Spark and Machine Learning

Feng Li

feng.li@cufe.edu.cn

School of Statistics and Mathematics
Central University of Finance and Economics

Revised on December 20, 2017

Today we are going to learn...

1. Linear Methods
2. Clustering
3. Dimensionality Reduction
4. Feature Extraction and Transformation
5. Spark Streaming

Linear Methods I

- **Logistic regression** Logistic regression is widely used to predict a binary response. MLlib implemented two algorithms to solve logistic regression: mini-batch gradient descent and L-BFGS. We recommend L-BFGS over mini-batch gradient descent for faster convergence.

```
from pyspark.mllib.classification import \
     LogisticRegressionWithLBFGS, LogisticRegressionModel
from pyspark.mllib.regression import LabeledPoint

# Load and parse the data
def parsePoint(line):
    values = [float(x) for x in line.split(' ')]
    return LabeledPoint(values[0], values[1:])
data = sc.textFile("data/mllib/sample_svm_data.txt")
parsedData = data.map(parsePoint)
# Build the model
model = LogisticRegressionWithLBFGS.train(parsedData)
# Evaluating the model on training data
labelsAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
trainErr = labelsAndPreds.filter(lambda lp: lp[0] != lp[1]).count() / \
                                    float(parsedData.count())
print("Training Error = " + str(trainErr))
# Save and load model
model.save(sc, "target/tmp/pythonLogisticRegressionWithLBFGSModel")
sameModel = LogisticRegressionModel.load(sc, \
            "target/tmp/pythonLogisticRegressionWithLBFGSModel")
```

Linear Methods II

- **Linear Regression** `LinearRegressionWithSGD` is used to build a simple linear model to predict label values. We compute the mean squared error at the end to evaluate goodness of fit. Note that the Python API does not yet support model save/load but will in the future.

```
from pyspark.mllib.regression import LabeledPoint, \
                        LinearRegressionWithSGD, LinearRegressionModel

# Load and parse the data
def parsePoint(line):
    values = [float(x) for x in line.replace(',', ' ').split(' ')]
    return LabeledPoint(values[0], values[1:])

data = sc.textFile("data/mllib/ridge-data/lpsa.data")
parsedData = data.map(parsePoint)

# Build the model
model = LinearRegressionWithSGD.train(parsedData, iterations=100, step=0.00000001)

# Evaluate the model on training data
valuesAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
MSE = valuesAndPreds \
    .map(lambda vp: (vp[0] - vp[1])**2) \
    .reduce(lambda x, y: x + y) / valuesAndPreds.count()
print("Mean Squared Error = " + str(MSE))

# Save and load model
```

Linear Methods III

```
model.save(sc, "target/tmp/pythonLinearRegressionWithSGDModel")
sameModel = LinearRegressionModel.load(sc, "target/tmp/pythonLinearRegressionWithSGDModel"
```

Clustering I

- **K-means** K-means is one of the most commonly used clustering algorithms that clusters the data points into a predefined number of clusters. The **spark.mllib** implementation includes a parallelized variant of the `k-means++` method called `kmeans||`.

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

# Load and parse the data
data = sc.textFile("data/mllib/kmeans_data.txt")
parsedData = data.map(lambda line: array([float(x) for x in line.split(' ')]))
# Build the model (cluster the data)
clusters = KMeans.train(parsedData, 2, maxIterations=10, initializationMode="random")
# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(point):
    center = clusters.centers[clusters.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))
WSSSE = parsedData.map(lambda point: error(point)).reduce(lambda x, y: x + y)
print("Within Set Sum of Squared Error = " + str(WSSSE))
# Save and load model
clusters.save(sc, "target/org/apache/spark/PythonKMeansExample/KMeansModel")
sameModel = KMeansModel.load(sc,\
            "target/org/apache/spark/PythonKMeansExample/KMeansModel")
```

Clustering II

- **Gaussian mixture** A Gaussian Mixture Model represents a composite distribution whereby points are drawn from one of k Gaussian sub-distributions, each with its own probability. The `spark.mllib` implementation uses the expectation-maximization algorithm to induce the maximum-likelihood model given a set of samples.

```
from numpy import array
from pyspark.mllib.clustering import GaussianMixture, GaussianMixtureModel
# Load and parse the data
data = sc.textFile("data/mllib/gmm_data.txt")
parsedData = data.map(lambda line: array([float(x) for x in line.strip().split(' ')]))
# Build the model (cluster the data)
gmm = GaussianMixture.train(parsedData, 2)
# Save and load model
gmm.save(sc, "target/org/apache/spark/PythonGaussianMixtureExample/GaussianMixtureModel")
sameModel = GaussianMixtureModel\
    .load(sc, "target/org/apache/spark/PythonGaussianMixtureExample/GaussianMixtureModel")
# output parameters of model
for i in range(2):
    print("weight = ", gmm.weights[i], "mu = ", gmm.gaussians[i].mu,
          "sigma = ", gmm.gaussians[i].sigma.toArray())
```

Clustering III

- **Latent Dirichlet allocation** Latent Dirichlet allocation (LDA) is a topic model which infers topics from a collection of text documents.

```
from pyspark.mllib.clustering import LDA, LDAModel
from pyspark.mllib.linalg import Vectors
# Load and parse the data
data = sc.textFile("data/mllib/sample_lda_data.txt")
parsedData = data.map(lambda line: Vectors.dense([float(x) for x in line.strip().split(' '
# Index documents with unique IDs
corpus = parsedData.zipWithIndex().map(lambda x: [x[1], x[0]]).cache()
# Cluster the documents into three topics using LDA
ldaModel = LDA.train(corpus, k=3)
# Output topics. Each is a distribution over words (matching word count vectors)
print("Learned topics (as distributions over vocab of " + str(ldaModel.vocabSize())
      + " words):")
topics = ldaModel.topicsMatrix()
for topic in range(3):
    print("Topic " + str(topic) + ":")
    for word in range(0, ldaModel.vocabSize()):
        print(" " + str(topics[word][topic]))
# Save and load model
ldaModel.save(sc, \
    "target/org/apache/spark/PythonLatentDirichletAllocationExample/LDAModel")
sameModel = LDAModel\
    .load(sc, \
     "target/org/apache/spark/PythonLatentDirichletAllocationExample/LDAModel")
```

Dimensionality Reduction I

- **Singular value decomposition** `spark.mllib` provides SVD functionality to row-oriented matrices, provided in the `RowMatrix` class.

```
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.linalg.distributed import RowMatrix

rows = sc.parallelize([
    Vectors.sparse(5, {1: 1.0, 3: 7.0}),
    Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),
    Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0)
])

mat = RowMatrix(rows)

# Compute the top 5 singular values and corresponding singular vectors.
svd = mat.computeSVD(5, computeU=True)
U = svd.U       # The U factor is a RowMatrix.
s = svd.s       # The singular values are stored in a local dense vector.
V = svd.V       # The V factor is a local dense matrix.
```

Dimensionality Reduction II

- **Principal component analysis** Principal component analysis (PCA) is a statistical method to find a rotation such that the first coordinate has the largest variance possible, and each succeeding coordinate in turn has the largest variance possible.

```
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.linalg.distributed import RowMatrix

rows = sc.parallelize([
    Vectors.sparse(5, {1: 1.0, 3: 7.0}),
    Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),
    Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0)
])

mat = RowMatrix(rows)
# Compute the top 4 principal components.
# Principal components are stored in a local dense matrix.
pc = mat.computePrincipalComponents(4)

# Project the rows to the linear space spanned by the top 4 principal components.
projected = mat.multiply(pc)
```

Feature Extraction and Transformation I

- **Term frequency-inverse document frequency** (TF-IDF) is a feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus.

```
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame([
    (0.0, "Hi I heard about Spark"),
    (0.0, "I wish Java could use case classes"),
    (1.0, "Logistic regression models are neat")
], ["label", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
wordsData = tokenizer.transform(sentenceData)

hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=20)
featurizedData = hashingTF.transform(wordsData)
# alternatively, CountVectorizer can also be used to get term frequency vectors

idf = IDF(inputCol="rawFeatures", outputCol="features")
idfModel = idf.fit(featurizedData)
rescaledData = idfModel.transform(featurizedData)

rescaledData.select("label", "features").show()
```

Feature Extraction and Transformation II

- **Word2Vec** is an Estimator which takes sequences of words representing documents and trains a `Word2VecModel`. The model maps each word to a unique fixed-size vector. The `Word2VecModel` transforms each document into a vector using the average of all words in the document; this vector can then be used as features for prediction, document similarity calculations, etc.

```
from pyspark.ml.feature import Word2Vec

# Input data: Each row is a bag of words from a sentence or document.
documentDF = spark.createDataFrame([
    ("Hi I heard about Spark".split(" "), ),
    ("I wish Java could use case classes".split(" "), ),
    ("Logistic regression models are neat".split(" "), )
], ["text"])

# Learn a mapping from words to Vectors.
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text", outputCol="result")
model = word2Vec.fit(documentDF)

result = model.transform(documentDF)
for row in result.collect():
    text, vector = row
    print("Text: [%s] => \nVector: %s\n" % (", ".join(text), str(vector)))
```

Feature Extraction and Transformation III

- An **n-gram** is a sequence of n tokens (typically words) for some integer n. The `NGram` class can be used to transform input features into n-grams.

```
from pyspark.ml.feature import NGram

wordDataFrame = spark.createDataFrame([
    (0, ["Hi", "I", "heard", "about", "Spark"]),
    (1, ["I", "wish", "Java", "could", "use", "case", "classes"]),
    (2, ["Logistic", "regression", "models", "are", "neat"])
], ["id", "words"])

ngram = NGram(n=2, inputCol="words", outputCol="ngrams")

ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.select("ngrams").show(truncate=False)
```

Spark Streaming

- http://spark.apache.org/docs/latest/streaming-programming-guide.html