Clustering and Feature Extraction in MLlib

This tutorial goes over the background knowledge, API interfaces and sample code for clustering, feature extraction and data transformation algorithm in MLlib.

Clustering

MLlib supports K-means algorithm for clustering.

K-means Clustering

K-means Clustering partitions N data points into K clusters in which each data point belongs to the cluster with a nearest mean. A formal description is as follows:

Given a set of data points (x_1, x_2,..., x_n), k-means clustering aims to partition the n data points into k (<=n) sets S = {S_1, S_2, ..., S_k} so as to minimize the within-cluster sum of squares (WCSS). In other words, its objective is to find:

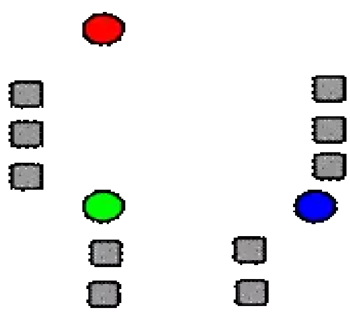
$$\arg min \sum_{i=1}^{K} \sum_{x \in S_i} ||x - \mu_i||^2$$

Where μ_i is the the mean of points in S_i

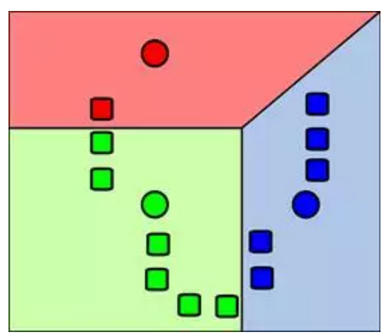
K-means Clustering is an NP hard problem and in reality is solved by heuristic algorithms. The most commonly used algorithm is [Standard algorithm](). Standard algorithm begins by assigning k random points in the domain as the mean of each cluster and then it iterates the following two steps until it reaches the convergence

1) Assigning each data point to the cluster with a nearest mean.

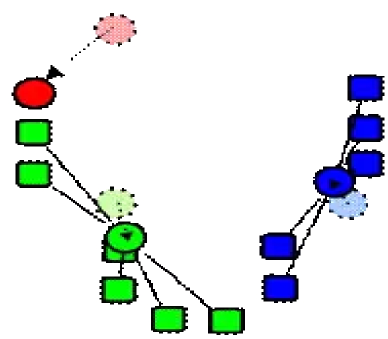
2) Update the mean of each cluster to be the centroid of data points assigned to that cluster.



1) k initial "means" (in this case k=3) are randomly generated within the data domain (shown in color).



2) K clusters are created by associating every data points with the nearest mean.



3) The centroid of each of the k clusters becomes the new mean.

Spark Mllib provides a clustering model that implements the K-means algorithm.

pyspark.mllib.clustering module

class **`pyspark.mllib.clustering.`**`KMeansModel`

Bases: **`object`**

A clustering model derived from the k-means method.

Method:

`clusterCenters`
    Get the cluster centers, represented as a list of NumPy arrays.

`predict`(*x*)
    Find the cluster to which x belongs in this model.

class ***`pyspark.mllib.clustering.`*** *`KMeans`*

Bases: **`object`**

Method:

train(*data*, *k*, *maxIterations=100*, *runs=1*, *initializationMode='k-means||'*)

Train a k-means clustering model.

k is the number of desired clusters.

maxIterations is the maximum number of iterations to run.

runs is the number of times to run the k-means algorithm (k-means is not guaranteed to find a globally optimal solution, and when run multiple times on a given dataset, the algorithm returns the best clustering result).

initialization mode can be either 'random'or 'k-meansII', 'random' means randomly generate K nodes in the domain to be the initial mean for each cluster while k-means|| generate initial mean by using kmeans||

Sample Code:

```
from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans
from numpy import array
from math import sqrt

sc = SparkContext()

#4 data points (0.0, 0.0), (1.0, 1.0), (9.0, 8.0) (8.0, 9.0)
data = array([0.0,0.0, 1.0,1.0, 9.0,8.0, 8.0,9.0]).reshape(4,2)

#Generate K means
model = KMeans.train(sc.parallelize(data), 2, maxIterations=10, runs=30, initializationMode="random")

#Print out the cluster of each data point
print (model.predict(array([0.0, 0.0])))
print (model.predict(array([1.0, 1.0])))
print (model.predict(array([9.0, 8.0])))
print (model.predict(array([8.0, 0.0])))
```

Output:

```
0
0 #First two nodes belong to cluster 0
1
1 #Last two nodes belongs to cluster 1
```

Feature Extraction

Feature Extraction converts vague features in the raw data into concrete numbers for further analysis. In this section, we introduce two feature extraction technologies: TF-IDF and Word2Vec.

TF-IDF

Term frequency-inverse document frequency (TF-IDF) reflects the importance of a term (word) to the document in corpus. Denote a term by t , a document by d, and the corpus by D. Term frequency $TF(t,d)$ is the number of times that term t appears in d while document frequency $DF(t,D)$ is the number of documents that contain the term.

If we only use term frequency to measure the importance, it is very easy to over-emphasize terms that appear very often but carry little information about the document, e.g., 'a', 'the', and 'of'. If a term appears very often across the corpus, it means it does not carry special information about a particular document. Inverse document frequency is a numerical measure of how much information a term provides:

$$IDF(t,D) = \log\frac{|D|+1}{DF(t,D)+1}$$

where $|D|$ is the total number of documents in the corpus. A smoothing term is applied to avoid dividing by zero for terms outside the corpus.

The TF-IDF measure is simply the product of TF and IDF:

$$TFIDF(t,d,D) = TF(t,d) \times IDF(t,D)$$

pyspark.mllib.feature module

class `pyspark.mllib.feature.`**`HashingTF`**

Bases: **object**

Maps a sequence of terms to their term frequencies using hashing algorithm.

Method:

indexOf(*term*)

Returns the index of the input term.

transform(*document*)

Transforms the input document (list of terms) to term frequency vectors, or transform the RDD of document to RDD of term frequency vectors.

class pyspark.mllib.feature.**IDFModel**

Bases: **pyspark.mllib.feature.JavaVectorTransformer**

Represents an IDF model that can transform term frequency vectors.

Method:

transform(*dataset*)

Transforms term frequency (TF) vectors to TF-IDF vectors.

If *minDocFreq* was set for the IDF calculation, the terms which occur in fewer than *minDocFreq* documents will have an entry of 0.

Parameters: **dataset** an RDD of term frequency vectors

Returns: an RDD of TF-IDF vectors

class pyspark.mllib.feature.**IDF** (*minDocFreq=0*)

Bases: **object**

Inverse document frequency (IDF).

The standard formulation is used: idf = log((m + 1) / (d(t) + 1)), where m is the total number of documents and d(t) is the number of documents that contain term t. This implementation supports filtering out terms which do not appear in a minimum number of documents (controlled by the variable minDocFreq). For terms that are not in at least minDocFreq documents, the IDF is found as 0, resulting in TF-IDFs of 0.

Method:

fit(*dataset*)

Computes the inverse document frequency.

Parameters: **dataset** an RDD of term frequency vectors

Sample Code:

```
from pyspark import SparkContext
from pyspark.mllib.feature import HashingTF
from pyspark.mllib.feature import IDF

sc = SparkContext()

# Load documents (one per line).
documents = sc.textFile("data/mllib/document").map(lambda line: line.split(" "))

#Computes TF
hashingTF = HashingTF()
tf = hashingTF.transform(documents)

#Computes tfidef
tf.cache()
idf = IDF().fit(tf)
tfidf = idf.transform(tf)

for r in tfidf.collect(): print r
```

Data in document:

```
1 1 1 1
1 2 2 2
```

Output:

```
(1048576, [485808], [0.0])
# 1048576 and [485808] are total numbers of hash bracket and the hash bracket for this element respectively
# 0.0 is the TFIDF for word '1' in document 1.
(1048576, [485808, 559923], [0.0, 1.21639532432])
# 0.0 and 1.21639532432 is the TFIDF for word '1' and word '2' in document 2.
```

Word2Vec

Word2Vec converts each word in documents into a vector. This technology is useful in many natural language processing applications such as named entity recognition, disambiguation, parsing, tagging and machine translation.

Mllib uses skip-gram model that is able to convert word in similar contexts into vectors that are close in vector space. Given a large dataset, skip-gram model can predict synonyms of a word with very high accuracy.

pyspark.mllib.feature module

class pyspark.mllib.feature.**Word2Vec**

Bases: **object**

Word2Vec creates vector representation of words in a text corpus.

Word2Vec used skip-gram model to train the model.

Method:

fit(*data*)

Computes the vector representation of each word in vocabulary.

Parameters: **data** training data. RDD of list of string

Returns: Word2VecModel instance

setLearningRate(*learningRate*)

Sets initial learning rate (default: 0.025).

setNumIterations(*numIterations*)

Sets number of iterations (default: 1), which should be smaller than or equal to number of partitions.

setNumPartitions(*numPartitions*)

Sets number of partitions (default: 1). Use a small number for accuracy.

setSeed(*seed*)

Sets random seed.

setVectorSize(*vectorSize*)

Sets vector size (default: 100).

class pyspark.mllib.feature.**Word2VecModel**

Bases: **pyspark.mllib.feature.JavaVectorTransformer**

class for Word2Vec model

Method:

findSynonyms(*word*, *num*)

Find synonyms of a word

Note: local use only

Parameters: **word** a word or a vector representation of word

num number of synonyms to find

Returns: array of (word, cosineSimilarity)

transform(*word*)

Transforms a word to its vector representation

Note: local use only

Parameters: **word** a word

Returns: vector representation of word(s)

Sample Code:

```
from pyspark import SparkContext
from pyspark.mllib.feature import Word2Vec

#Pippa Passes
sentence = "The year is at the spring \
        And the day is at the morn; \
        Morning is at seven;  \
        The hill-side is dew-pearled; \
            The lark is on the wing; \
            The snai is on the thorn; \
            God's in His heaven; \
        All's right with the world "

sc = SparkContext()

#Generate doc
localDoc = [sentence, sentence]
doc = sc.parallelize(localDoc).map(lambda line: line.split(" "))

#Convect word in doc to vectors.
model = Word2Vec().fit(doc)

#Print the vector of "The"
vec = model.transform("The")
print vec
```

```
#Find the synonyms of "The"
syms = model.findSynonyms("The", 5)
print [s[0] for s in syms]
```

Output:

```
[-0.00352853513323,0.00335159664974,-0.00598029373214,0.00399478571489,-0.00198440207168,-0.00294396048412,-0
05,0.000757675385103,-0.00189483352005,-0.00201138551347,0.00030658338801,0.00328158447519,-0.00367985945195,
05,-0.00372383627109,0.00685756560415,0.00612043589354,-0.000518668384757,0.000620941573288,0.00244942889549,
[u'', u'the', u'\t', u'is', u'at'] #The synonyms of "The"
```

Data Transformation

Data Transformation manipulates values in each dimension of vectors according to a predefined rule. Vectors that have gone through transformation can be used for future processing.

We introduce two types of data transformation: StandardScaler and Normalizer in this section.

StandardScaler

StandardScaler makes vectors in the dataset have zero-mean (when subtracting the mean in the enumerator) and unit-variance.

pyspark.mllib.feature module

class pyspark.mllib.feature.**StandardScalerModel**

Bases: **pyspark.mllib.feature.JavaVectorTransformer**

Represents a StandardScaler model that can transform vectors.

Method:

transform(*vector*)

Applies standardization transformation on a vector.

Parameters: **vector** Vector or RDD of Vector to be standardized.

Returns: Standardized vector. If the variance of a column is zero, it will return default *0.0* for the column with zero variance.

class pyspark.mllib.feature.**StandardScaler**(*withMean=False*, *withStd=True*)

Bases: **object**

Standardizes features by removing the mean and scaling to unit variance using column summary statistics on the samples in the training set.

If withMean is true, all the dimension of each vector subtract the mean of this dimension.

If withStd is true, all the dimension of each vector divides the length of the vector.

Method:

fit(*dataset*)

Computes the mean and variance and stores as a model to be used for later scaling.

Parameters: **data** The data used to compute the mean and variance to build the transformation model.

Returns: a StandardScalarModel

Sample Code:

```
from pyspark.mllib.feature import Normalizer
from pyspark.mllib.linalg import Vectors
from pyspark import SparkContext
from pyspark.mllib.feature import StandardScaler

sc = SparkContext()

vs = [Vectors.dense([-2.0, 2.3, 0]), Vectors.dense([3.8, 0.0, 1.9])]

dataset = sc.parallelize(vs)

#all false, do nothing.
standardizer = StandardScaler(False, False)
model = standardizer.fit(dataset)
```

```
result = model.transform(dataset)
for r in result.collect(): print r

print("\n")

#deducts the mean
standardizer = StandardScaler(True, False)
model = standardizer.fit(dataset)
result = model.transform(dataset)
for r in result.collect(): print r

print("\n")

#divides the length of vector
standardizer = StandardScaler(False, True)
model = standardizer.fit(dataset)
result = model.transform(dataset)
for r in result.collect(): print r

print("\n")

#Deducts min first, divides the length of vector later
standardizer = StandardScaler(True, True)
model = standardizer.fit(dataset)
result = model.transform(dataset)
for r in result.collect(): print r

print("\n")
```

Output:

```
#all false, do nothing.
[-2.0,2.3,0.0]
[3.8,0.0,1.9]

#deducts the mean
[-2.9,1.15,-0.95]
[2.9,-1.15,0.95]

#divides the length of vector
[-0.487659849094,1.41421356237,0.0]
[0.926553713279,0.0,1.41421356237]

#Deducts min first, divides the length of vector later
[-0.707106781187,0.707106781187,-0.707106781187]
[0.707106781187,-0.707106781187,0.707106781187]
```

Normalizer

Normalizer scales vectors by divide each dimension of the vector with a L^p norm.

For 1 <= p <= infinite, L^p norm is calculated as follows: sum(abs(vector)^p)$^{(1/p)}$.

For p = infinite, L^p norm is max(abs(vector))

pyspark.mllib.feature module

class `pyspark.mllib.feature.`**`Normalizer`**(*p=2.0*)

Bases: **`pyspark.mllib.feature.VectorTransformer`**

Method:

`transform`(*vector*)

Applies unit length normalization on a vector.

Parameters: **vector** vector or RDD of vector to be normalized.

Returns: normalized vector. If the norm of the input is zero, it will return the input vector.

Sample Code:

```
from pyspark.mllib.feature import Normalizer
from pyspark.mllib.linalg import Vectors
from pyspark import SparkContext

sc = SparkContext()

# v = [0.0, 1.0, 2.0]
```

```
v = Vectors.dense(range(3))

# p = 1
nor = Normalizer(1)
print (nor.transform(v))

# p = 2
nor = Normalizer(2)
print (nor.transform(v))

# p = inf
nor = Normalizer(p=float("inf"))
print (nor.transform(v))
```

Output:

```
[0.0, 0.3333333333, 0.666666667]
[0.0, 0.4472135955, 0.894427191]
[0.0, 0.5, 1.0]
```