark Raasveldt and Hannes Mühleisen. 2017. Don’t Hold My Data Hostage-A Case For Client Protocol Redesign. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1022–1033.
[9] The pgAdmin Development Team. 2018. pgAdmin 4. https://www.pgadmin.org/docs/pgadmin4/dev/l