CS-734/834 Introduction to Information Retrieval: Assignment #5: 10.3, 10.6, 10.9, SVM^{light}

Due on Friday, December 16, 2016

Dr. Michael L. Nelson

Plinio Vargas
pvargas@cs.odu.edu

Contents

List of Figures

Listings

1 Exercise 10.3

Compute five iterations of HITS (see Algorithm 3) and PageRank (see Figure 4.11) on the graph in Figure 10.3. Discuss how the PageRank scores compare to the hub and authority scores produced by HITS.

Figure 1: HITS vs PageRank 5 Iteration Calculation

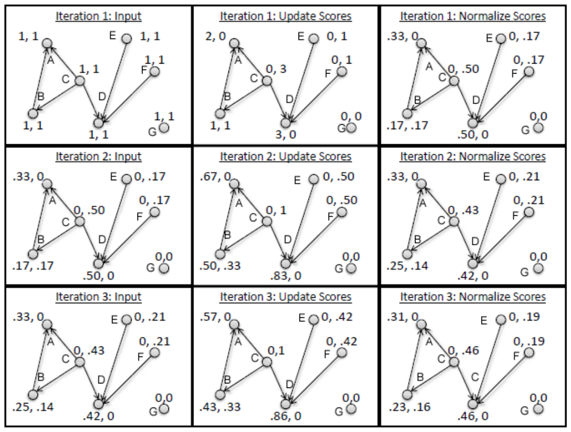

Iteration 1: Input
Iteration 1: Update Scores
Iteration 1: Normalize Scores
Iteration 2: Input
Iteration 2: Update Scores
Iteration 2: Normalize Scores
Iteration 3: Input
Iteration 3: Update Scores
Iteration 3: Normalize Scores

Fig. 10.3. Illustration of the HITS algorithm. Each row corresponds to a single iteration of the algorithm and each column corresponds to a specific step of the algorithm.

1.1 Approach

Vertices identification was added to the graph in order to better reference the nodes for further analysis. Graph G on Figure 1 is composed of the following vertices: {A, B, C, D, E, F, G}.

1.1.1 HITS

To calculate 5 iterations of the HITS algorithm for the graph shown on Figure 1, the *Python* scripts *hits.py* was developed (See Listing 1). The data structure to construct the graph is a dictionary, shown on lines 30 through 36. Each *vertex* in the graph has another dictionary whose value contains the authority and hub score calculated during the iteration. The *vertex* has an outbound connection to another on. A dictionary with the key value ’o’ has an array containing all outbound *vertices* connections. An additional dictionary, **node_value** on line 37, has the authority and hub tuple initial iteration values. The remaining lines are the implementation of the algorithm provided in textbook [3]

Listing 1: Script Generate n Number of HITS Iterations

```
28      # creating the graph
29      graph = {’A’: {’A’:[0, 0]},
30               ’B’: {’o’:[’A’], ’B’:[0, 0]},
31               ’C’: {’o’:[’A’, ’B’, ’D’], ’C’:[0, 0]},
32               ’D’: {’D’:[0, 0]},
33               ’E’: {’o’:[’D’], ’E’: [0, 0]},
34               ’F’: {’o’:[’D’], ’F’: [0, 0]},
```

Exercise 10.3 continued on next page. . .

```
                'G': {'G':[0, 0]}}

    node_value = {'A': [1, 1], 'B': [1, 1],'C': [1, 1],'D': [1, 1],'E': [1, 1],'F': [1, 1],'G': [1, 1]}

    for k in range(5):
        hub_total = 0
        auth_total = 0
        for node in graph:
            if 'o' in graph[node]:
                for vertex in graph[node]['o']:
                    graph[node][node][1] += node_value[vertex][0]
                    graph[vertex][vertex][0] += node_value[node][1]
                    hub_total += node_value[vertex][0]
                    auth_total += node_value[node][1]

        for node in graph:
            node_value[node][0] = graph[node][node][0] / auth_total
            node_value[node][1] = graph[node][node][1] / hub_total
            graph[node][node][0] = 0
            graph[node][node][1] = 0

        print('\nfor k=%d HITS Calculation is:' % (k + 1))
        for node in sorted(node_value):
            print('%c = (%.2f, %.2f)' % (node, node_value[node][0], node_value[node][1]))

    print('\nEnd Time:  %s' % strftime("%a,  %b %d, %Y at %H:%M:%S", localtime()))
    print('Execution Time: %.4f seconds' % (time()-start))
    return
```

1.1.2 PageRnk

To calculate five iterations of the PageRank algorithm for the graph shown on Figure 1, we developed the *Python* scripts *pagerank.py*. It has a difference the data structure as the one implemented in **HITS**. Here the *vertices* contain the values of the inward connection with them. Also, the dictionary contains a key *size* indicating the number of outbound connections, and the key *pr* indicating the calculated PR scored during the iteration. Additionally, the PR score at the beginning of the iteration is captured in the dictionary **pr** on line 45.

Listing 2: Script Generate n Number of PageRank Iterations

```
20 cachedStopWords = stopwords.words("english")
21
22
23 def main():
24     # record running time
25     start = time()
26     print('Starting Time: %s\n' % strftime("%a,  %b %d, %Y at %H:%M:%S", localtime()))
27
28     # creating the graph
29
```

Exercise 10.3 continued on next page...

```
    graph = {'A': {'links': ['B', 'C'], 'size': 0, 'pr': 0},
             'B': {'links': ['C'], 'size': 1, 'pr': 0},
             'C': {'links': [], 'size': 3, 'pr': 0},
             'D': {'links': ['C', 'E', 'F'], 'size': 0, 'pr': 0},
             'E': {'links': [], 'size': 1, 'pr': 0},
             'F': {'links': [], 'size': 1, 'pr': 0},
             'G': {'links': [], 'size': 0, 'pr': 0}}
    """
    graph = {'A': {'links':['C'], 'size': 2, 'pr': 0},
             'B': {'links': ['A'], 'size':1, 'pr': 0},
             'C': {'links': ['A', 'B'], 'size': 1, 'pr': 0}}
    """
    landa = 0.15
    n = len(graph)
    # pr = {'A': 1/n, 'B': 1/n, 'C': 1/n}
    pr = {'A': 1/n, 'B': 1/n, 'C': 1/n, 'D': 1/n, 'E': 1/n, 'F': 1/n, 'G': 1/n}
    y = lambda x: graph[x]['links']
    l = lambda x: 1 if graph[x]['size'] == 0 else graph[x]['size']
    page_rank = lambda k: landa/n + (1 - landa) * sum([pr[x]/l(x) for x in y(k)])

    for k in range(5):
        print('Iteration k=%d:' % k)
        for node in sorted(graph):
            graph[node]['pr'] = page_rank(node)
            print('PR(%c) = %.4f' % (node, graph[node]['pr']))

        for node in graph:
            pr[node] = graph[node]['pr']
        print()
```

Exercise 10.3 continued on next page. . .

2 Solution

Looking at Figure 2, we can find some similarities and difference between **HITS** and **PageRank** algorithm results. In the HITS algorithm at the fifth iteration, *vertex* **D** shows the highest authority score, similarly the same *vertex* has the highest PR rank scored through the same number of iterations. The same was noticed for *vertex* B, which has the second highest authority rank scored on the **HITS** algorithm. It also has the second high PR score in the **PageRank** algorithm.

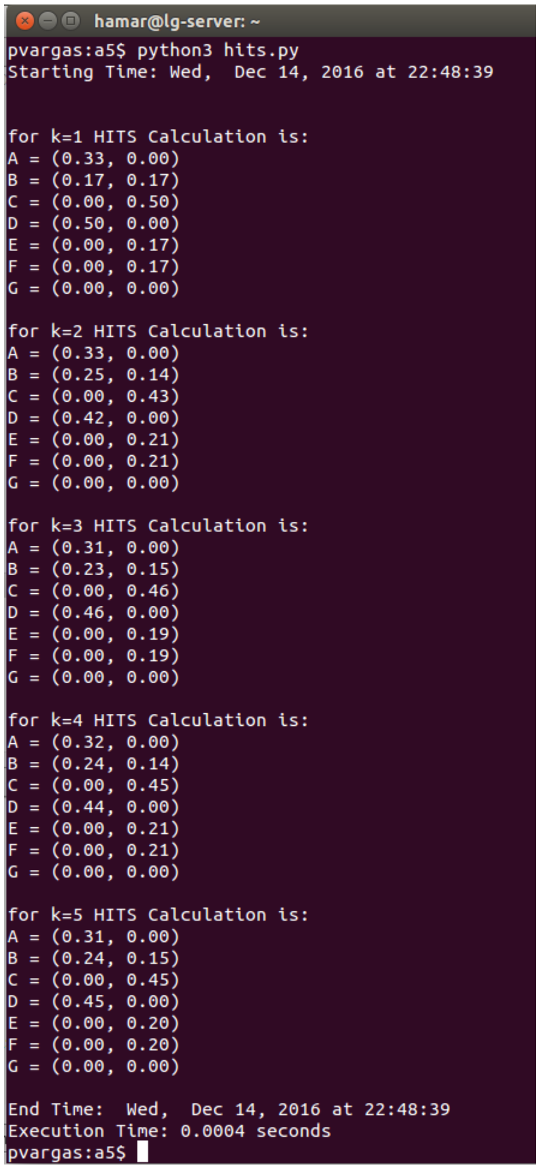
PageRank seemed to converged at the third iteration while **HITS** appeared to converged at the fifth iteration.

HITS providesa more descriptive information about why a particular node is important. It tells whether a node is providing information as an authority, or whether it is a hub pointing to the correct authority.

In **PageRank**, *vertex* G has a very small PR score even-though G is not connected to any of the other nodes in the graph. This is due to the λ factor or the damping factor which, in our case, was set to 0.15. However, the same *vertex* shows no scores, either as an authority or as a hub.

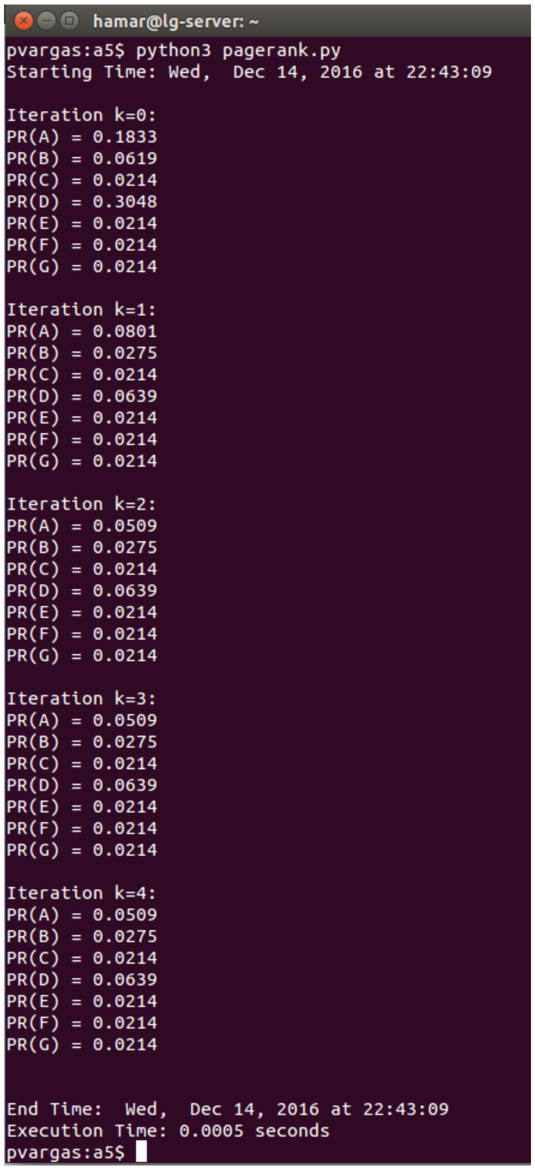
Exercise 10.3 continued on next page. . .

Figure 2: Result After 5 Iterations for HITS and PageRank Algorithms



```
pvargas:a5$ python3 hits.py
Starting Time: Wed,  Dec 14, 2016 at 22:48:39


for k=1 HITS Calculation is:
A = (0.33, 0.00)
B = (0.17, 0.17)
C = (0.00, 0.50)
D = (0.50, 0.00)
E = (0.00, 0.17)
F = (0.00, 0.17)
G = (0.00, 0.00)

for k=2 HITS Calculation is:
A = (0.33, 0.00)
B = (0.25, 0.14)
C = (0.00, 0.43)
D = (0.42, 0.00)
E = (0.00, 0.21)
F = (0.00, 0.21)
G = (0.00, 0.00)

for k=3 HITS Calculation is:
A = (0.31, 0.00)
B = (0.23, 0.15)
C = (0.00, 0.46)
D = (0.46, 0.00)
E = (0.00, 0.19)
F = (0.00, 0.19)
G = (0.00, 0.00)

for k=4 HITS Calculation is:
A = (0.32, 0.00)
B = (0.24, 0.14)
C = (0.00, 0.45)
D = (0.44, 0.00)
E = (0.00, 0.21)
F = (0.00, 0.21)
G = (0.00, 0.00)

for k=5 HITS Calculation is:
A = (0.31, 0.00)
B = (0.24, 0.15)
C = (0.00, 0.45)
D = (0.45, 0.00)
E = (0.00, 0.20)
F = (0.00, 0.20)
G = (0.00, 0.00)

End Time:  Wed,  Dec 14, 2016 at 22:48:39
Execution Time: 0.0004 seconds
pvargas:a5$
```

(a) HITS 5 Iteration Calculation Results



```
pvargas:a5$ python3 pagerank.py
Starting Time: Wed,  Dec 14, 2016 at 22:43:09

Iteration k=0:
PR(A) = 0.1833
PR(B) = 0.0619
PR(C) = 0.0214
PR(D) = 0.3048
PR(E) = 0.0214
PR(F) = 0.0214
PR(G) = 0.0214

Iteration k=1:
PR(A) = 0.0801
PR(B) = 0.0275
PR(C) = 0.0214
PR(D) = 0.0639
PR(E) = 0.0214
PR(F) = 0.0214
PR(G) = 0.0214

Iteration k=2:
PR(A) = 0.0509
PR(B) = 0.0275
PR(C) = 0.0214
PR(D) = 0.0639
PR(E) = 0.0214
PR(F) = 0.0214
PR(G) = 0.0214

Iteration k=3:
PR(A) = 0.0509
PR(B) = 0.0275
PR(C) = 0.0214
PR(D) = 0.0639
PR(E) = 0.0214
PR(F) = 0.0214
PR(G) = 0.0214

Iteration k=4:
PR(A) = 0.0509
PR(B) = 0.0275
PR(C) = 0.0214
PR(D) = 0.0639
PR(E) = 0.0214
PR(F) = 0.0214
PR(G) = 0.0214


End Time:  Wed,  Dec 14, 2016 at 22:43:09
Execution Time: 0.0005 seconds
pvargas:a5$
```

(b) PageRank 5 Iteration Calculation Results

3 Problem 2

Using `http://www.cs.cornell.edu/People/tj/svm_light/` work through the "Inductive SVM" example, discuss in detail the steps and resulting output.

The "Inductive SVM" example from [2] used the Reuters-21578 collection. 1000 positive and 1000 negative examples were used for the machine-learning training, in order to determine if a particular document in the

Problem 2 continued on next page. . .

collection should be classified as "corporate acquisition". A test file < *test.dat* > representing 600 documents, it served as the input to evaluate the accuracy of the created model.

3.1 Input File Format

The training file has the form:

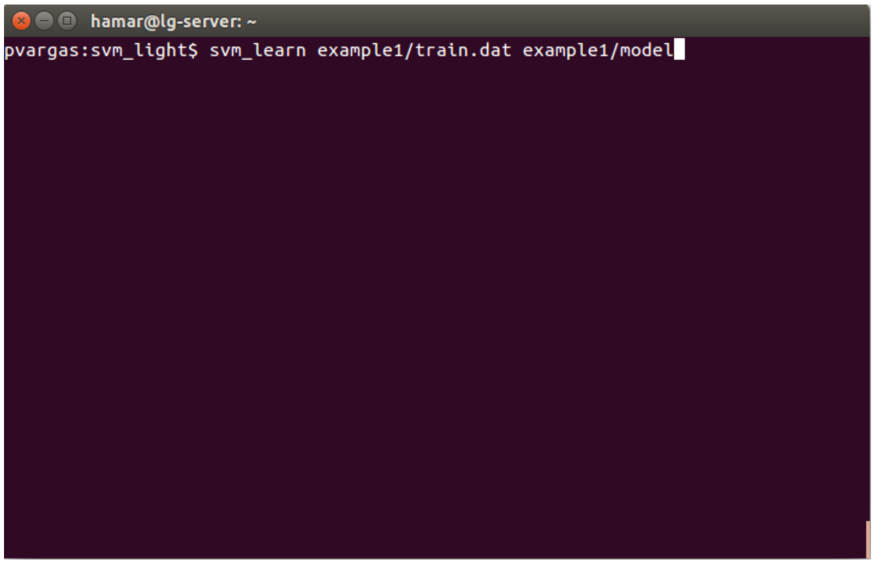
```
<line> .=. <target> <feature>:<value> <feature>:<value> ... <feature>:<value> # <info>
<target> .=. +1 | -1 | 0 | <float>
<feature> .=. <integer> | "qid"
<value> .=. <float>
<info> .=. <string>
```

Considering that the **positive** and **negative** examples are included in a single file, in order to distinguish if the training document is for a positive example, the value of **1** is placed at the beginning of the record. On the contrary, a **-1** value indicates the training record is related to a negative example.

In order to create our prediction model, the following instruction should be placed in the command-line:

svm_learn example1/train.dat example1/model

Figure 3: Training Module



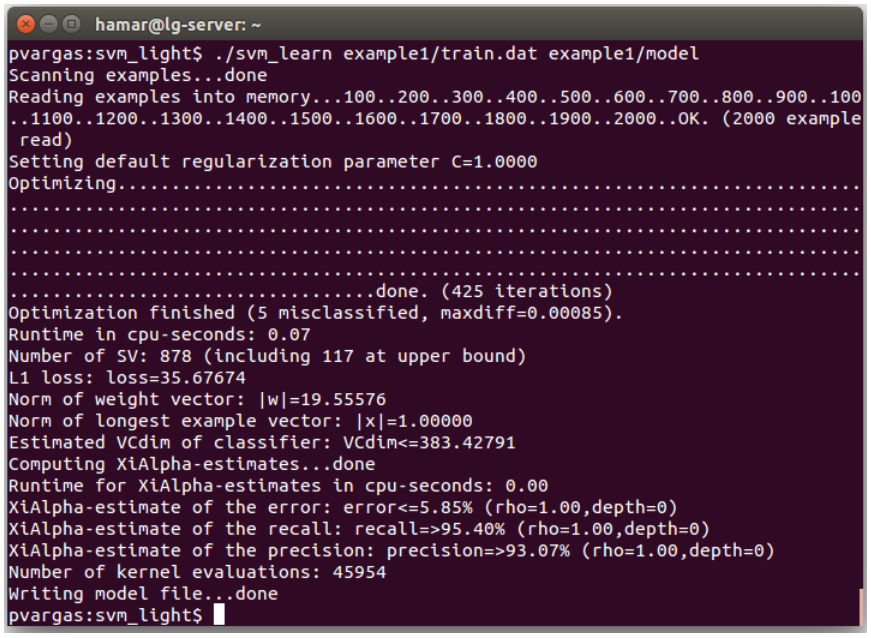
```
pvargas:svm_light$ svm_learn example1/train.dat example1/model
```

Problem 2 continued on next page...

3.2 Resulting Output

The accuracy of the report is shown on Figure 4

Figure 4: Data Training Result



```
pvargas:svm_light$ ./svm_learn example1/train.dat example1/model
Scanning examples...done
Reading examples into memory...100..200..300..400..500..600..700..800..900..100
..1100..1200..1300..1400..1500..1600..1700..1800..1900..2000..OK. (2000 example
 read)
Setting default regularization parameter C=1.0000
Optimizing.....................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
.................................done. (425 iterations)
Optimization finished (5 misclassified, maxdiff=0.00085).
Runtime in cpu-seconds: 0.07
Number of SV: 878 (including 117 at upper bound)
L1 loss: loss=35.67674
Norm of weight vector: |w|=19.55576
Norm of longest example vector: |x|=1.00000
Estimated VCdim of classifier: VCdim<=383.42791
Computing XiAlpha-estimates...done
Runtime for XiAlpha-estimates in cpu-seconds: 0.00
XiAlpha-estimate of the error: error<=5.85% (rho=1.00,depth=0)
XiAlpha-estimate of the recall: recall=>95.40% (rho=1.00,depth=0)
XiAlpha-estimate of the precision: precision=>93.07% (rho=1.00,depth=0)
Number of kernel evaluations: 45954
Writing model file...done
pvargas:svm_light$ 
```

3.2.1 Number of SV

Although the entire collection has **9,947** words, the number of features required to make the classification for any particular document in the collection as a "corporate acquisition" was calculated at: SV = **878**. Therefore, the model only requires **878** features to obtain our classification.

3.2.2 Loss Function

Measurement of empirical error, define the error the classifier made on the training set.

3.2.3 Norm of Weight Vector

According to [3], for all document pairs in the rank data, we would like the score for the document with the higher relevance rating (or rank) to be greater than the score for the document with the lower relevance rating. For our data $|x| = 19.55576$.

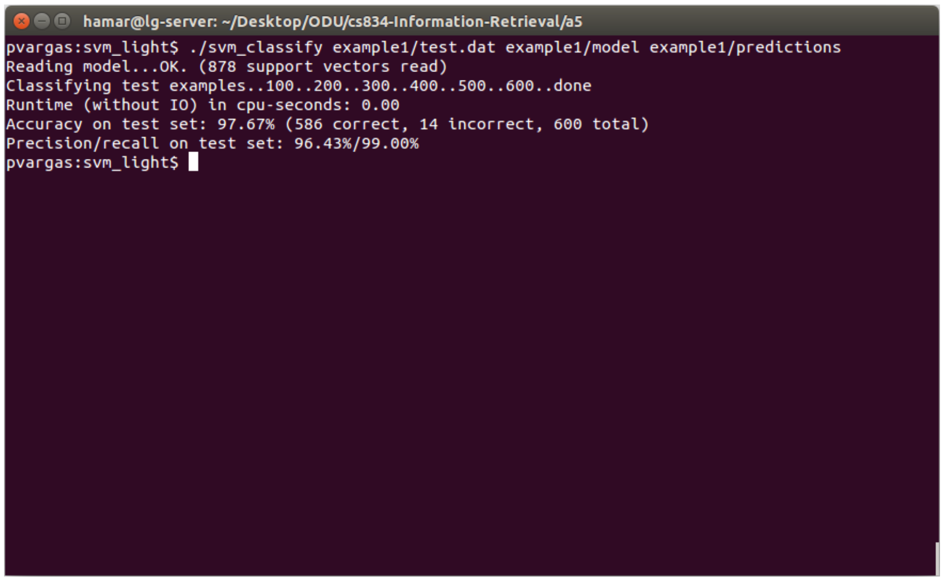
3.2.4 VC Dimension Result

The VC dimension or **VapnikChervonenkis dimension** measures the number of points required to shatter the classification model. For this example: $VCdim <=$ **383.42791**

3.2.5 Testing the Model

Looking at Figure 5, we can see that 14 out of 600 documents were incorrectly predicting, bringing the accuracy of this model to **97.6%** accuracy. This error falls into the predicted *XiAlpha-estimate* error of less than 5.85%.

Figure 5: Data Test Result


hamar@lg-server: ~/Desktop/ODU/cs834-Information-Retrieval/a5

```
pvargas:svm_light$ ./svm_classify example1/test.dat example1/model example1/predictions
Reading model...OK. (878 support vectors read)
Classifying test examples..100..200..300..400..500..600..done
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 97.67% (586 correct, 14 incorrect, 600 total)
Precision/recall on test set: 96.43%/99.00%
pvargas:svm_light$ 
```

4 Problem 3

- create your own example modeled after the "Inductive SVM" example
- pick a topic (e.g., "Australia") and provide 100 positive and 100 negative examples for training data:
    a. using the Reuters-21578 collection (linked from the SVMlight page)
    b. or, create your own collection with crawled web pages
- pick 30 documents not in the training set for your test data
- stem the words in the collection, using TFIDF as the features (compute for the 230 documents)
- train, classify, and discuss the results

4.1 Approach

4.1.1 Topic Selection

The collection selected for this exercise was **Reuters-21578**. This collection was accessed via the python library **nltk.corpus**. The topic chosen was "**exports**." In order to obtain 100 positive examples for data training, a *Python* function was creating. The script *svm-light.py* contains this function (*find_features*) which is shown on Figure 3.

4.1.2 Selecting Positive Example Documents

The first line for all the documents in the corpus is the title, which is representative its content. The function *find_features* takes for argument a string, which is the term scanned in the document title (line 140). In order to obtain a document with plenty features, a minimum length of 500 characters was required for the document size to be considered for selection. The number of documents in the corpus under this criteria came out to be **136**.

The function returns an array containing the index for all the documents in the corpus where the term appears. The content of those documents were visually inspected for validity, and the first 100 indexes were stored in a file: *positive.dat*. Later on, this file was going to be used as a input for the 100 positive training data examples.

Listing 3: Scan for Feature in Document Title

```
132 def find_features(feature):
133     feature = tokenize(feature)[0]
134     documents = reuters.fileids()
135 
136     id_array = []
137     for id in documents:
138         doc = reuters.raw(id)
139         first_line = doc.split('\n')[0]
140         if feature in tokenize(first_line) and len(doc) > 500:
141             print('<----------- %s -------------->' % id)
142             print(doc)
```

Problem 3 continued on next page...

```python
            print(len(doc))
            id_array.append(id)

    print(id_array)
    print(len(id_array))
    categories = reuters.categories()
    print(categories)

    return id_array
```

4.1.3 Selecting Negative Example Documents

A similar approach was used to obtain the negative training examples. The term "**bank**" was entered as the feature into the function *find_features*. The result was an array of 229 document indexes. The first 100 documents were viewed for content accuracy. In this case, it was ensured the content was not related to the term **export**. The document indexes were saved into the file: *negative.dat*.

4.1.4 Selecting Testing Documents

Considering that only 100 indexes from 136 available were taken to be trained as positive examples, and the next 15 indexes were selected to serve as input for the testing data. The same was done to find negative documents for the testing stage. In total, 30 documents were used for testing: 15 positive and 15 negative. The indexes for those documents were saved under the files: *positive-test.dat* and *negative-test.dat* respectively.

4.1.5 Building Word Index

The online resource on [1] was very useful to stem the words in the collection. The PorterStemmer function from the **nltk** library facilitated the stemming action. The function *create_vocabulary* completed the stemming task. The function shown on Listing 4 reads all the documents in the corpus, then it tokenized each of the document and it placed the stemmed word into a set object(lines 305-306). Considering that the set does not have duplicate elements, the resulting set becomes the vocabulary. Its content was written to the file: *words*.

Listing 4: Stemming Words in the Collection

```python
301 def create_vocabulary():
302     documents = reuters.fileids()
303     for document_id in documents:
304         text = tokenize(reuters.raw(document_id))
305         for word in text:
306             word_set.add(word)
307 
308     with open('words', 'w') as f:
309         for record in word_set:
310             f.write('%s\n' % record)
```

Problem 3 continued on next page. . .

4.1.6 Creating Training Feature Records

SVM^{light} requires the training input file in a specific format, as it is shown in section 3.1. The resource cited on [1] contained a function that calculates **TFIDF** which is the value for the features in our training document. This function named *tf_idf* was incorporated into the script *svm-light.py*, and it takes a string as a parameter, which in our case, it is the entire document. This function generates a **TfidVectorizer** object available in the **sklearn.feature_extraction** library.

The function *tf_idf* calls the function *feature_values* prior to passing the calculated vector. This last function takes all elements in the vector and converts them into a set of tuples (w, v), where w is the tokenized word, and v is its calculated **TFIDF** value.

Finally, the function *write_train_data*, shown on Listing 5, for each index marked as positive or negative in the training files, *positive.dat* and *negative.dat* respectively, is opened and passed as an argument to the *tf_idf* (line 113). The returned tuples (w, v) are placed into a dictionary (lines 119-120), where the word w is converted to the index in the vocabulary.

Since SVM^{light} requires the features to be placed in ascending order, the dictionary is sorted by key value prior to writing the features into *train.dat*, which it is the file that it will be used for training data.

Listing 5: Creating Training Data

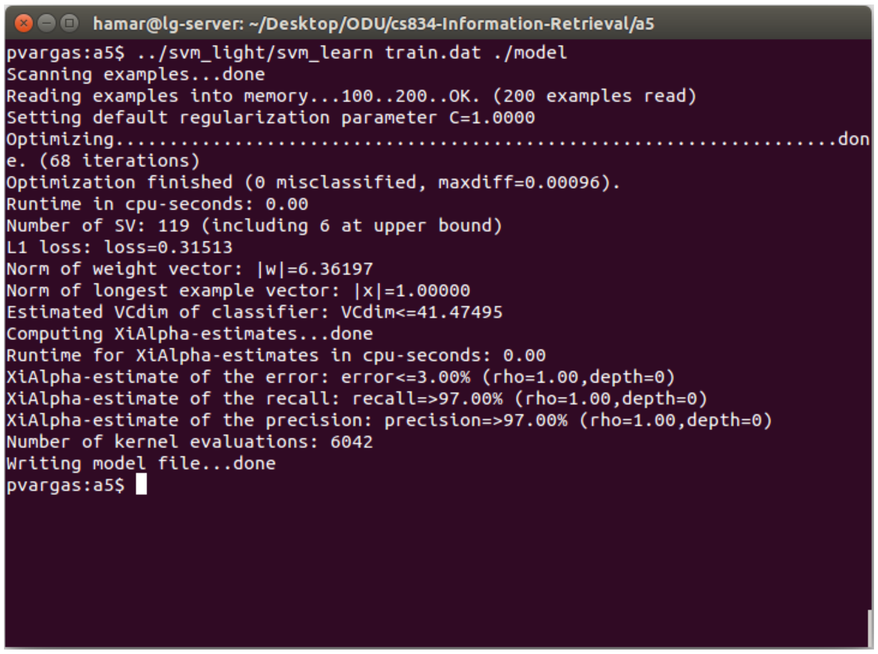
```
102 def write_train_data(train_file, mode, train_data_file, pos_neg):
103     # get training data
104     train_data = []
105     with open(train_file, mode) as w_file:
106         with open(train_data_file, ’r’) as f:
107             for record in f:
108                 record = record.strip()
109                 # include index
110                 train_data.append(record)
111 
112                 # get TFIDF features for document
113                 features = tf_idf(reuters.raw(record))
114 
115                 unorder_list= {}
116 
117                 print(pos_neg, end=’ ’)
118                 w_file.write(’%s ’ % pos_neg)
119                 for key, value in features:
120                     unorder_list[word_dic[key]] = value
121                     # print(’%d:%f’ % (word_dic[key], value), end=’ ’)
122 
123                 for key in sorted(unorder_list):
124                     print(’%d:%f’ % (key, unorder_list[key]), end=’ ’)
125                     w_file.write(’ %d:%f’ % (key, unorder_list[key]))
126 
127                 print()
128                 w_file.write(’\n’)
129     return
```

4.2 Solution

4.2.1 Training SMV^{light}

To create our model, we typed the command **svm_learn train.dat model**, as it is shown on Figure 6. The file *train.dat* is the training data, and *model* is the file that resulted from the machine learning. The estimated error is within **3%**, therefore we are expected to have an accuracy greater or equal to **97%**.

Figure 6: Data Training Result



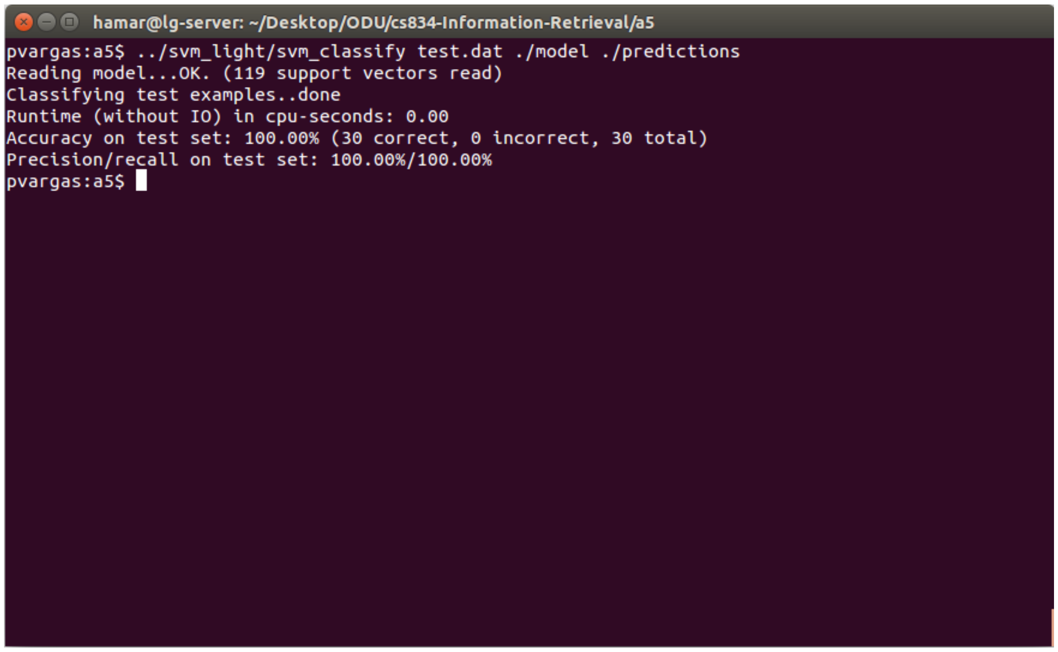
```
hamar@lg-server: ~/Desktop/ODU/cs834-Information-Retrieval/a5
pvargas:a5$ ../svm_light/svm_learn train.dat ./model
Scanning examples...done
Reading examples into memory...100..200..OK. (200 examples read)
Setting default regularization parameter C=1.0000
Optimizing..................................................................don
e. (68 iterations)
Optimization finished (0 misclassified, maxdiff=0.00096).
Runtime in cpu-seconds: 0.00
Number of SV: 119 (including 6 at upper bound)
L1 loss: loss=0.31513
Norm of weight vector: |w|=6.36197
Norm of longest example vector: |x|=1.00000
Estimated VCdim of classifier: VCdim<=41.47495
Computing XiAlpha-estimates...done
Runtime for XiAlpha-estimates in cpu-seconds: 0.00
XiAlpha-estimate of the error: error<=3.00% (rho=1.00,depth=0)
XiAlpha-estimate of the recall: recall=>97.00% (rho=1.00,depth=0)
XiAlpha-estimate of the precision: precision=>97.00% (rho=1.00,depth=0)
Number of kernel evaluations: 6042
Writing model file...done
pvargas:a5$
```

4.2.2 Testing the Model

To run the test, we typed the command **svm_classify test.dat predictions**, as shown on Figure 7. The file *test.dat* contained the test data, and *predictions* is the file that resulted after running the classification application. The accuracy of our test came out to be **100%**, which agrees with the prediction on section 4.2.1, which predicted that the results should be greater than or equal to **97%**.

Problem 3 continued on next page. . .

Figure 7: Testing Results



hamar@lg-server: ~/Desktop/ODU/cs834-Information-Retrieval/a5

```
pvargas:a5$ ../svm_light/svm_classify test.dat ./model ./predictions
Reading model...OK. (119 support vectors read)
Classifying test examples..done
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 100.00% (30 correct, 0 incorrect, 30 total)
Precision/recall on test set: 100.00%/100.00%
pvargas:a5$
```

References

[1] *Classifying reuters-21578 collection with python. (n.d.) retrieved december 14, 2016, from https://miguel-malvarez.com/2015/03/20/classifying-reuters-21578-collection-with-python-representing-the-data/.*

[2] *Svm light. (n.d.) retrieved december 13, 2016, from http://www.cs.cornell.edu/people/tj/svm_light/.*

[3] T. S. W.B. Croft, D. Metzler, *Search Engine Information Retrieval in Practice*, Pearson Education, 2015.