Cloud_Computing HW3

r09922102 資工所碩一 韓秉勳

1. Environment
   Docker engine on ubuntu20.04,
   RAM - 16G,
   CPU - Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
   Base image: bitnami Spark
2. Github link:
   https://github.com/nba556677go/cloud_computing2020/tree/main/hw4
3. installation & run guide: Check github README.md
4. Report questions & Discussions
   - Reproduce the results using your own spark cluster
     reproduce training error: **0.007288629737609329**

```
root@spark-master:/opt/bitnami/spark/my_files# python reproduce.py
20/11/18 07:34:00 WARN NativeCodeLoader: Unable to load native-hadoop library for y
our platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(n
ewLevel).
20/11/18 07:34:03 WARN BLAS: Failed to load implementation from: com.github.fommil
netlib.NativeSystemBLAS
20/11/18 07:34:03 WARN BLAS: Failed to load implementation from: com.github.fommil
netlib.NativeRefBLAS
labeland pred PythonRDD[209] at RDD at PythonRDD.scala:53
1372
10
Training Error = 0.007288629737609329
```

     The original training error should be 0.0211370262391.
     However, my reproduction works better even though I didn't alter anything (only altered codes to let it run, which were hints by TA, like importing LabeledPoint) I think it's because the origin website supported an older logistic model, but spark mllib has updated its model. Errors solved are listed in troubleshooting. (I used local mode in reduce.py for simplicity. However, you can setmaster to MASTER_URL once init.sh is executed in every spark node)

```
def mapper(line):
    """
    Mapper that converts an input line to a feature vector
    """
    feats = line.strip().split(",")
    # labels must be at the beginning for LRSGD, it's in the end i
    # putting it in the right place
    label = feats[len(feats) - 1]
    feats = feats[: len(feats) - 1]
    feats.insert(0,label)
    features = [ float(feature) for feature in feats ] # need floa
    return LabeledPoint(label, features)

parsedData = data.map(mapper)
```

   - Troubleshooting:
     - Syntax error: lambda tuple error

```
trainErr = (labelsAndPreds.filter(lambda kv: kv[0] != kv[1]).count() )/ (float(parsedData.count()
```

new version of python don't support lambda (k, v): k!= v, so we need to alter that

- sc.textfile not found when set.master(MASTER_URL) instead of set.master("local")

```
def getSparkContext():
    """
    Gets the Spark Context
    """
    conf = (SparkConf()
         .setMaster(MASTER_URL) # run on master/local
         .setAppName("Logistic Regression") # Name of App
         .set("spark.executor.memory", "1g")) # Set 1 gig
    sc = SparkContext(conf = conf)
    return sc
```

  - sol: https://stackoverflow.com/questions/24735516/spark-how-to-use-sparkcontext-textfile-for-local-file-system
    I used master as the driver for job submission. Our file is stored in spark-master's local file system. Since Bitnami Spark image does not included HDFS, Each node should contain a whole file. In this case local file system will be logically indistinguishable from the HDFS, in respect to this file. So the same input file has to be at all workers and the same local path in order to run successfully

- SPARK_LOCAL_IP has to be commented in set.master(MASTER_URL)
-
- WARNING emerges when executing program

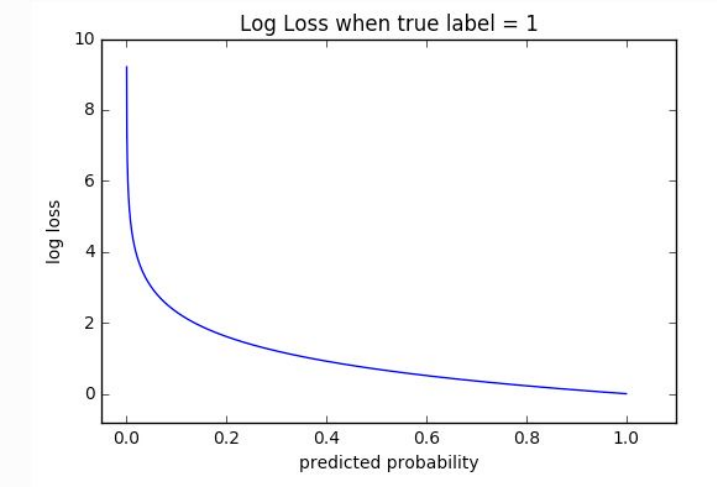
```
WARN BLAS: Failed to load implementation
from:com.github.fommil.netlib.NativeSystemBLAS
WARN BLAS: Failed to load implementation
from:com.github.fommil.netlib.NativeRefBLAS
```

these warning came out since MLlib default use these two libraries to accelerate linear algebra processing, but they are not native supported in spark. pure JVM implementation will be used instead.
(https://spark.apache.org/docs/latest/ml-guide.html#dependencies)

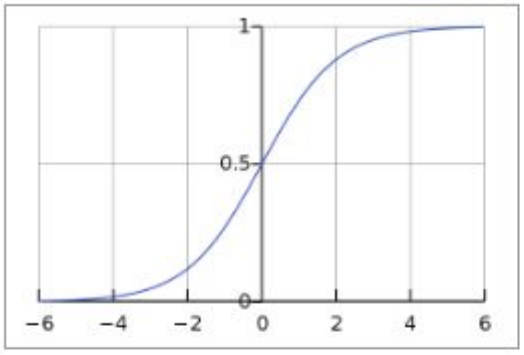
- Write your own SGD (stochastic gradient descent or simple gradient descent) function of logistic regression. And compare the results.
  - Method - simple gradient descend
  - loss function - cross-entropy

cross entropy is suitable in binary classification since the log loss decreases when prediction is closer to ground truth, equation is as follow:

$$CE = -\sum_{i=1}^{C'=2} t_i log(f(s_i)) = -t_1 log(f(s_1)) - (1 - t_1) log(1 - f(s_1))$$



- activation function - sigmoid



$$f(s_i) = \frac{1}{1 + e^{-s_i}}$$

- Sigmoid outputs from 0 to 1, so we can classify our prediction by setting the threshold at 0.5

```
@staticmethod
def sigmoid(z):
    res = 1 / (1.0 + np.exp(-z))
    return np.clip(res, 1e-8, 1-(1e-8))
```

- gradient update:
  - 利用step functiond將sigmoid後的output 得到prediction的結果，再用這結果與正確值相減*x(即微分)，再利用此gradient update weight

```
@staticmethod
def Sparkstep(data, w, l_rate):
    #print("insparkstep", data.shape)
    #print("datashape", data.shape)
    x = data[:, 1:]
    y = data[:, 0]
    #print("x", x)
    #print("w", w)
    func = myGD.sigmoid(np.dot(x , w))
    grad = -np.dot(x.transpose(), (y - func))
    w = w - l_rate*grad
    #print("w after update:", w)
    return w
```

- Spark RDD processing
    - This is the core part of this homework. However, I found it troublesome to map weight * input concurrently. For example, if you store features as RDD by parallelize like this:

```
features = self.sc.parallelize(features)
```

and weight initialized in your driver as numpy array:

```
w = np.zeros(self.feature_size - 1)# - label
```

when doing training mapping like this:

```
for i in range(1, iteration + 1):
    w = self.train_spark.map(lambda _data: myGD.Sparkstep(_data, w, l_rate)).reduce(myGD.add)
```

weight cannot be updated concurrently since they are not synced together, resulting in zero updation. I avoided this problem by reading feature with mappartition like this:

```
def readPointBatch(iterator):
        strs = list(iterator)
        matrix = np.zeros((len(strs), D + 1))
        for i, s in enumerate(strs):
            matrix[i] = np.fromstring(s.replace(',', ' '), dtype=np.float32, sep=' ')
        return [matrix]
```

```
def read_norm_feature(train_data):

    spark = SparkSession\
        .builder\
        .appName("PythonLR")\
        .getOrCreate()

    points = spark.read.text(train_data).rdd.map(lambda r: r[0])\
    .mapPartitions(readPointBatch).cache()

    return points
```

this will ensure training data maps with the entire input array(i.e. shape = (1372, feature_size)) and send into Sparkstep() function updates. However, this method requires reading another preprocessed file, resulting in more file IO. There should be a better way, but this is applicable

- feature preprocessing - including reorganize features, feature normalization, and store in another file for spark to mappartition
    - reorganize features
      read input file and put label in front of features.

```
with open(os.path.join(".",textfile)) as f:
    for line in f.readlines():
        row = line.strip().split(",")
        label =  row[len(row) - 1]
        labels.append(label)
        #put label as first feature
        feats = row[:len(row) - 1]
        feats = [float(i) for i in feats]
        #print(feats)
        #feats.insert(0, label)
        features.append(feats)
```

    - feature normalization
      calculate mean, std, and update features accordingly. This is important since sigmoid may overflow without normalizing

```
#normalization
features = np.array(features)
mean = np.mean(features, axis = 0)
std = np.std(features, axis = 0)
#print(mean)
#print(std)
for i in range(features.shape[0]):
    for j in range(features.shape[1]):
        if not std[j] == 0 :
            features[i][j] = (features[i][j]- mean[j]) / std[j]
```

    - add bias and write preprocessed file
      write to another file with labels and features. also adding bias = 1 in every input. This improves 1% of performance. spark reads this file as input

```
with open(outfile, 'w') as f:
    for i in range(features.shape[0]):
        f.write(labels[i])
        for j in range(features.shape[1]):
            f.write(','+ str(features[i][j]))
        #add bias
        f.write(",1")
        if i ≠ features.shape[0] - 1:
            f.write("\n")
```

- prediction
    - just predicts by transforming rdd to numpy array to save effort. use cross entropy to measure the loss and records the error rate

```
def Sparkpredict(self, w):

    data = np.array(self.train_spark.collect())
    x = data[0, :, 1:]
    y = data[0, :, 0]

    #res = 1 / (1.0 + np.exp(-np.dot(x , w)))
    #func = np.clip(res, 1e-8, 1-(1e-8))
    self._loss = self.Spark_cross_entropy(x ,y , w , self.sigmoid(np.dot(x , w))
    pred = self.sigmoid(np.dot(x , w))
    p = pred
    p[pred < 0.5] = 0.0
    p[pred ≥ 0.5] = 1.0
    return np.mean(np.abs(y - p))
```

- training process and results

```
def train(self, l_rate = 0.1, iteration = 2, regularize = 0.0):
    #init para
    iteration = int(iteration)
    if self.spark:
        #get feature size
        print("shape", np.array(self.train_spark.collect()).shape)
        self.feature_size = np.array(self.train_spark.collect()).shape[2]
        #input()

        self._lambdda = regularize
        w = np.zeros(self.feature_size - 1)# - label
        for i in range(1, iteration + 1):
            w = self.train_spark.map(lambda _data: myGD.Sparkstep(_data, w, l_rate)).reduce(myGD.add)
            #Sparkstep()
            if (i % 20 == 0):
                #err = self.train_spark.map(lambda d: self.Sparkpredict(d, w)).reduce(myGD.add)
                err = self.Sparkpredict(w)
                print('[iteration {:5d}] - training loss: {:.5f}, error_rate: {:.18f}'.format(i, self._loss, err))
        #final prediction
        print("final prediction...")
        #np.save("logistic"+str(iteration)+"-regularize-"+str(self._lambdda)+".npy" , self._w)
        err = self.Sparkpredict(w)
        print('[iteration {:5d}] - training loss: {:.5f}, error_rate: {:.18f}'.format(iteration, self._loss, err))
        print("final w:", w)
```

integrates all functions and record error rate. Final output looks like this (iterating 100 times)

```
root@spark-master:/opt/bitnami/spark/my_files# python my_gradient_descent.py dat
a_banknote_authentication.txt 100
20/11/20 05:17:42 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLeve
l(newLevel).
shape (1, 1372, 6)
[iteration    20] - training loss: 0.02104, error_rate: 0.010204081632653060
[iteration    40] - training loss: 0.02096, error_rate: 0.010204081632653060
[iteration    60] - training loss: 0.02087, error_rate: 0.010204081632653060
[iteration    80] - training loss: 0.02087, error_rate: 0.010204081632653060
[iteration   100] - training loss: 0.02087, error_rate: 0.010204081632653060
final prediction...
[iteration   100] - training loss: 0.02087, error_rate: 0.010204081632653060
final w: [-29.16764717 -31.49067677 -30.00695536  -1.83648167 -14.13538061]
```

final error rate is **0.010204081632653060 = 1%,** which is slightly higher than reproduce error(but lower then the website's 2%) If we use adagrad, we can improve to error rate 0% (but I implemented the nonspark version for adagrad only since it faces a lot of RDD issues, better to use mllib library!)