PySpark Machine Learning Demo

Yupeng Wang, Ph.D., Data Scientist

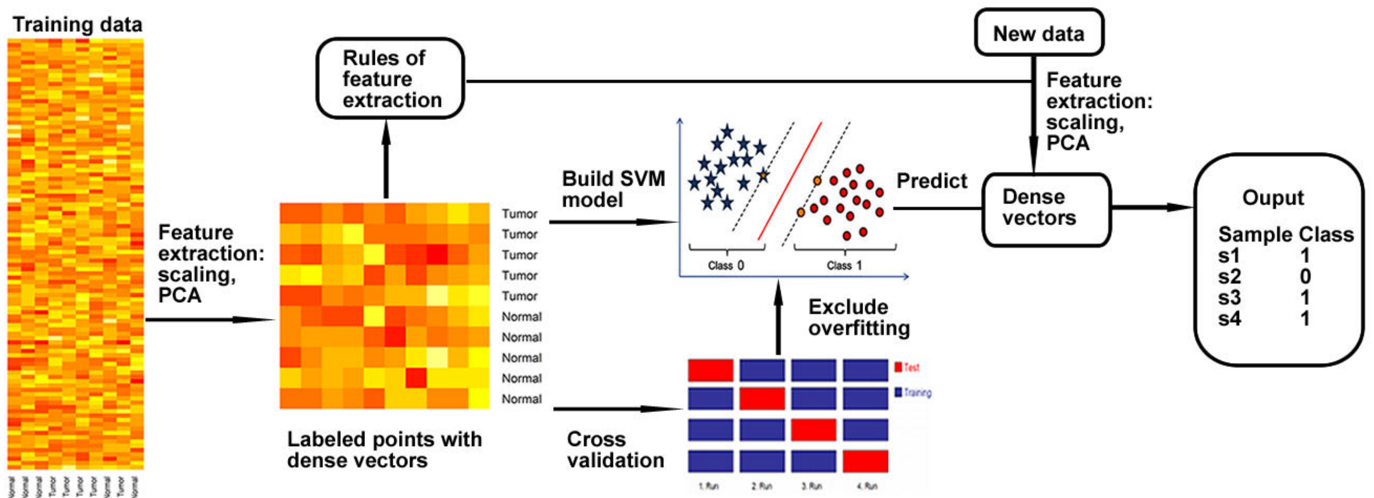
Overview

Apache Spark is an emerging big data analytics technology. Machine learning (ML) frameworks built on Spark are more scalable compared with traditional ML frameworks. In this demo, I build a Support Vector Machine (SVM) model using Spark Python API (PySpark) to classify normal and tumor microarray samples. Microarray measures expression levels of thousands of genes in a tissue or cell type. The raw data contains 102 microarray samples and 12625 genes. Feature extraction and cross-validation are employed to ensure effectiveness. The SVM model achieves an accuracy>80%.

The source code of this demo can be downloaded from https://github.com/wyp1125/Python-Spark-MapReduce-ML.

Key techniques: Python, Spark, machine learning, Support Vector Machine, feature extraction, feature scaling, dimension reduction, Principal Component Analysis, NumPy, cross-validation

General flowchart



Detailed procedure

1. Raw data

The data file is pheno_exp.txt, which is a tab-delimited text file. The dataset contains 102 microarray samples, of which 50 are normal samples and 52 are prostate tumor samples. Each microarray sample has 12625 genes, and occupies a column. The first column contains gene IDs. The first row contains sample IDs, while the second row contains label (i.e. sample class: 0: normal, 1: tumor). All other cells are expression levels, composing a matrix with a dimension of 12625×102. A small part of the dataset is displayed below:

	N01	N02	N03	N04	N05	T01	T02
Label	0	0	0	0	0	1	1
1000_at	7.664007	7.457289	7.290592	7.447533	7.188223	6.927656	7.261257
1001_at	3.783702	4.118153	4.216858	4.223811	4.364807	3.69376	3.837382
1002_f_at	3.152019	3.633566	3.767178	3.90114	4.00167	3.335858	3.448102
1003_s_at	5.452293	6.716626	6.431925	6.612021	6.34816	5.623835	6.284552

We do not want to use all of the 102 samples to build the SVM model. We need some data to mimic the scenario of making predictions on new data. Thus, we randomly divide the dataset into a training dataset and a prediction dataset according to 8:2 partition.

Program name: raw_div_dataset.py
Linux command: python raw_div_dataset.py pheno_exp.txt demo 0.8

```
from __future__ import print_function
import sys
import numpy as np
from sklearn.utils import resample
if len(sys.argv)<4:
    print("python raw_div_dataset.py input_file output_prefix train_ratio")
    quit()
with open(sys.argv[1],'r') as fl:
    line=fl.readlines()
word=line[0].strip("\n").split("\t")
id=range(1,len(word))
nid=resample(id,replace=False)
of1=sys.argv[2]+".train"
of2=sys.argv[2]+".predict"
r=float(sys.argv[3])
c=len(id)*r
with open(of1,'w') as fl1:
    with open(of2,'w') as fl2:
        for i in range(len(line)):
            word=line[i].strip("\n").split("\t")
            fl1.write(word[0])
            fl2.write(word[0])
            for j in range(len(word)-1):
                if j<c:
                    fl1.write('\t'+word[nid[j]])
                else:
                    fl2.write('\t'+word[nid[j]])
            fl1.write('\n')
            fl2.write('\n')
```

Here we obtain two output files: demo.train and demo.predict, containing 82 and 20 samples respectively.

2. Feature extraction on training data

If the raw data were directly used as features, the feature number (12625) would be too much higher than the sample number (102), which is prone to over-fitting. Thus, dimension reduction is needed. We first do feature scaling, and then use PCA for dimension reduction to obtain first 10 principal components.

Program name: convert_train_data.py
Linux command: python convert_train_data.py demo.train demo.train.labeledpoint

```
from __future__ import print_function
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import sys
if len(sys.argv)<3:
    print("python convert_train_data.py raw_training_data output")
    quit()
with open(sys.argv[1],"r") as fl:
    line=fl.readlines()
    lab=line[1].strip("\n").replace('\r','').split("\t")[1:]
exp=[]
for i in range(2,len(line)):
    gene=line[i].strip("\n").split("\t")[1:]
    exp.append(gene)
exp1=np.transpose(exp)
scaler = StandardScaler()
scaler.fit(exp1)
exp2=scaler.transform(exp1)
pca = PCA(n_components=10)
pca.fit(exp2)
exp3=pca.transform(exp2)
with open(sys.argv[2],"w") as of:
    for i in range(len(lab)):
        of.write(lab[i])
        for j in range(len(exp3[i])):
            of.write(" "+str(exp3[i][j]))
        of.write('\n')
```

Here we obtain the output file demo.train.labeledpoint. The file is in Spark's 'labeled points' format, with features being dense vectors. Each row represents a microarray sample. The first column is label (sample class: 0 or 1). The other columns are features (first 10 principal components).

3. Build an SVM model using PySpark ML

We now build an SVM model using the feature file obtained from the previous step. We want to use Spark which is very powerful for big data applications. We need to read the feature file into a Spark RDD dataset, and then parallelize the ML procedure using the Spark MapReduced framework. We need to import the PySpark ML library. We use the SVMWithSGD method, which is an optimization-based linear SVM.

Program name: train_svm.py
Linux command: python train_svm.py demo.train.labeledpoint demo_model

```
from __future__ import print_function
from pyspark import SparkContext
from pyspark.mllib.classification import SVMWithSGD, SVMModel
from pyspark.mllib.regression import LabeledPoint
import sys
if len(sys.argv)<3:
  print("python train_svm.py input_dens output_model_dir #_iterations(default:100)")
  quit()
n_ite=100
if len(sys.argv)==4:
  n_ite=int(sys.argv[3])
if __name__ == "__main__":
    sc = SparkContext(appName="PythonSVMWithSGDE")
    def parsePoint(line):
        values = [float(x) for x in line.split(' ')]
        return LabeledPoint(values[0], values[1:])
    data = sc.textFile(sys.argv[1])
    parsedData = data.map(parsePoint)
    model = SVMWithSGD.train(parsedData, iterations=n_ite)
    # 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 model
    model.save(sc, sys.argv[2])
```

The generated SVM model is saved under the directory demo_model. The screen output shows a training error of 0.04878. Note that the training error may be a little different for different executions due to randomly selected training samples.

4. Cross-validation to verify the generated SVM model

Over-fitting often occurs when the trained ML models are too complicated (i.e. having too many parameters). Here we use cross-validation to verify the generated SVM model. If the cross-validation generates acceptable accuracies, we can be assured to use the generated SVM model to make predictions on new data.

Here we use 4-fold cross-validation. We first generate the cross-validation datasets (75% training and 25% testing) using the training data demo.train.labeledpoint.

Program name: cv_div_dataset.py
Linux command: python cv_div_dataset.py demo.train.labeledpoint demo_cv 4

```
from __future__ import print_function
import sys
import numpy as np
import os
from sklearn.utils import resample
if len(sys.argv)<4:
    print("python cv_div_dataset.py input_file output_dir fold")
    quit()
with open(sys.argv[1],'r') as fl:
    line=fl.readlines()
n=len(line)
nline=resample(line,replace=False)
nfold=int(sys.argv[3])
unit=int(n/nfold)
if n%nfold>0:
    unit=unit+1
if not os.path.exists(sys.argv[2]):
    os.makedirs(sys.argv[2])
for i in range(nfold):
    dir1=sys.argv[2]+'/'+str(i)+'.train'
    dir2=sys.argv[2]+'/'+str(i)+'.test'
    with open(dir1,'w') as ftn:
        with open(dir2,'w') as fts:
            t_sta=i*unit
            t_end=(i+1)*unit
            if t_end>n:
                t_end=n
            for j in range(n):
                if j>=t_sta and j<t_end:
                    fts.write(nline[j])
                else:
                    ftn.write(nline[j])
```

Then we iteratively run SVM models on the cross-validation datasets (use the training part to train the SVM model while the testing part to assess the model accuracy).

Program names: test_svm.py, execute_cv_svm.py
Linux command: python execute_cv_svm.py demo_cv demo_cv_model

```
from __future__ import print_function
from pyspark import SparkContext
from pyspark.mllib.classification import SVMWithSGD, SVMModel
from pyspark.mllib.regression import LabeledPoint
import sys
if len(sys.argv)<3:
    print("python test_svm.py input_dense_vector_file model_directory")
    quit()
if __name__ == "__main__":
    sc = SparkContext(appName="PythonSVMWithSGD")
    def parsePoint(line):
        values = [float(x) for x in line.split(' ')]
        return LabeledPoint(values[0], values[1:])
    sameModel = SVMModel.load(sc, sys.argv[2])
    ndata = sc.textFile(sys.argv[1])
    nparsedData = ndata.map(parsePoint)
    # Evaluating the model on training data
    nlabelsAndPreds = nparsedData.map(lambda p: (p.label, sameModel.predict(p.features)))
    trainAcc = nlabelsAndPreds.filter(lambda lp: lp[0] == lp[1]).count() / float(nparsedData.count())
    print("Model accuracy = " + str(trainAcc))
```

```
from __future__ import print_function
import sys
import os
import subprocess
if len(sys.argv)<3:
    print("python execute_cv_svm.py cv_datasets_folder model_folder")
    quit()
if not os.path.exists(sys.argv[2]):
    os.makedirs(sys.argv[2])
cmd=[]
cmd.append('ls')
cmd.append(sys.argv[1])
process = subprocess.Popen(cmd, stdout=subprocess.PIPE)
out, err = process.communicate()
word=out.strip("\n").split("\n")
max_n=0
for temp in word:
    temp1=temp.split(".")
    if int(temp1[0])>max_n:
        max_n=int(temp1[0])
n_cv=max_n+1
print(str(n_cv)+" fold cross validation found!")
for i in range(n_cv):
    os.system("python train_svm.py "+sys.argv[1]+"/"+str(i)+".train "+sys.argv[2]+"/"+str(i)+" 100")
    os.system("python test_svm.py "+sys.argv[1]+"/"+str(i)+".test "+sys.argv[2]+"/"+str(i))
```

We obtain model accuracies from the screen output:

```
Run #1: 0.9047  Run #2: 0.9523  Run #3: 0.7619  Run #4: 0.8947
```

The average model accuracy is 0.878. Model accuracy>0.7 indicates effectiveness of ML models. The cross-validation ensures us to use the generated SVM model.

5. Make predictions on new data

We now make predictions on the 20% prediction data made at step 1. We need to transform the prediction data into features, according to the feature extraction rules obtained at step 2.

Program name: convert_pred_data.py
Linux command: python convert_pred_data.py demo.train demo.predict demo.pred.dense

```
from __future__ import print_function
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import sys
if len(sys.argv)<3:
    print("python convert_pred_data.py raw_training_data raw_prediction_data output")
    quit()
#read in training dataset
with open(sys.argv[1],"r") as fl1:
    line1=fl1.readlines()
t_exp=[]
for i in range(2,len(line1)):
    gene1=line1[i].strip("\n").split("\t")[1:]
    t_exp.append(gene1)
t_exp1=np.transpose(t_exp)
#read in predicted dataset
with open(sys.argv[2],"r") as fl2:
    line2=fl2.readlines()
lab=line2[0].strip("\n").split("\t")[1:]
n_exp=[]
for i in range(2,len(line2)):
    gene2=line2[i].strip("\n").split("\t")[1:]
    n_exp.append(gene2)
n_exp1=np.transpose(n_exp)
#normalize training data then adjust prediction data
scaler = StandardScaler()
scaler.fit(t_exp1)
t_exp2=scaler.transform(t_exp1)
n_exp2=scaler.transform(n_exp1)
pca = PCA(n_components=10)
pca.fit(t_exp2)
n_exp3=pca.transform(n_exp2)
with open(sys.argv[3],"w") as of:
    for i in range(len(lab)):
        of.write(lab[i])
        for j in range(len(n_exp3[i])):
            of.write(" "+str(n_exp3[i][j]))
        of.write('\n')
```

Here we get the feature file demo.pred.dense. Note that the first column of this file is sample ID, while in the training feature file demo.train.labeledpoint the first column is label (sample class). We make predictions on this file using the SVM model generated at step 3. We need to implement the Spark MapReduce framework to parallelize predictions.

Program name: predict_svm.py

Linux command: python predict_svm.py demo.pred.dense demo_model

```
from __future__ import print_function
from pyspark import SparkContext
from pyspark.mllib.classification import SVMWithSGD, SVMModel
from pyspark.mllib.linalg import Vectors
import sys
if len(sys.argv)<3:
 print("python svm_dens_test.py input_dense_vector_file model_directory")
 quit()
if __name__ == "__main__":
    sc = SparkContext(appName="PythonSVMWithSGD")
    def denseFea(line):
        word=line.split(' ')
        values = [float(x) for x in word[1:]]
        return (word[0],Vectors.dense(values))
    sameModel = SVMModel.load(sc, sys.argv[2])
    ndata = sc.textFile(sys.argv[1])
    nparsedData = ndata.map(denseFea)
    # Predict on new data
    nPreds = nparsedData.map(lambda p: (p[0],sameModel.predict(p[1])))
    for x in nPreds.collect():
      print(x)
```

The prediction outcome is printed as Python tuples on the screen.

```
(u'N33', 0)
(u'T19', 1)
(u'T42', 1)
(u'T31', 1)
(u'N44', 0)
(u'N02', 0)
(u'N21', 0)
(u'T10', 1)
(u'T22', 1)
(u'T36', 1)
(u'T39', 0)
(u'N01', 0)
(u'T59', 1)
(u'N60', 0)
(u'N24', 1)
(u'N13', 0)
(u'T54', 0)
(u'N39', 0)
(u'T27', 1)
(u'T13', 1)
```

From the prediction outcome, we can see that most predictions are correct: ‘N’ prefix represents normal samples with 0 being correct, while ‘T’ prefix represents tumor samples with 1 being correct. This demo shows that when a robust ML model is constructed, its predictions are mostly trustable. In real-world problems, usually it is impossible to know whether a specific prediction is 100% correct, but we are able to treat positive predictions as highly-risky instances.