SDIL Platform Tutorials

A supplementary document to the SDIL Platform Documentation

Version 1.0

Copyright © 2016 Smart Data Innovation Lab (SDIL)

PUBLISHED BY SDIL

WWW.SDIL.DE

Contents

1. Document History

Revision	Date	Description
0.1	26.05.2015	First draft
0.2	03.06.2015	New Sections: • Introduction • Terracotta BigMemory Max tutorials • IBM InfoSphere Big Insights tutorials using the Web Console and the InfoSphere BigInsights Tools for Eclipse.
0.3	02.07.2015	New Section: • IBM SPSS Modeler - Example using „Essential for R“ plugin
0.4	15.12.2016	Changes: • New document template • Removed chapter for IBM InfoSphere Big Insights Web Console and eclipse plugin - not anymore available on the SDIL Platform • Fixed references and IBM SPSS Modeler version number for the R plugin
0.5	26.01.2017	New Section: • Data Mining • Tutorial 1 - Classification of the iris flowers dataset on the Jupyter Notebook using Python
0.6	30.01.2017	Added a new tutorial: • Tutorial 2 - Clustering Customers using K-Means in HANA

0.7	08.02.2017	Added a new tutorial: • Tutorial 3 - Heart Disease Prediction Model using Decision Tree in Apache Spark
0.8	15.03.2017	Edit HANA tutorial: • added paragraph "Perform the K-means with SAP HANA Application Function Modeler (AFM)"
0.9	12.05.2017	Edit HANA tutorial: • added a pre-requisite for activation of a flowgraph in the 5.3.4 paragraph, step 14.
1.0	12.12.2018	• hided the Terracotta Software AG tutorial

2. Introduction

This is a supplementary document to the SDIL Platform Documentation user guide you should read first. The scope of the document is to provide you step-by-step tutorials and practical examples to get started with the main tools available on the SDIL Platform, e.g. IBM SPSS Modeler.
In addiction you find three data mining tutorials about typical mining tasks such as clustering and classification by the use of different powerful tools and languages (Python and the scikit-learn library, the Predictive Analysis Library on SAP HANA, the Machine Learning Library (MLlib) on Apache Spark). These tutorials are based on sample open dataset available on Internet. They provide step-by-step instructions how to load, filter and analyze the data, enable the data mining functions and perform mining models.

Before starting these tutorials, make sure you have an SDIL account.

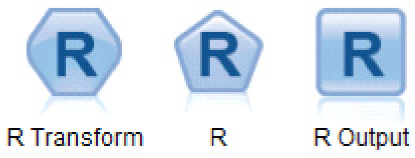
The tutorials are mainly basic and cover only the main features of each tool. Please check the official online documentation to learn more on the product itself and advanced features.

3. IBM SPSS Modeler

Read first the "IBM SPSS Modeler" chapter on the SDIL Platform Documentation to learn how to start the SPSS Modeler v 18 on the SDIL Platform.

3.1 Example using „Essential for R" plugin

The nodes in SPSS Modeler where you can insert R scripts are:



In this simple example, the R Transform node is used to implement a custom R algorithm that adds one day to a given date. This example is taken from the official user guide IBM SPSS Modeler R Nodes you can check for more information and additional examples.

1. Add a User Input node, from the Sources palette, to the stream canvas.
2. Double-click the User Input node to open the node dialog box.
3. In the table, enter dob in the Field cell, select Date in the Storage cell, and enter 2001-01-01 in the Values cell.
4. Click OK to close the User Input node.
5. Add an R Transform node, from the Record Ops palette, to the stream canvas and connect it to the User Input node.
6. Double-click the R Transform node to open the node dialog box.

7. In the R Transform Syntax field on the Syntax tab, enter the following R script:

```
day<-as.Date(modelerData$dob, format="%Y-%m-%d")
next_day<-day + 1
modelerData<-cbind(modelerData,next_day)
var1<-c(fieldName="Next ↙
    day",fieldLabel="",fieldStorage="date",fieldMeasure="",fieldFormat="",
fieldRole="")
modelerDataModel<-data.frame(modelerDataModel,var1)
```

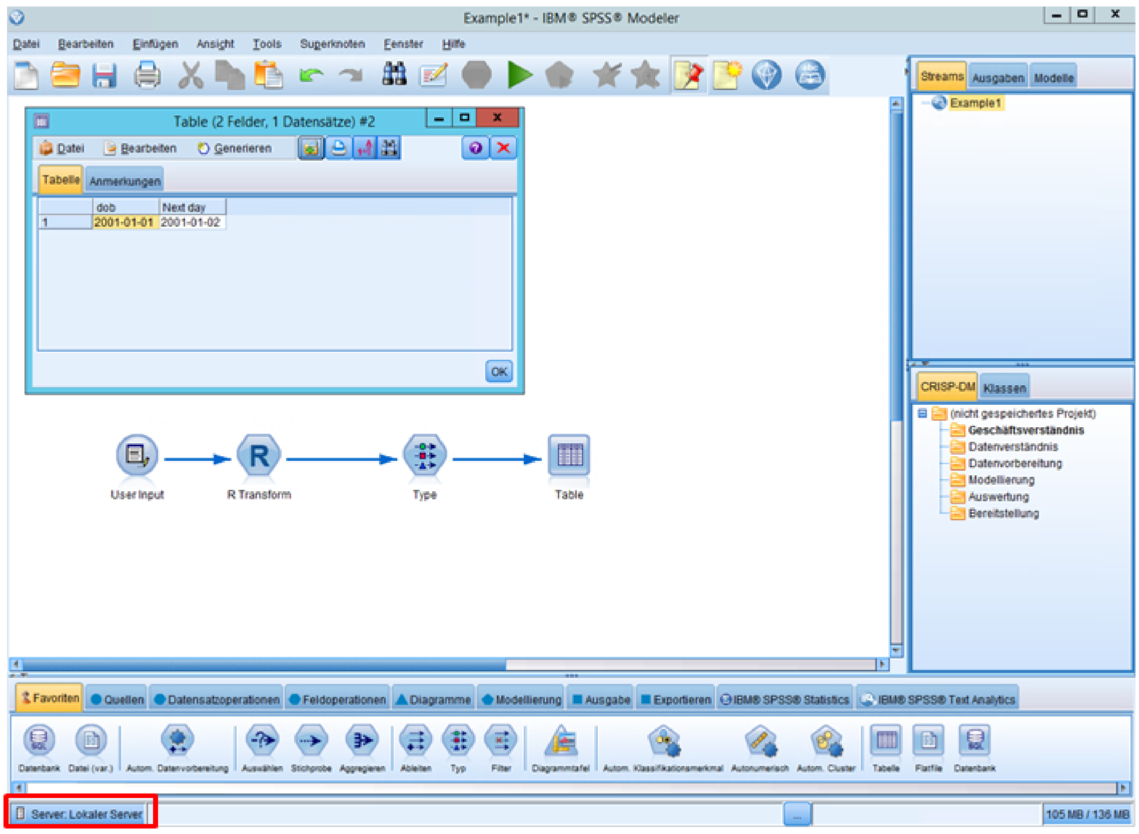
8. Select Convert date/time fields to R classes with special control for time zones. Keep the default option POSIXct selected.

9. Add a Type node, from the Field Ops palette, to the stream canvas and connect it to the R Transform node.

10. Add a Table node, from the Output palette, to the stream canvas.

11. To see the result of executing the R script in the R Transform node, connect the Table node to the Type node, double-click the Table node, and click Run.

12. The table contains the original date, and the new date in the field named *Next day*; this field was created by the R script


Example1* - IBM® SPSS® Modeler
Datei Bearbeiten Einfügen Ansicht Tools Superknoten Fenster Hilfe
Table (2 Felder, 1 Datensätze) #2
Datei Bearbeiten Generieren
Tabelle Anmerkungen
OK
User Input
R Transform
Type
Table
Streams Ausgaben Modelle
Example1
CRISP-DM Klassen
(nicht gespeichertes Projekt)
Geschäftsverständnis
Datenverständnis
Datenvorbereitung
Modellierung
Auswertung
Bereitstellung
Favoriten Quellen Datensatzoperationen Feldoperationen Diagramme Modellierung Ausgabe Exportieren IBM® SPSS® Statistics IBM® SPSS® Text Analytics
Datenbank Datei (var.) Autom. Datenvorbereitung Auswählen Stichprobe Aggregieren Ableiten Typ Filter Diagrammtafel Autom. Klassifikationsmerkmal Autonumerisch Autom. Cluster Tabelle Flatfile Datenbank
Server: Lokaler Server
105 MB / 136 MB

	dob	Next day
1	2001-01-01	2001-01-02

Important note: *the "Essential for R" plugin works only on the local server where the version 3.1.2 of R is installed.*

4. Data Mining

Data mining is the process to discover interesting knowledge from large amounts of data. It is an interdisciplinary field with contributions from many areas, such as statistics, machine learning, information retrieval, pattern recognition and bioinformatics. Data mining is widely used in many domains, such as retail, finance, telecommunication and social media. The main techniques for data mining include classification and prediction, clustering, outlier detection, association rules, sequence analysis, time series analysis and text mining, and also some new techniques such as social network analysis and sentiment analysis.

In real world applications, a data mining process can be broken into six major phases:

- Business understanding: get a clear understanding of the problem you're out to solve, how it impacts your organization, and your goals for addressing it.
- Data understanding: review the data that you have, document it, identify data management.
- Data preparation: set your data ready to use for modeling (selecting data,cleaning data...)
- Modeling: use mathematical techniques to identify patterns within your data (building models)
- Evaluation: review the patterns you have discovered and assess their potential for business use
- Deployment: put your discoveries to work in everyday business (reporting final results,reviewing final results)

In general we have a dataset of examples (called **instances**), each of which comprises the values of a number of variables, which in data mining are often called **attributes**.

There are two types of data, which are treated in radically different ways.

For the first type there is a specially designated attribute and the aim is to use the data given to predict the value of that attribute for instances that have not yet been seen. Data of this kind is called **labelled**. Data mining using labelled data is known as **supervised learning**. If the designated attribute is categorical, i.e. it must take one of a number of distinct values such as ‘very good’, ‘good’ or ‘poor’, or (in an object recognition application) ‘car’, ‘bicycle’, ‘person’, ‘bus’ or ‘taxi’ the task is called **classification**. If the designated attribute is numerical, e.g. the expected sale price of a house or the opening price of a share on tomorrow’s stock market, the task is called **regression**.

Data that does not have any specially designated attribute is called **unlabelled**. Data mining of unlabelled data is known as **unsupervised learning**.The goal in such problems may be to discover groups of similar examples within the data, where it is called **clustering** or to determine the distribution of data within the input space, known as **density estimation**.

4.1 Introduction

The following tutorials provide just some examples of possible use cases for data mining using different algorithms on open datasets. The main purpose of this section is to familiarize you with the SDIL Platform and the data analysis tools available, e.g. Python on the Jupyter Notebook, SAP HANA and Apache Spark. The use cases are the following:

1. Supervised learning - Classification of the iris flowers dataset using the Python scikit-learn library
2. Unsupervised learning - Clustering Customers using K-means in HANA with the Predictive Analysis Library (PAL)
3. Supervised learning - Heart Disease Prediction Model using Decision Tree with the Apache Spark's MLlib machine learning library.

Tutorial scripts

The code snippets used in the tutorials are available on the *SDIL Git repository* under the project *sdil_tutorials*. You will find a separate folder for each tutorial. Each folder contains in addition to the file script also the dataset used, even if it is open and available on the web. Please check the paragraph 2.4.5 of the *SDIL Platform Documentation* to check how to access and use the SDIL Git Repository.

4.2 Tutorial 1 - Classification of the iris flowers dataset on the Jupyter Notebook using Python

In this tutorial we are going to perform a supervised learning task: the classification of the iris flowers dataset using different algorithms. This famous iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris *setosa*, *versicolor*, and *virginica*.

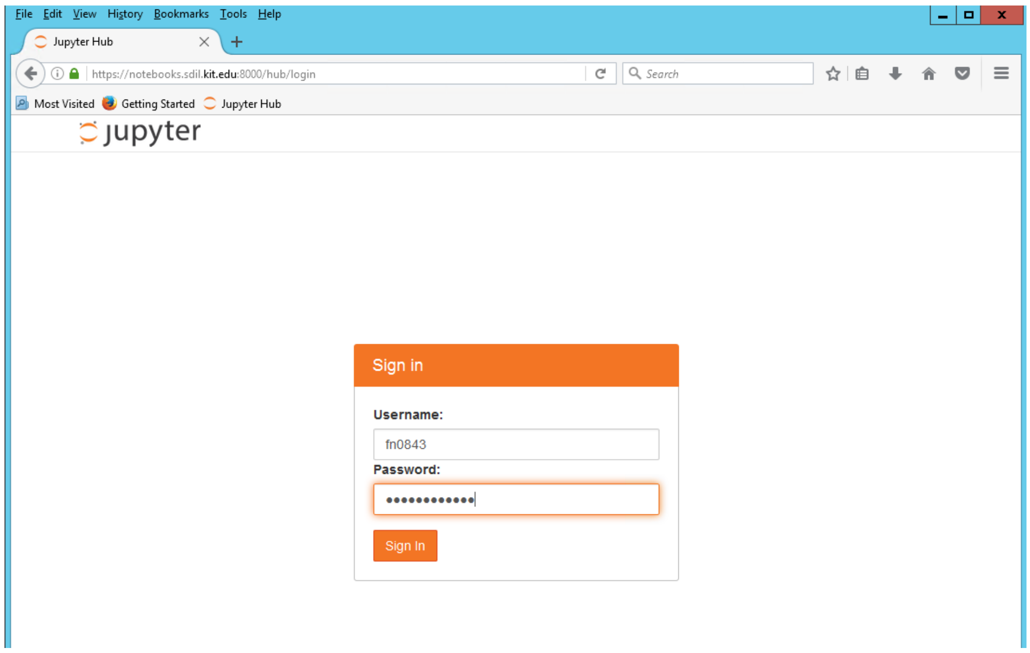
We will use the **scikit-learn** library of Python and we run the code using the Jupyter Notebook on the SDIL Platform.

The dataset is available in a standard installation of Python and it can be also found on the **UCI Machine Learning Repository - Iris Data Set**.

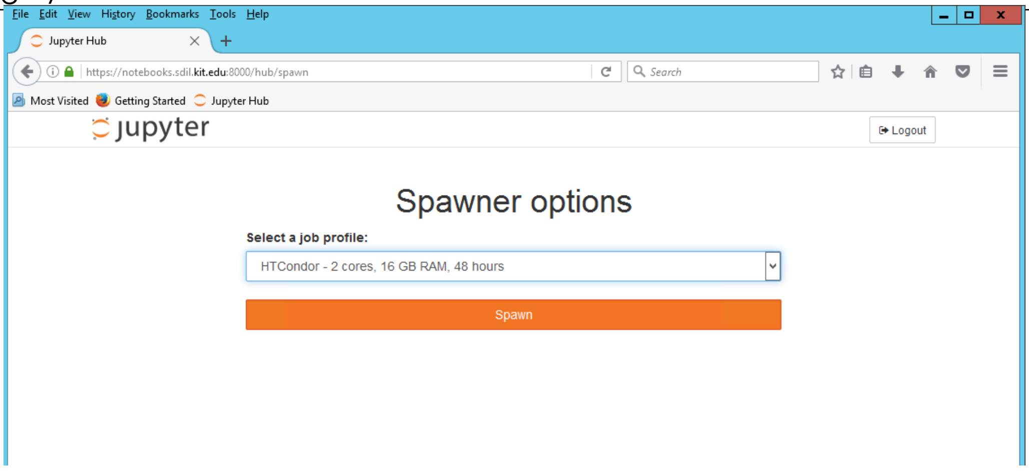
To access the code snippets used in the tutorial please visit the iris_tutorial folder on the *SDIL Git repository* and check out the *iris_tutorial.ipynb* file. Log in with your SDIL credentials. Please check the paragraph 2.4.5 of the *SDIL Platform Documentation* to check how to access and use the SDIL Git Repository.

4.2.1 Start the Jupyter Notebook

In order to access the Jupyter Notebook on the SDIL Platform log in to the Windows or Linux login server (see section 2.3 of the *SDIL Platform Documentation*), open a web browser and connect to *https://notebooks.sdil.kit.edu*. Log in with your SDIL credentials.



Once you are logged in, you have to select one of the "HTCondor" profile from the Profiles Spawner list. Click the button "Spawn"


File Edit View History Bookmarks Tools Help
Jupyter Hub
https://notebooks.sdil.kit.edu:8000/hub/spawn
Search
Most Visited
Getting Started
Jupyter Hub
jupyter
Logout

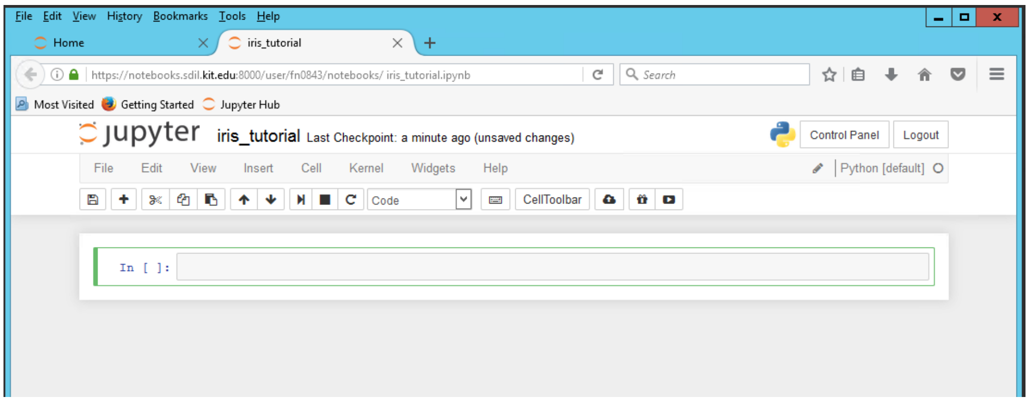
Spawner options

Select a job profile:

HTCondor - 2 cores, 16 GB RAM, 48 hours

After your server is started you can create a new Jupyter notebook using the Python kernel. Select "Python" from the "New" menu.

When you create a new notebook document, you will be presented with the notebook name, a menu bar, a toolbar and an empty code cell. Click on the notebook name to rename it to *iris_tutorial*. This renames also the file to *iris_tutorial.ipynb*.


File Edit View History Bookmarks Tools Help
Home
iris_tutorial
https://notebooks.sdil.kit.edu:8000/user/fn0843/notebooks/ iris_tutorial.ipynb
Search
Most Visited
Getting Started
Jupyter Hub
jupyter
Control Panel
Logout
File Edit View Insert Cell Kernel Widgets Help
Python [default]
Code
CellToolbar

iris_tutorial Last Checkpoint: a minute ago (unsaved changes)

```
In [ ]:
```

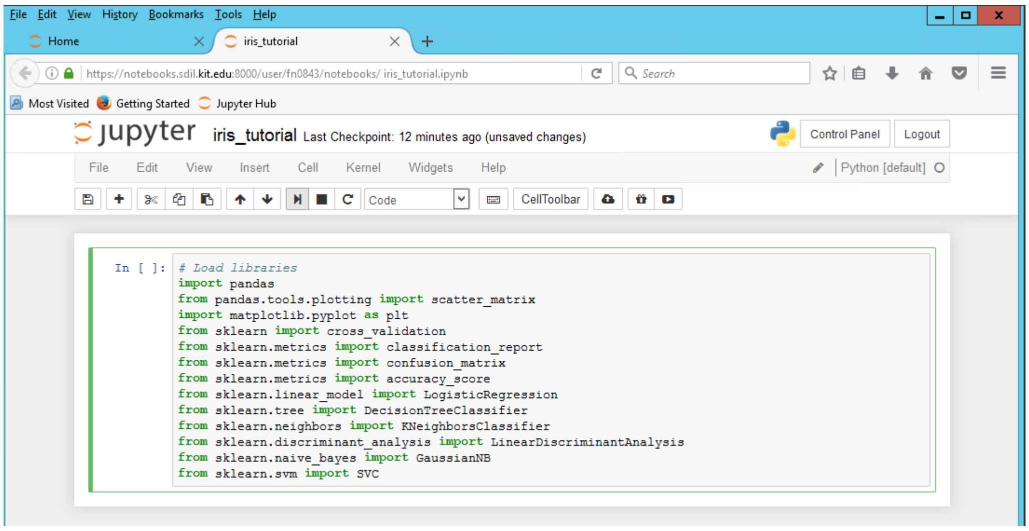
A code cell allows you to edit and write new code, with full syntax highlighting and tab completion. The language associated to a code cell is Python since we selected the Python kernel. When a code cell is executed, code that it contains is sent to the kernel. The results that are returned from this computation are then displayed in the notebook as the cell's output.

To run a cell click the button on the tool bar or Shift-Enter will execute the current cell, show output (if any), and jump to the next cell below.

Now first, let's import all of the modules, functions and objects we are going to use in this tutorial.

```
# Load libraries
import pandas
from pandas.tools.plotting import scatter_matrix
import matplotlib.pyplot as plt
from sklearn import cross_validation
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
```

Run the cell clicking the arrow button on the tool bar or press Shift-Enter.


File Edit View History Bookmarks Tools Help
Home
iris_tutorial
https://notebooks.sdil.kit.edu:8000/user/fn0843/notebooks/ iris_tutorial.ipynb
Search
Most Visited
Getting Started
Jupyter Hub
jupyter
iris_tutorial Last Checkpoint: 12 minutes ago (unsaved changes)
Control Panel
Logout
File Edit View Insert Cell Kernel Widgets Help
Python [default]
Code
CellToolbar

```
In [ ]: # Load libraries
        import pandas
        from pandas.tools.plotting import scatter_matrix
        import matplotlib.pyplot as plt
        from sklearn import cross_validation
        from sklearn.metrics import classification_report
        from sklearn.metrics import confusion_matrix
        from sklearn.metrics import accuracy_score
        from sklearn.linear_model import LogisticRegression
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.neighbors import KNeighborsClassifier
        from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
        from sklearn.naive_bayes import GaussianNB
        from sklearn.svm import SVC
```

Everything should load without error.

4.2.2 Load the Data

We are using *Pandas*(Pandas is a Python library for data analysis and data manipulation) to load the data. We will also use Pandas next to explore the data both with descriptive statistics and data visualization. Note that we are specifying the names of each column when loading the data. This will help later when we explore the data.

```
# Load dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
dataset = pandas.read_csv(url, names=names)
```

4.2.3 Data Exploration

In this step we are going to take a look at the data. We first check the size and structure of data.

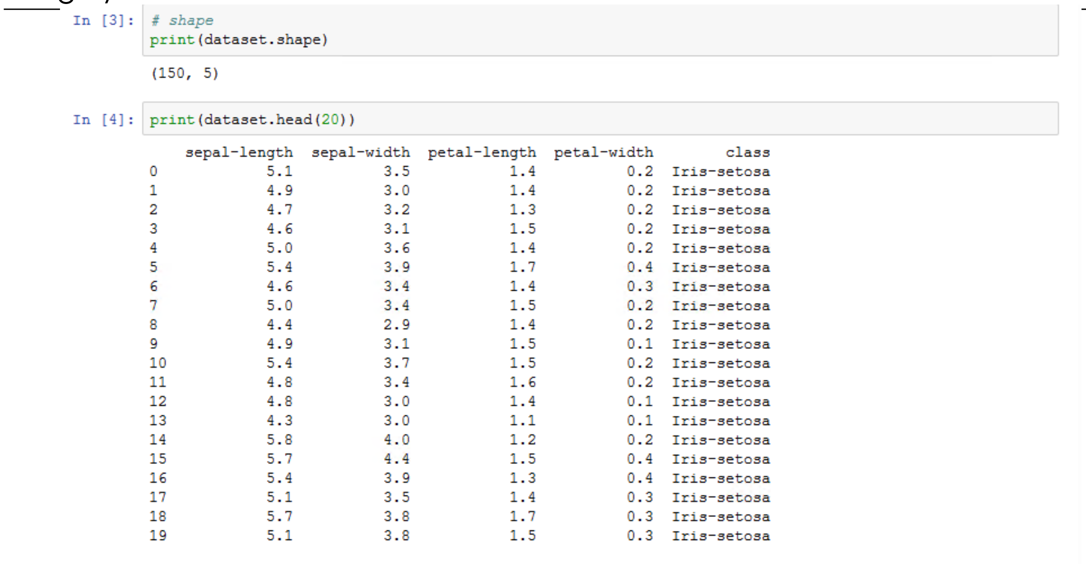
```
# shape
print(dataset.shape)
```

You should see 150 instances and 5 attributes.

It is also always a good idea to see the first 20 rows of data. The first or last rows of data can be retrieved with head or tail.

```
# head
print(dataset.head(20))
```

You should see the first 20 rows of the data:



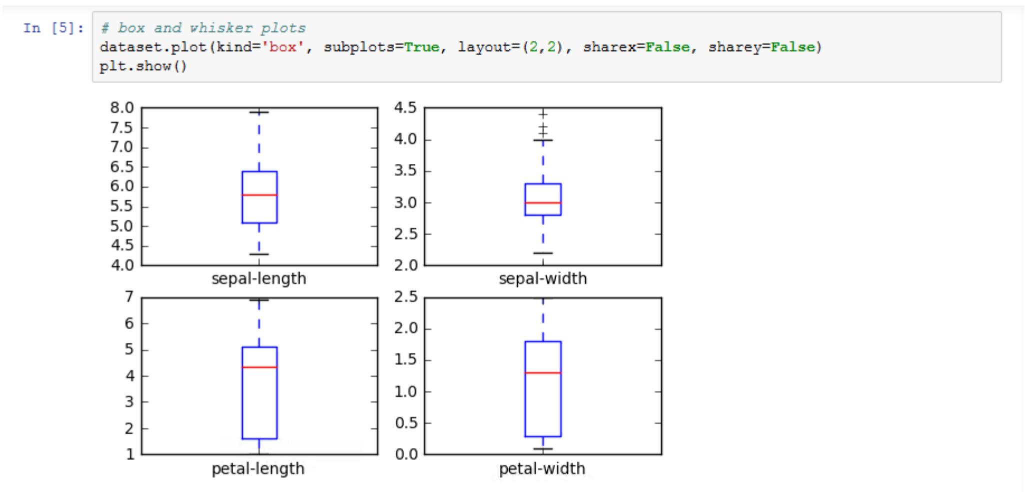
```
In [3]: # shape
        print(dataset.shape)

(150, 5)

In [4]: print(dataset.head(20))

    sepal-length  sepal-width  petal-length  petal-width        class
0            5.1          3.5           1.4          0.2  Iris-setosa
1            4.9          3.0           1.4          0.2  Iris-setosa
2            4.7          3.2           1.3          0.2  Iris-setosa
3            4.6          3.1           1.5          0.2  Iris-setosa
4            5.0          3.6           1.4          0.2  Iris-setosa
5            5.4          3.9           1.7          0.4  Iris-setosa
6            4.6          3.4           1.4          0.3  Iris-setosa
7            5.0          3.4           1.5          0.2  Iris-setosa
8            4.4          2.9           1.4          0.2  Iris-setosa
9            4.9          3.1           1.5          0.1  Iris-setosa
10           5.4          3.7           1.5          0.2  Iris-setosa
11           4.8          3.4           1.6          0.2  Iris-setosa
12           4.8          3.0           1.4          0.1  Iris-setosa
13           4.3          3.0           1.1          0.1  Iris-setosa
14           5.8          4.0           1.2          0.2  Iris-setosa
15           5.7          4.4           1.5          0.4  Iris-setosa
16           5.4          3.9           1.3          0.4  Iris-setosa
17           5.1          3.5           1.4          0.3  Iris-setosa
18           5.7          3.8           1.7          0.3  Iris-setosa
19           5.1          3.8           1.5          0.3  Iris-setosa
```

4.2.4 Data Visualization

We now have a basic idea about the data. We need to extend that with some visualizations.

We start with some univariate plots, that is, plots of each individual variable. Given that the input variables are numeric, we can create box known as box-and-whisker plot.

```
# box and whisker plots
dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)
plt.show()
```

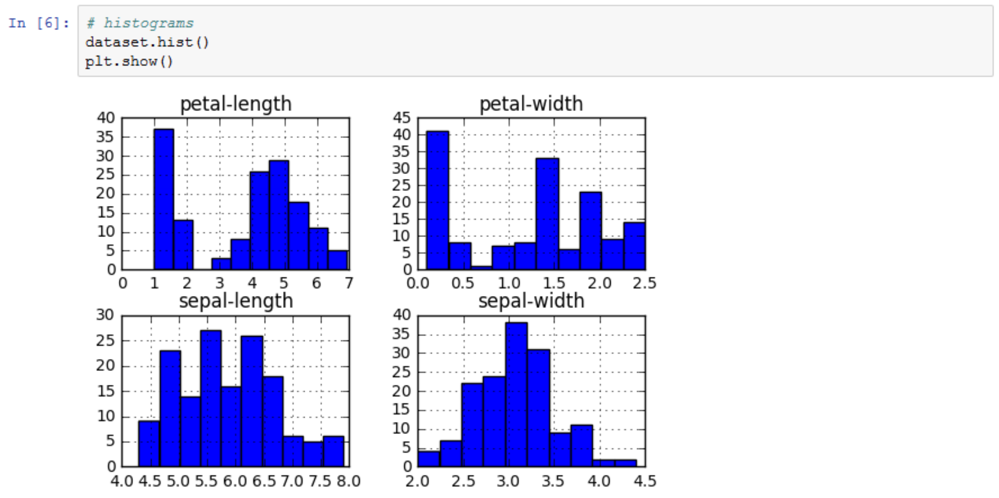


```
In [5]: # box and whisker plots
        dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)
        plt.show()
```

The box shows the median, first and third quartile of a distribution (i.e., the 50%, 25% and 75% points in cumulative distribution), and outliers. The bar in the middle is the median.

We can also create a histogram of each input variable to get an idea of the distribution.

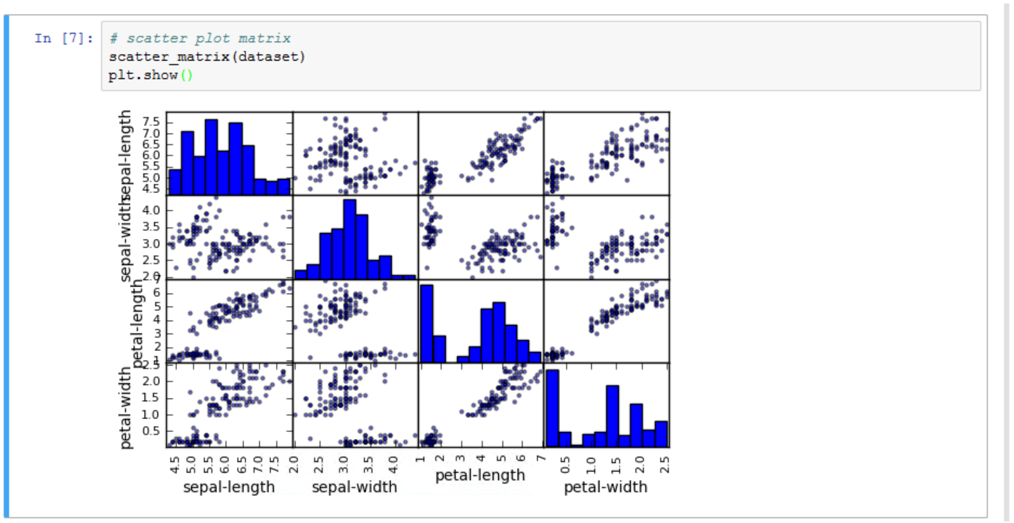
```
# histograms
dataset.hist()
plt.show()
```



```
In [6]: # histograms
        dataset.hist()
        plt.show()
```

Now we can look at the interactions between the variables. First let's look at scatterplots of all pairs of attributes. This can be helpful to spot structured relationships between input variables.

```
# scatter plot matrix
scatter_matrix(dataset)
plt.show()
```



```
In [7]: # scatter plot matrix
        scatter_matrix(dataset)
        plt.show()
```

Note the diagonal grouping of some pairs of attributes. This suggests a high correlation and a predictable relationship.

4.2.5 Evaluate some algorithms

Now it is time to create some models of the data and estimate their accuracy on unseen data. Here is what we are going to cover in this step:

1. Separate out a validation dataset.
2. Prepare metrics to use 10-fold cross validation.
3. Build 5 different models to predict species from flower measurements.

4. Select the best model.

Create a Validation Dataset

We need to know that the model we created is any good. Later, we will use statistical methods to estimate the accuracy of the models that we create on unseen data. We also want a more concrete estimate of the accuracy of the best model on unseen data by evaluating it on actual unseen data. That is, we are going to hold back some data that the algorithms will not get to see and we will use this data to get a second and independent idea of how accurate the best model might actually be. We will split the loaded dataset into two, 80% of which we will use to train our models and 20% that we will hold back as a validation dataset.

```
# Split-out validation dataset
array = dataset.values
X = array[:,0:4]
Y = array[:,4]
validation_size = 0.20
seed = 7
X_train, X_validation, Y_train, Y_validation = ↙
    cross_validation.train_test_split(X, Y, test_size=validation_size, ↙
    random_state=seed)
```

You now have training data in the X_train and Y_train for preparing models and a X_validation and Y_validation sets that we can use later.

Prepare Cross Validation metrics

We will use 10-fold cross validation to estimate accuracy. With this method we divide randomly the dataset into 10 parts. We use 9 of those parts for training and reserve one tenth for testing. We repeat this procedure 10 times each time reserving a different tenth for testing.

```
# Test options and evaluation metric
seed = 7
scoring = 'accuracy'
```

We are using the metric of ‘accuracy‘ to evaluate models. This is a ratio of the number of correctly predicted instances in divided by the total number of instances in the dataset multiplied by 100 to give a percentage (e.g. 95% accurate). We will be using the scoring variable when we run build and evaluate each model next.

Build Models

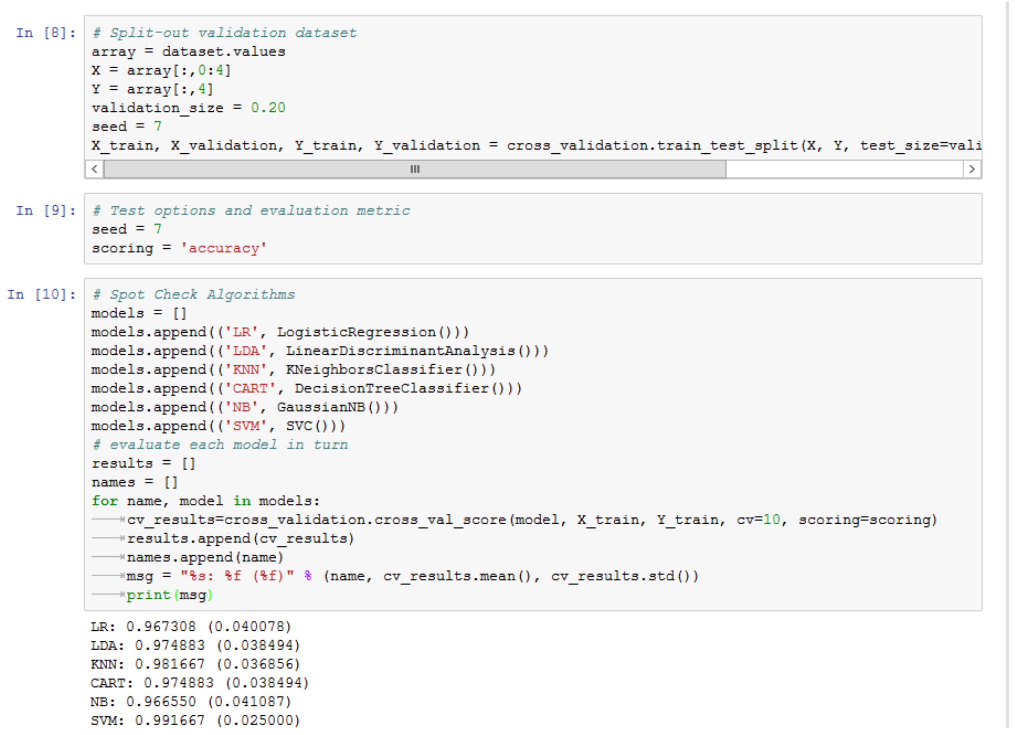
We don’t know which algorithms would be good on this problem or what configurations to use. We get an idea from the plots that some of the classes are partially linearly separable in some dimensions, so we are expecting generally good results.

Let’s evaluate 6 different algorithms:

- Logistic Regression (LR)
- Linear Discriminant Analysis (LDA)
- K-Nearest Neighbors (KNN)
- Classification and Regression Trees (CART)
- Gaussian Naive Bayes (NB)
- Support Vector Machines (SVM)

This is a good mixture of simple linear (LR and LDA), nonlinear (KNN, CART, NB and SVM) algorithms. We reset the random number seed before each run to ensure that the evaluation of each algorithm is performed using exactly the same data splits. It ensures the results are directly comparable. Let's build and evaluate our five models:

```
# Spot Check Algorithms
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
# evaluate each model in turn
results = []
names = []
for name, model in models:
   cv_results=cross_validation.cross_val_score(model, X_train, Y_train, cv=10, ↙
       scoring=scoring)
   results.append(cv_results)
   names.append(name)
   msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
   print(msg)
```



```
In [8]:  # Split-out validation dataset
          array = dataset.values
          X = array[:,0:4]
          Y = array[:,4]
          validation_size = 0.20
          seed = 7
          X_train, X_validation, Y_train, Y_validation = cross_validation.train_test_split(X, Y, test_size=vali

In [9]:  # Test options and evaluation metric
          seed = 7
          scoring = 'accuracy'

In [10]: # Spot Check Algorithms
          models = []
          models.append(('LR', LogisticRegression()))
          models.append(('LDA', LinearDiscriminantAnalysis()))
          models.append(('KNN', KNeighborsClassifier()))
          models.append(('CART', DecisionTreeClassifier()))
          models.append(('NB', GaussianNB()))
          models.append(('SVM', SVC()))
          # evaluate each model in turn
          results = []
          names = []
          for name, model in models:
              cv_results=cross_validation.cross_val_score(model, X_train, Y_train, cv=10, scoring=scoring)
              results.append(cv_results)
              names.append(name)
              msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
              print(msg)

          LR: 0.967308 (0.040078)
          LDA: 0.974883 (0.038494)
          KNN: 0.981667 (0.036856)
          CART: 0.974883 (0.038494)
          NB: 0.966550 (0.041087)
          SVM: 0.991667 (0.025000)
```

Select Best Model

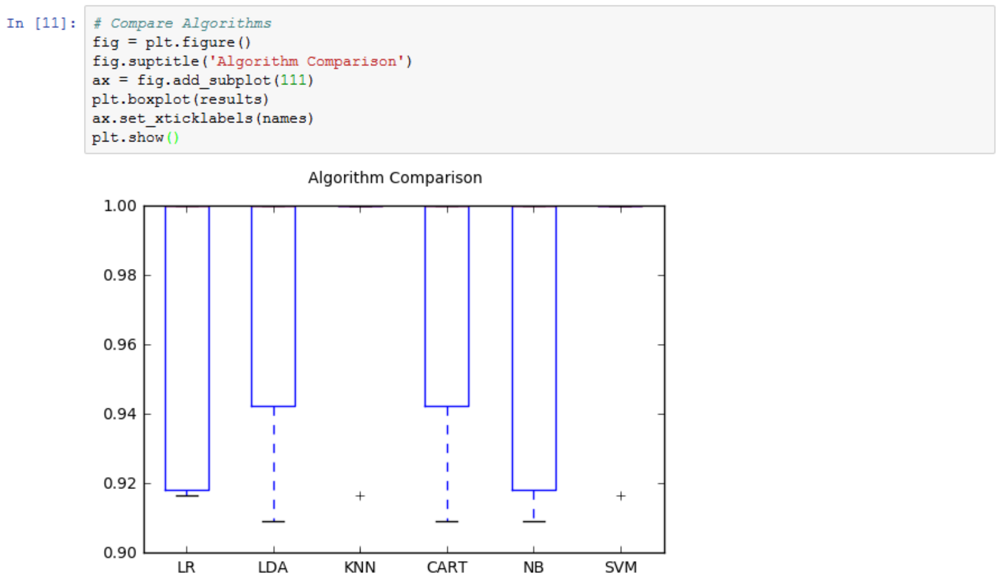
We now have 6 models and accuracy estimations for each. We need to compare the models to each other and select the most accurate. Running the example above, we get the following raw results:

LR: 0.966667 (0.040825)
LDA: 0.975000 (0.038188)

```
KNN: 0.983333 (0.033333)
CART: 0.975000 (0.038188)
NB: 0.975000 (0.053359)
SVM: 0.981667 (0.025000)
```

We can see that it looks like KNN has the largest estimated accuracy score. We can also create a plot of the model evaluation results and compare the spread and the mean accuracy of each model. There is a population of accuracy measures for each algorithm because each algorithm was evaluated 10 times (10 fold cross validation).

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```



```
In [11]: # Compare Algorithms
         fig = plt.figure()
         fig.suptitle('Algorithm Comparison')
         ax = fig.add_subplot(111)
         plt.boxplot(results)
         ax.set_xticklabels(names)
         plt.show()
```

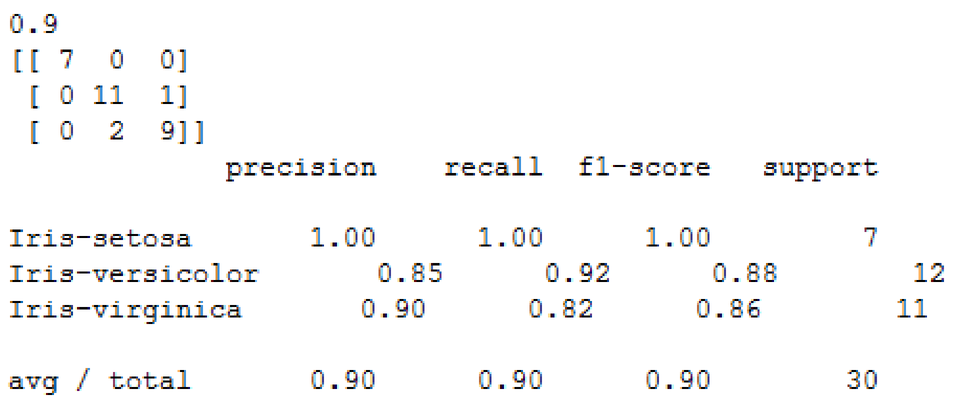
You can see that the box and whisker plots are squashed at the top of the range, with many samples achieving 100% accuracy.

4.2.6 Make Predictions

The KNN algorithm was the most accurate model that we tested. Now we want to get an idea of the accuracy of the model on our validation set. This will give us an independent final check on the accuracy of the best model. It is valuable to keep a validation set just in case you made a slip during training, such as overfitting to the training set or a data leak. Both will result in an overly optimistic result. We can run the KNN model directly on the validation set and summarize the results as a final accuracy score, a confusion matrix and a classification report.

```
# Make predictions on validation dataset
knn = KNeighborsClassifier()
knn.fit(X_train, Y_train)
predictions = knn.predict(X_validation)
```

```python
print(accuracy_score(Y_validation, predictions))
print(confusion_matrix(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

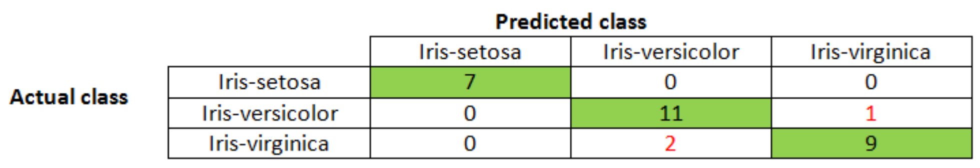


```
0.9
[[ 7  0  0]
 [ 0 11  1]
 [ 0  2  9]]
             precision    recall  f1-score   support

Iris-setosa       1.00      1.00      1.00         7
Iris-versicolor       0.85      0.92      0.88        12
Iris-virginica       0.90      0.82      0.86        11

avg / total       0.90      0.90      0.90        30
```

We can see that the accuracy is 0.9 or 90% that is the number of tests correctly classified divided per the total number of test cases.

The confusion matrix provides an indication of the three errors made. The rows of the confusion matrix represent the actual class of the test cases, the columns represent what our classifier predicted.

You can see that there was 1 error where an Iris-versicolor was classified as an Iris-virginica, 2 cases where Iris-virginica was classified as an Iris-versicolor (a total of 3 errors). The accuracy is (7+11+9)/30= 0.9 =90%



		Predicted class		
		Iris-setosa	Iris-versicolor	Iris-virginica
Actual class	Iris-setosa	7	0	0
	Iris-versicolor	0	11	1
	Iris-virginica	0	2	9

Finally the classification report provides a breakdown of each class by precision, recall, f1-score and support showing excellent results (granted the validation dataset was small).

- The precision is the ratio tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

- The recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

- The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0. The F-beta score weights recall more than precision by a factor of beta. beta == 1.0 means recall and precision are equally important.

- The support is the number of occurrences of each class in y_true.

4.2.7 References

- Book - Han, J. and Kamber, M. (2000). Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, U
- Book - Principles of Data Mining - Undergraduate Topics in Computer Science. Max Bramer, Springer.
- Homepage on Internet - *Scikit Learn - Machine Learning in Python*
- Homepage on Internet - *UCI Machine Learning Repository - Iris Dataset*
- Homepage on Internet- *Machine Learning Mastery - Python Machine Learning*

4.3 Tutorial 2 - Clustering Customers using K-means in HANA

In this tutorial we will use the Predictive Analysis Library (PAL) of HANA and in particular the K-means algorithm, to cluster customers by similar behaviour.

Clustering can be considered the most important unsupervised learning problem and it is the process of grouping a set of physical or abstract objects into similar groups. A cluster is a collection of data objects that are similar to one another within the same cluster and are dissimilar to the objects in other clusters.

The K-means algorithm calculates the distance between all objects and the ones that are close to each other are grouped in the same cluster. Each cluster is associated with a centroid and each point is assigned to the cluster with the closest centroid. The centroid is the mean of the points in the cluster. The closeness can be measured using:

- Euclidean Distance (most used)
- Manhattan Distance
- Minkowski Distance

Every time a point is assigned to a cluster the centroid is recalculated. This is repeated in multiple iterations until centroids don't change anymore (meaning all points have been assigned to a corresponding cluster) or until relatively few points change clusters. Usually most of the centroid movement happens in the first iterations.

One of the main drawbacks of the K-means Algorithm is that you need to specify the number of Ks (or clusters) upfront as an input parameter. Knowing this value is usually very hard, that is why it is important to run quality measurement functions to check the quality of your clustering.

The dataset we use is available on the SAP HANA Academy Github Repository.

To access the code snippets used in the tutorial please visit the customers_tutorial folder on the *SDIL Git repository* and check out the *customers_tutorial.sql* SQLscript. Log in with your SDIL credentials. Please check the paragraph 2.4.5 of the *SDIL Platform Documentation* to check how to access and use the SDIL Git Repository.

4.3.1 Introduction

The Predictive Analysis Library (PAL) is a component of SAP HANA and it enables modelers to perform predictive analysis over big volumes of data. It includes classic predictive analysis algorithms such as: clustering, classification, regression, association etc.. which can be called from within SQLScript procedures.
In general this means writing a SQLScript to:

- Generate the PAL procedure
- Run the PAL procedure

It can be quite discouraging to use the PAL if the user is less experienced in SQLScript and/or with the mechanisms of the predictive method(s) used. For this reason, it is possible to use the Application Function Modeler (AFM), an extension of HANA Studio which supports functions from PAL in flowgraph models. With the AFM it is very easy to add PAL function components to your flowgraph, configure its parameters and input/output tables and generate the stored procedure, all without writing lines of SQLScript code.

Please check from the SAP Help Portal the topic "Using PAL in SAP HANA AFM".

The next paragraph describes how to run the K-means algorithm with the CUSTOMERS dataset using the Application Function Modeler (AFM). Anyway, if you are interested to perform the same algorithm writing an SQLScript, please check the paragraph 5.3.5 of this section.

For more information about PAL and for a complete list of all available algorithms please check the official documentation: **SAP HANA Predictive Analysis Library (PAL)**.
You can find a lot of tutorials about the usage of PAL on the **SAP HANA Academy YouTube channel**.

4.3.2 Start your HANA instance on the SDIL Platform

We suppose you have already an HANA user and you added your HANA System in SAP HANA Studio. For more information see chapter 4 on the SDIL Platform Documentation.

Please follow the steps below to start your HANA instance on the SDIL Platform:

1. Login to the Windows Login Server (see section 2.3.2 on SDIL Platform Documentation)

2. Launch the SAP HANA Studio by clicking the icon on the desktop

3. Select your System

4. Start the SQL Console. From the bar choose the SQL button

4.3.3 Prepare the data

The first step is create the customers table and fill it with data from the csv file.

On the SQL Console execute the following statement to create the table structure:

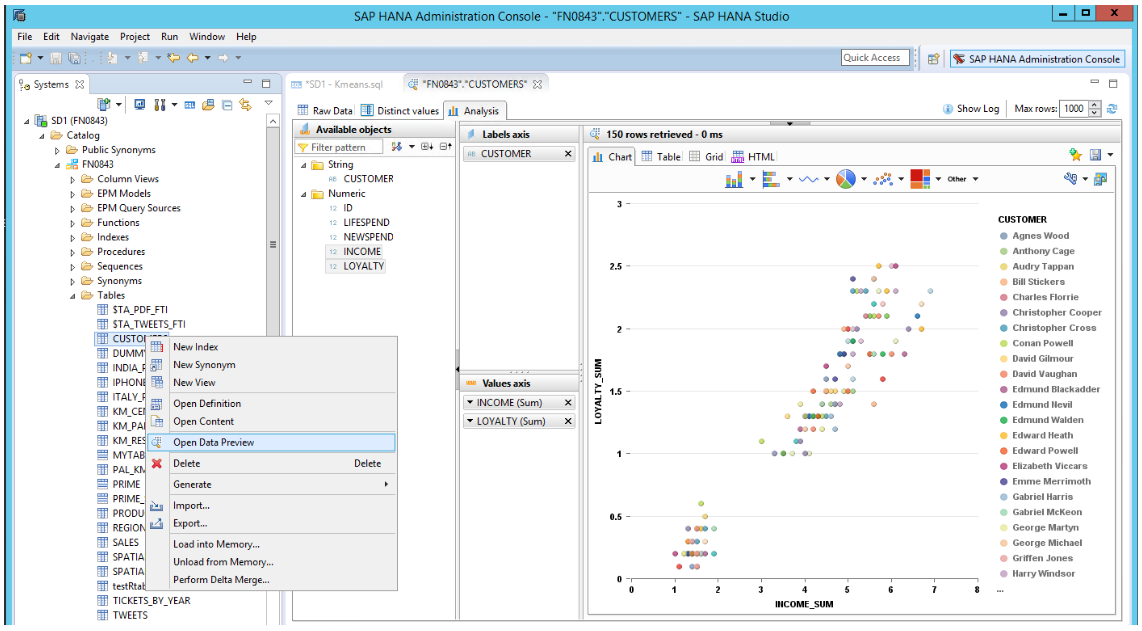
```
CREATE COLUMN TABLE "CUSTOMERS" (
    "ID" INTEGER NOT NULL,
    "CUSTOMER" NVARCHAR(60),
    "LIFESPEND" DOUBLE,
    "NEWSPEND" DOUBLE,
    "INCOME" DOUBLE,
    "LOYALTY" DOUBLE,
    PRIMARY KEY ("ID")
```

Now you need first to import the CUSTOMERS.csv file to HANA on the SDIL Platform. Please check the SDIL Platform Documentation paragraph 4.4 Importing data into the HANA database.

Run the import sql statement:

```
IMPORT FROM CSV FILE
'/gpfs/hana_dataimport/<your output import path>/CUSTOMERS.csv' into CUSTOMERS
```

We can already do a quick analysis. For each customer we look at the income and the loyalty factor. Right click on the CUSTOMERS table and choose "Open Data Preview". Click Analysis on the tool bar and add to the Labels axis the CUSTOMER and to the Values Axis the LOYALTY and INCOME. Change the Chart type and click the button "Scatter charts".



We can see we have already two natural grouping of customers.

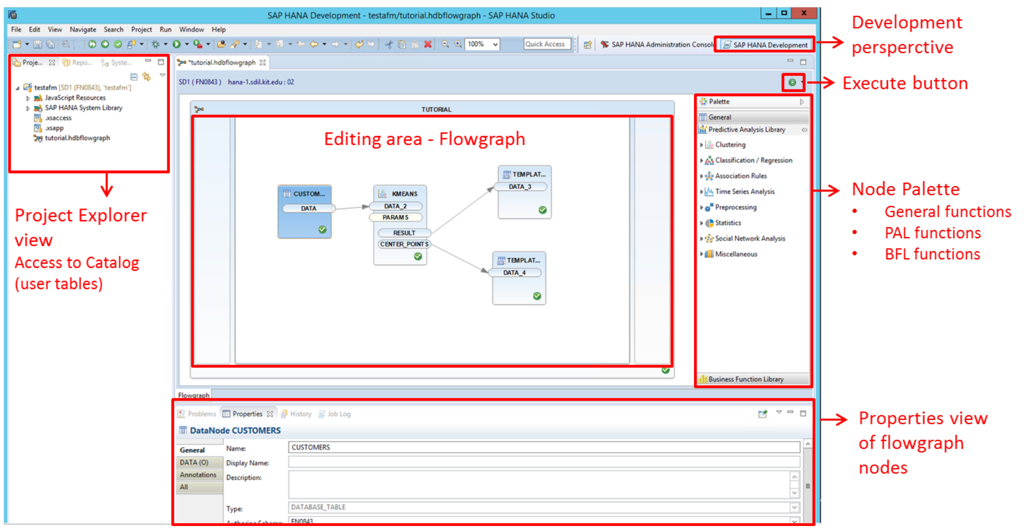
The following two paragraphs explain two possible and different ways to perform the K-means algorithm of PAL in HANA:

1. Method 1: use the SAP HANA Application Function Modeler (AFM) tool to model a flowgraph

2. Method 2: write a SQLScript to create and call the PAL procedure. (A basic knowledge of SQLScript is required)

4.3.4 Method 1 - Perform the K-means with SAP HANA Application Function Modeler (AFM)

This section leads you through the most common steps of using the HANA Application Function Modeler (AFM).
The HANA AFM supports standard graphical editing operations like move, copy, paste, and delete on the elements of a flowgraph. Detailed properties of these elements are edited in the Properties view.
The following image gives you an overview of the AFM in HANA Studio.



Here are all the steps needed to perform the K-means algorithm on our customers dataset.

1. Before you can start working with the flowgraph, you must create an XS project. If you have already an XS project, open it. Otherwise you need to create a new one. Here the steps to create a new project:

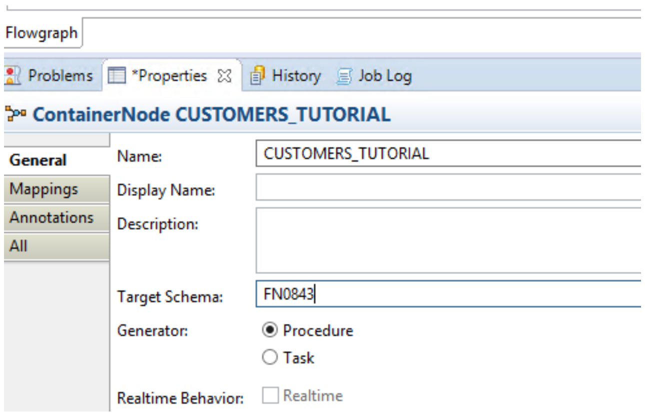
   - Open the SAP HANA Development perspective.
   - Choose the Project Explorer view.
   - Choose File > New > XS Project from the menu bar.
   - Enter the required details for the new XS project (e.g.: project name, repository workspace).

2. Create a new flowgraph or open an existing flowgraph in the Project Explorer view.

   - Right click on the Project in the Project Explorer view, and choose: New > Other > SAP HANA > Database Development > Flowgraph Model.
   - Enter or select the parent folder, and enter the file name.
   - Choose Finish.
   - The new .hdbflowgraph file will appear in the Project Explorer view.

3. Double click on the .hdbflowgraph file. The flowgraph is opened in the Editing Area of the AFM.

4. Specify the target schema by selecting the flowgraph container and editing Target Schema in the Properties view.


Flowgraph

Problems | *Properties | History | Job Log

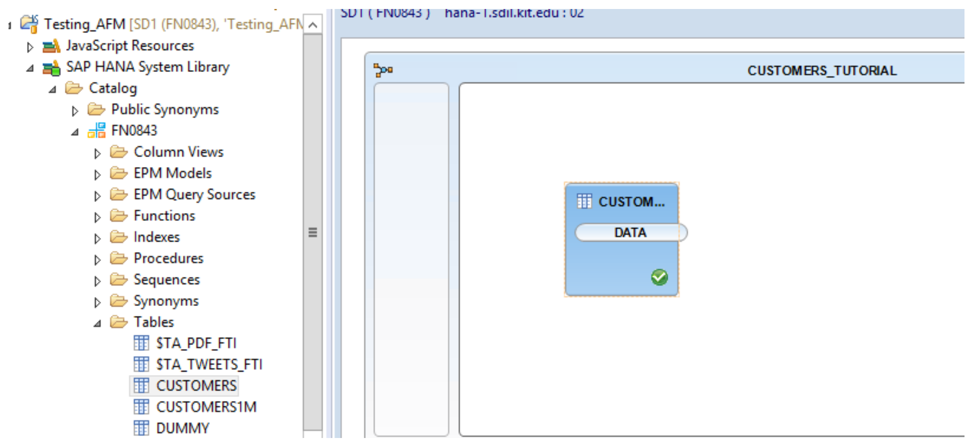
ContainerNode CUSTOMERS_TUTORIAL

General | Mappings | Annotations | All

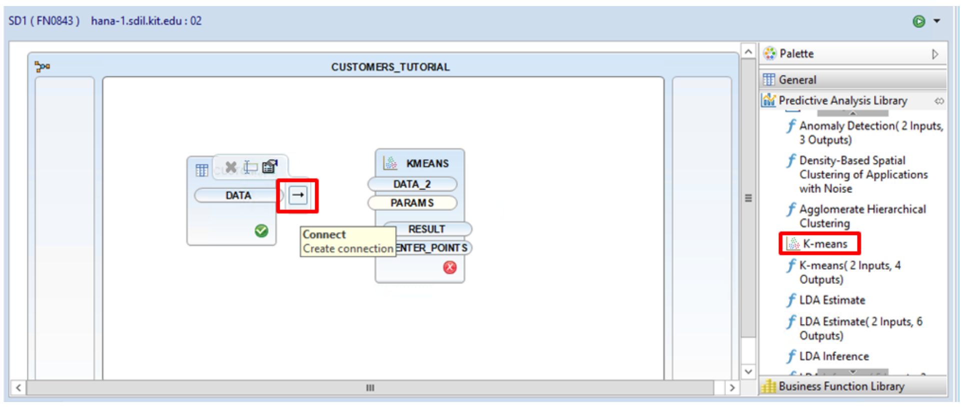
Name: CUSTOMERS_TUTORIAL
Display Name:
Description:
Target Schema: FN0843
Generator: [✓ Procedure] Task
Realtime Behavior: Realtime

5. Add the CUSTOMER table as input for the flowgraph by doing the following:

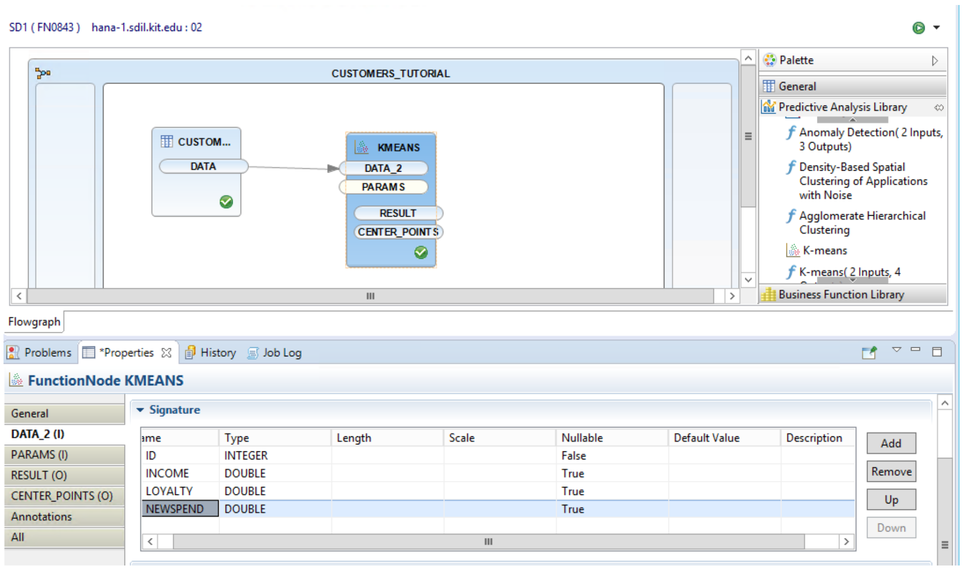
   - Navigate in the Project View to SAP HANA System Library > "your schema" > Tables.
   - Drag and drop the CUSTOMER table on the editing area of the flowgraph.



6. Drag the function node from the Predictive Analysis Library compartment of the Palette to the flowgraph editing area. Choose K-means under Predictive Analysis Library > Clustering > K-means.
   Connect the CUSTOMERS input table to the K-means PAL function node.


SD1 (FN0843) hana-1.sdil.kit.edu : 02
CUSTOMERS_TUTORIAL
DATA
Connect
Create connection
KMEANS
DATA_2
PARAMS
RESULT
CENTER_POINTS
Palette
General
Predictive Analysis Library
Anomaly Detection(2 Inputs, 3 Outputs)
Density-Based Spatial Clustering of Applications with Noise
Agglomerate Hierarchical Clustering
K-means
K-means(2 Inputs, 4 Outputs)
LDA Estimate
LDA Estimate(2 Inputs, 6 Outputs)
LDA Inference
Business Function Library

7. Select the K-means function node and check the input data in the Properties view. Remove the two features: LIFESPEND and NEWSPEND in order to do the clustering only in a bi-dimensional space.

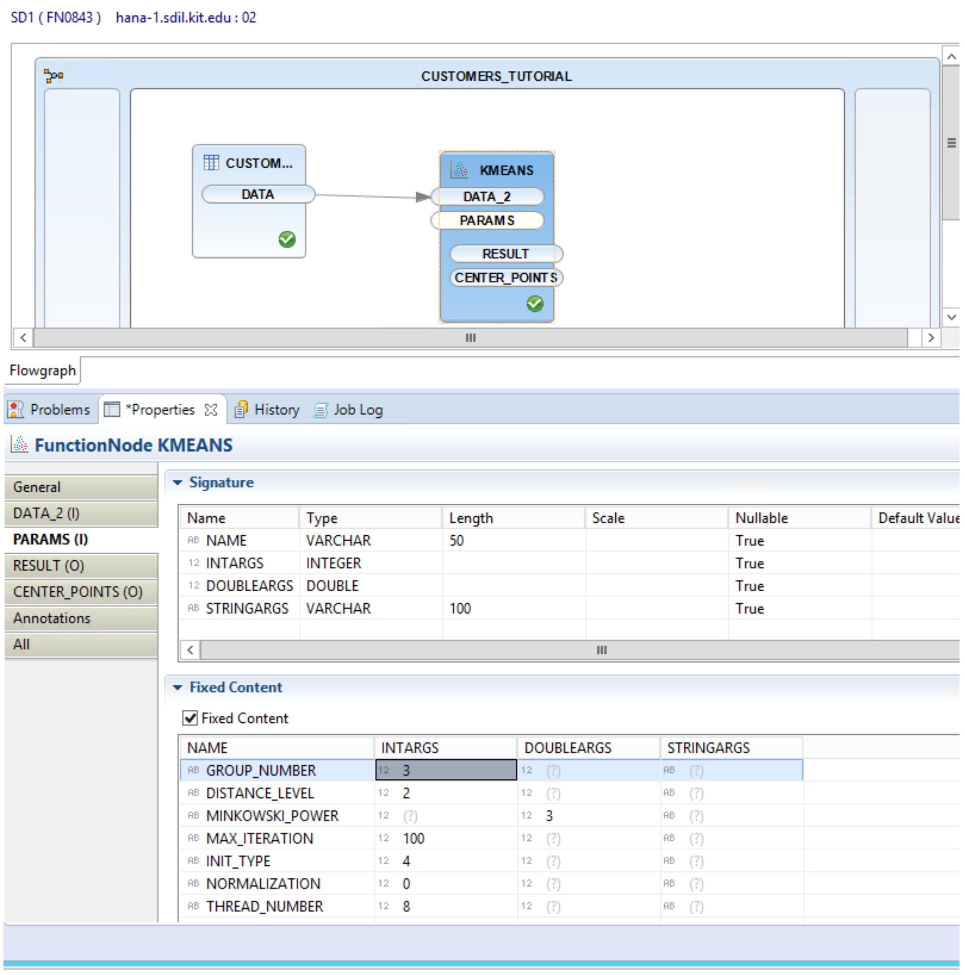


FunctionNode KMEANS

Signature

ame	Type	Length	Scale	Nullable	Default Value	Description
ID	INTEGER			False		
INCOME	DOUBLE			True		
LOYALTY	DOUBLE			True		
NEWSPEND	DOUBLE			True		

8. Specify the K-means parameters by selecting the parameter in the Properties view. Change the GROUP_NUMBER to 3.



FunctionNode KMEANS

Signature

Name	Type	Length	Scale	Nullable	Default Value
NAME	VARCHAR	50		True	
INTARGS	INTEGER			True	
DOUBLEARGS	DOUBLE			True	
STRINGARGS	VARCHAR	100		True	

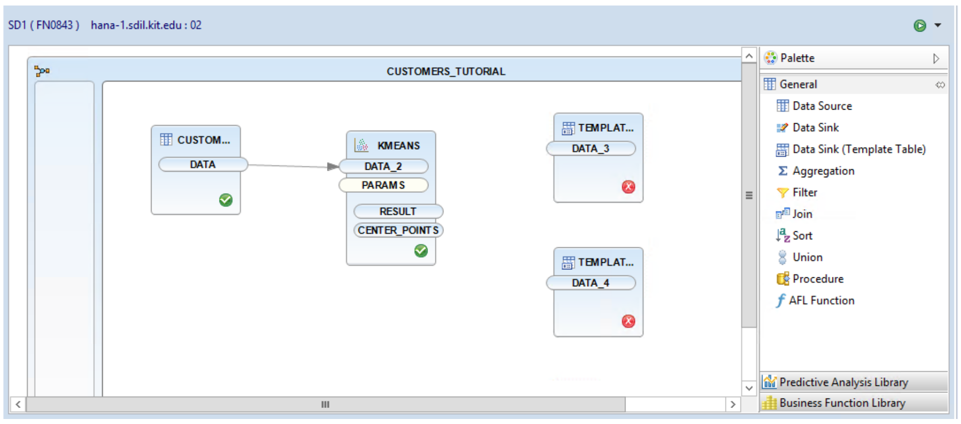
Fixed Content

☑ Fixed Content

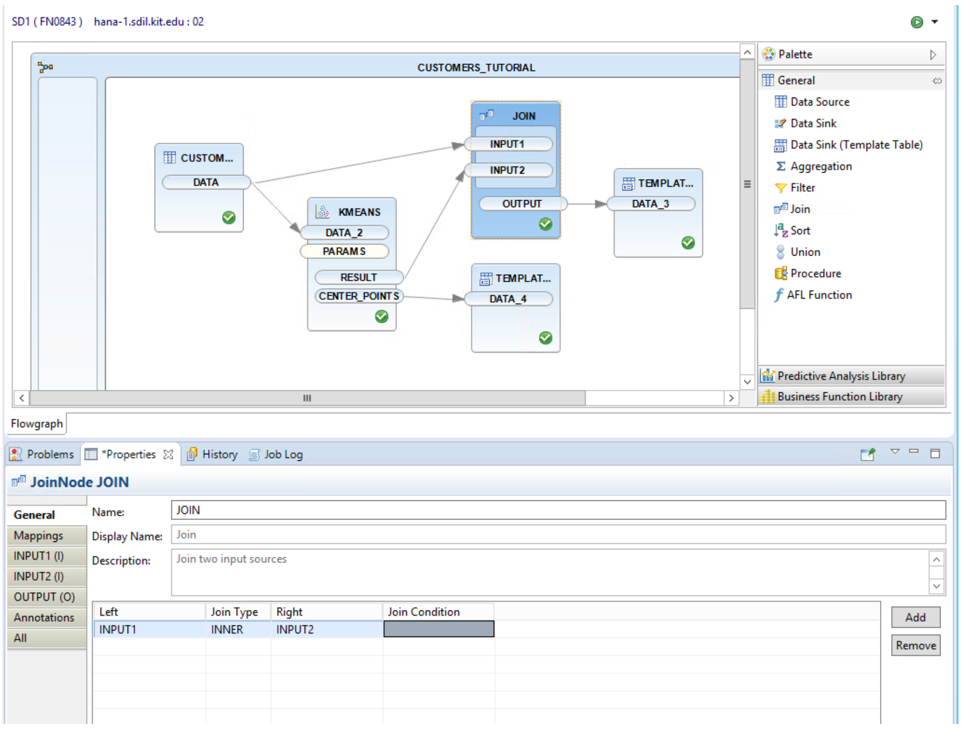
NAME	INTARGS	DOUBLEARGS	STRINGARGS
GROUP_NUMBER	3	(?)	(?)
DISTANCE_LEVEL	2	(?)	(?)
MINKOWSKI_POWER	(?)	3	(?)
MAX_ITERATION	100	(?)	(?)
INIT_TYPE	4	(?)	(?)
NORMALIZATION	0	(?)	(?)
THREAD_NUMBER	8	(?)	(?)

9. The K-means PAL functions has two output tables (RESULT and CENTER_POINTS). You can check the properties on the official page: HANA PAL K-means.

From the Palette view on the right panel choose General > Data Sink (Template Table) and drag and drop this node to the editing area. Similarly add a second one.



10. Connect the CENTER_POINTS output of K-means with the second template table. Before to connect the RESULT table to the output table, add a Join node from the Palette view. We are going to join the customer table with the result table of the K-means in order to have not only the list of cluster id and set its definitions:

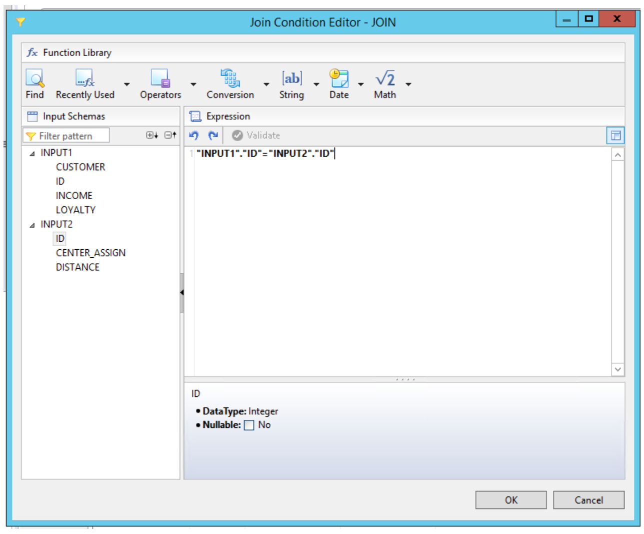


JoinNode JOIN

Name: JOIN
Display Name: Join
Description: Join two input sources

Left	Join Type	Right	Join Condition
INPUT1	INNER	INPUT2	

11. Set the join condition from the General view of the Join node. Click add, specify the left and right tables, and the join condition with the following expression: "INPUT1"."ID"="INPUT2"."ID"



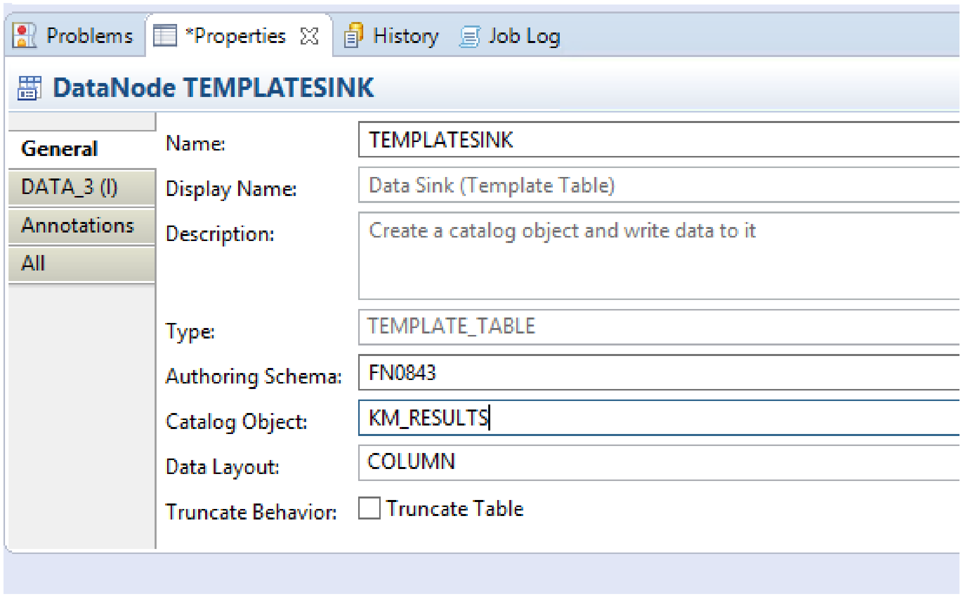
INPUT1
- CUSTOMER
- ID
- INCOME
- LOYALTY

INPUT2
- ID
- CENTER_ASSIGN
- DISTANCE

```
"INPUT1"."ID"="INPUT2"."ID"
```

ID
- **DataType:** Integer
- **Nullable:** ☐ No

12. Set in the two output tables the Authoring Schema (your Schema) and the Catalog Object name (the name of the output table).



DataNode TEMPLATESINK

Name:	TEMPLATESINK
Display Name:	Data Sink (Template Table)
Description:	Create a catalog object and write data to it
Type:	TEMPLATE_TABLE
Authoring Schema:	FN0843
Catalog Object:	KM_RESULTS
Data Layout:	COLUMN
Truncate Behavior:	☐ Truncate Table

13. Save the flowgraph. Select File Save in the HANA Studio main menu.

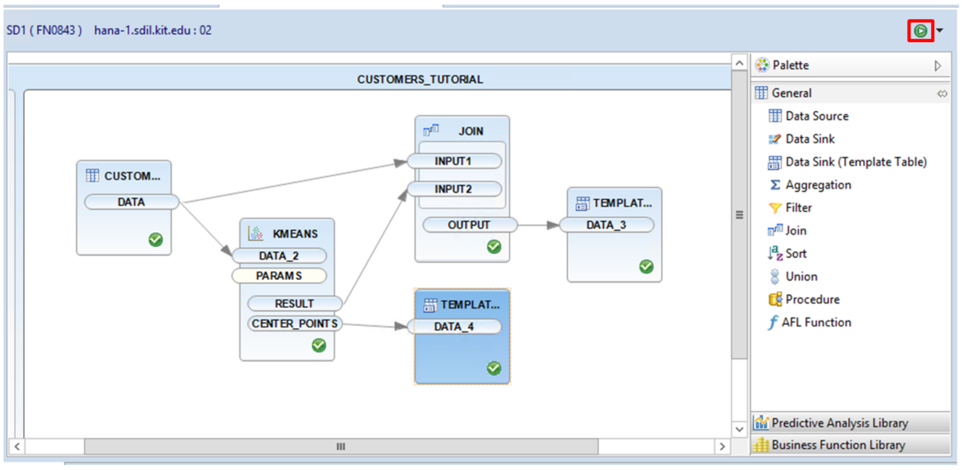
14. Activate the flowgraph. First of all it's necessary to execute following SQL to grant authorization to _SYS_REPO user:

    ```
    GRANT SELECT ON SCHEMA "your schema name" TO _SYS_REPO WITH GRANT OPTION;
    ```

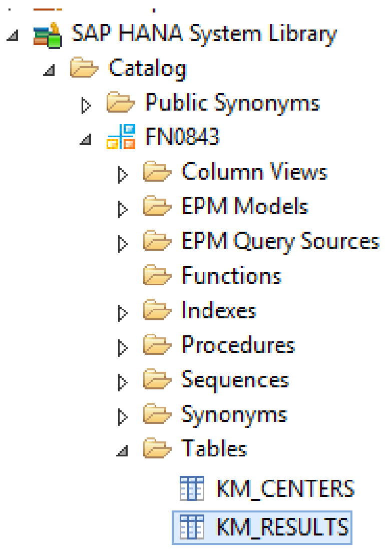
    This user is responsible for activation and for creating run-time objects.

    Now activate the flowgraph: in the Project Explorer view, right-click the flowgraph object and choose Team Activate... from the context-sensitive menu. A new procedure is generated in the target schema which is specified in the properties of the flowgraph container.

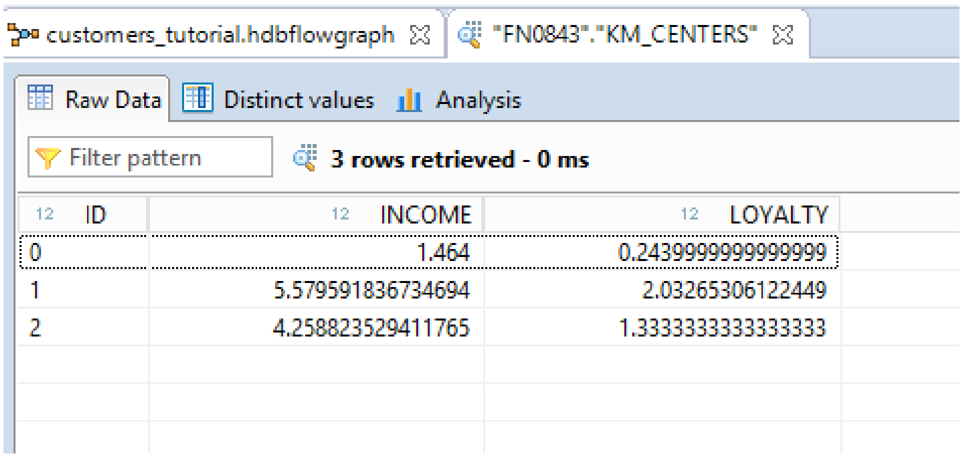
15. Click on the Execute button in the top right corner of the AFM.



16. If you now go to the list of tables you will see the output tables KM_RESULTS, KM_CENTERS. Let's have a look first to the KM_CENTERS table. Right click on the table and choose "Open Data preview".


SAP HANA System Library
- Catalog
  - Public Synonyms
  - FN0843
    - Column Views
    - EPM Models
    - EPM Query Sources
    - Functions
    - Indexes
    - Procedures
    - Sequences
    - Synonyms
    - Tables
      - KM_CENTERS
      - KM_RESULTS

If you now refresh the list of tables you will see the output tables KM_RESULTS, KM_CENTERS. Let's have a look first to the KM_CENTERS table. Right click on the table and choose "Open Data preview".


customers_tutorial.hdbflowgraph "FN0843"."KM_CENTERS"

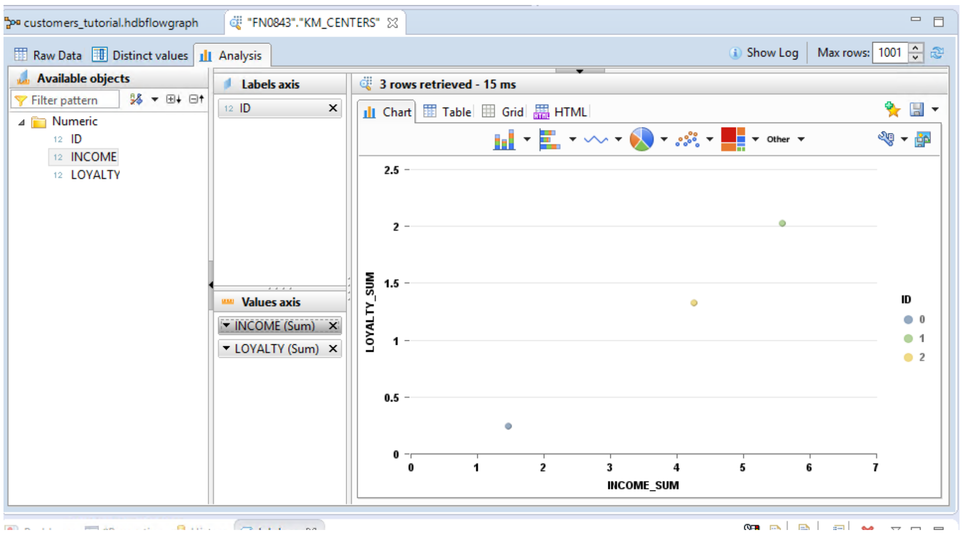
Raw Data Distinct values Analysis

Filter pattern 3 rows retrieved - 0 ms

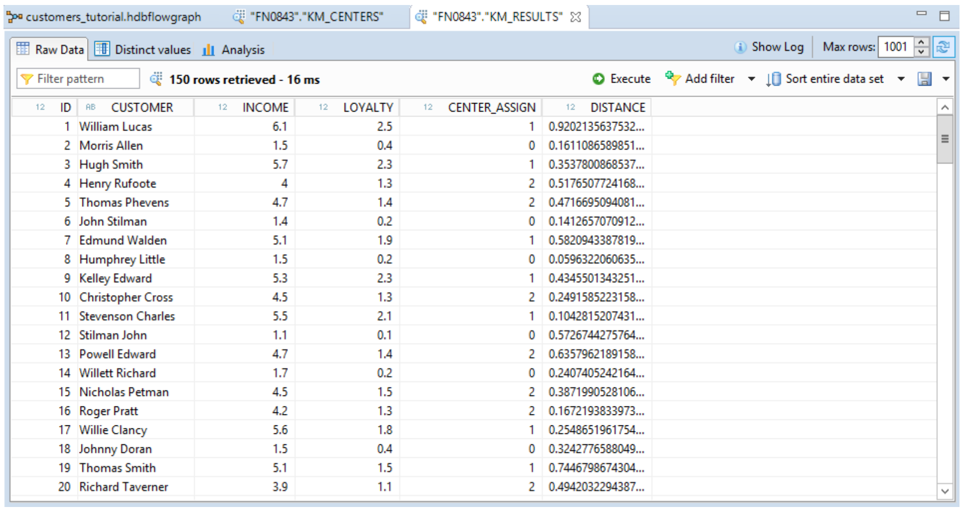
ID	INCOME	LOYALTY
0	1.464	0.243999999999999
1	5.579591836734694	2.0326530612244
2	4.258823529411765	1.3333333333333333

We can see we have 3 centers and we see for each of the four input variables where is the center.

Let's show the Scatter chart for Income and Loyalty. Click Analysis on the tool and add to the Labels axis the CENTER_ID and to the Values Axis the LOYALTY and INCOME. Choose "Scatter chart". We see where the K-means algorithm identifies the centers of the 3 clusters.



Now we look at the KM_RESULTS table. Right click on the table and choose "Open Data preview".

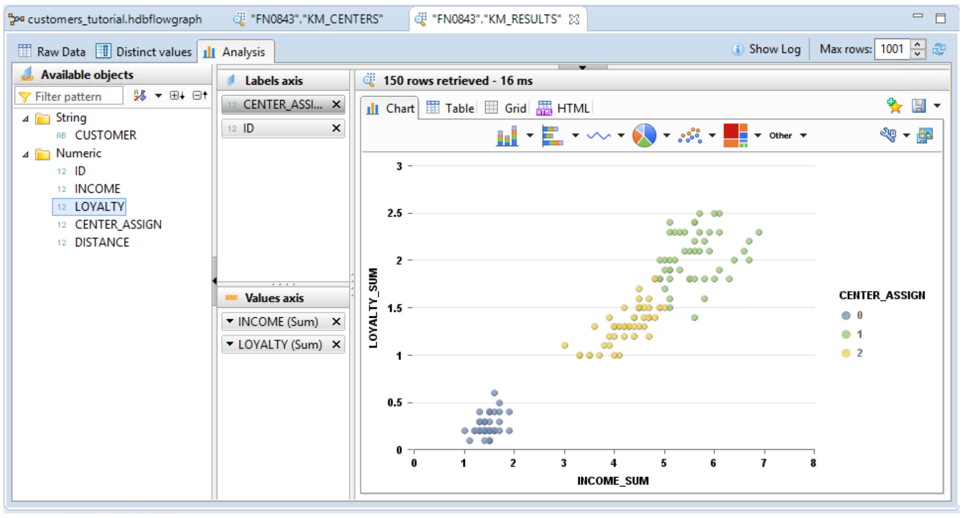


150 rows retrieved - 16 ms

ID	CUSTOMER	INCOME	LOYALTY	CENTER_ASSIGN	DISTANCE
1	William Lucas	6.1	2.5	1	0.9202135637532...
2	Morris Allen	1.5	0.4	0	0.1611086589851...
3	Hugh Smith	5.7	2.3	1	0.3537800868537...
4	Henry Rufoote	4	1.3	2	0.5176507724168...
5	Thomas Phevens	4.7	1.4	2	0.4716695094081...
6	John Stilman	1.4	0.2	0	0.1412657070912...
7	Edmund Walden	5.1	1.9	1	0.5820943387819...
8	Humphrey Little	1.5	0.2	0	0.0596322060635...
9	Kelley Edward	5.3	2.3	1	0.4345501343251...
10	Christopher Cross	4.5	1.3	2	0.2491585223158...
11	Stevenson Charles	5.5	2.1	1	0.1042815207431...
12	Stilman John	1.1	0.1	0	0.5726744275764...
13	Powell Edward	4.7	1.4	2	0.6357962189158...
14	Willett Richard	1.7	0.2	0	0.2407405242164...
15	Nicholas Petman	4.5	1.5	2	0.3871990528106...
16	Roger Pratt	4.2	1.3	2	0.1672193833973...
17	Willie Clancy	5.6	1.8	1	0.2548651961754...
18	Johnny Doran	1.5	0.4	0	0.3242776588049...
19	Thomas Smith	5.1	1.5	1	0.7446798674304...
20	Richard Taverner	3.9	1.1	2	0.4942032294387...

We can see that for every customer has been assigned the cluster id and the distance that is how far is the customer from the center point.

Now we look at the result View table- Open the KM_RESULTS. Click Analysis on the tool bar and add to the Labels axis first the CLUSTER_NUMBER and as second CUSTOMERS to the Values Axis the INCOME and LOYALTY. We see where the K-means algorithm identifies the 3 clusters.



4.3.5 Method 2 - Perform the K-means from within SQLScript code

This section shows you an alternative way to use PAL functions through the creation and execution of the procedure within a SQLScript.

Prerequisite

Your HANA user must be assigned the following roles and privileges to execute the functions in the library:

```
--Granted Roles:
AFL__SYS_AFL_AFLPAL_EXECUTE
AFLPM_CREATOR_ERASER_EXECUTE

--Object Privileges:
AFL_WRAPPER_ERASER
AFL_WRAPPER_GENERATOR
AFLPM_CREATOR
```

Get in contact with the SDIL support team (see section 2.1 of the *SDIL Platform Documentation*) if you are not sure about your HANA roles and privileges.

Generate the PAL Procedure

The first thing we need to do is generate the PAL procedure by calling the AFL Wrapper Generator. To do so we need to create a number of Table Types (they are HANA table without records) that will be used to define the structure of the data used as input and output parameters.

Let‘s use only two of the four attributes available in the dataset to have a better visualization in a two-dimensional space. We are going to use the INCOME and LOYALTY attributes.

The AFL_WRAPPER_GENERATOR('<procedure_name>', '<area_name>', '<function_name>', <signature_table>) function takes as inputs:

- <procedure_name>: A name for the PAL procedure. This can be anything you want.
- <area_name>: Always set to AFLPAL.
- <function_name>: A PAL built-in function name. In our case K-means.
- <signature_table>: A user-defined table variable. The table contains records to describe the input table type, parameter table type, and result table type. The records of this table change according to the specified <function_name>.

```
/* Table Type that will be used as the input parameter
that will contain the data that we would like to cluster */
CREATE TYPE PAL_T_KM_DATA AS TABLE (ID INTEGER, INCOME DOUBLE, LOYALTY DOUBLE);

/* Table Type that will be used to specify
the different parameters to run the KMeans algorithm */
CREATE TYPE PAL_T_KM_PARAMS AS TABLE (NAME VARCHAR(60), INTARGS INTEGER, ↙
    DOUBLEARGS DOUBLE, STRINGARGS VARCHAR (100));

/* Table Type that will be used as the output parameter
that will contain which cluster has been assigned to each
customer and what is the distance to the mean of the cluster */
CREATE TYPE PAL_T_KM_RESULTS AS TABLE (ID INTEGER, CENTER_ID INTEGER, DISTANCE ↙
    DOUBLE);

/* Table Type that will be used as the output parameter
that will contain the centers for each cluster */
CREATE TYPE PAL_T_KM_CENTERS AS TABLE (CENTER_ID INTEGER, INCOME DOUBLE, ↙
    LOYALTY DOUBLE);


/* This table is used to generate the KMeans procedure
and will point the AFL Wrapper Generator to the different
table types that we just created */
CREATE COLUMN TABLE PAL_KM_SIGNATURE (ID INTEGER, TYPENAME VARCHAR(100), ↙
    DIRECTION VARCHAR(100));
INSERT INTO PAL_KM_SIGNATURE VALUES (1, '<your schema name>.PAL_T_KM_DATA', ↙
    'in');
INSERT INTO PAL_KM_SIGNATURE VALUES (2, '<your schema name>.PAL_T_KM_PARAMS', ↙
    'in');
INSERT INTO PAL_KM_SIGNATURE VALUES (3, '<your schema name>.PAL_T_KM_RESULTS', ↙
    'out');
INSERT INTO PAL_KM_SIGNATURE VALUES (4, '<your schema name>.PAL_T_KM_CENTERS', ↙
    'out');

/* Creates the KMeans procedure that executes the KMeans Algorithm */
CALL SYSTEM.AFL_WRAPPER_GENERATOR ('PAL_KM', 'AFLPAL', 'KMEANS', ↙
    PAL_KM_SIGNATURE);
```

Run the above code to create the procedure.

Run the K-means Procedure

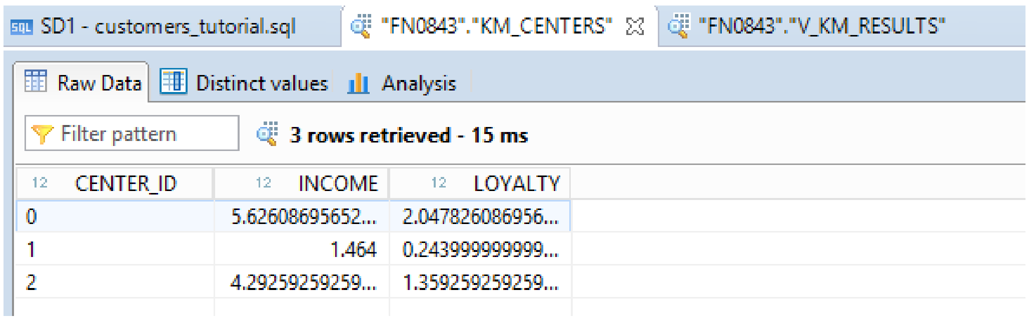
We have generated the K-means procedure so now we need the code that will execute it. We will set the number of cluster to 3 and as distance function the Euclidean distance.

```
-- app setup
CREATE VIEW V_KM_DATA AS
   SELECT ID, INCOME, LOYALTY
     FROM CUSTOMERS;

/* This table will contain the parameters that will be used
during the execution of the KMeans procedure. */
CREATE COLUMN TABLE KM_PARAMS LIKE PAL_T_KM_PARAMS;

/* Fill the parameters table */
```

```
--Number of threads to be used during the execution
INSERT INTO KM_PARAMS VALUES ('THREAD_NUMBER', 2, null, null);
--Number of clusters
INSERT INTO KM_PARAMS VALUES ('GROUP_NUMBER', 3, null, null);
--This parameter will specify how to select the initial center of each cluster
INSERT INTO KM_PARAMS VALUES ('INIT_TYPE', 1, null, null);
--Which distance to use. In this case Im using Euclidean
INSERT INTO KM_PARAMS VALUES ('DISTANCE_LEVEL', 2, null, null);
--   > Maximum Iterations
INSERT INTO KM_PARAMS VALUES ('MAX_ITERATION', 100, null, null);
-- How to normalize the data. In this case Im not normalizing at all
INSERT INTO KM_PARAMS VALUES ('NORMALIZATION', 0, null, null);
-- Threshold value for exiting an iteration
INSERT INTO KM_PARAMS VALUES ('EXIT_THRESHOLD', null, 0.0001, null);

/* This table will contain the resulting cluster
assignment for each customer and the distance to the center of the cluster */
CREATE COLUMN TABLE KM_RESULTS LIKE PAL_T_KM_RESULTS;

/* This table will contain the resulting centers for each cluster */
CREATE COLUMN TABLE KM_CENTERS LIKE PAL_T_KM_CENTERS;

/* create some views to analyze later the results */
CREATE VIEW V_KM_RESULTS AS
   SELECT a.ID, b.CUSTOMER, b.INCOME, b.LOYALTY, a.CENTER_ID + 1 AS ↙
       CLUSTER_NUMBER
     FROM KM_RESULTS a, CUSTOMERS b
     WHERE a.ID = b.ID;
```

```
/* Execute the K-means procedure */
CALL _SYS_AFL.PAL_KM (V_KM_DATA, KM_PARAMS, KM_RESULTS, KM_CENTERS) WITH ↙
    OVERVIEW;
```

If you now refresh the list of tables you will see the output tables KM_RESULTS, KM_CENTERS. Let's have a look first to the KM_CENTERS table. Right click on the table and choose "Open Data preview".

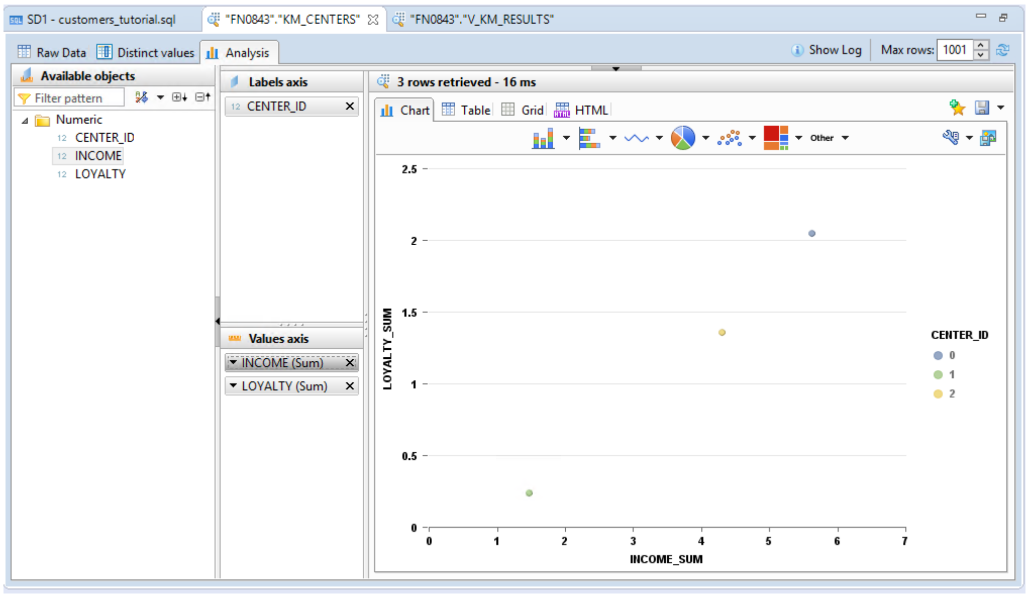


3 rows retrieved - 15 ms

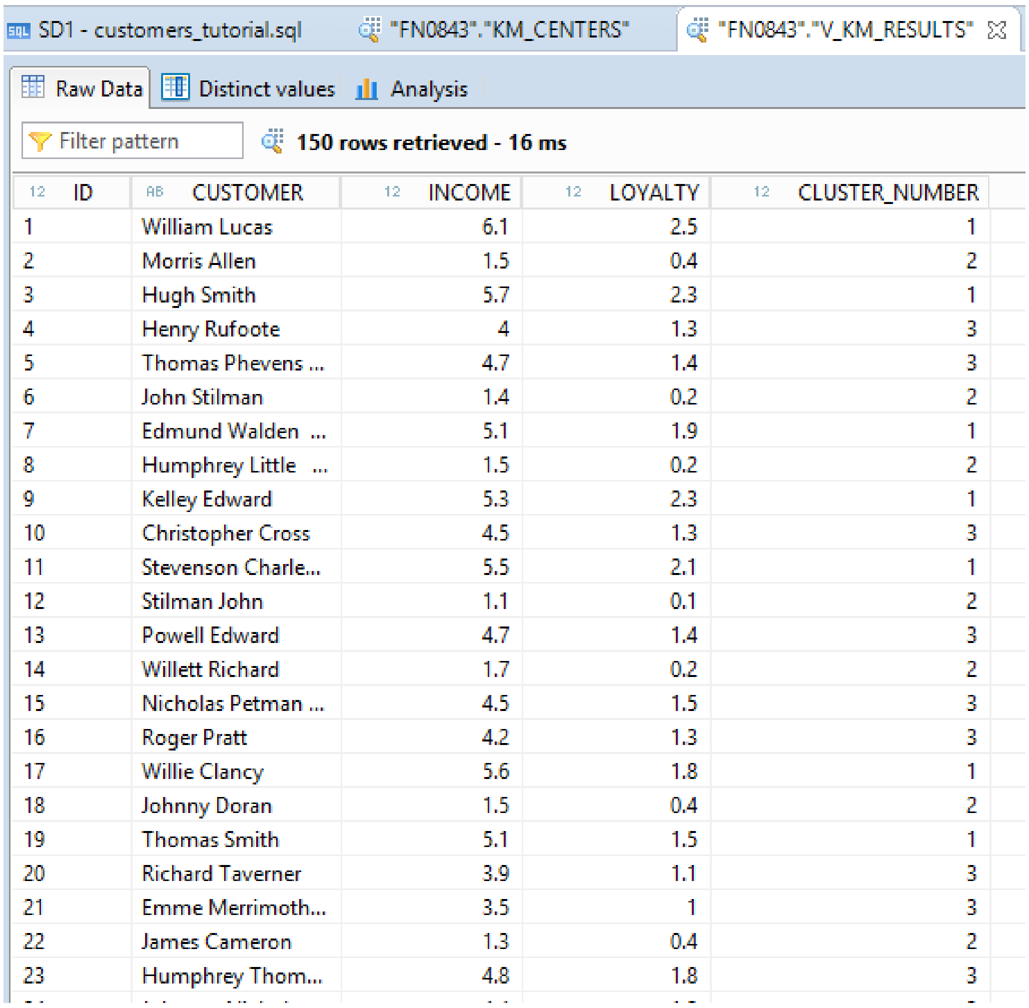
CENTER_ID	INCOME	LOYALTY
0	5.62608695652...	2.047826086956...
1	1.464	0.243999999999...
2	4.29259259259...	1.359259259259...

We can see we have 3 centers and we see for each of the four input variables where is the center.

Let's show the Scatter chart for Income and Loyalty. Click Analysis on the tool and add to the Labels axis the CENTER_ID and to the Values Axis the LOYALTY and INCOME. Choose "Scatter chart". We see where the K-means algorithm identifies the centers of the 3 clusters.



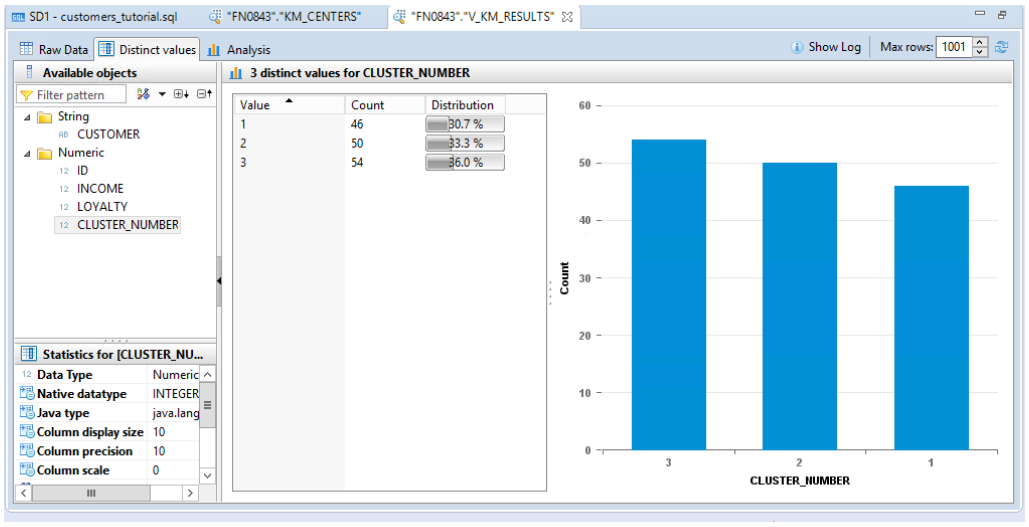
Now we look at the KM_RESULTS table. Right click on the table and choose "Open Data preview".



ID	CUSTOMER	INCOME	LOYALTY	CLUSTER_NUMBER
1	William Lucas	6.1	2.5	1
2	Morris Allen	1.5	0.4	2
3	Hugh Smith	5.7	2.3	1
4	Henry Rufoote	4	1.3	3
5	Thomas Phevens ...	4.7	1.4	3
6	John Stilman	1.4	0.2	2
7	Edmund Walden ...	5.1	1.9	1
8	Humphrey Little ...	1.5	0.2	2
9	Kelley Edward	5.3	2.3	1
10	Christopher Cross	4.5	1.3	3
11	Stevenson Charle...	5.5	2.1	1
12	Stilman John	1.1	0.1	2
13	Powell Edward	4.7	1.4	3
14	Willett Richard	1.7	0.2	2
15	Nicholas Petman ...	4.5	1.5	3
16	Roger Pratt	4.2	1.3	3
17	Willie Clancy	5.6	1.8	1
18	Johnny Doran	1.5	0.4	2
19	Thomas Smith	5.1	1.5	1
20	Richard Taverner	3.9	1.1	3
21	Emme Merrimoth...	3.5	1	3
22	James Cameron	1.3	0.4	2
23	Humphrey Thom...	4.8	1.8	3

We can see that for every customer has been assigned the cluster id and the distance that is how far is the customer from the center point.

This is also important to check how many customers we have in each cluster and that the clusters are well balance. Click on "Distinct values" label. We see that the number of customers is balances in the three clusters.



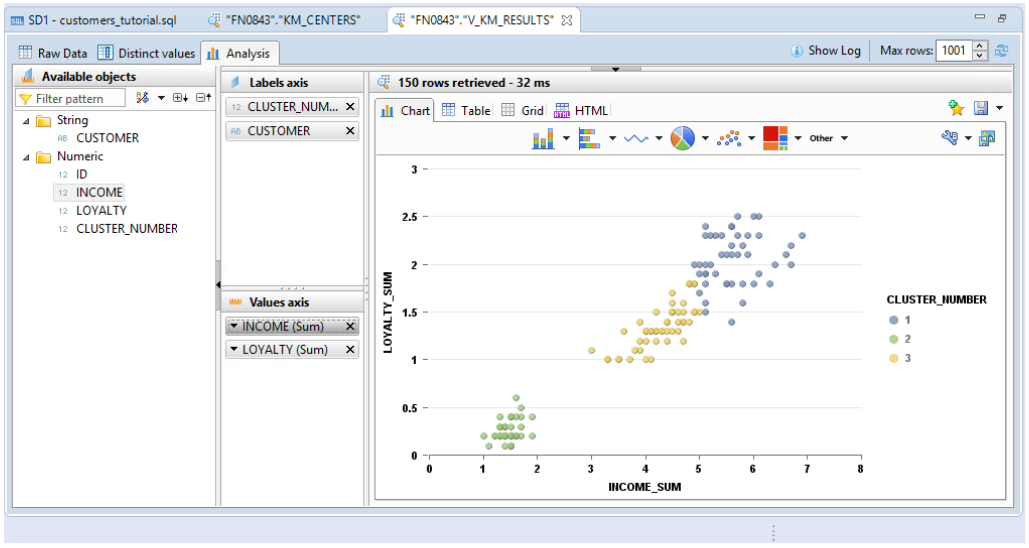
3 distinct values for CLUSTER_NUMBER

Value	Count	Distribution
1	46	30.7 %
2	50	33.3 %
3	54	36.0 %

Statistics for [CLUSTER_NU...

Data Type	Numeric
Native datatype	INTEGER
Java type	java.lang
Column display size	10
Column precision	10
Column scale	0

Now we look at the result View table- Open the V_KM_RESULTS. Click Analysis on the tool bar and add to the Labels axis first the CLUSTER_NUMBER and as second CUSTOMERS to the Values Axis the INCOME and LOYALTY. We see where the K-means algorithm identifies the 3 clusters.



Measuring the Quality

PAL gives us the ability to measure the quality of a clustering process by providing the VALIDATEKMEANS function. This function calculates the Clusters' Silhouette.

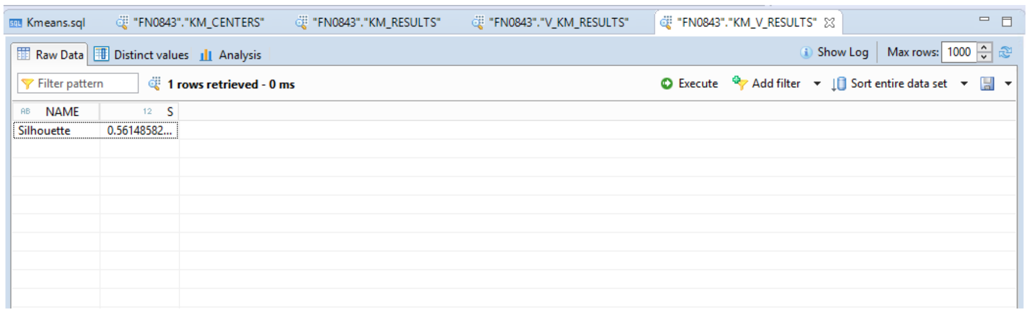
For each point it calculates the average distance to all other points in the same cluster, this is called the average dissimilarity. Then it calculates the average dissimilarity of each point to all other clusters where the point is not a member. The cluster with the lowest average dissimilarity is said to be the "neighboring cluster", as it is, aside from the cluster it has been assigned to, the cluster in which fits best. The average distance to cluster members is then compared to the average distance to members of the neighboring cluster. The resulting value is the silhouette score.

The silhouette score can go from -1 to 1. A negative value means that the record is more similar to the records of its neighboring cluster than to other members of its own cluster. A score near to 0 means that the record is in the border of the two clusters. A score near to 1 means that the record has been appropriately clustered. The silhouette of the entire dataset is the average of the silhouette scores of all the individual records. So if this number is close to 1, it means we used the right number of clusters.

First we need to generate the Validation Procedure using the AFL Wrapper Generator the same way we did with the Clustering Procedure:

```
/* This type will be used as the input parameter
of the table that contains the result of the clustering process */
CREATE TYPE PAL_T_KM_V_TYPE_ASSIGN AS TABLE (ID INTEGER, TYPE_ASSIGN INTEGER);

/* This is the type of the output parameter that will
show the silhouette score of the entire data set */
CREATE TYPE PAL_T_KM_V_RESULTS AS TABLE (NAME VARCHAR(50), S DOUBLE);

/* This is the table that contains the input and output tables */
CREATE COLUMN TABLE PAL_KM_V_SIGNATURE (ID INTEGER, TYPENAME VARCHAR(100), ↙
    DIRECTION VARCHAR(100));
INSERT INTO PAL_KM_V_SIGNATURE VALUES (1, '<your schema name>.PAL_T_KM_DATA', ↙
    'in');
INSERT INTO PAL_KM_V_SIGNATURE VALUES (2, '<your schema ↙
    name>.PAL_T_KM_V_TYPE_ASSIGN', 'in');
INSERT INTO PAL_KM_V_SIGNATURE VALUES (3, '<your schema ↙
    name>.PAL_T_KM_PARAMS', 'in');
INSERT INTO PAL_KM_V_SIGNATURE VALUES (4, '<your schema ↙
    name>.PAL_T_KM_V_RESULTS', 'out');

/* Generate the PAL function by calling the AFL Wrapper Generator */
CALL SYSTEM.AFL_WRAPPER_GENERATOR ('PAL_KM_V', 'AFLPAL', 'VALIDATEKMEANS', ↙
    PAL_KM_V_SIGNATURE);

-- app setup
CREATE VIEW V_KM_TYPE_ASSIGN AS
   SELECT ID, CENTER_ID AS TYPE_ASSIGN
      FROM KM_RESULTS
   ;
CREATE COLUMN TABLE KM_V_PARAMS LIKE PAL_T_KM_PARAMS;
CREATE COLUMN TABLE KM_V_RESULTS LIKE PAL_T_KM_V_RESULTS;
/* Fill the Parameters Table */
INSERT INTO KM_V_PARAMS VALUES ('VARIABLE_NUM', 4, null, null);
INSERT INTO KM_V_PARAMS VALUES ('THREAD_NUMBER', 1, null, null);

/* Call the Validate KMeans procedure */
CALL _SYS_AFL.PAL_KM_V (V_KM_DATA, V_KM_TYPE_ASSIGN, KM_V_PARAMS, ↙
    KM_V_RESULTS) WITH OVERVIEW;
```

If you refresh the Table folder you see a new table KM_V_RESULTS. Open the data preview right clicking on it.



1 rows retrieved - 0 ms

NAME	S
Silhouette	0.56148582...

The Silhouette value is above 0.5 means that we do a fairly good group grouping.

4.3.6 References

- Homepage on Internet - *SAP HANA Academy - PAL: Clustering Using K-means*
- Homepage on Internet - *SAP HANA Academy - PAL: Clustering Using Validate K-means*
- Homepage on Internet - *SAP HANA Academy Git Repository*
- Homepage on Internet - *SAP HANA Predictive Analysis Library (PAL) Documentation*
- SAP HELP Portal - *Modeling a flowgraph*
- Book - Principles of Data Mining - Undergraduate Topics in Computer Science. Max Bramer, Springer.

4.4 Tutorial 3 - Heart Disease Prediction Model using Decision Tree in Apache Spark

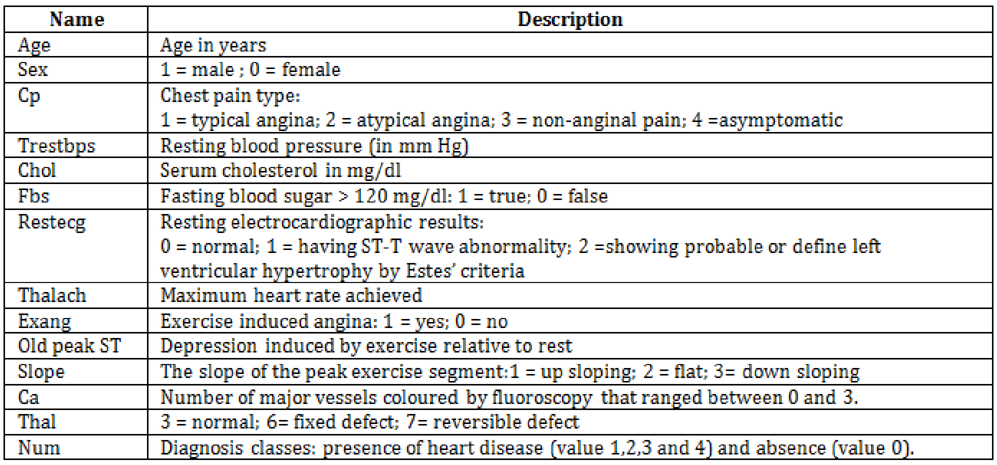
This tutorial demonstrates how to use MLLib, Spark's built-in machine learning libraries, to perform a simple predictive analysis on an open dataset. We are going to classify heart diseases by using a Decision Tree algorithm. The decision tree approach is one of the most effective and efficient classification methods available.

MLLib is a core Spark library that provides a number of utilities that are useful for machine learning tasks. This tutorial provides just one example of a possible use case for MLlib. For more examples about the machine learning and Spark, see the *MLlib official guide*.

We will run the application using PySpark, the Spark Python API, available on the Jupyter Notebook on the SDIL Platform.

We use the Heart Disease dataset of the Cleveland Clinic from the UCI Machine Learning repository available *here*. The dataset is described in detail *here*.

We have total 302 instances of which 164 instances belonged to the healthy and 139 instances belonged to the heart disease. 14 clinical features have been recorded for each instance. The following table shows the 14 clinical features and their description.



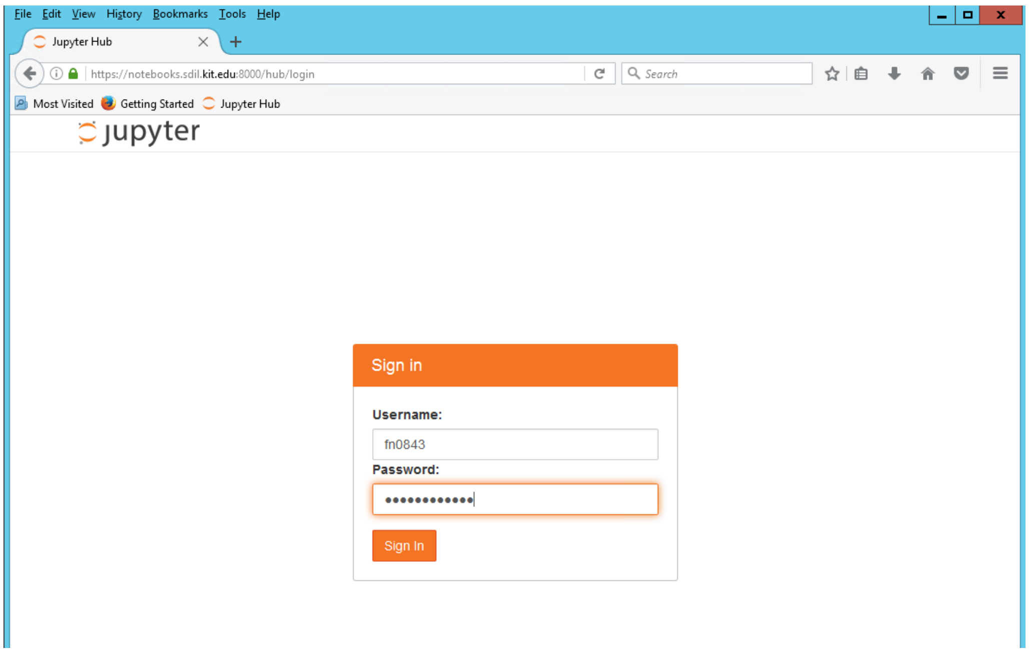
Name	Description
Age	Age in years
Sex	1 = male ; 0 = female
Cp	Chest pain type: 1 = typical angina; 2 = atypical angina; 3 = non-anginal pain; 4 =asymptomatic
Trestbps	Resting blood pressure (in mm Hg)
Chol	Serum cholesterol in mg/dl
Fbs	Fasting blood sugar > 120 mg/dl: 1 = true; 0 = false
Restecg	Resting electrocardiographic results: 0 = normal; 1 = having ST-T wave abnormality; 2 =showing probable or define left ventricular hypertrophy by Estes' criteria
Thalach	Maximum heart rate achieved
Exang	Exercise induced angina: 1 = yes; 0 = no
Old peak ST	Depression induced by exercise relative to rest
Slope	The slope of the peak exercise segment:1 = up sloping; 2 = flat; 3= down sloping
Ca	Number of major vessels coloured by fluoroscopy that ranged between 0 and 3.
Thal	3 = normal; 6= fixed defect; 7= reversible defect
Num	Diagnosis classes: presence of heart disease (value 1,2,3 and 4) and absence (value 0).

We focus on detecting the presence of heart disease (value 1,2,3 and 4) and absence (value 0). The variable we want to predict is Num. We assume that every value with 0 means heart is okay, and 1,2,3,4 means heart disease.

To access the code snippets used in the tutorial please visit the heart_diseases_tutorial folder on the *SDIL Git repository* and check out the *heart_disease_tutorial.ipynb* file. Log in with your SDIL credentials. Please check the paragraph 2.4.5 of the *SDIL Platform Documentation* to check how to access and use the SDIL Git Repository.

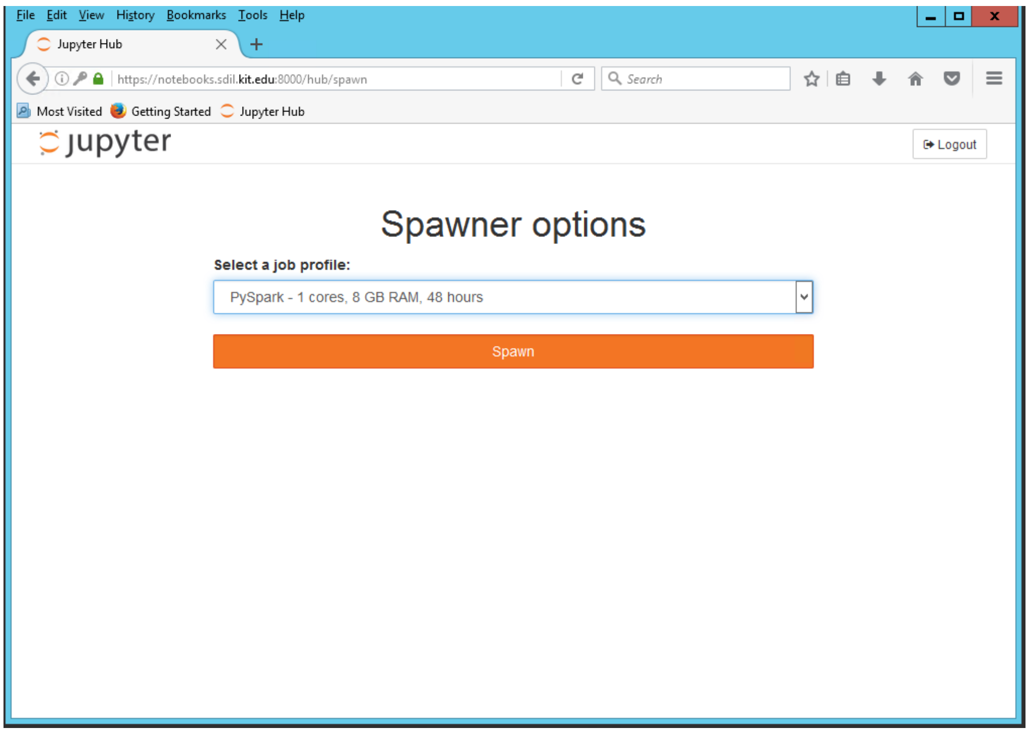
4.4.1 Start PySpark in the Jupyter Notebook

In order to access the Jupyter Notebook on the SDIL Platform log in to the Windows or Linux login server (see section 2.3 of the *SDIL Platform Documentation*), open a web browser and connect to *https://notebooks.sdil.kit.edu*. Log in with your SDIL credentials.


File Edit View History Bookmarks Tools Help
Jupyter Hub
https://notebooks.sdil.kit.edu:8000/hub/login
Search
Most Visited
Getting Started
Jupyter Hub
jupyter
Sign in
Sign In

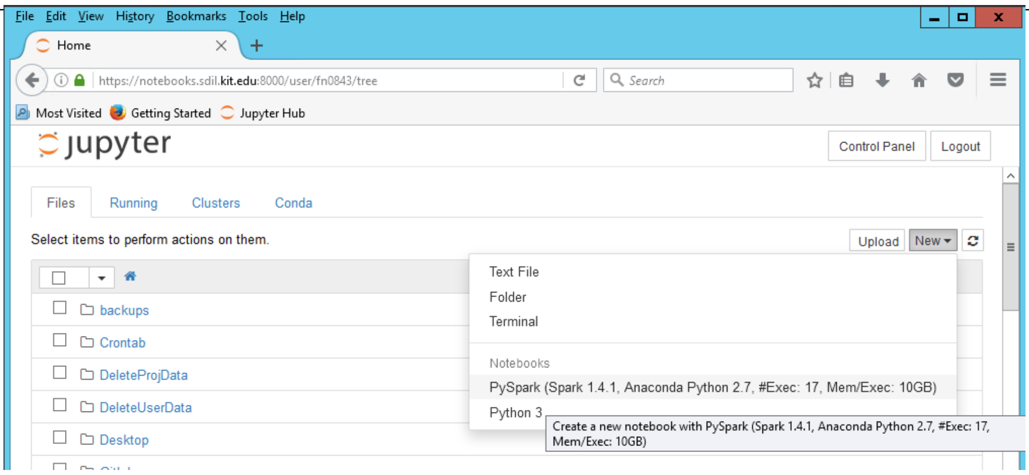
Username: fn0843
Password: ••••••••••••

Once you are logged in, you have to select one of the "PySpark" profile from the Profiles Spawner list. Click the button "Spawn".


File Edit View History Bookmarks Tools Help
Jupyter Hub
https://notebooks.sdil.kit.edu:8000/hub/spawn
Search
Most Visited
Getting Started
Jupyter Hub
jupyter
Logout
Spawner options
Spawn

Select a job profile: PySpark - 1 cores, 8 GB RAM, 48 hours

After your server is started you can create a new Jupyter notebook using the PySpark kernel. Select "PySpark" from the "New" menu.


File Edit View History Bookmarks Tools Help
Home
https://notebooks.sdil.kit.edu:8000/user/fn0843/tree
Search
Most Visited
Getting Started
Jupyter Hub
jupyter
Control Panel
Logout
Files
Running
Clusters
Conda
Select items to perform actions on them.
Upload
New
backups
Crontab
DeleteProjData
DeleteUserData
Desktop
Text File
Folder
Terminal
Notebooks
PySpark (Spark 1.4.1, Anaconda Python 2.7, #Exec: 17, Mem/Exec: 10GB)
Python 3
Create a new notebook with PySpark (Spark 1.4.1, Anaconda Python 2.7, #Exec: 17, Mem/Exec: 10GB)

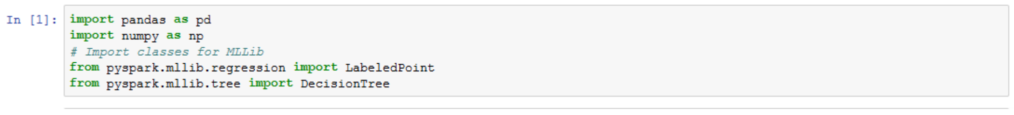
4.4.2 Load and Parse the Data

Now we have a new notebook document we can rename to *heart_disease_tutorial*. This renames also the file to *heart_disease_tutorial.ipynb*.

First, we will import the mllib packages.

Insert the following code in the empty cell and run it by clicking the button on the tool bar or Shift-Enter to execute the current cell, show output (if any), and jump to the next cell below.

```
import pandas as pd
import numpy as np
# Import classes for MLLib
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.tree import DecisionTree
```

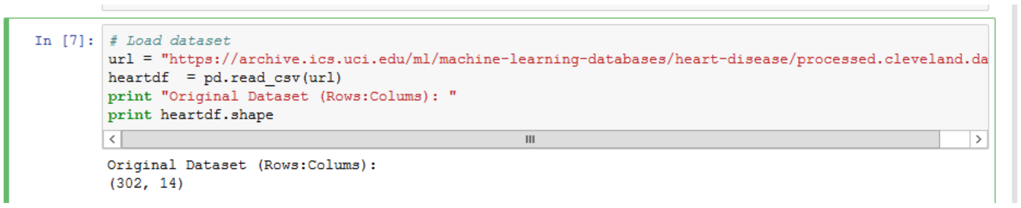


```
In [1]: import pandas as pd
        import numpy as np
        # Import classes for MLLib
        from pyspark.mllib.regression import LabeledPoint
        from pyspark.mllib.tree import DecisionTree
```

Now we can import the dataset and check the number of rows and columns.

```
# Load dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/
heart-disease/processed.cleveland.data"
heartdf = pd.read_csv(url)
print "Original Dataset (Rows:Colums): "
print heartdf.shape
```

You should see as output the number of rows and columns.



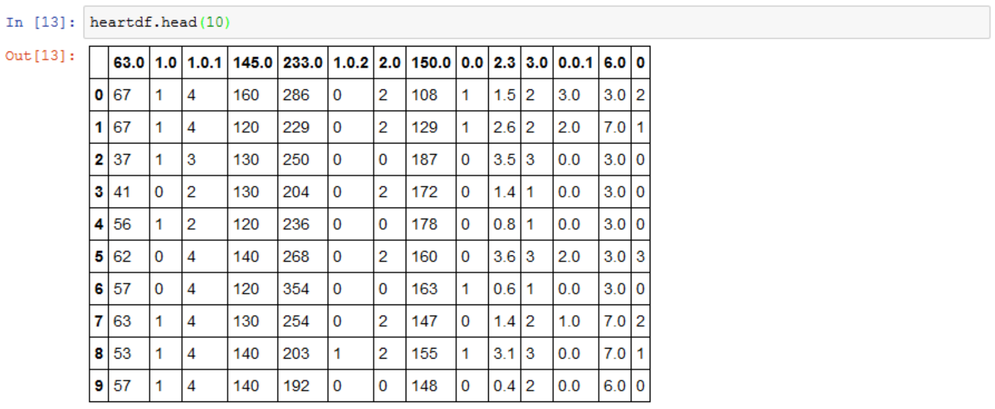
```
In [7]: # Load dataset
        url = "https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.da
        heartdf  = pd.read_csv(url)
        print "Original Dataset (Rows:Colums): "
        print heartdf.shape

        Original Dataset (Rows:Colums):
        (302, 14)
```

We check the unique possible values of the column containing the Diagnosis of heart disease which are 0, 1, 2, 3 and 4.

```
print "Categories of Diagnosis of heart disease (angiographic disease status) ↙
    that we are predicting"
print "-- Value 0: < 50% diameter narrowing"
print "-- Value 1: > 50% diameter narrowing "
print heartdf.ix[:,13].unique() #Column containing the Diagnosis of heart disease
```

Have a look at the first 10 rows of our dataset:

```
heartdf.head(10)
```

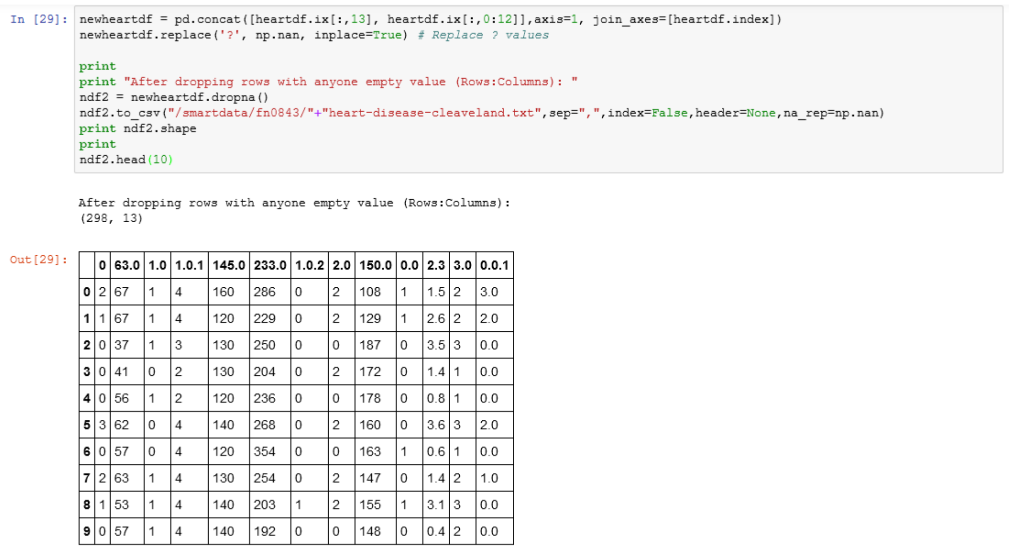


```
In [13]: heartdf.head(10)
Out[13]:
```

	63.0	1.0	1.0.1	145.0	233.0	1.0.2	2.0	150.0	0.0	2.3	3.0	0.0.1	6.0	0
0	67	1	4	160	286	0	2	108	1	1.5	2	3.0	3.0	2
1	67	1	4	120	229	0	2	129	1	2.6	2	2.0	7.0	1
2	37	1	3	130	250	0	0	187	0	3.5	3	0.0	3.0	0
3	41	0	2	130	204	0	2	172	0	1.4	1	0.0	3.0	0
4	56	1	2	120	236	0	0	178	0	0.8	1	0.0	3.0	0
5	62	0	4	140	268	0	2	160	0	3.6	3	2.0	3.0	3
6	57	0	4	120	354	0	0	163	1	0.6	1	0.0	3.0	0
7	63	1	4	130	254	0	2	147	0	1.4	2	1.0	7.0	2
8	53	1	4	140	203	1	2	155	1	3.1	3	0.0	7.0	1
9	57	1	4	140	192	0	0	148	0	0.4	2	0.0	6.0	0

Now we create a new dataset from the original one in which the first column is the num value followed by all other columns. If we look at the entire dataset we see that there are rows with ? values. We replace these symbol with a Not a Number (NaN) value and then we remove rows with empty values from the dataset.

```
newheartdf = pd.concat([heartdf.ix[:,13], heartdf.ix[:,0:12]],axis=1, ↙
    join_axes=[heartdf.index])
# replacing Not a Number (NaN)
#  replace(to_replace, value, subset=None)
newheartdf.replace('?', np.nan, inplace=True) # Replace ? values

print
print "After dropping rows with anyone empty value (Rows:Columns): "
ndf2 = newheartdf.dropna() # remove NaN values
ndf2.to_csv("<your homedirectory path>"+"heart-disease-cleaveland.txt",sep=",",
index=False,header=None,na_rep=np.nan)
print ndf2.shape
print ndf2.head(10)
```



```
In [29]: newheartdf = pd.concat([heartdf.ix[:,13], heartdf.ix[:,0:12]],axis=1, join_axes=[heartdf.index])
         newheartdf.replace('?', np.nan, inplace=True) # Replace ? values

         print
         print "After dropping rows with anyone empty value (Rows:Columns): "
         ndf2 = newheartdf.dropna()
         ndf2.to_csv("/smartdata/fn0843/"+"heart-disease-cleaveland.txt",sep=",",index=False,header=None,na_rep=np.nan)
         print ndf2.shape
         print
         ndf2.head(10)

         After dropping rows with anyone empty value (Rows:Columns):
         (298, 13)
```

Out[29]:

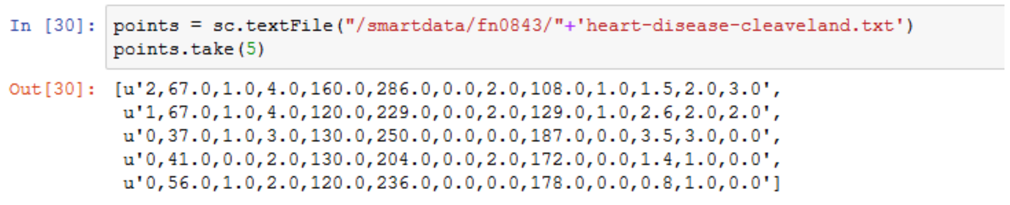
	0	63.0	1.0	1.0.1	145.0	233.0	1.0.2	2.0	150.0	0.0	2.3	3.0	0.0.1
0	2	67	1	4	160	286	0	2	108	1	1.5	2	3.0
1	1	67	1	4	120	229	0	2	129	1	2.6	2	2.0
2	0	37	1	3	130	250	0	0	187	0	3.5	3	0.0
3	0	41	0	2	130	204	0	2	172	0	1.4	1	0.0
4	0	56	1	2	120	236	0	0	178	0	0.8	1	0.0
5	3	62	0	4	140	268	0	2	160	0	3.6	3	2.0
6	0	57	0	4	120	354	0	0	163	1	0.6	1	0.0
7	2	63	1	4	130	254	0	2	147	0	1.4	2	1.0
8	1	53	1	4	140	203	1	2	155	1	3.1	3	0.0
9	0	57	1	4	140	192	0	0	148	0	0.4	2	0.0

The new dataset has now 298 rows.

4.4.3 Create Labeled Points

The basic data abstraction for Spark is an RDD (Resilient Distributed Dataset) which is a distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. An RDD is, essentially, the Spark representation of a set of data, spread across multiple machines. An RDD could come from any datasource, e.g. text files, a database via JDBC, etc. We now read data as an RDD and named it "points".

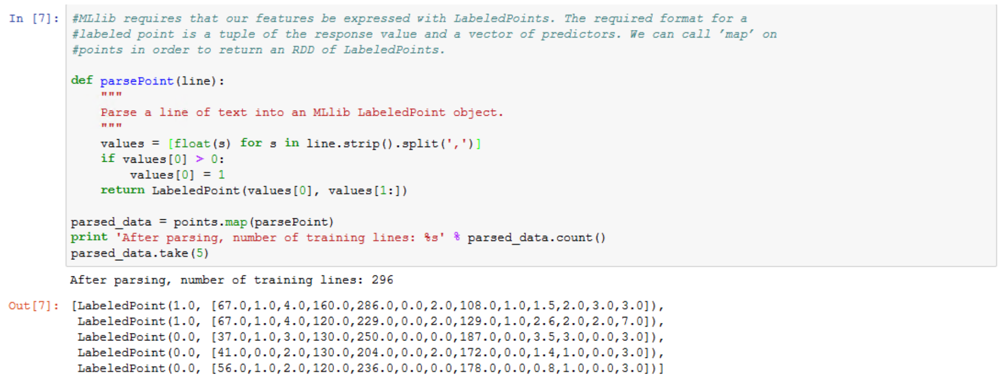
```
points = sc.textFile("<your homedirectory path>"+"heart-disease-cleaveland.txt")
points.take(5)
```



```
In [30]: points = sc.textFile("/smartdata/fn0843/"+'heart-disease-cleaveland.txt')
         points.take(5)

Out[30]: [u'2,67.0,1.0,4.0,160.0,286.0,0.0,2.0,108.0,1.0,1.5,2.0,3.0',
          u'1,67.0,1.0,4.0,120.0,229.0,0.0,2.0,129.0,1.0,2.6,2.0,2.0',
          u'0,37.0,1.0,3.0,130.0,250.0,0.0,0.0,187.0,0.0,3.5,3.0,0.0',
          u'0,41.0,0.0,2.0,130.0,204.0,0.0,2.0,172.0,0.0,1.4,1.0,0.0',
          u'0,56.0,1.0,2.0,120.0,236.0,0.0,0.0,178.0,0.0,0.8,1.0,0.0']
```

MLlib requires that our features be expressed with LabeledPoints. The required format for a labeled point is a tuple of the response value and a vector of predictors. We can call 'map' on points in order to return an RDD of LabeledPoints.

```
def parsePoint(line):
    """
    Parse a line of text into an MLlib LabeledPoint object.
    """
    values = [float(s) for s in line.strip().split(',')]
    #consider only two classes
    if values[0] > 0:
        values[0] = 1
    return LabeledPoint(values[0], values[1:])

parsed_data = points.map(parsePoint)
print 'After parsing, number of training lines: %s' % parsed_data.count()
parsed_data.take(5)
```



```
In [7]: #MLlib requires that our features be expressed with LabeledPoints. The required format for a
        #labeled point is a tuple of the response value and a vector of predictors. We can call 'map' on
        #points in order to return an RDD of LabeledPoints.

        def parsePoint(line):
            """
            Parse a line of text into an MLlib LabeledPoint object.
            """
            values = [float(s) for s in line.strip().split(',')]
            if values[0] > 0:
                values[0] = 1
            return LabeledPoint(values[0], values[1:])

        parsed_data = points.map(parsePoint)
        print 'After parsing, number of training lines: %s' % parsed_data.count()
        parsed_data.take(5)

        After parsing, number of training lines: 296

Out[7]: [LabeledPoint(1.0, [67.0,1.0,4.0,160.0,286.0,0.0,2.0,108.0,1.0,1.5,2.0,3.0,3.0]),
         LabeledPoint(1.0, [67.0,1.0,4.0,120.0,229.0,0.0,2.0,129.0,1.0,2.6,2.0,2.0,7.0]),
         LabeledPoint(0.0, [37.0,1.0,3.0,130.0,250.0,0.0,0.0,187.0,0.0,3.5,3.0,0.0,3.0]),
         LabeledPoint(0.0, [41.0,0.0,2.0,130.0,204.0,0.0,2.0,172.0,0.0,1.4,1.0,0.0,3.0]),
         LabeledPoint(0.0, [56.0,1.0,2.0,120.0,236.0,0.0,0.0,178.0,0.0,0.8,1.0,0.0,3.0])]
```

4.4.4 Perform Classification using a Decision Tree

Next, we split the dataset into training and test sets, train on the first dataset, and then evaluate on test set.

The model is trained by making associations between the input features and the labeled output associated with those features. We train the model using the DecisionTree.trainClassifier method which returns a DecisionTreeModel.

We train the decision tree model with Gini impurity as an impurity measure and a maximum tree depth of 3.

For more information check the Apache Spark Decision Trees - RDD-based API.

```
# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = parsed_data.randomSplit([0.7, 0.3])
# Train a DecisionTree model.
# Empty categoricalFeaturesInfo indicates all features are continuous.
model = DecisionTree.trainClassifier(trainingData, numClasses=2, ↙
    categoricalFeaturesInfo={}, impurity='gini', maxDepth=3, maxBins=32)
```

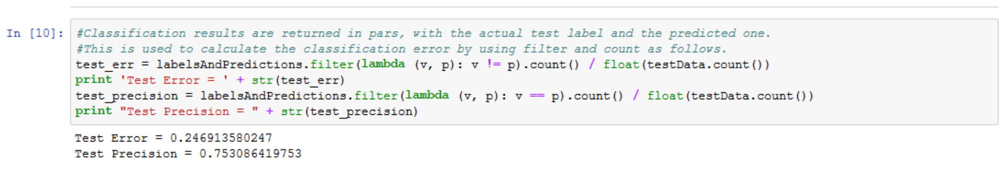
4.4.5 Evaluate model on test instances and compute test error

In order to measure the classification error on our test data, we use map on the testData RDD and the model to predict each test point class.

```
predictions = model.predict(testData.map(lambda x: x.features))
labelsAndPredictions = testData.map(lambda lp: lp.label).zip(predictions)
```

Classification results are returned in pars, with the actual test label and the predicted one. This is used to calculate the classification error by using filter and count as follows.

```
testErr = labelsAndPredictions.filter(lambda (v, p): v != p).count() / ↙
    float(testData.count())
print('Test Error = ' + str(testErr))
test_precision = labelsAndPredictions.filter(lambda (v, p): v == p).count() / ↙
    float(testData.count())
print "Test precision = " + str(test_precision)
```



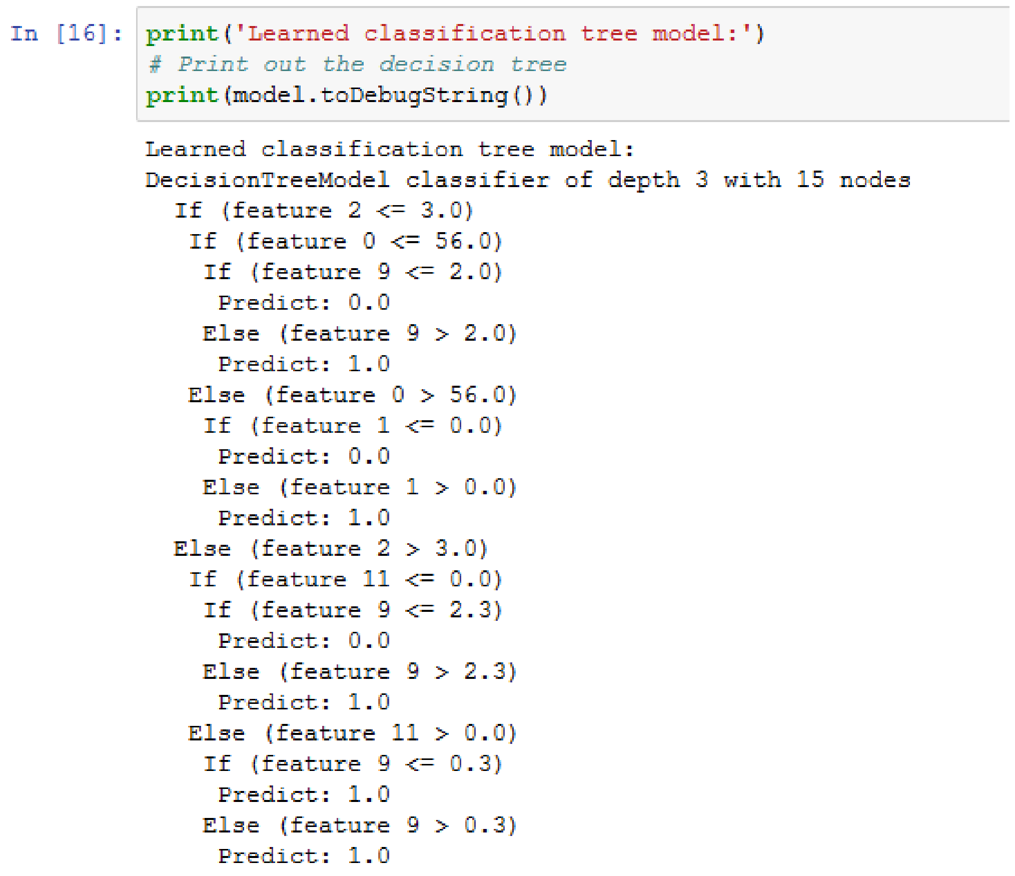
```
In [10]: #Classification results are returned in pars, with the actual test label and the predicted one.
         #This is used to calculate the classification error by using filter and count as follows.
         test_err = labelsAndPredictions.filter(lambda (v, p): v != p).count() / float(testData.count())
         print 'Test Error = ' + str(test_err)
         test_precision = labelsAndPredictions.filter(lambda (v, p): v == p).count() / float(testData.count())
         print "Test Precision = " + str(test_precision)

         Test Error = 0.246913580247
         Test Precision = 0.753086419753
```

4.4.6 Interpreting the model

The toDebugString() function provides a print of the tree's decision nodes and final prediction outcomes at the end leafs.

```python
print('Learned classification tree model:')
# Print out the decision tree
print(model.toDebugString())
```

Learned classification tree model:



```
In [16]: print('Learned classification tree model:')
         # Print out the decision tree
         print(model.toDebugString())

         Learned classification tree model:
         DecisionTreeModel classifier of depth 3 with 15 nodes
           If (feature 2 <= 3.0)
            If (feature 0 <= 56.0)
             If (feature 9 <= 2.0)
              Predict: 0.0
             Else (feature 9 > 2.0)
              Predict: 1.0
            Else (feature 0 > 56.0)
             If (feature 1 <= 0.0)
              Predict: 0.0
             Else (feature 1 > 0.0)
              Predict: 1.0
           Else (feature 2 > 3.0)
            If (feature 11 <= 0.0)
             If (feature 9 <= 2.3)
              Predict: 0.0
             Else (feature 9 > 2.3)
              Predict: 1.0
            Else (feature 11 > 0.0)
             If (feature 9 <= 0.3)
              Predict: 1.0
             Else (feature 9 > 0.3)
              Predict: 1.0
```

4.4.7 References

- Homepage on Internet - *MLlib Decision Trees*
- Homepage on Internet - *UCI Machine Learning Repository - Heart Disease Dataset*