CS777

Big Data Analytics

Spring 2021

Professor Kia Teymourian

Assignment 5

Kunfei Chen

U15575304

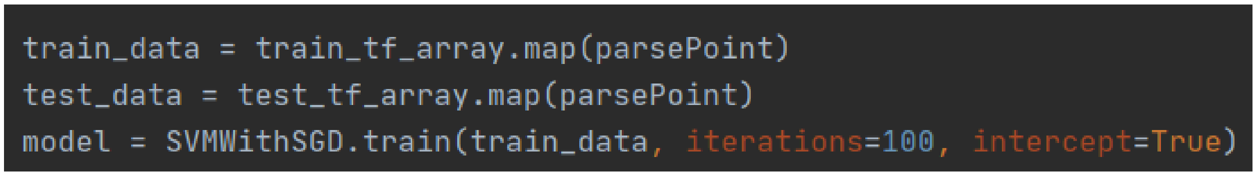
Introduction

This project aims to implement support vector machines via PySpark to achieve to classify text documents. The whole data set consists of two categories which are Wikipedia pages and Australian court cases.

Task 1 – Using Spark Mllib to train SVM

Firstly, the TF arrays of both training set and testing set are calculated, each array represents the terms frequency value of top 20000 words in the training set. Then, the "SVMWithSGD" is imported to train the data. As shown in Figure 1, the number of iterations is set to be 100, and the intercept is added to the model.

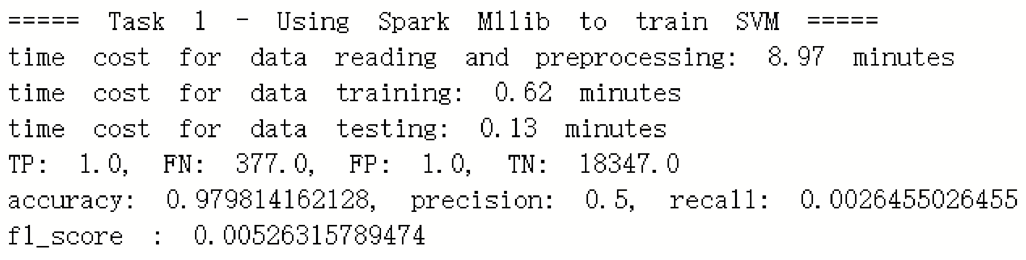
Figure 1



```
train_data = train_tf_array.map(parsePoint)
test_data = test_tf_array.map(parsePoint)
model = SVMWithSGD.train(train_data, iterations=100, intercept=True)
```

As shown in Figure 2, the time cost for data reading and preprocessing are about 9 minutes, the time cost for data training are 0.62 minutes and the time cost for data testing are 0.13 minutes. After 100 iterations, the final accuracy is about 98%, however the precision rate and recall rate are only 50% and 0.26% respectively, and the F1 score is only 0.005. Compared with the 99.7% precision rate, 99.4% precision rate, and 0.996 F1 score achieved by Logistic Regression model on Assignment 4, the performance of SVM with SGD is bad. One possible reason for this result could be the unbalanced proportion of Wikipedia cases and Australian court cases. In addition, compared with SGD optimization method, the Sequential Minimal Optimization (SMO) might be a better method for SVM model.

Figure 2

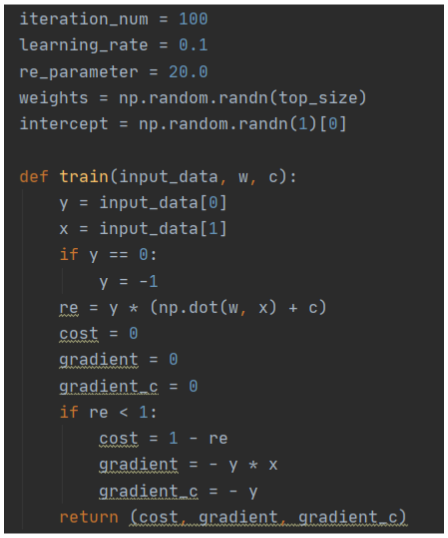


```
===== Task 1 - Using Spark Mllib to train SVM =====
time cost for data reading and preprocessing: 8.97 minutes
time cost for data training: 0.62 minutes
time cost for data testing: 0.13 minutes
TP: 1.0, FN: 377.0, FP: 1.0, TN: 18347.0
accuracy: 0.979814162128, precision: 0.5, recall: 0.0026455026455
f1_score : 0.00526315789474
```

Task 2 – Implementing SVM model from Scratch

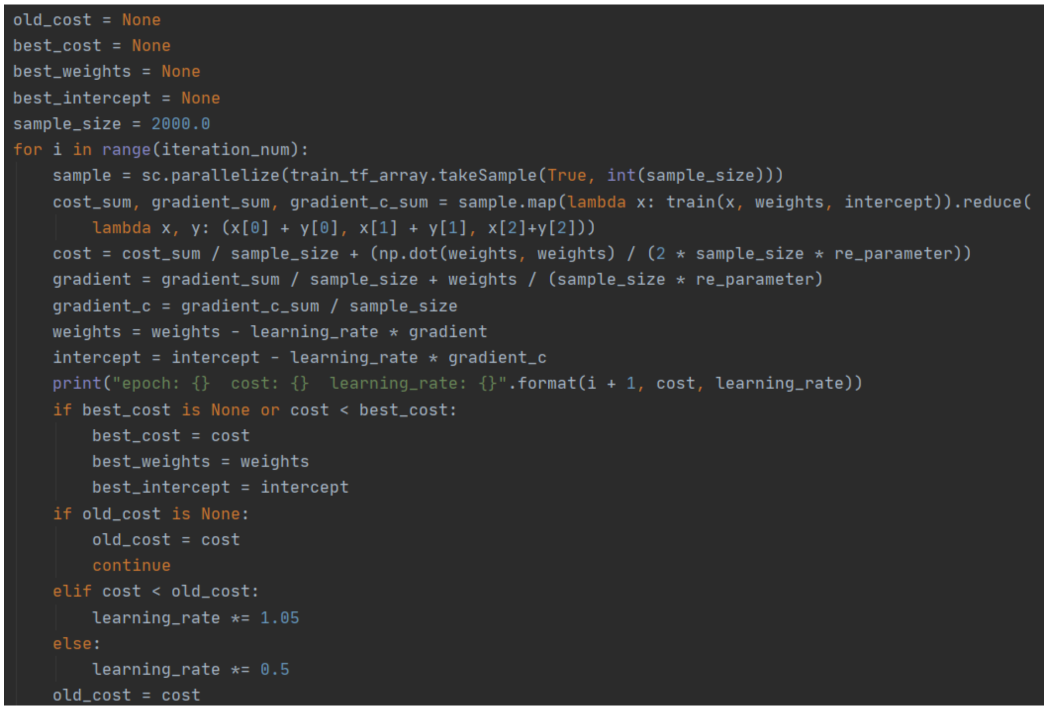
Then, a new SVM model is built for the training via pyspark and numbpy (See Figure 3). The learning rate is set to be 0.1, the regularizationn parameter is set to be 20. The initial weights and initial intercept are generated by the random function of numpy in which the value follows the normal distribution. The model is trained for 100 interation.

Figure 3



```python
iteration_num = 100
learning_rate = 0.1
re_parameter = 20.0
weights = np.random.randn(top_size)
intercept = np.random.randn(1)[0]

def train(input_data, w, c):
    y = input_data[0]
    x = input_data[1]
    if y == 0:
        y = -1
    re = y * (np.dot(w, x) + c)
    cost = 0
    gradient = 0
    gradient_c = 0
    if re < 1:
        cost = 1 - re
        gradient = - y * x
        gradient_c = - y
    return (cost, gradient, gradient_c)
```

Figure 4

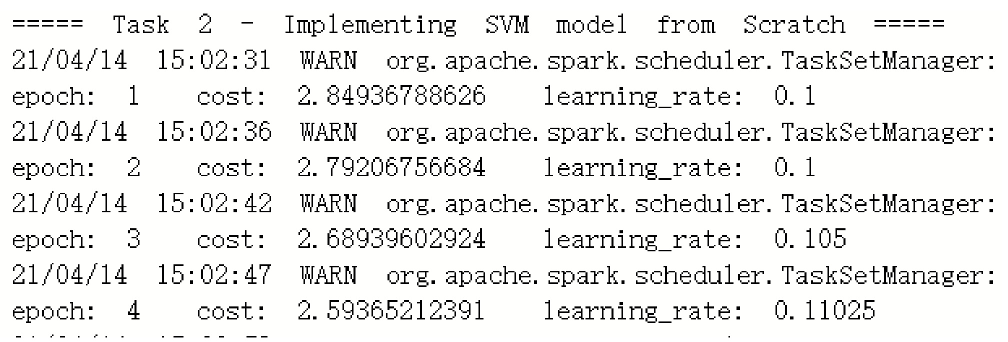


```python
old_cost = None
best_cost = None
best_weights = None
best_intercept = None
sample_size = 2000.0
for i in range(iteration_num):
    sample = sc.parallelize(train_tf_array.takeSample(True, int(sample_size)))
    cost_sum, gradient_sum, gradient_c_sum = sample.map(lambda x: train(x, weights, intercept)).reduce(
        lambda x, y: (x[0] + y[0], x[1] + y[1], x[2]+y[2]))
    cost = cost_sum / sample_size + (np.dot(weights, weights) / (2 * sample_size * re_parameter))
    gradient = gradient_sum / sample_size + weights / (sample_size * re_parameter)
    gradient_c = gradient_c_sum / sample_size
    weights = weights - learning_rate * gradient
    intercept = intercept - learning_rate * gradient_c
    print("epoch: {}  cost: {}  learning_rate: {}".format(i + 1, cost, learning_rate))
    if best_cost is None or cost < best_cost:
        best_cost = cost
        best_weights = weights
        best_intercept = intercept
    if old_cost is None:
        old_cost = cost
        continue
    elif cost < old_cost:
        learning_rate *= 1.05
    else:
        learning_rate *= 0.5
    old_cost = cost
```

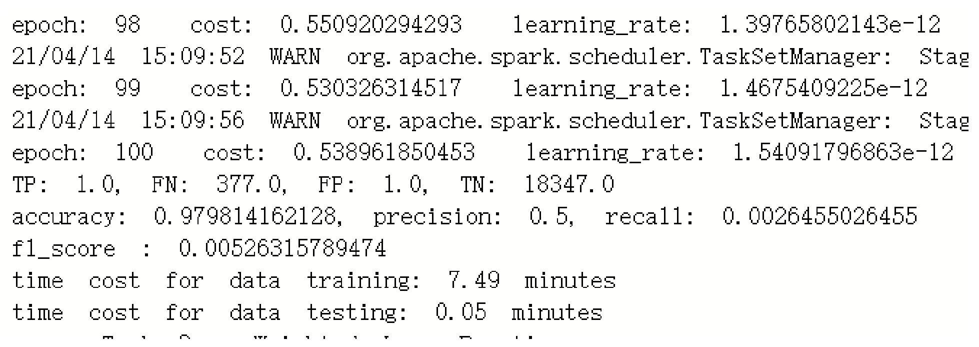
Since this is a batch gradient descent model, for each iteration, 2000 samples are taken from the whole dataset for each epoch of training. In addition, the training of this model applys "Bold Driver" technique to dynamically change the learning rate (See Figure 4).

The costs and model parameters in each iteration is shown in Figure 5. It can be seen that after training 100 iterations, the cost decreased from 2.85 to 0.53, the final accuracy is about 98%, the precision rate and recall rate are 50% and 0.26% respectively, and the F1 score is 0.005, which shows the same results of SVM model in task 1. In addition, the time cost for data training are 7.49 minutes and the time cost for data testing are 0.05 minutes

Figure 5



```
=====  Task  2  -  Implementing  SVM  model  from  Scratch  =====
21/04/14  15:02:31  WARN  org.apache.spark.scheduler.TaskSetManager:
epoch:  1    cost:  2.84936788626    learning_rate:  0.1
21/04/14  15:02:36  WARN  org.apache.spark.scheduler.TaskSetManager:
epoch:  2    cost:  2.79206756684    learning_rate:  0.1
21/04/14  15:02:42  WARN  org.apache.spark.scheduler.TaskSetManager:
epoch:  3    cost:  2.68939602924    learning_rate:  0.105
21/04/14  15:02:47  WARN  org.apache.spark.scheduler.TaskSetManager:
epoch:  4    cost:  2.59365212391    learning_rate:  0.11025
```

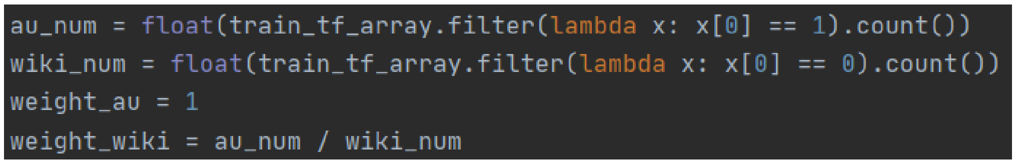


```
epoch:  98    cost:  0.550920294293    learning_rate:  1.39765802143e-12
21/04/14  15:09:52  WARN  org.apache.spark.scheduler.TaskSetManager:  Stag
epoch:  99    cost:  0.530326314517    learning_rate:  1.4675409225e-12
21/04/14  15:09:56  WARN  org.apache.spark.scheduler.TaskSetManager:  Stag
epoch:  100    cost:  0.538961850453    learning_rate:  1.54091796863e-12
TP:  1.0,  FN:  377.0,  FP:  1.0,  TN:  18347.0
accuracy:  0.979814162128,  precision:  0.5,  recall:  0.0026455026455
f1_score  :  0.00526315789474
time  cost  for  data  training:  7.49  minutes
time  cost  for  data  testing:  0.05  minutes
```

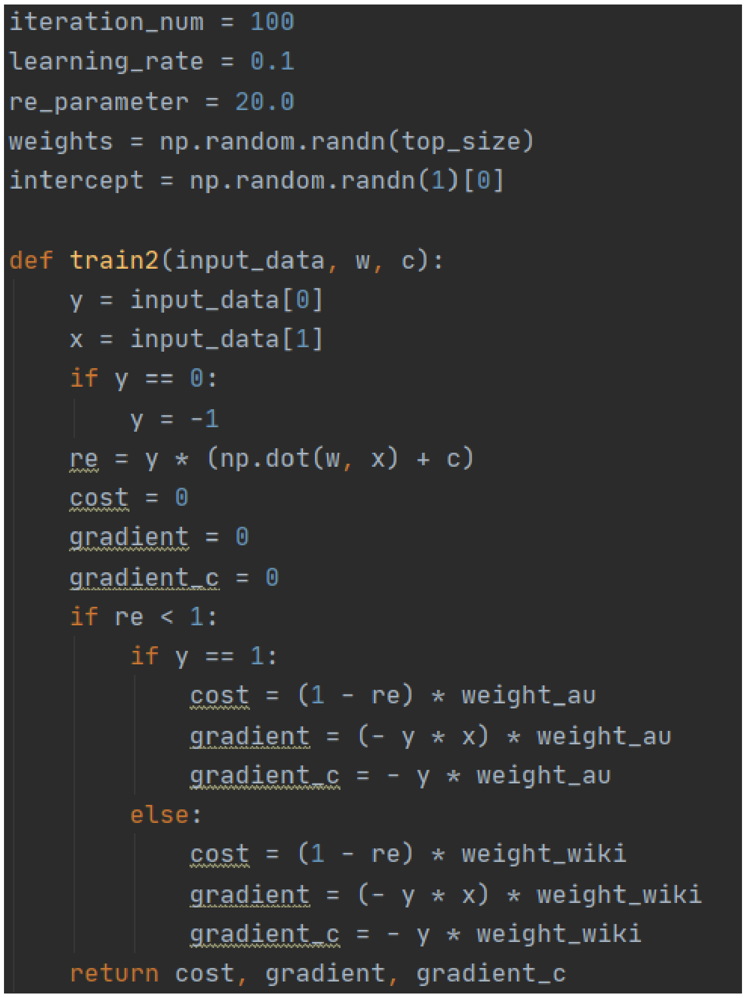
Task 3 – Weighted Loss Function

As shown in Figure 6, a new SVM model is designed with weighted loss function which aims to solve the issues with unbalanced data set. The weight of Australian court case is set to be 1, and that of Wikipedia page is based on the relative ration of the two classes' counts which is 0.02. Other parameters remain the same setting, the learning rate is set to be 0.1, the regularizationn parameter is set to be 20, the initial weights and initial intercept are generated by the random function of numpy in which the value follows the normal distribution. The model is trained for 100 interation.

Figure 6



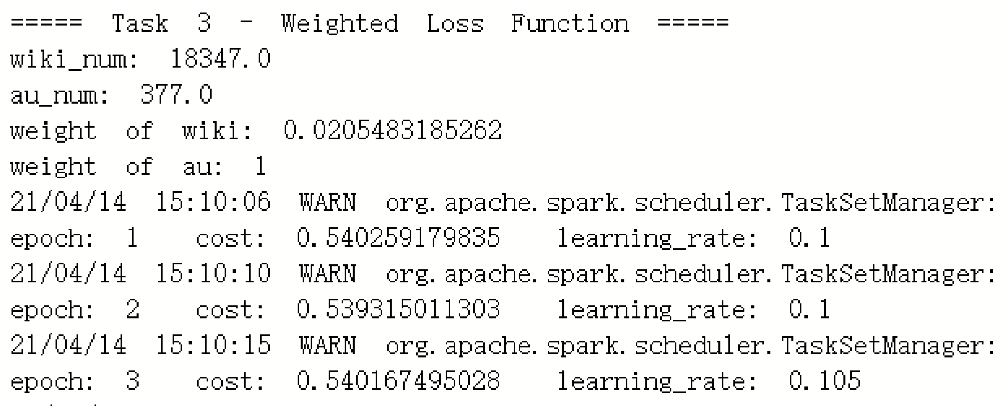
```
au_num = float(train_tf_array.filter(lambda x: x[0] == 1).count())
wiki_num = float(train_tf_array.filter(lambda x: x[0] == 0).count())
weight_au = 1
weight_wiki = au_num / wiki_num
```



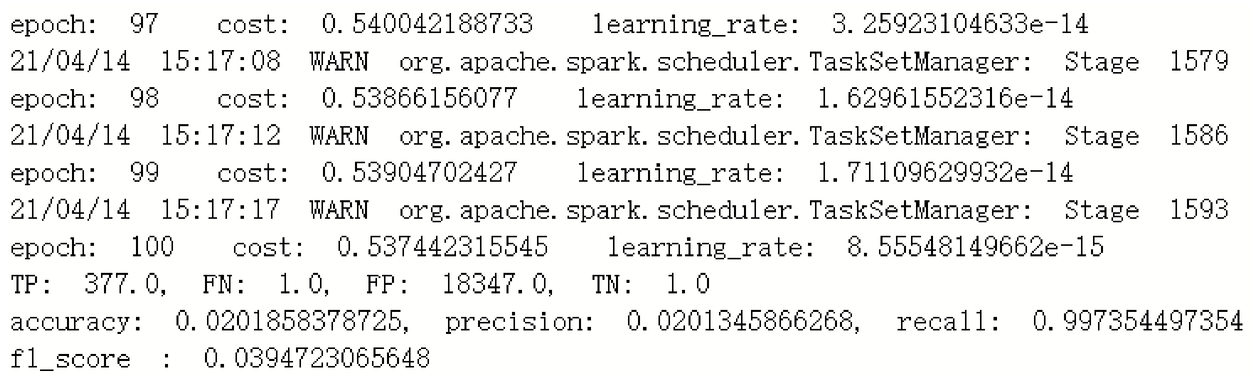
```
iteration_num = 100
learning_rate = 0.1
re_parameter = 20.0
weights = np.random.randn(top_size)
intercept = np.random.randn(1)[0]

def train2(input_data, w, c):
    y = input_data[0]
    x = input_data[1]
    if y == 0:
        y = -1
    re = y * (np.dot(w, x) + c)
    cost = 0
    gradient = 0
    gradient_c = 0
    if re < 1:
        if y == 1:
            cost = (1 - re) * weight_au
            gradient = (- y * x) * weight_au
            gradient_c = - y * weight_au
        else:
            cost = (1 - re) * weight_wiki
            gradient = (- y * x) * weight_wiki
            gradient_c = - y * weight_wiki
    return cost, gradient, gradient_c
```

As shown in Figure 7, after training 100 iterations, the cost decreased from 0.54 to 0.53, the final accuracy is only 2%, the precision rate is 2%, but the recall rate is increased to 99.73%, and the final F1 score is also increased to 0.039. The final result shows the new model pay more attention to the correct classification of Australian court cases. This is caused by the relative high weight of loss for this case.

Figure 7



```
===== Task 3 - Weighted Loss Function =====
wiki_num: 18347.0
au_num: 377.0
weight of wiki: 0.0205483185262
weight of au: 1
21/04/14 15:10:06 WARN org.apache.spark.scheduler.TaskSetManager:
epoch: 1   cost: 0.540259179835   learning_rate: 0.1
21/04/14 15:10:10 WARN org.apache.spark.scheduler.TaskSetManager:
epoch: 2   cost: 0.539315011303   learning_rate: 0.1
21/04/14 15:10:15 WARN org.apache.spark.scheduler.TaskSetManager:
epoch: 3   cost: 0.540167495028   learning_rate: 0.105
```



```
epoch: 97   cost: 0.540042188733   learning_rate: 3.25923104633e-14
21/04/14 15:17:08 WARN org.apache.spark.scheduler.TaskSetManager: Stage 1579
epoch: 98   cost: 0.53866156077   learning_rate: 1.62961552316e-14
21/04/14 15:17:12 WARN org.apache.spark.scheduler.TaskSetManager: Stage 1586
epoch: 99   cost: 0.53904702427   learning_rate: 1.71109629932e-14
21/04/14 15:17:17 WARN org.apache.spark.scheduler.TaskSetManager: Stage 1593
epoch: 100   cost: 0.537442315545   learning_rate: 8.55548149662e-15
TP: 377.0, FN: 1.0, FP: 18347.0, TN: 1.0
accuracy: 0.0201858378725, precision: 0.0201345866268, recall: 0.997354497354
f1_score : 0.0394723065648
```