MET CS 777
Big Data Analytics

Lecture - 4

Dimitar Trajanov

Agenda

- Data Partitioning and Accessing data partitions
- Map-Side Join Operation
- Using python libraries with Spark
  - NumPy
  - SciPy
- Spark linalg package
- TF-IDF algorithm

Data Partitioning and Accessing data partitions

- BigDataAnalytics/Spark-Example-08-Data-Partitioning.ipynb at master · trajanov/BigDataAnalytics (github.com)
- BigDataAnalytics/Spark-Example-09-Data-Partitioning-TreeAggregate.ipynb at master · trajanov/BigDataAnalytics (github.com)

Map-Side Join Operation

- If we are joining a small RDD with a large RDD and the small RDD can fit into the main memory of a single executor, then we can convert the Join operation into a map operation. This is then called Map-Side Join operation.

1. Collect the small RDD as Map (a dict in python)
2. Broadcast the small dictionary so that a copy of it is available on each worker.
3. Do the map to run the join operation instead of the actual join operation

- BigDataAnalytics/Spark-Example-16-Map-Side-Join.ipynb at master · trajanov/BigDataAnalytics (github.com)

Map-Side Join Operation

- In case that the RDD is large so that it can not fit into the memory, then maybe the keys only can fit into the memory. This would allow us to keep the keys of the medium size RDD in memory and use it to reduce the size of the large RDD and then run the join operation on it.

1. Collect the Keys of the Medium size RDD into a set of keys
2. Use the keys to filter the large RDD and reduce the size of
3. Then run the join on the smaller RDD

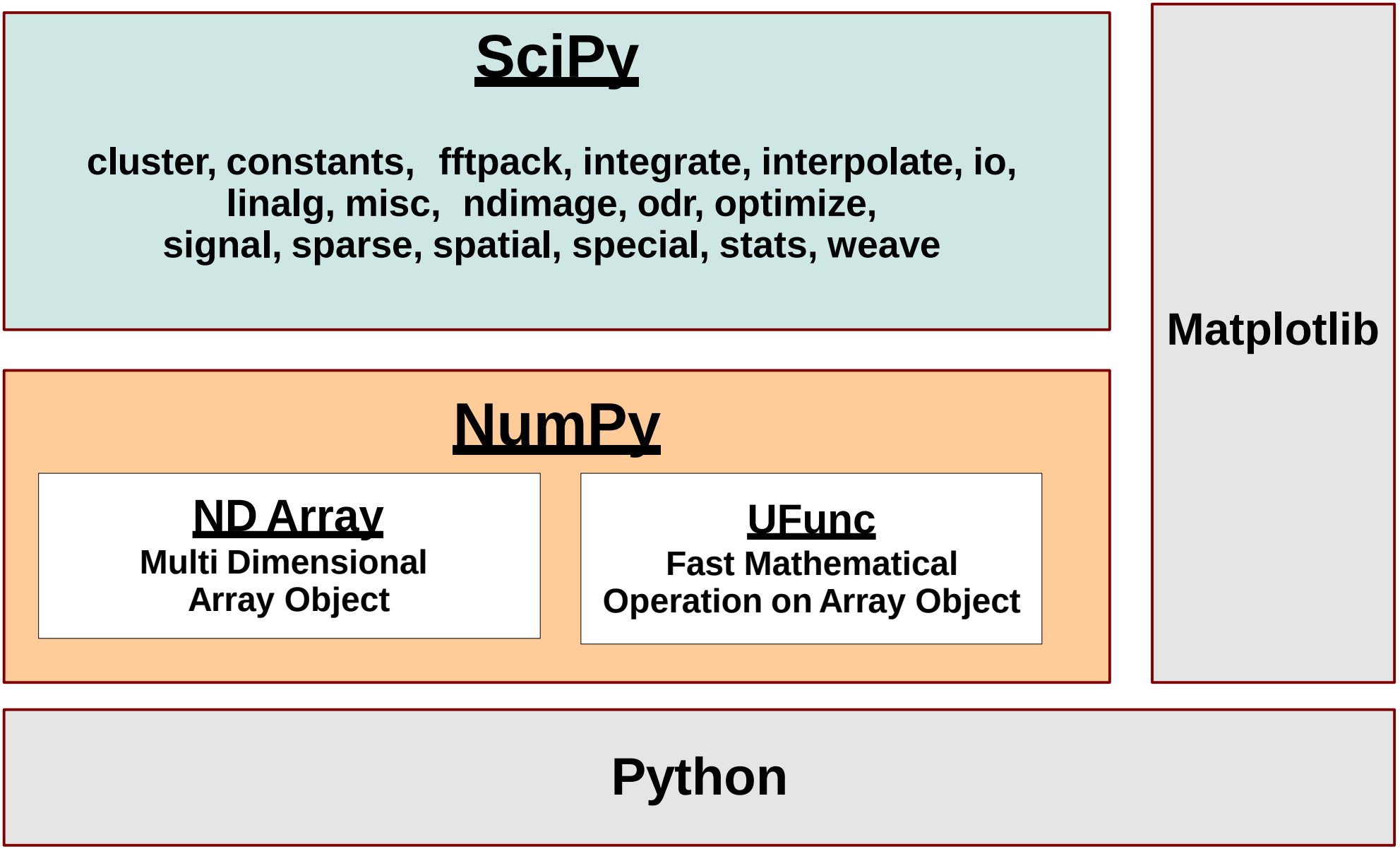
- BigDataAnalytics/Spark-Example-16-Map-Side-Join.ipynb at master · trajanov/BigDataAnalytics (github.com)

NumPy & SciPy

- ***NumPy and SciPy*** are powerful libraries for mathematics, science and engineering computing including data analysis.

- **NumPy** provides Core Data Structure libraries
  - **http://www.numpy.org/**
- **SciPy** provides Scientific Algorithm libraries
  - **http://scipy.org/**
- Documentation about NumPy and Scipy
  - **http://docs.scipy.org/doc/numpy/**

Python, NumPy, SciPy, Matplotlib



Importing NumPy

- **Import numpy in your program**

  >>> *from numpy import **

  or

  >>> from numpy import array

  or

  >>> import numpy as np

- **numpy** is the top package name, and ***"import numpy"*** doesn't import **submodules like numpy.f2py**

- **Running ipython with "--pylab" imports all of NumPy**

  # ipython --pylab

Importing NumPy

- If you **imported numpy:**

  >>> array

  <function numpy.core.multiarray.array>

- If you imported **numpy as np**:

  >>> **np.array**

  <function numpy.core.multiarray.array>

- If you **have not Imported numpy,** you will get an error message like this:

  >>> array

  NameError Traceback (most recent call last)

  <ipython-input-1-a7cf24f7419f> in <module>()

  ----> 1 array

  NameError: name 'array' is not defined

Arrays in NumPy

- **A very simple array:**

  >>> a = np.array([0,1,2,3])

  >>> a

  array([0, 1, 2, 3])

- **Size:**

  >>> a.size

  4

- **Array Shape:**

  shape returns a tuple listing the length of array along each dimension of it

  >>> a.shape

  (4,)

- Ndim – number of dimensions

  >>> a.ndim

  1

- **Type:**

  >>> type(a)

  <type 'numpy.ndarray'>

- **Type of Elements:**

  >>> a.dtype

  dtype('int64')

- **Bytes per Elements:**

  >>> a.itemsize

  8

- **nbytes:**

  Return the number of bytes

  >>> a.nbytes

  32

Creating Arrays

- **Creating an Array**

  >>> a = np.array([1,2,3,4])

  >>> a

  array([1, 2, 3, 4])

- **Creating an Array including float numbers**

  >>> a = np.array([1, 4, 5, 8], float)

  >>> a

  array([1., 4., 5., 8.])

- **Checking types:**

  >>> type(a)

  numpy.ndarray

- ***arange***

  >>> np.arange(3)

  array([0, 1, 2])

  >>> np.arange(3.0)

  array([0., 1., 2.])

  >>> np.arange(3,7)

  array([3, 4, 5, 6])

  >>> arange(3,7,2)

  array([3, 5])

- ***numpy.arange([start,] stop, [step,]dtype=None)***

Creating Arrays - linspace

- **linspace**

```
>>> np.linspace(2.0, 3.0, num=5)
    array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
    array([ 2. ,  2.2,  2.4,  2.6,  2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
    (array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)
```

- **numpy.linspace (start, stop, num=50, endpoint=True, retstep=False, dtype=None)**

Operations on Arrays

- **Simple Operations on Arrays**

```
>>> a =np.array([1,2,3,4])
>>> b = a + a
>>> b
array([2, 4, 6, 8])
>>> c= a* a
>>> c
np.array([ 1,  4,  9, 16])

>>> d= a**a
>>> d
array([  1,   4,  27, 256])
```

- **Simple Math Functions**

```
>>> x=2
>>> y=x+a
>>> y
np.array([3, 4, 5, 6])
>>> y=x*a
>>> y
np.array([2, 4, 6, 8])


>>> y=sin(a)
>>> y
array([ 0.84147098,  0.90929743,
0.14112001, -0.7568025 ])
```

Operations on Arrays

- **NumPy defines PI and e constants:**

  pi = 3.14159265359

  e = 2.71828182846

```
>>> a = np.array([1,2,3,4])
>>> c=(2*pi)*a
>>> c
array([  6.28318531,  12.56637061,
18.84955592,  25.13274123])

>>> y=np.sin(c)
>>> y
array([ -2.44929360e-16,  -4.89858720e-
16, -7.34788079e-16,  -9.79717439e-16])
```

- **in-place operations**

```
>>> x= np.arange(6)
>>> z = 2*pi
>>> x *=z
>>> x
array([ 0,  6, 12, 18, 25, 31])
```

- **On Float Elements:**

```
>>> x= np.arange(6.)
>>> x *=z
>>> x
array([  0. ,   6.28318531,  12.56637061,
18.84955592, 25.13274123,  31.41592654])
```

Setting Array Elements – Filling Arrays

- **Indexing**

  >>> a = array([1,2,3,4])

  >>> a[0]

  1

- **Setting values**

  >>> a[0]=2*pi

  >>> a[0]

  6.283185307179586

  >>> a

  [6.283185307179586, 2, 3, 4]

- **Fill - Fill the array with a scalar value.**

  >>> a.fill(0)

  >>> a

  array([0, 0, 0, 0, 0])

- **Data Types are important:**

  >>> a.dtype

  dtype('int64')

- **Assigning a float into a Integer array**

  >>> a[0]=2.34

  >>> a

  array([2, 2, 3, 4])

  >>> a.fill(2.3)

  >>> a

  Array([2, 2, 2, 2])

- **Filling by using slice**

  >>> a[:]=1

  >>> a

  array([1, 1, 1, 1])

Slicing Arrays

- **Slicing can extract a portion of an array by using a lower and upper bound.**

  var[lower:upper:step]

  >>> **a=array([3,21,1,6,7])**

  >>> a[1:3]

  array([21, 1])

- **Negative Indexes:**

  >>> a[1:-2]

  array([21, 1])

  >>> a[-3:-1]

  array([1, 6])

- **Omitting indices:**

  start or end or both

  >>> a[:2]

  array([3, 21])

  >>> a[-2:]

  array([6, 7]

- **Slicing with steps:**

  >>> a[::2]

  array([3, 1, 7])

  >>> a[1::2]

  array([21, 6])

- a1[::2] – elements even indexes
- a[1::2] – elements with odd indexes

Slices are References

- Slices are simple references to the original memory. Any changes on the slices would change the original array.

```
>>> a=array([1,2,3,4,5])
>>> b=a[2:4]
>>> b
array([3, 4])
>>> b[0]=100
>>> a
array([  1,   2, 100,   4,   5])
```

N-Dimensional Arrays

- **A 2-D array:**

```
>>> b = array([[ 10, 11, 12, 13], [20, 21, 22, 23] ])
>>> b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

- **Shape, size, ndim:**

```
>>> b.shape
(2, 4)
>>> b.size
8
>>> b.ndim
2
```

- **Indexing in 2-D:**

```
>>> b[1,3]
23
```

- **Setting Values:**

```
>>> b[1,3]=2.3
>>> b
array([[10, 11, 12, 13],
       [20, 21, 22,  2]])
```

- **Addressing the rows using single index**

```
>>> b[1]
array([20, 21, 22,  2])
```

Multidimensional Array Slicing

- **Similar to 1-D**

```
>>> c[1:3]
array([[20, 21, 22, 23],
       [30, 31, 32, 33]])
```

- **Addressing a column:**

```
>>> c[:,2]
array([12, 22, 32, 42])
```

- **With steps:**

```
>>> c[2::2,::2]
array([[30, 32]])
```

c=array …

10	11	12	13
20	21	22	23
30	31	32	33
40	41	42	43

Advanced indexing

- **Indexing by using positions:**

```
>>> a = arange(10,40,2)
>>> a
array([10, 12, 14, 16, 18, 20, 22, 24, 26,
28, 30, 32, 34, 36, 38])
>>> indexes=array([1,3,-4])
>>> x=a[indexes]
>>> x
array([12, 16, 32])
```

- ***Indexing with Booleans:***

```
>>> mymask=array([1,0,1,1,1],dtype=bool)
>>> y=a[mymask]
>>> y
array([10, 14, 16, 18])
```

- ***Create a mask by condition:***

```
>>> mymask2= a<20
>>> mymask2
array([ True, True, True, True, True,
False, False, False, False, False, False,
False, False, False, False], dtype=bool)
>>> y=a[mymask2]
>>> y
array([10, 12, 14, 16, 18])
```

Where Method - Finding Indexes

- **Where :**

  finds indexes in array where expression is True.

  a = array([1.19, 2.42, 3.91, 4.66])

  >>> a>2

  array([False, True, True, True], dtype=bool)

  >>> where(a>2)

  (array([1, 2, 3]),)

- ***Where 2D***

  a= array([[0, 1, 2, 3],

  [4, 5, 6, 7]])

  >>> a>2

  array([[False, False, False, True],

  [True, True, True, True]], dtype=bool)

  >>> mindex=where(a>2)

  >>> mindex

  (array([0, 1, 1, 1, 1]), array([3, 0, 1, 2, 3]))

  >>> y=a[mindex]

  >>> y

  array([3, 4, 5, 6, 7])

Reshaping Arrays

- **Arrays can be reshaped:**

```
>>> a=arange(10)
>>> a.shape
(10,)
>>> a.shape=(2,5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

- **Dimension - 1**

```
>>> a.shape=(2,1,5)
>>> a
array([[[0, 1, 2, 3, 4]],


       [[5, 6, 7, 8, 9]]])
```

- **Reshape – Returns a new array from the original array**

```
>>> b=a.reshape(5,2)
>>> b
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
```

- **Reshape do not remove elements:**

```
>>> a.reshape(3,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

Flatten Arrays and Flat Attribute

- **Flatten()**

  Converts Multidimensional arrays into one dimensional array

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> b=a.flatten()
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

a.ravel() is the same as **a.flatten(),** but works a reference on the original array.

- **Attribute – flat**

  a.flat is an attribute that can be used like an iterator to access elements in a N-D array as one dimensional array.

```
>>> a=arange(10)
>>> a.shape=(5,2)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> b=a.flat
>>> b[2]=100
>>> a
array([[  0,   1],
       [100,   3],
       [  4,   5],
       [  6,   7],
       [  8,   9]])
```

Array Transpose

- **Transpose**

```
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a.transpose()
array([[0, 5],
       [1, 6],
       [2, 7],
       [3, 8],
       [4, 9]])
>>> a.T
array([[0, 5],
       [1, 6],
       [2, 7],
       [3, 8],
       [4, 9]])
```

- Strides are Tuple of bytes to step in each dimension
- 8 bytes (1 value) to move to the next column, but 40 bytes (5 values) to get to the same position in the next row.
- Transpose only changes the values of "Strides" in the array memory.

```
>>> a.strides
(40, 8)
>>> a.T.strides
(8, 40)
```

Array Calculation Methods

- **Sum – sum up the elements**

```
a= array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> sum(a)
45
```

- **Sum along the axis:**

```
a = array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> sum(a, axis=0)
array([20, 25])
>>> sum(a, axis=1)
array([ 1,  5,  9, 13, 17])
```

- **Product – calculate product of columns**

```
a = array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> a.prod(axis=0)
array([  0, 945])
>>> a.prod(axis=1)
array([ 0,  6, 20, 42, 72])
```

Min/Max

- **Min – Find the minimum**

  a= array([[0, 1, 2, 3, 4],

  [5, 6, 7, 8, 9]])

  >>> a.min(axis=0)

  array([0, 1, 2, 3, 4])

  >>> a.min(axis=1)

  array([0, 5])

- **argmin – finding the index of Minimum element**

  >>> a.argmin(axis=1)

  array([0, 0])

  >>> a.argmin(axis=0)

  array([0, 0, 0, 0, 0])

- **Max – Find the Maximum**

  similar to min

  >>> a.max(axis=0)

  array([5, 6, 7, 8, 9])

  >>> a.max(axis=1)

  array([4, 9])

  >>> a.argmax(axis=0)

  array([1, 1, 1, 1, 1])

  >>> a.argmax(axis=1)

  array([4, 4])

- **Diagonal - Extract the diagonal from array**

  >>> a.diagonal()

  array([0, 6])

Statistics Array Methods

- **Mean**

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a.mean(axis=0)
array([ 4.,  5.,  6.])
>>> average(a, axis=0)
array([ 4.,  5.,  6.])
```

- **Average with weights**

```
>>> average(a, weights=[1,2,4], axis=0)
array([ 5.28571429,  6.28571429,
7.28571429])
```

- **Standard Deviation**

```
>>> a.std(axis=0)
array([ 2.44948974,  2.44948974,
2.44948974])
>>> a.std(axis=1)
array([ 0.81649658,  0.81649658,
0.81649658])
```

- **Variance**

```
>>> a.var(axis=0)
array([ 6.,  6.,  6.])
>>> a.var(axis=1)
array([ 0.66666667,  0.66666667,
0.66666667])
```

Further Useful Array Methods

- **Clip – limit the values in an array.**

  a=array([20, 21, 22, 23, 24, 25, 26, 27, 28])

  - set values less than 22 to 22

  - set values bigger than 27 to 27

  >>> a.clip(22,27)

  array([22, 22, 22, 23, 24, 25, 26, 27, 27])

- **Peak to Peak - Range of values (maximum - minimum) along an axis.**

  >>> a.ptp(axis=0)

  array([3, 3, 3])

  >>> a.ptp(axis=1)

  array([2, 2])

  >>> a.ptp()

  5

- **Rounding Float Numbers**

  >>> a=array([1.19, 2.42, 3.91, 4.66])

  >>> a.round()

  array([1., 2., 4., 5.])

  >>> a.round(decimals=1)

  array([1.2, 2.4, 3.9, 4.7])

Combination of Spark RDD and numpy Array

- You can have inside a Spark RDD any kind of Objects including a numpy array.

- How large can the numpy Array be in inside RDD?
  - It should not be larger than max memory of a spark executors memory. This is cluster configuration setting.
  - But it is not larger than the highest RAM memory of one of the machines.
  - In a heterogeneous cluster, not larger than RAM of the smallest machine.

Spark Dataframe and numpy Array

- You can have inside your spark Dataframe any objects including **numpy Array or matrix**.

- You can convert one of the columns of a Spark Dataframe to a numpy Array.

  import numpy as np

  np.array(sparkDF.select('COL_NAME').collect())

- Or convert it to a numpy Matrix. Numpy matrices are strictly 2-dimensional

  np.array(sparkDF.select('COL1', 'COL2').collect())

- **Note:** Remember that sparkDF is distributed on a large Cluster while numpy array is on one of the worker process (single machine) so that it should not get large.

Combination of Spark Dataframe and Pandas

- Create a Spark Dataframe from Pandas Dataframe

  sparkDF = sqlContex.createDataFrame(pandaDf)

- Convert a Spark Dataframe back to Pandas Dataframe

  pandaDf = sparkDF.toPandas()

- **Note:**

  Remember that sparkDF is distributed on the large Cluster

  while pandasDF is on one of the worker process (single machine).

SciPy

Toolboxes dedicated to scientific computing.

http://scipy.org/

Scipy Tutorial

http://docs.scipy.org/doc/scipy/reference/tutorial/index.html

http://www.scipy-lectures.org/

SciPy Organization - Subpackages

Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
IO	Input and Output
linalg	**Linear algebra**
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions
weave	C/C++ integration

Finding Documentation

- SciPy and NumPy documentation
- Use **numpy.info()**

  >>> **info**(np.polyval)

```
>>> np.info(np.polyval)
 polyval(p, x)

Evaluate a polynomial at specific values.

If `p` is of length N, this function returns the value:

    ``p[0]*x**(N-1) + p[1]*x**(N-2) + ... + p[N-2]*x + p[N-1]``

If `x` is a sequence, then `p(x)` is returned for each element of `x`.
If `x` is another polynomial then the composite polynomial `p(x(t))`
is returned.
Parameters
----------
p : array_like or poly1d object
   1D array of polynomial coefficients (including coefficients equal
   to zero) from highest degree to the constant term, or an
   instance of poly1d.
x : array_like or poly1d object
   A number, a 1D array of numbers, or an instance of poly1d, "at"
   which to evaluate `p`.

Returns, See Also, Notes, References,
Examples
---------

>>> np.polyval([3,0,1], 5)  # 3 * 5**2 + 0 * 5**1 + 1
76
```

Linear Algebra (**scipy.linalg**)

- Functions related to Linear Algebra
- **scipy.linalg** includes all the functions in numpy.linalg plus some other advances functions.
- A very good reference is:
  - http://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html
- **Basic routines:**
  - **Finding Inverse -** linalg.inv
  - **Solving linear system -** linalg.solve
  - **Finding Determinant -** linalg.det
  - **Computing norms** - linalg.norm

Linear Algebra (scipy.linalg) - Examples

- **Inverse of a Matrix**

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2],[3,4]])
array([[1, 2],
       [3, 4]])
>>> linalg.inv(A)
array([[-2. ,  1. ],
          [ 1.5, -0.5]])
>>> A.dot(linalg.inv(A))
array([
[  1.00000000e+00, 0.00000000e+00],
[ 4.44089210e-16 , 1.00000000e+00]
])
```

- **Rounding Float Numbers**

```
>>> A = np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> b = np.array([[5],[6]])
>>> b
array([[5],
       [6]])
>>> np.linalg.solve(A,b)
array([[-4. ],
          [ 4.5]])
```

Sparse matrices (scipy.sparse)¶

- SciPy 2-D sparse matrix package for numeric data.

  https://docs.scipy.org/doc/scipy/reference/sparse.html

Sparse matrix classes

- **bsr_matrix**(arg1[, shape, dtype, copy, blocksize]) Block Sparse Row matrix
- **coo_matrix**(arg1[, shape, dtype, copy]) A sparse matrix in COOrdinate format.
- **csc_matrix**(arg1[, shape, dtype, copy]) Compressed Sparse Column matrix
- **csr_matrix**(arg1[, shape, dtype, copy]) Compressed Sparse Row matrix
- **dia_matrix**(arg1[, shape, dtype, copy]) Sparse matrix with DIAgonal storage
- **dok_matrix**(arg1[, shape, dtype, copy]) Dictionary Of Keys based sparse matrix.
- **lil_matrix**(arg1[, shape, dtype, copy]) Row-based linked list sparse matrix
- **spmatrix**([maxprint]) This class provides a base class for all sparse matrices.

Compressed Sparse Column matrix

- **scipy.sparse.csc_matrix**

  https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csc_matrix.html#scipy.sparse.csc_matrix

```
>>> import numpy as np
>>> from scipy.sparse import csc_matrix
>>> csc_matrix((3, 4), dtype=np.int8).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)

>>> row = np.array([0, 2, 2, 0, 1, 2])
>>> col = np.array([0, 0, 1, 2, 2, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> csc_matrix((data, (row, col)), shape=(3, 3)).toarray()
array([[1, 0, 4],
       [0, 0, 5],
       [2, 3, 6]])
```

Statistics (scipy.stats)

- A large number of **probability distributions** as well as a growing library of **statistical functions**
- **Distributions like:** norm, bernoulli, poisson
- **Statistical functions like:**
  - stats: Return mean, variance
  - PDF: Probability Density Function
  - PPF: Percent Point Function (Inverse of CDF)

- **Documentation and Tutorial**
  - Reference http://docs.scipy.org/doc/scipy/reference/stats.html
  - http://docs.scipy.org/doc/scipy/reference/tutorial/stats.html

Spark linalg

https://spark.apache.org/docs/latest/mllib-data-types.html

- Local vector
- Labeled point
- Local matrix
- Distributed matrix
  - RowMatrix
  - IndexedRowMatrix
  - CoordinateMatrix
  - BlockMatrix

Local vector

- MLlib recognizes the following types as dense vectors:
  - NumPy's array
  - Python's list, e.g., [1, 2, 3]
- and the following as sparse vectors:
  - MLlib's SparseVector.
  - SciPy's csc_matrix with a single column
- NumPy arrays are recommended over lists for efficiency, and using the factory methods implemented in Vectors to create sparse vectors.

- For example, a vector (1.0, 0.0, 3.0) can be represented in dense format as [1.0, 0.0, 3.0] or in sparse format as (3, [0, 2], [1.0, 3.0]), where 3 is the size of the vector.

Labeled point

- A labeled point is a local vector, either dense or sparse, associated with a label/response.
- Spark uses a double to store a label, so Spark can use labeled points in both regression and classification.
- For binary classification, a label should be either 0 (negative) or 1 (positive).

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense feature
vector.
pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])

# Create a labeled point with a negative label and a sparse feature
vector.
neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

Sparse data

- It is very common in practice to have sparse training data.

  MLlib supports reading training examples stored in **LIBSVM** format, which is the default format used by **LIBSVM** and **LIBLINEAR**.

  It is a text format in which each line represents a labeled sparse feature vector using the following format:

  label index1:value1 index2:value2 ...

Local Matrix

- A local matrix has integer-typed row and column indices and double-typed values, stored on a single machine.
- MLlib supports dense matrices, whose entry values are stored in a single double array in column-major order, and sparse matrices, whose non-zero entry values are stored in the Compressed Sparse Column (CSC) format in column-major order.

```
from pyspark.mllib.linalg import Matrix, Matrices


# Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
dm2 = Matrices.dense(3, 2, [1, 3, 5, 2, 4, 6])


# Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 2, 1], [9, 6, 8])
```

Distributed Matrix

- A distributed matrix has **long-typed row and column indices** and **double-typed values**, stored distributively in one or more RDDs.
- It is very important to choose the right format to store large and distributed matrices.
- Converting a distributed matrix to a different format may require a global shuffle, which is quite expensive.

- Four types of distributed matrices have been implemented so far.
  - **RowMatrix**
  - **IndexedRowMatrix**
  - **CoordinateMatrix**
  - **BlockMatrix**

TF-IDF

- Term Frequency–Inverse Document Frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus
- TF-IDF can create an embedding of the text documents: Vector of values that represents the properties of the textual document
- The approach is used in Natural Language Processing for:
  - Information retrieval
  - Text summarization & keyword extraction
  - Vectors & Word Embeddings

- Pros and cons of using TF-IDF
  - TF-IDF is simple and easy to use. It is simple to calculate, computationally cheap, and a simple starting point for similarity calculations (via TF-IDF vectorization + cosine similarity).
  - TF-IDF does not carry semantic meaning. It considers the importance of the words due to how it weighs them, but it cannot necessarily derive the contexts and understand importance that way.
  - TF-IDF ignores word order
  - TF-IDF can suffer from the curse of dimensionality. The length of TF-IDF vectors is equal to the vocabulary size.

Term frequency

•Term frequency, $\mathrm{tf}(t,d)$, is the relative frequency of term t within document d,

$$\mathrm{tf}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

- where $\mathrm{f}_{t,d}$ is the raw count of a term in a document, i.e., the number of times that term t occurs in document d.
- The denominator is the total number of terms in document *d*

Inverse document frequency

- The inverse document frequency measures the information provided by a given word regarding the prediction of the document.
  - If a given word exists only in a small number of documents, it provides a lot of information to identify the corresponding documents; contrary, it is common across all documents, then it cannot be used to determine the documents.
- It is the logarithmically scaled inverse fraction of the documents that contain the word
  - It is calculated by dividing the total number of documents by the number of documents containing the word and then taking the logarithm of that quotient).

$$\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

  - N: total number of documents in the corpus $N = |D|$
  - $|\{d \in D: t \in d\}|$: number of documents where the term t appears

Term frequency–inverse document frequency

- TF–IDF is calculated as

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D)$$

- A high value of tf–idf results from a high term frequency (in the given document), and a low document frequency of the term in the whole collection of documents;

- The value tends to filter out common terms.

TF-IDF example

- The following Google Sheet gives an example of
  - How the TF-IDF is calculated and
  - How it can be applied to predict the Category of a document using the KNN algorithm.
- This is a live example, create your own copy so you can play with the numbers.

- TF-IDF/KNN Example - Google Sheets
  - https://docs.google.com/spreadsheets/d/1SBYw38f_eC5ualJD_HBb0ekde2OF4e9iBdSVIM2hc1c/edit?usp=sharing