Incorporating Scikit-Learn Libraries into AIDA

Shelley Xia

1. Introduction

As the popularity of artificial intelligence increases, machine learning inevitably becomes a heated topic. The rapid growth in data science and machine learning fields gives rise to the birth of AIDA, a Python based in-database framework for data scientists. Able to perform analysis near-data, AIDA facilitates both relational operations and linear algebra through a unified data abstraction. In order to adapt to the growing interest in ML, AIDA furthermore embarks on the mission to introduce existing ML libraries into its system. Among these libraries, scikit-learn is one of the most recognized Python libraries with simple usage. Therefore, our work focuses on leveraging the scikit-learn library to construct an environment in which users can work with ML models.

It is worth mentioning that most of the work involved in the models is accomplished at AIDA's server, with as little participation of AIDA's client as possible. This aligns with AIDA's idea of database-internal development environment, reducing the overheads in data transfer. When training and testing models, both accuracy and efficiency are indispensable characteristics. However, the current version of AIDA does not support any pre-defined ML algorithms. Such algorithms need to be written from scratch using basic operators by users, presenting significant complexity and inefficiency. Thanks to scikit-learn, we obtain a model blueprint that is ready to use. While accuracy is provided by scikit-learn, efficiency can be further improved by supporting trained models as first-class citizens. To additionally promote utilizability of trained models, we establish explicit mechanisms to persist models in our database.

In this article, we first present some background information about AIDA that is related to our work. We will then delineate our design to integrate the scikit-learn library within AIDA. Next, we would like to verify whether such an approach is advantageous by comparing it to a traditional model training method. At the same time, we wish to find the optimal solution to model storage.

2. Background

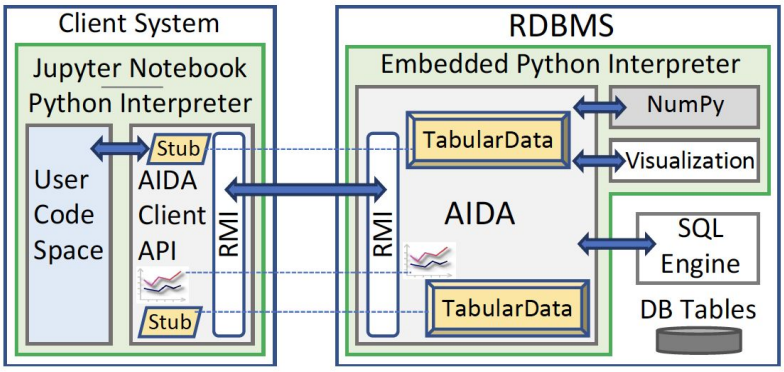


Figure 3.1: AIDA - conceptual layout

The above figure demonstrates the conceptual architecture of AIDA. AIDA's server is embedded in the python interpreter in RDBMS, and users can connect to it through regular python IDEs. The blue arrows represent interactions between different locations. For instance, users communicate with AIDA's client via user code space and AIDA's client and server communicate with each other through RMI in AIDA. The sci-kit learn package is imported at AIDA's server; users could make use of it via writing python programs at the client.

As our project requires the use of memory-resident objects and stub objects, we will list and define some additional terms.

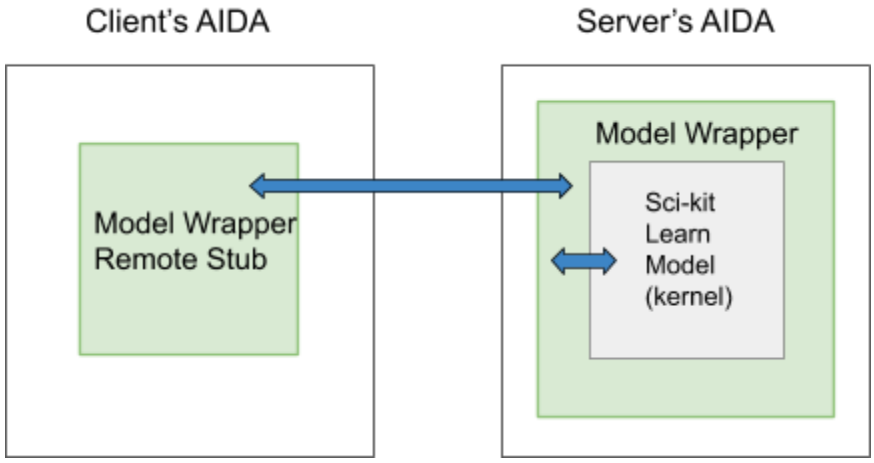
DMRO: DMRO, database memory resident objects, are objects maintained in the RDBMS memory at AIDA's server. Users can operate on them through client's API.

TabularData: TabularData object is a type of DMRO, which encapsulates all the datasets in AIDA. It could be performed linear and relational operations upon.

Remote Stub: Remote Stub object acts as a communicator through which AIDA's client API could interact with the remote objects at AIDA's server. Hence, these remote stub objects reside at AIDA's client and associate with their corresponding remote objects, such as TabularData objects, at AIDA's server.

3. Design

Scikit-learn library is fairly thorough in terms of machine learning algorithms and models. It provides various utilities for model fitting, data processing, model selection and evaluation. Therefore, we hope to leave these built-in functionalities unchanged in our AIDA database. On top of this, to facilitate the communication between AIDA's client and server, we create wrapper classes at the server which 'wrap' around the actual scikit-learn model objects. Users could access the kernel scikit-learn model by calling the function 'get_model' on a wrapper object. To avoid confusion, names of the wrapper classes are carefully chosen to match with their kernel classes. For example, 'sklearn.linear_model.LinearRegression' has a wrapper class 'LinearRegressionModel'. Both wrapper objects and scikit-learn objects reside at the server. At the client, we add corresponding remote stub classes, through whose objects users could interact with wrapper objects and in turn, scikit-learn objects. In this manner, we are able to provide users with the same syntax and usage as the original scikit-learn package, minimizing the learning curve and maximizing user experience.

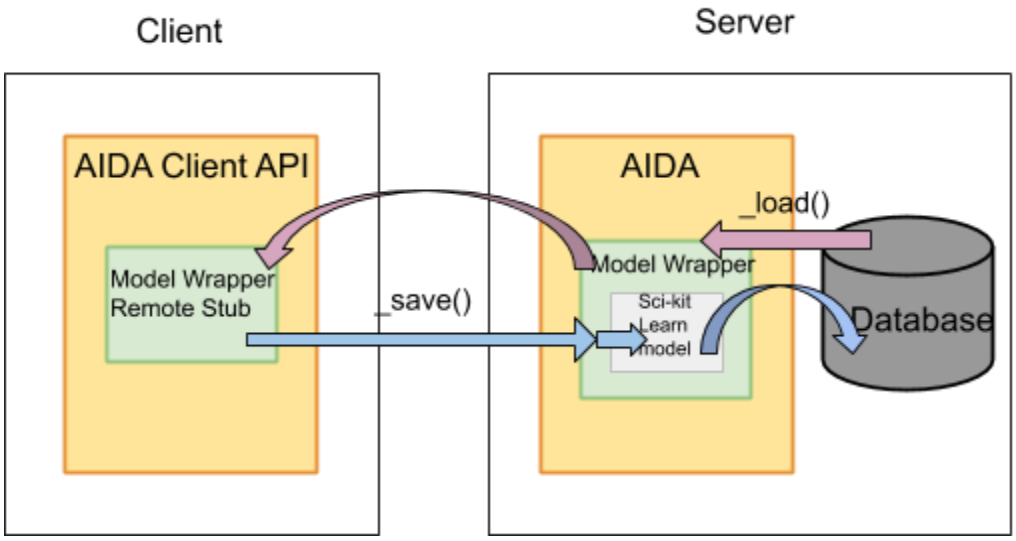


The following screenshot displays the usage of such implementation at AIDA's client. We first connect to AIDA's server and obtain a database workspace object 'dw'. We next call the function '_linearRegression' to construct a linear regression wrapper object along with its scikit-learn linear regression object at the server. Afterwards, we could perform various

scikit-learn functionalities on the wrapper object, as we would perform on the actual scikit-learn object.

```
root@405d45e79c70:/home# python3
Python 3.5.2 (default, Apr 16 2020, 17:47:17)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from aida.aida import *
>>> host='server-fxia6'
>>> dbname='bixi'
>>> user='bixi'
>>> passwd='bixi'
>>> jobName='connect'
>>> port=55660
>>>
>>> dw=AIDA.connect(host,dbname,user,passwd,jobName,port)
>>>
>>> linear_model = dw._linearRegression()
>>> linear_model.get_model()
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
>>> linear_model
<aidacommon.rdborm.LinearRegressionModelRemoteStub object at 0x7f207a67f2b0>
>>>
>>> distance = dw.gmdata2017[:,2]
>>> duration = dw.gmdata2017[:,3]
>>> distance.cdata
OrderedDict([('gdistm', array([3568, 3821, 1078, ..., 1996, 2410, 4009], dtype=int32))])
>>> duration.cdata
OrderedDict([('gduration', array([596, 704, 293, ..., 615, 618, 944], dtype=int32))])
>>>
>>> linear_model.fit(distance,duration)
<aidacommon.rdborm.LinearRegressionModelRemoteStub object at 0x7f207a684400>
>>> linear_model.coef_
array([0.21220577])
>>> linear_model.score(distance,duration)
0.9108887497281143
```

With the above implementation, users are able to build various ML models inside AIDA. However, these models only exist in the session in which they are constructed. If users were to use them at a later point or another session, we need to construct a workflow to persist these models. As a result, we introduce two functions: '_save' and '_load'. As the names suggest, '_save' takes two parameters 'model_name' and 'model', and saves both information in the database; '_load' takes one parameter 'model_name' and loads the saved model back from the database. This is achieved by utilizing the 'pickle' module in python, and what is saved as an identity for a model is its pickled byte representation. These two processes could be done in any session, allowing data scientists to publish and access any model across time and session.

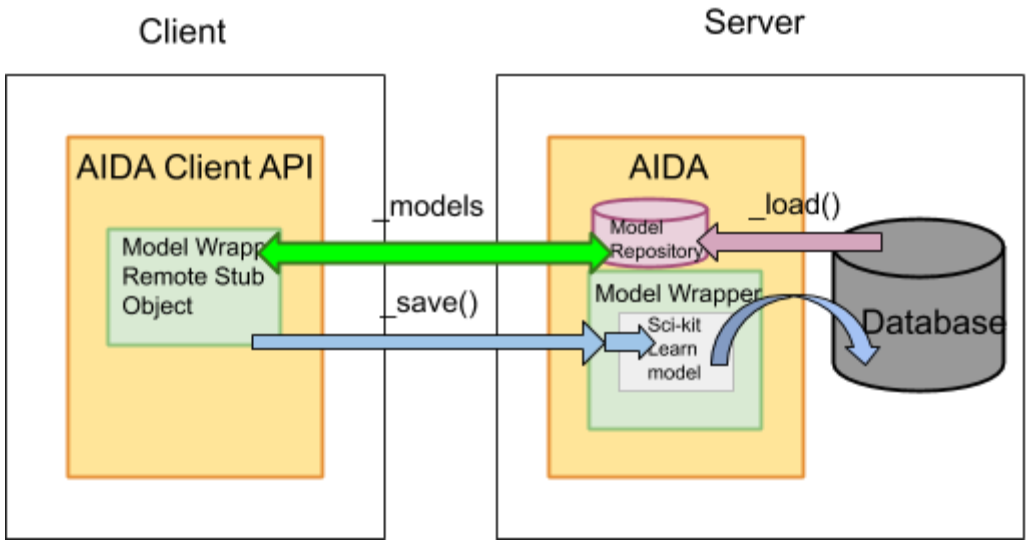


In the example below, we construct a linear regression model, save it to the database and capture it afterwards. We verify whether the two models are identical through comparing their coefficients. Note that in this example, we retrieve the model from the database within the same session. We could also load the model back in another session.

```
>>> linear_model = dw._linearRegression()
>>>
>>> distance = dw.gmdata2017[:,2]
>>> duration = dw.gmdata2017[:,3]
>>> linear_model.fit(distance,duration)
 <aidacommon.rdborm.LinearRegressionModelRemoteStub object at 0x7f64478849b0>
>>> linear_model.coef_
 array([0.21220577])
>>>
>>> dw._save('a_model',linear_model)
>>> the_model = dw._load('a_model')
>>> the_model
 <aidacommon.rdborm.LinearRegressionModelRemoteStub object at 0x7f64478849b0>
>>> the_model.coef_
 array([0.21220577])
```

It might be argued that the '_load' function could be replaced by the rudimentary UDF approach. Supposedly, a UDF using a single SQL statement 'SELECT FROM' may be faster than '_load' when loading a model once. That being said, when it comes to loading a model for a hundred times, things become quite different. The UDF approach requires loading a model from the database every time it needs to be used, and this incurs an

unnecessary overhead. We would like to employ the idea of a model repository, in which any loaded model is automatically saved and immediately available for future requests. As a result, the maximum time of loading any model is one. If users want to retrieve a model more than once, a model will be directly accessed through the repository instead of the database. In AIDA, this concept is made possible by introducing the '_models' attribute to the database workspace object. Indeed, this is an advantage given by the concept of DMRO. In order to verify our proposed approach, we performed several tests to compare the performances using model repository and UDF.




```
>>> linear_model.coef_
array([0.21220577])
>>> dw._save('a_model',linear_model)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/AIDA/aidacommon/rop.py", line 474, in wrap
    reraise(*exception);
  File "/usr/local/lib/python3.5/dist-packages/six.py", line 703, in reraise
    raise value
  File "/home/AIDA/aidacommon/rop.py", line 106, in handle
    result = func(*args, **kwargs);
  File "/home/AIDA/aidacommon/dbAdapter.py", line 479, in _save
    # throw an error if update=False and there is another model already saved with <model_name>
Exception: There already exists a model in the database with the same model_name. Please set 'update' to True to overwrite
>>> dw._save('a_model',linear_model,True)
>>> the_model_from_repo = dw._models.a_model
>>> the_model_from_repo.coef_
array([0.21220577])
```

In the code snippet, we would like to save the model constructed in the previous example again with the same name. To avoid accidental deletions, '_save' does not allow such operation unless the user specifies the 'overwrite' attribute to 'True'. After successfully

overwritting the previously saved model, we load the model back through the model repository instead of the database this time by calling '_models' attribute.

4. Evaluations

Our first experiment concerns efficiency and accuracy of scikit-learn models in comparison with those of conventional algorithms. Specifically, we analyze and juxtapose the behaviours of the sklearn linear regression model and gradient descent. The results are the times taken to complete the three phases common in building an ML model and r_squared which indicates how well the model fits some given data.

	Feature Engineering (sec)	Model Training (sec)	Model Testing (sec)	R^2 (accuracy of the model)
Scikit-Learn Linear Regression	2.300524553	0.079542478	0.369495233	0.331794754
Scikit-Learn SGD Regression	2.416093667	0.117973487	0.479581594	0.331794754
Gradient Descent (1000 iterations)	2.6653862	36.29207969	0.520232439	0.290201277
Gradient Descent (10000 iterations)	2.720980803	351.6176795	0.468488852	0.331808346

From the table, we could observe a significant difference in time spent on model training between scikit-learn models and gradient descent (GD). The calculations involved with scikit-learn models are completed internally within the scikit-learn library, whereas the GD algorithm is written from scratch using AIDA's linear algebra operators over TabularData objects. As the number of iterations increases, GD is outperformed by scikit-learn models more. Between the two scikit-learn models, linear regression and SGD (Stochastic Gradient Descent) regression, the former has the better time performance. On the other hand, elapsed time in feature engineering and model testing between scikit-learn and GD are fairly comparable. This could be explained by the fact that these two processes

require few sophisticated calculations; they involve mainly manipulations on TabularData Object and vector multiplication, respectively. In terms of accuracy, both scikit-learn models have identical r^2 while r^2 increases along with the number of iterations in GD. GD using 10000 iterations has slightly higher r^2 than that of the scikit-learn models. However, such negligible lead is far from enough to compensate for the drawbacks in efficiency of GD. Evidently, the scikit-learn linear regression model has overall the most desirable traits.

The previous assessment strengthens the reasoning behind our choice. Next, we perform another set of tests aiming to explore various means to retrieve a model from the database. Again, we use a linear regression model as our testing model for its simplicity. There are following candidate methods to 'load' a model from the database: calling '_models' at the server, calling '_models' at the client, calling '_models' through '_X' so that '_models' would be executed at the server, calling UDF inside the database. To verify whether our assumption on the benefits of model repository is proper, we examine these candidates in two scenarios, 1 iteration V.S. 100 iterations, rendering eight test cases in total. Each test case is repeated for 15 times to account for variations and ensure reliability. The table below shows the calculated time averages for the eight cases.

# of Iterations	_models (server)	_models (client)	_X()	UDF
1	0.017190997	0.021874682	0.014932028	0.036085333
100	0.17195735	0.221819878	0.015184704	3.246318923

As expected, executions at the client perform generally worse than those at the server, due to network delay. Furthermore, using '_X' accelerates the model retrieval process, as calling '_models' through '_X' is faster than '_models' on its own. To our surprise, UDF is slower than '_models', even when loading a model for a single iteration. This may be attributed to the fact that utilizing UDF to load a model involves two user defined functions, one responsible for selecting and unpickling a model while the other responsible for calling the former function. Within the same column, duration increases as the number of

iterations goes up. However, the scale factor by which time increments differs among the four methods. From 1 iteration to 100 iterations, loading the model with UDF escalates by a scale factor of approximately 100. Both '_models' at server and client have a scale factor of 10, while time spent with '_X' remains fairly stable. Such behaviour could be ascribed to the fact that UDF uses a 'SELECT FROM' clause to communicate with the database each time it loads a model, whereas '_models' only in effect interacts with the database once no matter how many times it is called. Accordingly, executing the UDF for 100 times requires proportioanlly 100 times longer. On the other hand, 99 out of 100 times using '_models' does not involve the database; the system simply reads the saved attribute from the model repository. On the whole, '_X' has conspicuous advantages over the others, making it the most favourable candidate.

5. Conclusion

In this report, we reviewed our motivation to incorporate scikit-learn library into AIDA's system. Subsequently, we depicted the scheme to achieve this with the help of 'wrapper' classes. To further facilitate model usage across time and session, we introduce two functions for persisting any user created model. Yet the '_load' function could be optimized, leading us to consider the possibility of model repository. As illustrated in our evaluations, the implementation of repository successfully accelerates model retrieval through reducing the overheads from redundant unpickling.

Henceforward, we would like to explore more functionalities inside the scikit-learn package and enrich AIDA's machine learning working environment. As there are bountiful ML resources in the CS community, we also look forward to investigating some other ML libraries, such as TensorFlow and SciPy. In the end, we aspire to leverage existing resources.

References

1. Joseph Vinish D'Silva, *AIDA An Agile Abstraction for Advanced In-database Analytics*, 2020