# In-Database Analytics with ibmdbpy

Edouard Fouché
Karlsruhe Institute of Technology
Karlsruhe, Germany
edouard.fouche@kit.edu

Alexander Eckert
IBM Deutschland R&D
Böblingen, Germany
aeck@de.ibm.com

Klemens Böhm
Karlsruhe Institute of Technology
Karlsruhe, Germany
klemens.boehm@kit.edu

## ABSTRACT

The increasing size of the available data and database volumes represents a real challenge for the data management community. In general, current approaches in data mining require the data to be first extracted from an underlying database. From a practical point of view, this presents many drawbacks. In this short article, we present a possible solution to bridge the gap between data repositories and end user analysis. We demonstrate the interestingness of this approach with *ibmdbpy*, an open source Python interface developed by IBM for database administration and data analytics.

## CCS CONCEPTS

• **Information systems** → **Data analytics**; *Call level interfaces*; *Database administration*; *Data mining*;

## KEYWORDS

Database, Data Analytics, Data Mining, SQL-Pushdown

## 1 INTRODUCTION

To conduct data analysis, one typically must first extract the data locked in a database management system (DBMS). This is problematic, for several reasons: First, the volume of data can be very large, and the data can be heterogeneous, so that it cannot be obtained fast and easily. Second, if the data is big, it may not fit in memory. Third, if the data contains sensitive information such as credit card numbers, passwords or business-related information, the data needs to be properly encrypted. This also holds for all copies of the data. Fourth, when new data becomes available, one must update the duplicates, which requires expensive synchronization episodes. Finally, data manipulation and mining often is an interactive process, involving different programming languages, where short response times are preferred.

As a result, one often resorts to the extraction of small samples, or transfers the data to a cluster system for further processing. However, samples may be unrepresentative of the real data distribution. Working with computer clusters in turn gives way to high infrastructure expenses.

| SL | SW | PL | PW | CLASS |
|-----|-----|-----|-----|------------|
| 5.0 | 3.6 | 1.4 | 0.2 | setosa |
| 6.8 | 2.8 | 4.8 | 1.4 | versicolor |
| 4.9 | 3.1 | 1.5 | 0.1 | setosa |
| 6.9 | 3.1 | 5.4 | 2.1 | virginica |

**Table 1: Sample of the Iris data set [5]**

While modern DBMSs do incorporate analytical components, interacting with a DBMS can be cumbersome. From the end user perspective, it requires the knowledge of database specific languages such as SQL. Despite the maturity of database technology, database languages are not high-level and may be unsuitable for extensive data analysis. For example, consider the sample of the Iris data set [5] shown in Table 1. To compute the mean of each attribute with respect to the class, one could create the following SQL query:

```sql
SELECT AVG("SL"), AVG("SW"), AVG("PL"), AVG("PW")
FROM IRIS
GROUP BY "CLASS"
```

This query is very simple. However, for each table with other columns, the query needs to be rewritten. If we want to express more complex operations, such as filtering, sorting or joining over other aggregated values, writing SQL queries becomes cumbersome. As a comparison, this task is achieved with Python using a short and schema-independent syntax:

```python
iris.groupby("CLASS").mean()
```

In this work, we present a general framework to cope with these issues, with the core idea to "bring" analytics to data, instead of data to analytics. We connect to distant databases and provide high-level Python analytics functions that are translated in SQL queries and pushed to the database for execution, leaving the data where it is. Thus, one can take advantage of in-database performance-enhancing features, such as in-memory column store and parallel processing. Our implementation - *ibmdbpy* - is a Python library that imitates the high-level syntax and methods of standard libraries for data analysis such as *Pandas* and *Scikit-learn*. So far, *ibmdbpy* offers basic statistical functions, such as aggregated average, standard deviation, variance, as well as unsupervised data mining algorithms like *K-means* or supervised algorithms like *Naive Bayes* classification. The latest development focused on in-database correlation coefficients for feature selection, such as the *Pearson correlation coefficient*, *Chi statistics*, *Gini index* and *mutual information*.

*Ibmdbpy* also wraps existing in-database analytics algorithms, making them accessible and usable outside of the database environment. Being a high-level interface, *ibmdbpy* can easily be extended

to include additional features. While *ibmdbpy* is currently compatible with IBM DB2® and IBM dashDB® database instances, its principles could be extended to any SQL database as well.

We organize this paper as follows: in Section 2, we review the related work. In Section 3, we explain the architecture of the *ibmdbpy* framework. In Section 4, we present our demonstration and focus on a relevant part of data analysis, i.e. the in-database computation of various correlation coefficients. In Section 5, we present the results of the experiments. Finally, we conclude in Section 6.

## 2 RELATED WORK

Interfaces for in-database analytics exist. For example, the *Blaze* ecosystem [1] provides an interface for multiple backends, such as SQL databases, NoSQL data stores, *Spark*, *Hive*, *Impala*, and raw data files. The drawback of supporting so many backends is that it reduces the available functions to the common subset. For example, it cannot accommodate platform-specific functionality such as custom SQL keywords and procedures. Also, the *Ibis Project* [2] is related to *ibmdbpy*, but mainly supports *Impala* as a backend, and its analytics capabilities are so far limited.

In parallel, commercial and non-commercial database developers put effort in in-database frameworks, such as the *SAP HANA* database [6] or *Oracle Data Miner* [7]. Those frameworks provide end users with specific SQL keywords or functions to call in-database machine learning algorithms. In general, the algorithms are platform-specific and can only be used as a black-box. Nonetheless, these approaches are orthogonal and compatible with ours: They can be wrapped again and accessed through our interface.

## 3 FRAMEWORK

The *ibmdbpy* framework is an open source Python library. It can connect to distant IBM DB2 or IBM dashDB instances and translates a subset of Python functions into SQL automatically. The queries are sent through a middleware using ODBC or JDBC. Figure 1 illustrates the workflow of *ibmdbpy*.

End users, typically analysts or data scientists, can interact with IBM DB2/IBM dashDB from their workstation environment. They can issue basic Python commands (1) that the client transforms into SQL queries (2) then pushes to the database. IBM DB2/IBM dashDB processes the queries (3) and the results are retrieved back in raw format, typically as a list of tuples. They are then parsed (4) into the corresponding Python data structures and presented to the user. In our example, the user asks for the list of the available tables: The answer is IRIS and SWISS. With this approach, we combine the advantages of the expressiveness of high-level programming languages like Python and the performance of database systems, as our experiments in Section 5 will show.

Internally, *ibdmbpy* uses objects with similar methods as *Pandas* objects, such as the well-known *DataFrame*, but in fact, the data lies in a distant database. Calling a method leads to the construction of a string that should be a valid SQL query. We want to illustrate this on a simple example, i.e., computing the Pearson correlation matrix of the Iris data set [5].

We can retrieve the correlation matrix of an *ibmdbpy DataFrame* easily by calling the *corr* method. For the Iris data set (see sample from Table 1), this would produce the following SQL query:
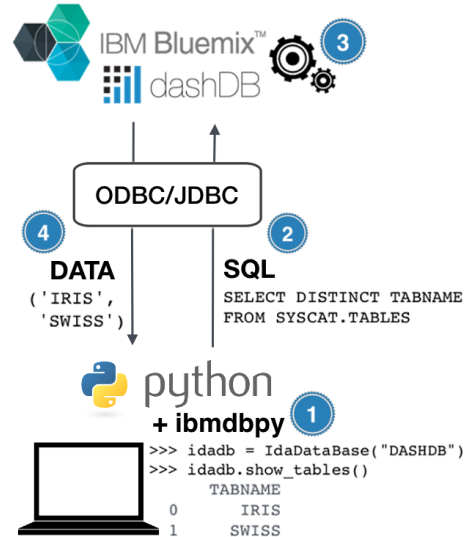


**Figure 1: The *ibmdbpy* workflow**

```
SELECT CORRELATION("SL","SW"),
CORRELATION("SL","PL"), CORRELATION("SL","PW"),
CORRELATION("SW","PL"), CORRELATION("SW","PW"),
CORRELATION("PL","PW")
FROM IRIS
```

The query is then sent to the database for execution, and the result is mapped back into a two-dimensional correlation matrix. We name the whole approach SQL-Pushdown. Algorithm 1 is the pseudo-code that constructs this query.

---

**Algorithm 1** Correlation-query construction

---

**Input:** Table *tab*
**Output:** Query for the computation of correlations in *tab*
1: **function** CORR
2:      *attributes* ← GET-ATTRIBUTES(*tab*)
3:      *tabname* ← GET-NAME(*tab*)
4:      *string* ← []
5:      **for** $p1, p2$ ← *combinations*(*attributes*, 2) **do**
6:          *string* += "CORRELATION(p1,p2)"
7:      *corrs* ← *string* join with ", "
8:      *select* ← "SELECT "
9:      *from* ← CONCATENATE(" FROM ", *tabname*)
10:      **return** CONCATENATE(*select*, *corrs*, *from*)

---

Apart from the connectivity layer, *ibmdbpy* works independently from the underlying database system, since it generates standard SQL code. It can be used jointly with *Jupyter notebook*, a web application for creating and sharing documents, containing live code, visualizations and explanatory text. This makes the data analysis interactive and human-readable.

## 4 DEMO: CORRELATION ANALYSIS

This section features a case study to show the relevance of this framework for data analysis. This will also shed light on the structure of our demonstration.

In many scenarios, it is useful to estimate the correlation between the different variables of a data set. However, the Pearson correlation coefficient, as presented in the previous section, is limited to the detection of linear dependencies. Instead, one can use *mutual information*. In information theory, *mutual information* is a measure of mutual dependence of two variables $X$ and $Y$ [4] and estimates the amount of information that the two variables share:

$$MI(X, Y) = H(X) - H(X|Y) \tag{1}$$

where $H(X)$ is the *entropy* of $X$ and $H(X|Y)$ is the *conditional entropy* of the attribute $X$, given that we know the value of an attribute $Y$, they are defined as:

$$H(X) = - \sum_i p_i * log_2(p_i), \tag{2}$$

$$H(X|Y) = - \sum_j p_j \sum_i p_{i/j} * log_2(p_{i/j}), \tag{3}$$

where $p_i$ is the prior probability for each Class $i$ of attribute $X$, $p_{i/j}$ is the probability of the value of $X$ being from Class $i$, given that the value of $Y$ is from Class $j$. The *conditional entropy* quantifies the amount of information needed to describe the outcome of a random variable $X$ given the value of another random variable $Y$. Using the *chain rule* [4], we can also formulate it the following way:

$$H(X|Y) = H(X, Y) - H(Y) \tag{4}$$

where $H(X, Y)$ is the joint entropy of the classes of $X$ and $Y$. Using Equation 4, the *mutual information* can be expressed as:

$$MI(X, Y) = H(X) + H(Y) - H(X, Y) \tag{5}$$

It is easy to compute the entropy of a list of categorical attributes $attr1, \ldots, attrN$ in the database. The following query corresponds to the SQL implementation of Equation 2:

```
SELECT SUM(-prob*LOG(prob)/LOG(2))
FROM (SELECT CAST(COUNT(*) AS FLOAT)/@n AS prob
FROM @tab
GROUP BY @attr1,...,@attrN)
```

where @n is the number of rows in table @tab, which is known by accessing metadata. Using this query as a template and Equation 5, it is easy to compute the *mutual information* of two attributes by parameterizing the variables @tab and the attribute variables @attr1,...,@attrN. Algorithm 2 show the respective pseudo-code.

---

**Algorithm 2** Mutual information

---

**Input:** Table *tab*, X *attr1*, Y *attr2*
**Output:** Mutual information of attributes X and Y in *tab*
 1: **function** MUTUALINFORMATION(*tab*, $X$, $Y$)
 2:    $H(X) \leftarrow$ ENTROPY-SQL-PUSHDOWN(*tab*, $X$)
 3:    $H(Y) \leftarrow$ ENTROPY-SQL-PUSHDOWN(*tab*, $Y$)
 4:    $H(X, Y) \leftarrow$ ENTROPY-SQL-PUSHDOWN(*tab*, $X$, $Y$)
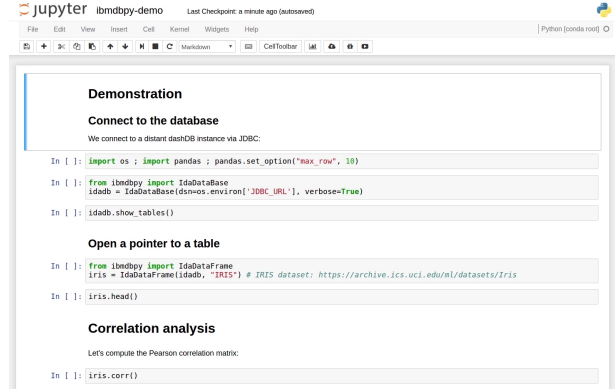 5:    **return** $H(X) + H(Y) - H(X, Y)$

---



**Figure 2: Interactive Data Analysis with *ibmdbpy***

This simple algorithm represents a general method to estimate the *mutual information* of two categorical attributes, without extracting the data. More information on the implemented measures is available in the online documentation [3] and the GitHub repository[1] of *ibmdbpy*.

In our demonstration, we will connect to a distant database through a *Jupyter notebook* and we will show how one can use *ibmdbpy* to explore the correlation relationships between table attributes via *mutual information* and conduct exploratory data analysis. We will show the real-time automated construction of the SQL queries combined with filtering and sorting. We will also make use of IBM DB2 specific in-database algorithms such as *K-Means* and *Naive Bayes*. For illustration purposes, we will use the USCensus1990 data set from the UCI Machine Learning Repository [5]. To give an idea of the interface, we show in Figure 2 a snapshot of the first section of the *Jupyter notebook* we will present during the demonstration.

## 5 EXPERIMENTAL RESULTS

In this section, we report on a runtime analysis to show the advantage of in-database computation for large data sets. The scalability was assessed using a notebook running a 64-bits operating system. The machine contains 16 GB RAM and a quad-core processor at 2.60 GHz. We use Python version 3.5. By the time of the experiments, the version 0.1.0 of *ibmdbpy* is installed. We connect to a distant database using ODBC. The distant database is an IBM dashDB enterprise instance, hosted on IBM Bluemix®. It is a virtual environment with 64 GB dedicated RAM and 16 cores at 2.60 GHz. We create additional versions of the USCensus1990 data set with 1, 5, 10 and 50 millions rows via random sampling from the original data. Originally, the USCensus1990 data set has 69 attributes and 2,458,285 instances.

Figure 3 shows the average time required to compute various correlation matrices when the number of rows in the table scales from 1 to 50 millions rows. Since the data set contains 69 columns, $69 \cdot 68 = 4692$ and $\frac{4692}{2} = 2346$ values are computed for asymmetric and symmetric measures respectively.
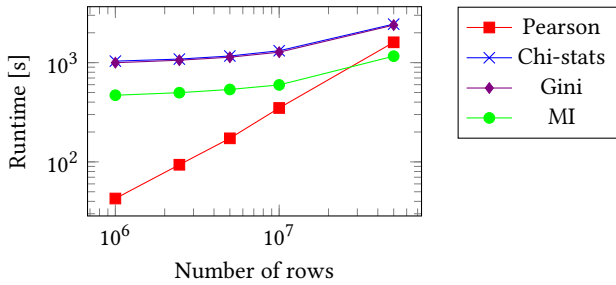
---

[1]https://github.com/ibmdbanalytics/ibmdbpy

Figure 3: Scalability of correlation computation



Figure 4: MI estimation with equal resources

|       | Scenario | 1M | 2.5M | 5M | 10M | 50M |
|-------|----------|------|--------|---------|---------|----------|
| Local | Load | 8.8 | 21.2 | 42.5 | 107.3 | 562.3 |
|       | Execute | 1937.4 | 5662.5 | 12720.0 | 29068.1 | 150069.7 |
| Total | Local | 1946.1 | 5683.8 | 12762.5 | 29175.4 | 150632.7 |
|       | SMP | **118.2** | **165.5** | 295.9 | 894.8 | 959.2 |
|       | MPP | 146.3 | 204.6 | **281.6** | **354.6** | **761.2** |

Table 2: Scalability experiment results (in seconds)

We can see that the runtimes of *Chi-statistics* and *Gini index* matrix are similar and relatively higher. This is explained by their asymmetric property. Because of symmetry, the *mutual information* matrix is nearly twice as fast to compute. Interestingly, the *Pearson correlation coefficient* shows the worst scalability, despite a comparatively low runtime for small instances.

## 5.1 Comparison local and in-database

It is also interesting to compare the runtime of the following three scenarios: First, the local computation of the *mutual information* matrix with Python 3.5. Second, the in-database computation using *ibmdbpy* on a SMP dashDB Local instance, and third, the performance of the same computation on a MPP dashDB multi-node cluster. For a better comparison, the local computation is executed on the same machine where dashDB Local is deployed. In the case of a MPP dashDB system, the head node is used. The SMP machine runs in a virtual environment with 16 cores at 2.27 GHz and 16 GB RAM. The MPP system has the same CPU configuration with 64 GB RAM and consists of 3 identical nodes in total.

In addition to the runtime of a local *mutual information* matrix computation, the time to load the data into the Python environment has to be considered too. Data sets typically are loaded locally as csv files. We measure the time to load the data in memory using the function *pandas.read_csv* from the *Pandas* library. After the data is loaded into the memory, we compute the *mutual information* score using the function *sklearn.metrics.mutual_info_score* from the *Scikit-learn* library as baseline.

We show the results of the experiment in Table 2. The local computation time increases rapidly with the number of row processed, and even for 1 million rows it is one order of magnitude slower compared to the in-database computation. For large data sets like the 50 millions rows sample in particular, reading the data set from a file takes more than half of the time of an actual complete in-database computation. When we compare the in-database results directly, we can see that the MPP dashDB system provides a better performance on data sets above 2.5 millions rows.
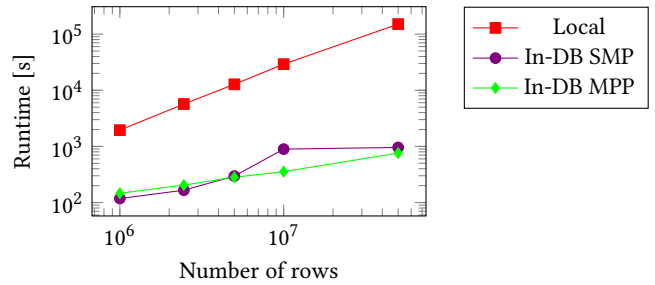
Figure 4 graphs the results. We can see the clear difference between the local and in-database computation. Figure 4 also shows that the in-database approach on MPP systems becomes beneficial as the table grows.

## 6 CONCLUSIONS

In our use case we have shown that the *ibmdbpy* framework bridges the syntactical gap between SQL databases and the Python programming language: *Ibmdbpy* provides a familiar Python interface for manipulating and mining data stored in relational databases. The framework offers performance advantages by carrying out mining and transformation tasks directly in the database, without the need for a full data transfer to the Python environment. While our presentation has focused on the computation of correlation measures, *ibmdbpy* offers a range of different supervised and unsupervised data mining algorithms. Moreover, it can be easily extended by wrapping SQL queries and user-defined stored procedures.

In future work, we will investigate how to extend the *ibmdbpy* library to support the execution of user-defined Python functions in the database. Besides the advantage of parallel Python code execution, the collocation of the Python runtimes with the data nodes of a multi-node DBMS also promises fast transfer rates. In this context, it will be interesting to leverage the Spark capabilities in IBM dashDB and create synergies with the *PySpark* library.

## REFERENCES

[1] Continuum Analytics. 2018. The Blaze Ecosystem. http://blaze.pydata.org.
[2] Inc. Cloudera. 2018. Ibis Project Blog. http://www.ibis-project.org.
[3] IBM Corporation. 2018. ibmdbpy 0.1.4. https://pypi.python.org/pypi/ibmdbpy.
[4] Thomas M. Cover and Joy A. Thomas. 2006. *Elements of Information Theory*. Wiley-Interscience, New York, NY, USA.
[5] Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml
[6] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.
[7] Pablo Tamayo, C Berger, Marcos Campos, Joseph Yarmus, Boriana Milenova, A Mozes, M Taft, Mark Hornick, R Krishnan, S Thomas, M Kelly, D Mukhin, B Haberstroh, Susie Stephens, and J Myczkowski. 2005. Oracle Data Mining. In *Data mining and knowledge discovery handbook*. 1315–1329.