Zafar Mahmood
Distributed lab 8
16/June/2017

Q1 : Find an image histogram using aggregation

a) Initialize your spark context for gray values

- reading file from hdfs in binaries of only one channel
- decode the binaries into bytes using numpy with dtype=np.uint8
- decode image using openCV

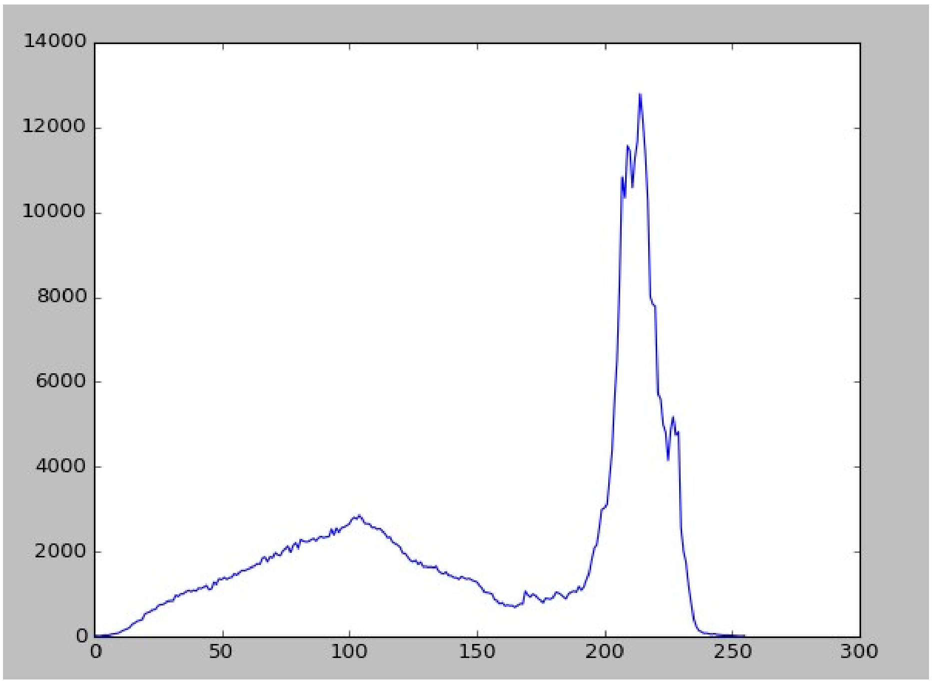
```
img_binary = sc.binaryFiles('hdfs:/user/zfar/exercise8/castle.jpg').take(1)

img_bytes = np.asarray(bytearray(img_binary[0][1]),dtype=np.uint8)

img = cv2.imdecode(img_bytes,0)
```

b) How you design your sequence and combination operation functions

- parallelize the Image → flatmap each digit → map each digit , 1 → aggregation w.r.t each key
- collect method to display the graph using plot function

```
rdd = sc.parallelize(img).flatMap(lambda word:(word)).map(lambda item : (item ,
1)).aggregateByKey(0,(lambda k,v:v+k),(lambda v,k:v+k)).collect()
```

c) Implement only for gray scale histogram



Q 2 : Using Apache Spark Mllib

2.a) Explain your pipeline by following standard machine learning approach.

- Tokenizer
- Stopword
- Hashing
- Machine Learning Model (linear Regression , Naive Bayesian)
- Model Fit

2.b) Explain your preprocessing steps and how much each added step improves accuracy. You can use a table to list your results for each technique.

Initially without remover ,stop words have created the difference

Then other techniques like ngrams , hashingTF , IDF , normlization also included in the model with leads for little bit better prediction

2.c) Develop a pipeline for textual data pre-processing and Naive Bayes model.

```
tokenizer = Tokenizer(inputCol="SentimentText", outputCol="words")
remover = StopWordsRemover(inputCol=tokenizer.getoutputCol()     ,
outputCol="filtered")
hashingTF = HashingTF(inputCol=remover.getOutputCol(), outputCol="features")
#lr = LogisticRegression(maxIter=10, regParam=0.001)
nb = NaiveBayes(smoothing=1.0)
pipeline = Pipeline(stages=[tokenizer, remover ,hashingTF, nb])
model = pipeline.fit(training)


other


tokenizer = Tokenizer(inputCol="SentimentText", outputCol="words")
remover    =    StopWordsRemover(inputCol=    tokenizer.getOutputCol()    ,
outputCol="filtered")
ngrams = NGram(n=2, inputCol= remover.getOutputCol() , outputCol="ngrams")
hashingTF = HashingTF(inputCol=ngrams.getOutputCol(), outputCol="rawfeatures")
idf = IDF(inputCol= hashingTF.getOutputCol() , outputCol="idffeatures")
normalizer  =  Normalizer(inputCol=  idf.getOutputCol()  ,  outputCol="features",
p=1.0)


#lr = LogisticRegression(maxIter=10, regParam=0.001)
nb = NaiveBayes(smoothing=1.0)
pipeline  =  Pipeline(stages=[tokenizer,  remover  ,  ngrams,  hashingTF,  idf  ,
normalizer , nb])
model = pipeline.fit(training)
```

2.d) Report evaluation on prediction classification accuracy.

	Executor 1	2	3	4
Performance	0.375	0.375	0.375	0.375
Better	0.53	0.53	0.53	0.53

2.e) Effect of varying number of executors on the classification accuracy

The accuracy remains the same with varying number of executors

	Executor 1	2	3	4
Performance	0.375	0.375	0.375	0.375
Better Performance	0.53	0.53	0.53	0.53

Exercise 3 : Matrix Factorization with Coordinate Descent using Apache Spark

3.a) Data Division Strategy

As in this algorithm we have given user's and items in case of movielen dataset.
Now for this data parallelization I have followed the paper algorithm, as it parallelizes using the latent features of each user and Item (k which is given in the format t)

```
    for t in range(0,K):
            temp_p = P[:,t] ## Broadcast latent of user
            temp_q = Q[:,t]       ## Broadcast latent of item

            for t_i in range(0,len(temp_p)):

               for t_j in range(0,len(temp_q)):

                   ## parallel Update u_star

                   u_star     =     ((reference[t_i][t_j]     -     temp_p[t_i]*temp_q[t_j]     +
temp_p[t_i]*temp_q[t_j] ) * temp_q[t_j])/( lemda + np.sum(np.square(temp_q)))

                   ## parallel update v_star


                   v_star     =     ((reference[t_i][t_j]     -     temp_p[t_i]*temp_q[t_j]     +
temp_p[t_i]*temp_q[t_j] ) * temp_p[t_j])/( lemda + np.sum(np.square(temp_p)))


                   #update R

                   reference[t_i][t_j]   =   reference[t_i][t_j]   +   temp_p[t_i]*temp_q[t_j]   -
u_star*v_star


                   ## update latent fetures

                   if (u_star != 0): P[t_i,t] = u_star
                   if (v_star != 0 ): Q[t_j,t] = v_star

                   #print (u_star , v_star)
    print ("\n\nP ",P)

    return P, Q , new_arr
```

Now in each latent feature is transformed using the mapper function

(best of my knowledge and understanding)

now while transformation we can map using the latent as key ,

consider the example as if we take from above

```
array = sc.parallize(P[:,t]).map(lambda item : ( t , item ))
```

as using this function we can map array using the latent feature.

When it comes to action function we can apply combiner / reduction operation and then we can apply the our algorithm either by cashing it in memory as we will be needing that later working .