**STATISTICS AND TRANSFORMATIONS SPARKMLLIB**

**# Imports**

**from pyspark.mllib.linalg import Vectors**

**from pyspark.mllib.feature import ElementwiseProduct**

**from pyspark.mllib.stat import Statistics**

**from pyspark.mllib.feature import StandardScaler**

**# TODO:**

**# Load the raw data into the rawdata RDD**

**rawdata = sc.textFile("hdfs:///user/training/mldata/concrete.csv")**

**# TODO:**

**# Convert the rawdata RDD to an RDD of dense vectors called vecrdd**

**# Be sure to map the split values to a float before casting to a dense vector.**

**vecrdd = rawdata.map(lambda x: Vectors.dense([float(i) for i in x.split(',')]))**

**# TODO:**

**# Create a dense vector of weights, naming the vector "weights"**

**# Make sure that this vector has a weight for each of the 9 columns of the dataset.**

**weights = Vectors.dense([0.3, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 1.0])**

**# TODO:**

**# Create an instance of "ElementwiseProduct"**

**# Initialize the instance with the weights vector and store it in a variable called "ep"**

**ep = ElementwiseProduct(weights)**

**# TODO**

**# Transform vecrdd and store the output in an RDD called "weighted".**

**# Then print the first row of vecrdd and the weighted.**

**weighted = ep.transform(vecrdd)**

**print weighted.take(1)**

**print vecrdd.take(1)**

**# TODO:**

**# Compute basic statistics on the vecrdd using Statistics.colStats()**

**# Print the mean, variance, and numNonZeros**

**stats = Statistics.colStats(vecrdd)**

**print stats.mean()**

**print stats.variance()**

**print stats.numNonzeros()**

**ss = StandardScaler(withMean = True, withStd = True)**

**ss\_model = ss.fit(vecrdd)**

**scaled = ss\_model.transform(vecrdd)**

**stats = Statistics.colStats(scaled)**

**print stats.mean()**

**print stats.variance()**

**print stats.numNonzeros()**

**LINEAR REGRESSION SPARKMLLIB**

**# Imports**

**from pyspark.mllib.linalg import Vectors**

**from pyspark.mllib.feature import StandardScaler**

**from pyspark.mllib.regression import LinearRegressionWithSGD, LabeledPoint**

**# TODO:**

**# Load the data from the delimited file "concrete.csv" into the rdd rawdata**

**rawdata = sc.textFile('hdfs:///user/training/mldata/concrete\_train.csv')**

**# TODO:**

**# Map the RDD of strings to an RDD of dense vectors by splitting**

**# on commas, casting each value as floating point, then casting as a dense vector**

**vecrdd = rawdata.map(lambda x: Vectors.dense([float(i) for i in x.split(',')]))**

**# TODO:**

**# Create an instance of StandardScaler, setting withMean and withStd both True.**

**# Use it to scale the vector RDD create above, and store in a new RDD called scaled**

**# Note: Be sure to store the model created by the fit method of the StandardScaler**

**# in a variable as it will be used in a later step**

**ss = StandardScaler(withMean=True, withStd=True)**

**model = ss.fit(vecrdd)**

**scaled = model.transform(vecrdd)**

**# TODO:**

**# Create an RDD of labeledPoints using the last value in the list**

**# as the feature and all but the last value in the list as the features.**

**# Be sure to explicitly cast the features as a dense vector before**

**# using in the labeledPoint initialization.**

**lprdd = scaled.map(lambda x: LabeledPoint(label = x[-1], features=Vectors.dense(x[:-1])))**

**# TODO:**

**# Create three LinearRegressionWithSGD models**

**# by calling the train method with different parameters on the RDD of labeled points.**

**# The first should have no regularization, the second should use l1 regularization,**

**# and the third should use l2 regularization.**

**# For both l1 and l2 regularization, set the regParam to 0.2**

**lr\_model = LinearRegressionWithSGD.train(lprdd)**

**lr\_l1\_model = LinearRegressionWithSGD.train(lprdd, regType = 'l1', regParam = 0.2)**

**lr\_l2\_model = LinearRegressionWithSGD.train(lprdd, regType = 'l2', regParam = 0.2)**

**# TODO:**

**# Create a new RDD of labeled points called "test\_lprdd" from the "concrete\_test.csv" data**

**# using the same exact method used to create the "lprdd" RDD above.**

**# Be sure to use the same StandardScaler model used to scale the training data,**

**# i.e. do not create a new instance of StandardScaler for this step.**

**test\_rawdata = sc.textFile('hdfs:///user/training/mldata/concrete\_test.csv')**

**test\_vecrdd = test\_rawdata.map(lambda x: Vectors.dense([float(i) for i in x.split(',')]))**

**test\_scaled = model.transform(test\_vecrdd)**

**test\_lprdd = test\_scaled.map(lambda x: LabeledPoint(label = x[-1], features=Vectors.dense(x[:-1])))**

**# TODO:**

**# Create RDDs of predictions for all three models**

**pred\_lr = test\_lprdd.map(lambda x: (x.label, lr\_model.predict(x.features)))**

**pred\_lr.collect()**

**pred\_lr\_l1 = test\_lprdd.map(lambda x: (x.label, lr\_l1\_model.predict(x.features)))**

**pred\_lr\_l1.collect()**

**pred\_lr\_l2 = test\_lprdd.map(lambda x: (x.label, lr\_l2\_model.predict(x.features)))**

**pred\_lr\_l2.collect()**

**TRANSFORMATIONS WITH SPARKML**

**# Imports**

**from pyspark.mllib.linalg import Vectors**

**from pyspark.ml.feature import StandardScaler, PolynomialExpansion**

**from pyspark.ml import Pipeline**

**from pyspark.sql import Row**

**# TODO:**

**# Load the raw data from concrete.csv into the "rawdata" RDD**

**rawdata = sc.textFile("hdfs:///user/training/mldata/concrete.csv")**

**# TODO:**

**# Split the rawdata rdd on commas and save in an rdd called "splits"**

**# and map the split values to a float**

**splits = rawdata.map(lambda x: [float(i) for i in x.split(',')])**

**# TODO:**

**# Create a new DataFrame called "df" with two columns called "features"**

**# and "labels". The "labels" column should contain the last element of**

**# the lists in "concretesplit" and the "features" column should contain**

**# all other elements.**

**# (Hint: make sure to cast the features to a dense vector)**

**df = splits.map(lambda x: Row(labels=float(x[-1]), features=Vectors.dense(x[:-1]))).toDF()**

**# TODO:**

**# Create an instance of "StandardScaler" named "ss" to scale**

**# the "features" column. Output to a new column called "scaledfeatures"**

**ss = StandardScaler(withStd = True, withMean=True, inputCol = 'features', outputCol = 'scaledfeatures')**

**# TODO:**

**# Create an instance of "PolynomialExpansion" named "pe" to transform**

**# the "scaledfeatures" column and output to a new column called "expandedfeatures".**

**# Set "degree" = 2 when initializing.**

**pe = PolynomialExpansion(degree=2, inputCol = 'scaledfeatures', outputCol='expandedfeatures')**

**# TODO:**

**# Create a Pipeline estimator**

**# Use "ss" as the first stage and "pe" as the second stage.**

**pl = Pipeline(stages = [ss,pe])**

**# TODO:**

**# Call the "fit" method of the "pl" estimator on "df" to create a**

**# "PipelineModel" and then save it in a variable called "model".**

**model = pl.fit(df)**

**# TODO:**

**# Create a new DataFrame called "transformed" by calling the "transform"**

**# method of "model" on "df". Then print the columns using the "columns"**

**# attribute of the "transformed" dataframe.**

**transformed = model.transform(df)**

**print transformed.columns**

**DECISION TREE CLASSIFIERS WITH SPARKML**

**# Imports**

**from pyspark.mllib.linalg import Vectors**

**from pyspark.ml import Pipeline**

**from pyspark.sql import Row**

**from pyspark.ml.feature import StringIndexer, VectorAssembler**

**from pyspark.ml.classification import DecisionTreeClassifier**

**# TODO:**

**# Load the data from the delimited file "cars\_train.csv" into the rdd rawdata**

**# Notice the triple forward slashes**

**rawdata = sc.textFile("hdfs:///user/training/mldata/cars\_train.csv")**

**# TODO:**

**# Map the RDD of strings to an RDD of lists by splitting on commas**

**lrdd = rawdata.map(lambda x: x.split(','))**

**# TODO:**

**# Create an RDD of Rows by supplying the name of each column**

**# and the corresponding element from the lists in the lrdd.**

**# Recall that the rownames are buying, maint, doors, persons,**

**# lugboots, safety, and label.**

**# Also recall that the syntax for initializing a row is:**

**# Row(columnname=value)**

**# e.g.,:**

**# Row(firstcolname = x[0], secondcolname = x[1])**

**rowrdd = lrdd.map(lambda x: Row(**

**buying=x[0],**

**maint=x[1],**

**doors=x[2],**

**persons=x[3],**

**lugboots=x[4],**

**safety=x[5],**

**label=x[6]**

**))**

**# Convert rowrdd to a DataFrame, called "train\_df"**

**train\_df = rowrdd.toDF()**

**# TODO:**

**# Create 7 distinct StringIndexer transformers with the outputCol**

**# parameter set to be the name of the input column appended with the**

**# string "\_ix"**

**#**

**# e.g.,:**

**# MyStringIndexer = StringIndexer(inputCol = 'thiscol', outputCol = 'thiscol\_ix')**

**#**

**# Additionally, store the names of the transformed feature columns**

**# (excluding the "label\_ix" column) in an list named "indexedcols"**

**indexedcols = ["buying\_ix", "maint\_ix", "doors\_ix", "persons\_ix", "lugboots\_ix", "safety\_ix"]**

**si1 = StringIndexer(inputCol="buying",outputCol="buying\_ix")**

**si2 = StringIndexer(inputCol="maint",outputCol="maint\_ix")**

**si3 = StringIndexer(inputCol="doors",outputCol="doors\_ix")**

**si4 = StringIndexer(inputCol="persons",outputCol="persons\_ix")**

**si5 = StringIndexer(inputCol="lugboots",outputCol="lugboots\_ix")**

**si6 = StringIndexer(inputCol="safety",outputCol="safety\_ix")**

**si7 = StringIndexer(inputCol="label",outputCol="label\_ix")**

**# TODO:**

**# Create a VectorAssembler transformer to combine all of the indexed**

**# categorical features into a vector. Provide the "indexedcols" list**

**# created above as the inputCols parameter, and name the outputCol "features".**

**va = VectorAssembler(inputCols = indexedcols, outputCol = 'features')**

**# TODO:**

**# Create a DecisionTreeClassifier, setting the label column to your**

**# indexed label column ("label\_ix") and the features column to the**

**# newly created column from the VectorAssembler above ("features").**

**# Store the new StringIndexer transformers, the VectorAssembler,**

**# as well as the DecisionTreeClassifier in a list called "steps"**

**clf = DecisionTreeClassifier(labelCol = 'label\_ix', featuresCol = 'features')**

**steps = [si1, si2, si3, si4, si5, si6, si7, va, clf]**

**# TODO:**

**# Create a ML pipeline named "pl" using the steps list to set the stages parameter**

**pl = Pipeline(stages=steps)**

**# TODO:**

**# Run the fit method of the pipeline on the DataFrame**

**# "train\_df" to create a pipeline model, and save the**

**# model in a new variable called "plmodel"**

**plmodel = pl.fit(train\_df)**

**# TODO:**

**# Create a new DataFrame called "test\_df" from the "cars\_test.csv" data.**

**# using the same exact method used to create the "train\_df" DataFrame above.**

**test\_df = sc.textFile('hdfs:///user/training/mldata/cars\_test.csv').map(lambda x: x.split(',')).map(lambda x: Row(**

**buying=x[0],**

**maint=x[1],**

**doors=x[2],**

**persons=x[3],**

**lugboots=x[4],**

**safety=x[5],**

**label=x[6])).toDF()**

**# TODO:**

**# Run the transform method of the pipeline model created above**

**# on the "test\_df" DataFrame to create a new DataFrame called "predictions"**

**predictions = plmodel.transform(test\_df)**

**# Compare the first 15 values in the "prediction" column with those**

**# rows' values in the "label\_ix" column**

**predictions.select("label\_ix", "prediction").show(15)**

**CLUSTERING WITH K-MEANS IN SPARKML**

**# Imports**

**from pyspark.mllib.linalg import Vectors**

**from pyspark.ml.feature import StandardScaler**

**from pyspark.ml import Pipeline**

**from pyspark.sql import Row**

**from pyspark.ml.clustering import KMeans**

**# TODO:**

**# Load the data from the delimited file "wine.csv" into the rdd rawdata**

**rawdata = sc.textFile("hdfs:///user/training/mldata/wine.csv")**

**# TODO:**

**# Create an RDD of dense vectors by splitting on commas, casting to float,**

**# then casting to a dense vector**

**vecrdd = rawdata.map(lambda x: Vectors.dense([float(i) for i in x.split(",")]))**

**# TODO:**

**# Convert to a DataFrame with a single column called "features"**

**df = vecrdd.map(lambda x: Row(features = x)).toDF()**

**# TODO:**

**# Create a StandardScaler transformer that will**

**# scale and center the data, taking as input the**

**# features column and setting the output column**

**# to be called "scaled"**

**ss = StandardScaler(**

**withMean=True,**

**withStd = True,**

**inputCol = 'features',**

**outputCol = 'scaled')**

**# TODO:**

**# Create a KMeans estimator, setting the featuresCol**

**# to "scaled" and setting 3 clusters**

**km = KMeans(k = 3, featuresCol = 'scaled')**

**# TODO:**

**# Create a pipeline with two stages: the StandardScaler**

**# transformer and the KMeans estimator created above**

**pl = Pipeline(stages = [ss,km])**

**# TODO:**

**# Chain methods of the pipeline (and the resulting pipeline model)**

**# to create an RDD of predictions using only one line**

**clustered = pl.fit(df).transform(df)**

**# TODO:**

**# Select only the prediction column, collect all**

**# predictions to the driver and print them to screen**

**for line in clustered.select("prediction").collect():**

**print line**

**SVM WITH SPARKMLLIB**

**from pyspark.mllib.classification import SVMWithSGD, SVMModel**

**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 = SVMWithSGD.train(parsedData, iterations=100)**

**# Evaluating the model on training data**

**labelsAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))**

**trainErr = labelsAndPreds.filter(lambda (v, p): v != p).count() / float(parsedData.count())**

**print("Training Error = " + str(trainErr))**

**# Save and load model**

**model.save(sc, "target/tmp/pythonSVMWithSGDModel")**

**sameModel = SVMModel.load(sc, "target/tmp/pythonSVMWithSGDModel**

**NAÏVE BAYES WITH SPARKMLLIB**

**from pyspark.mllib.classification import NaiveBayes, NaiveBayesModel**

**from pyspark.mllib.util import MLUtils**

**import shutil**

**# Load and parse the data file.**

**data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")**

**# Split data approximately into training (60%) and test (40%)**

**training, test = data.randomSplit([0.6, 0.4])**

**# Train a naive Bayes model.**

**model = NaiveBayes.train(training, 1.0)**

**# Make prediction and test accuracy.**

**predictionAndLabel = test.map(lambda p: (model.predict(p.features), p.label))**

**accuracy = 1.0 \* predictionAndLabel.filter(lambda (x, v): x == v).count() / test.count()**

**#print('model accuracy {}'.format(accuracy)) # python 2.7 and up**

**print('model accuracy: ' + str(accuracy))**

**# Save and load model**

**output\_dir = 'target/tmp/myNaiveBayesModel'**

**shutil.rmtree(output\_dir, ignore\_errors=True)**

**model.save(sc, output\_dir)**

**sameModel = NaiveBayesModel.load(sc, output\_dir)**

**predictionAndLabel = test.map(lambda p: (sameModel.predict(p.features), p.label))**

**accuracy = 1.0 \* predictionAndLabel.filter(lambda (x, v): x == v).count() / test.count()**

**#print('sameModel accuracy {}'.format(accuracy)) # python 2.7 and up**

**print('sameModel accuracy: ' + str(accuracy))**

**GRADIENT BOOSTING CLASSIFICATION/REGRESSION IN SPARKMLLIB**

**from \_\_future\_\_ import print\_function**

**from pyspark import SparkContext**

**# $example on$**

**from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel**

**from pyspark.mllib.util import MLUtils**

**# $example off$**

**if \_\_name\_\_ == "\_\_main\_\_":**

**sc = SparkContext(appName="PythonGradientBoostedTreesClassificationExample")**

**# $example on$**

**# Load and parse the data file.**

**data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")**

**# Split the data into training and test sets (30% held out for testing)**

**(trainingData, testData) = data.randomSplit([0.7, 0.3])**

**# Train a GradientBoostedTrees model.**

**# Notes: (a) Empty categoricalFeaturesInfo indicates all features are continuous.**

**# (b) Use more iterations in practice.**

**model = GradientBoostedTrees.trainClassifier(trainingData,**

**categoricalFeaturesInfo={}, numIterations=3)**

**# Evaluate model on test instances and compute test error**

**predictions = model.predict(testData.map(lambda x: x.features))**

**labelsAndPredictions = testData.map(lambda lp: lp.label).zip(predictions)**

**testErr = labelsAndPredictions.filter(**

**lambda lp: lp[0] != lp[1]).count() / float(testData.count())**

**print('Test Error = ' + str(testErr))**

**print('Learned classification GBT model:')**

**print(model.toDebugString())**

**# Save and load model**

**model.save(sc, "target/tmp/myGradientBoostingClassificationModel")**

**sameModel = GradientBoostedTreesModel.load(sc,**

**"target/tmp/myGradientBoostingClassificationModel")**

**# $example off$**

**from \_\_future\_\_ import print\_function**

**from pyspark import SparkContext**

**# $example on$**

**from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel**

**from pyspark.mllib.util import MLUtils**

**# $example off$**

**if \_\_name\_\_ == "\_\_main\_\_":**

**sc = SparkContext(appName="PythonGradientBoostedTreesRegressionExample")**

**# $example on$**

**# Load and parse the data file.**

**data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")**

**# Split the data into training and test sets (30% held out for testing)**

**(trainingData, testData) = data.randomSplit([0.7, 0.3])**

**# Train a GradientBoostedTrees model.**

**# Notes: (a) Empty categoricalFeaturesInfo indicates all features are continuous.**

**# (b) Use more iterations in practice.**

**model = GradientBoostedTrees.trainRegressor(trainingData,**

**categoricalFeaturesInfo={}, numIterations=3)**

**# Evaluate model on test instances and compute test error**

**predictions = model.predict(testData.map(lambda x: x.features))**

**labelsAndPredictions = testData.map(lambda lp: lp.label).zip(predictions)**

**testMSE = labelsAndPredictions.map(lambda lp: (lp[0] - lp[1]) \* (lp[0] - lp[1])).sum() /\**

**float(testData.count())**

**print('Test Mean Squared Error = ' + str(testMSE))**

**print('Learned regression GBT model:')**

**print(model.toDebugString())**

**# Save and load model**

**model.save(sc, "target/tmp/myGradientBoostingRegressionModel")**

**sameModel = GradientBoostedTreesModel.load(sc, "target/tmp/myGradientBoostingRegressionModel")**

**# $example off$**

**LATENT DIRICHLET ALLOCATION IN SPARKMLLIB**

**from \_\_future\_\_ import print\_function**

**from pyspark import SparkContext**

**# $example on$**

**from pyspark.mllib.clustering import LDA, LDAModel**

**from pyspark.mllib.linalg import Vectors**

**# $example off$**

**if \_\_name\_\_ == "\_\_main\_\_":**

**sc = SparkContext(appName="LatentDirichletAllocationExample") # SparkContext**

**# $example on$**

**# 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")**

**# $example off$**

**sc.stop()**

**SPAM CLASSIFICATION WITH LOGISTIC REGRESSION AND HASHING WITH SPARKMLLIB**

**from pyspark import SparkContext**

**from pyspark.mllib.regression import LabeledPoint**

**from pyspark.mllib.classification import LogisticRegressionWithSGD**

**from pyspark.mllib.feature import HashingTF**

**if \_\_name\_\_ == "\_\_main\_\_":**

**sc = SparkContext(appName="PythonBookExample")**

**# Load 2 types of emails from text files: spam and ham (non-spam).**

**# Each line has text from one email.**

**spam = sc.textFile("file:///home/cloudera/machine\_learning/Spark-Text-Mining/spam.txt")**

**ham = sc.textFile("file:///home/cloudera/machine\_learning/Spark-Text-Mining/ham.txt")**

**# Create a HashingTF instance to map email text to vectors of 100 features.**

**tf = HashingTF(numFeatures = 100)**

**# Each email is split into words, and each word is mapped to one feature.**

**spamFeatures = spam.map(lambda email: tf.transform(email.split(" ")))**

**hamFeatures = ham.map(lambda email: tf.transform(email.split(" ")))**

**# Create LabeledPoint datasets for positive (spam) and negative (ham) examples.**

**positiveExamples = spamFeatures.map(lambda features: LabeledPoint(1, features))**

**negativeExamples = hamFeatures.map(lambda features: LabeledPoint(0, features))**

**training\_data = positiveExamples.union(negativeExamples)**

**training\_data.cache() # Cache data since Logistic Regression is an iterative algorithm.**

**# Run Logistic Regression using the SGD optimizer.**

**# regParam is model regularization, which can make models more robust.**

**model = LogisticRegressionWithSGD.train(training\_data)**

**# Test on a positive example (spam) and a negative one (ham).**

**# First apply the same HashingTF feature transformation used on the training data.**

**posTestExample = tf.transform("O M G GET cheap stuff by sending money to ...".split(" "))**

**negTestExample = tf.transform("Hi Dad, I started studying Spark the other ...".split(" "))**

**# Now use the learned model to predict spam/ham for new emails.**

**print "Prediction for positive test example: %g" % model.predict(posTestExample)**

**print "Prediction for negative test example: %g" % model.predict(negTestExample)**

**sc.stop()**

**TFIDF WITH SPARKMLLIB**

**from \_\_future\_\_ import print\_function**

**from pyspark import SparkContext**

**# $example on$**

**from pyspark.mllib.feature import HashingTF, IDF**

**# $example off$**

**if \_\_name\_\_ == "\_\_main\_\_":**

**sc = SparkContext(appName="TFIDFExample") # SparkContext**

**# $example on$**

**# Load documents (one per line).**

**documents = sc.textFile("data/mllib/kmeans\_data.txt").map(lambda line: line.split(" "))**

**hashingTF = HashingTF()**

**tf = hashingTF.transform(documents)**

**# While applying HashingTF only needs a single pass to the data, applying IDF needs two passes:**

**# First to compute the IDF vector and second to scale the term frequencies by IDF.**

**tf.cache()**

**idf = IDF().fit(tf)**

**tfidf = idf.transform(tf)**

**# spark.mllib's IDF implementation provides an option for ignoring terms**

**# which occur in less than a minimum number of documents.**

**# In such cases, the IDF for these terms is set to 0.**

**# This feature can be used by passing the minDocFreq value to the IDF constructor.**

**idfIgnore = IDF(minDocFreq=2).fit(tf)**

**tfidfIgnore = idfIgnore.transform(tf)**

**# $example off$**

**print("tfidf:")**

**for each in tfidf.collect():**

**print(each)**

**print("tfidfIgnore:")**

**for each in tfidfIgnore.collect():**

**print(each)**

**sc.stop()**

**WORD2VEC WITH SPARKMLLIB**

**from \_\_future\_\_ import print\_function**

**from pyspark import SparkContext**

**# $example on$**

**from pyspark.mllib.feature import Word2Vec**

**# $example off$**

**if \_\_name\_\_ == "\_\_main\_\_":**

**sc = SparkContext(appName="Word2VecExample") # SparkContext**

**# $example on$**

**inp = sc.textFile("data/mllib/text8\_lines").map(lambda row: row.split(" "))**

**word2vec = Word2Vec()**

**model = word2vec.fit(inp)**

**synonyms = model.findSynonyms('china', 5)**

**for word, cosine\_distance in synonyms:**

**print(word, cosine\_distance)**

**# $example off$**

**sc.stop()**

**LATENT FACTOR WITH ALS WITH SPARKMLLIB**

**from pyspark.mllib.recommendation import ALS, MatrixFactorizationModel, Rating**

**# Load and parse the data**

**data = sc.textFile("/user/hadoop/alstest.data")**

**ratings = data.map(lambda l: l.split(','))\**

**.map(lambda l: Rating(int(l[0]), int(l[1]), float(l[2])))**

**# Build the recommendation model using Alternating Least Squares**

**rank = 10**

**numIterations = 10**

**model = ALS.train(ratings, rank, numIterations)**

**# Evaluate the model on training data**

**testdata = ratings.map(lambda p: (p[0], p[1]))**

**predictions = model.predictAll(testdata).map(lambda r: ((r[0], r[1]), r[2]))**

**ratesAndPreds = ratings.map(lambda r: ((r[0], r[1]), r[2])).join(predictions)**

**MSE = ratesAndPreds.map(lambda r: (r[1][0] - r[1][1])\*\*2).mean()**

**print("Mean Squared Error = " + str(MSE))**

**# Save and load model**

**model.save(sc, "target/tmp/myCollaborativeFilter")**

**sameModel = MatrixFactorizationModel.load(sc, "target/tmp/myCollaborativeFilter")**

**LATENT FACTOR WITH ALS WITH SPARKML**

**from pyspark.ml.recommendation import ALS**

**from pyspark.sql import Row**

**ratings = spark.read.text("/user/hadoop/sample\_movielens\_ratings.txt")\**

**.rdd.toDF()\**

**.selectExpr("split(value , '::') as col")\**

**.selectExpr(**

**"cast(col[0] as int) as userId",**

**"cast(col[1] as int) as movieId",**

**"cast(col[2] as float) as rating",**

**"cast(col[3] as long) as timestamp")**

**training, test = ratings.randomSplit([0.8, 0.2])**

**# Train an Alternating Least Squares model**

**als = ALS()\**

**.setMaxIter(5)\**

**.setRegParam(0.01)\**

**.setUserCol("userId")\**

**.setItemCol("movieId")\**

**.setRatingCol("rating")**

**print als.explainParams()**

**alsModel = als.fit(training)**

**predictions = alsModel.transform(test)**

**# Generate recommendations (users and then items) - output top 10 recommendations for each user or movie.**

**alsModel.recommendForAllUsers(10)\**

**.selectExpr("userId", "explode(recommendations)").show()**

**alsModel.recommendForAllItems(10)\**

**.selectExpr("movieId", "explode(recommendations)").show()**

**# Evaluate rating predictions**

**# We can treat rating prediction as a standard regression problem and use the standard regression evaluators**

**from pyspark.ml.evaluation import RegressionEvaluator**

**evaluator = RegressionEvaluator()\**

**.setMetricName("rmse")\**

**.setLabelCol("rating")\**

**.setPredictionCol("prediction")**

**rmse = evaluator.evaluate(predictions)**

**print("Root-mean-square error = %f" % rmse)**

**# We can also use ranking metrics**

**# We will set 2.5 as our threshold**

**from pyspark.mllib.evaluation import RankingMetrics, RegressionMetrics**

**from pyspark.sql.functions import col, expr**

**perUserActual = predictions\**

**.where("rating > 2.5")\**

**.groupBy("userId")\**

**.agg(expr("collect\_set(movieId) as movies"))**

**# At this point we have a collection of users along with a truth set of previously ranked movies for each user**

**# Now, we will get our top 10 recommendations from our algorithm on a per-user basis**

**perUserPredictions = predictions\**

**.orderBy(col("userId"), expr("prediction DESC"))\**

**.groupBy("userId")\**

**.agg(expr("collect\_list(movieId) as movies"))**

**# We will now see if the top 10 recommendations show up in our truth set (if we have a well trained model, it will correctly recommend the movies a user already liked)**

**perUserActualvPred = perUserActual.join(perUserPredictions, ["userId"]).rdd\**

**.map(lambda row: (row[1], row[2][:15]))**

**ranks = RankingMetrics(perUserActualvPred)**

**# We can see the metrics from that ranking**

**ranks.meanAveragePrecision**

**ranks.precisionAt(5)**