

Predicting Stackoverflow tags

02807 Final project

December 1, 2016

1 Introduction

In this project two models predicting tags associating a given text will be proposed, implemented, and tested. The problem will be handled as a multi-label classification problem.

The models will be trained on posts from Stackoverflow where associated tags have been given.

The top 20 frequent tags and posts containing these tags will be used for training and prediction.

The first model is an unsupervised model where the distribution of the tags text types are attempted learned using the K-means clustering algorithm [1]. By clustering the texts into 20 clusters each cluster will (hopefully) represent a given tag. The prediction of tags can then be done by looking at a given text's closest clusters, where each cluster has been assigned the tag, which was most present in the cluster during training.

The second model is a supervised model based on Decision Trees [1]. Inspired by the *Random Forest* ensemble model an ensemble of decision trees are trained on subsets of the training data, and thereby resembles the structure of the *Random Forest* model. Prediction of tags are then based on the mean probabilities predicted by all decision trees.

2 The data

The dataset consists of two XML files, one containing all possible tags and their corresponding counts, and one containing posts with *title*, *body*, *tags* and some meta data.

The total size of the files are approximately 49GB in uncompressed format.

3 Preprocessing

The preprocessing step regards the transforming of questions in an `XML` file to processed questions in a `.csv` file. The question `.csv` file will consist of a list of word indices from the title and body including a list of tag indices for each question. This transforming process also includes disregarding questions that does not have any of the top N tags attached. At the same time two other `.csv` files are created: One containing all unique words in the extracted questions, and one containing the unique tags used.

The processing of each questions contains the following steps (code can be found in appendix A.1)

1. Replace all links with `<link>`
2. Remove certain unwanted symbols
3. Remove suffix from words (e.g. *haven't* \rightarrow *have*)
4. Remove line breaks
5. Replace digits with `<digit>`
6. Remove double whitespaces
7. Reduce words to their word stem (e.g. *lines* \rightarrow *line*)
8. Lemmatize words (e.g. *better* \rightarrow *good*)

Finally the unique words used as word dictionary were filtered by removing words occurring in more than 50% of the questions and words occurring in less than 0.1% of the questions. Also english stop words were removed making use of the NLTK library.

All these steps are used in order to reduce the dimensionality of the word space without removing much information. Here the assumption is, that e.g. words like *better* and *good* kind of adds the same meaning to the sentence, and the same with e.g. two numbers.

4 Methods

4.1 Distributed file loading

Since the size of the final processed questions file is approximately 11GB, it will not be feasible to load into memory on most laptops. Therefore it will be necessary to load the file in smaller chunks.

The following code illustrates how the file `posts.csv` can be divided into byte-chunks. I.e. the following generator yields a list of tuples (`from_byte`, `size`) where `from_byte` is the index in the file in bytes and `size` is the size of the given chunk in bytes.

```
1 || with open('posts.csv', 'rb') as f:
2 ||     while True:
```

```

3 |     start = f.tell()
4 |     f.seek(chunk_size, 1)
5 |     s = f.readline()
6 |     yield start, f.tell() - start
7 |     if not s: break

```

The `chunk_size` is a given minimum size of each chunk. The `f.readline()` makes sure the chunk ends at the end of a line.

A chunk of lines from the file can then be loaded using the following lines

```

1 | # Seek to chunk start bytes
2 | f.seek(from_bytes)
3 |
4 | # Read end of chunk until end of line
5 | chunk = f.read(size)
6 |
7 | # Split in lines (Removing the last newline)
8 | lines = chunk.rstrip('\n').split('\n')

```

4.2 Feature hashing

In order to work with the text data each line of word indices is transformed into a sparse matrix using the word dictionary and `scipy`'s *Compressed Sparse Row matrix* `scipy.sparse.csr_matrix`.

A file chunk can be converted to a sparse matrix representation in the following way (simplified code):

```

1 | indptr, indices, data, tags = [0], [], [], []
2 | for input_indices in chunk_to_indices(chunk):
3 |     for idx in input_indices:
4 |         indices.append(idx)
5 |         data.append(1)
6 |     indptr.append(len(indices))
7 |
8 | X = csr_matrix(
9 |     (data, indices, indptr),
10 |    shape=(len(indptr) - 1, word_count)
11 | )

```

4.3 K-means clustering

4.3.1 Serial

The regular serial in-memory version of K-means clustering algorithm is shown in algorithm 1.

Algorithm 1 Serial K-means clustering algorithm

```

1: procedure KMEANSCLUSTERING( $X, K$ )
2:    $\#$  Initialize cluster centers
3:   for  $k = 0$  to  $K - 1$  do
4:      $\mu_k \leftarrow$  random point in  $X$ 
5:    $\#$  Run iterations
6:   while  $iter < max\_iter$  do
7:      $\#$  Update cluster means
8:      $\mu_{old} = \mu$ 
9:     for  $k = 0$  to  $K - 1$  do
10:       $C_k \leftarrow \{\text{Points in } X \text{ closest to } \mu_k\}$ 
11:       $\mu_k \leftarrow \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$ 
12:      $\#$  Check convergence criteria
13:      $norm \leftarrow \|\mu - \mu_{old}\|$ 
14:     if  $norm < \epsilon$  then
15:       break

```

The distance measure used for finding closest points in X is the *cosine similarity*. I.e. the distance will be defined as

$$\text{dist}(x_1, x_2) = \frac{x_1 x_2}{\|x_1\| \|x_2\|} \quad (4.1)$$

4.3.2 Distributed

The proposed distributed K-means clustering algorithm, which loads the data matrix X in chunks, is shown in algorithm 2.

Algorithm 2 Distributed K-means clustering algorithm

```

1: procedure KMEANSCLUSTERINGDISTRIBUTED( $X, K$ )
2:   # Initialize cluster centers
3:   for  $k = 0$  to  $K - 1$  do
4:      $\mu_k \leftarrow$  random point in  $X$ 
5:   # Run iterations
6:   while  $iter < max\_iter$  do
7:     # Initialize shared cluster sums and cluster point counts.
8:     for  $k = 0$  to  $K - 1$  do
9:        $Csum_k \leftarrow 0$ 
10:       $Ccount_k \leftarrow 0$ 
11:     # Process each chunk in a distributed manner
12:     for all Chunks  $X_{chunk}$  in  $X$  do
13:       for  $k = 0$  to  $K - 1$  do
14:          $C_k \leftarrow \{\text{Points in } X_{chunk} \text{ closest to } \mu_k\}$ 
15:          $Csum_k \leftarrow Csum_k + \sum_{x_i \in C_k} x_i$ 
16:          $Ccount_k \leftarrow Ccount_k + |C_k|$ 
17:     # Gather results and update cluster means
18:      $\mu_{old} = \mu$ 
19:     for  $k = 0$  to  $K - 1$  do
20:        $\mu_k \leftarrow \frac{Csum_k}{Ccount_k}$ 
21:     # Check convergence criteria
22:      $norm \leftarrow \|\mu - \mu_{old}\|$ 
23:     if  $norm < \epsilon$  then
24:       break

```

4.3.3 Implementation

Simplified implementation of a single iteration of the distributed K-means (see full code in appendix A.2):

```

1 | cluster_sums = {k: np.zeros((1, word_count)) for k in
   | range(0, K)}
2 | cluster_counts = {k: 0 for k in range(0, K)}
3 |
4 | for chunk in chunks:
5 |
6 |     # Load chunk lines to sparse matrix
7 |     X = chunk_to_sparse_mat(chunk)
8 |
9 |     # Get closest cluster indices
10 |    max_idx = sparse_matrix_to_cluster_indices(X, mu)
11 |
12 |    # Assign points to clusters
13 |    mu_subs = collections.defaultdict(list)
14 |    for i, k in enumerate(max_idx):
15 |        mu_subs[k].append(X[i].toarray())
16 |
17 |    # Compute sub-means
18 |    for k in range(0, K):

```

```

19     mu_sub = mu_subs[k]
20     if len(mu_sub) == 0:         continue
21     cluster_sums[k] += np.asarray(mu_sub).mean(axis=0)
22     cluster_counts[k] += 1
23
24 # Save old means
25 mu_old = np.array(mu, copy=True)
26
27 # Update means
28 for k in range(0, K):
29     count = cluster_counts[k]
30     if count == 0: continue
31     mu[k] = cluster_sums[k] / cluster_counts[k]
32
33 # Check convergence criteria
34 mu_norm = np.linalg.norm(mu - mu_old)
35
36 if mu_norm < epsilon:
37     print('Converged after %d iterations' % (iteration+1))
38     break

```

4.4 Distributed decision tree ensemble

An alternative to the unsupervised approach is a supervised approach using an ensemble of decision trees. This idea is inspired by the *Random Forest* model which is an ensemble of decision trees trained on bootstrapped features.

For this specific task the decision trees are trained on each chunk of data. Hence the randomness here lies more in the fact that each tree will only see a subset of the training data.

The algorithm outline can be seen in algorithm 3.

Algorithm 3 Distributed decision tree ensemble algorithm

```

1: procedure DECISIONTREEENSEMBLEDISTRIBUTED( $X, Y, K$ )
2:   # Process each chunk in a distributed manner
3:   for all Chunks  $(X_{\text{chunk}}, Y_{\text{chunk}})$  in  $(X, Y)$  do
4:      $T \leftarrow$  Decision tree trained on  $(X_{\text{chunk}}, Y_{\text{chunk}})$ 
5:     # Dump decision tree to file

```

Finally the prediction of tags can be done by computing tag probabilities for each trained decision tree, and basing the prediction on the mean tag probabilities of all classifiers.

4.4.1 Implementation

Simplified implementation of the training of the distributed decision tree ensemble algorithm (see full code in appendix A.3):

```

1 from sklearn.tree import DecisionTreeClassifier
2 from sklearn.externals import joblib
3
4 for chunk in chunks:
5
6     # Convert to sparse matrix
7     X, Y = chunk_to_sparse_mat(chunk)
8
9     # Train decision tree
10    clf = DecisionTreeClassifier(
11        splitter='best',
12        max_features='auto',
13        max_depth=None,
14    )
15
16    # Fit data
17    clf.fit(X, Y)
18
19    # Save trained classifier
20    joblib.dump(clf, classifier_filename)

```

Here the *scikit-learn* library is used for the implementation of the decision tree class `DecisionTreeClassifier`.

4.5 Parallel processing

Since these proposed algorithms are implemented in a distributed manner it makes sense to run them in parallel. For this the Python multiprocessing library is used.

The general parallel implementations used in this project follow the following structure:

```

1 import multiprocessing
2
3 # Initialize shared variable manager
4 manager = multiprocessing.Manager()
5 lock = multiprocessing.Lock()
6
7 for chunks in list_of_chunks:
8     p = multiprocessing.Process(
9         target=process_chunks,
10        kwargs={
11            'chunks': chunks,
12            ...
13            'lock': lock
14        }
15    )
16    processes.append(p)
17
18 # Start processes
19 for p in processes:
20    p.start()

```

```

21 |
22 | # Wait for processes to finish
23 | for p in processes:
24 |     p.join()
25 |
26 | # Use results
27 | # ...

```

where `process_chunks` processes a set of chunks of the data file (and thereby varies from K-means to decision tree ensemble). The `manager` is used for sharing values between spawned processes and the `lock` is used for making sure multiple processes are not writing to a shared value simultaneously (which will result in only one of the values being actually written).

Example of how the multiprocessing lock works

```

1 | with lock:
2 |     shared_counter += 1

```

5 Results

For evaluating the performance of the models the *Precision at K* metric is used denoted $\text{Precision@}K$. This is given as the fraction of tags correctly retrieved in the top K predicted tags

$$\text{Precision@}K = \frac{\# \text{ of correctly retrieved tags in top } K \text{ predicted tags}}{\# \text{ of tags predicted}} \quad (5.1)$$

The provided results are obtained by evaluating the models on a testset which consists of 33% of the original dataset, which was left out during training.

In table 1 the obtained performance of the models can be seen.

Model	Precision@1	Precision@5	Precision@10
Baseline	0.050	0.250	0.500
K-means clustering	0.040	0.213	0.412
Decision tree ensemble	0.596	0.936	0.984

Table 1: $\text{Precision@}K$ values for the different models where *Baseline* is guessing a single tag at random.

In table 2 the run-times for the models can be seen.

Model	Training	Prediction
K-means clustering	$\approx 24\text{hours}$	$\approx 1\text{min}$
Decision tree ensemble	$\approx 10\text{min}$	$\approx 2\text{hours}$

Table 2: Approximate run-times for the models run on a 2.7 GHz Intel Core i5 with 4 physical cores, and 8GB memory.

6 Discussion

From the results shown in table 1 it is seen that the K-means clustering model performs worse than random guessing a single of the 20 tags for a question. Despite this not being completely comparable since some questions have more than a single tag it is still fair to say, that the model did not perform very well. This might be due to the fact that the K-means algorithm is very sensitive to the initialization of the cluster centers.

The decision tree ensemble is seen to perform much better than the K-means model and the baseline. When looking at the top 5 predicted tags the model retrieves $\approx 94\%$ of the test tags which is a very nice result.

From table 2 it is seen that training the decision tree ensemble is also significantly faster than the K-means model, but using the model for predicting takes much longer. This is because the model trains about 1,300 decision trees with no hard constraints on the depth of the tree, which results in each model being about 2.5MB which has to be loaded and used for prediction on all test observations, and finally the average class probabilities must be computed across the 1,300 decision trees.

7 Conclusion

From the project it can be concluded that for predicting tags from given Stackoverflow questions, an ensemble of decision trees trained on subsets of the data can be trained in a parallel, distributed manner, and will obtain a decent test performance.

It can also be concluded that using an unsupervised approach with K-means clustering, proper performance can be difficult to obtain and model training will be slow.

A Code snippets

A.1 Preprocess text

```

1 import re
2 import Stemmer
3 from nltk.stem import WordNetLemmatizer
4 lemmatizer = WordNetLemmatizer()
5
6 # Precompile regular expressions
7 reg_links = re.compile(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@
8 .&+]|[*%\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+')
9 reg_digits = re.compile(r'\b\d+\b')
10 reg_spaces = re.compile(r'\s{2,}')
11
12 reg_symbols = re.compile(r'^A-Za-z0-9(),!?\'\`')
13 reg_symb_1 = re.compile(r',')
14 reg_symb_2 = re.compile(r'!')
15 reg_symb_3 = re.compile(r'\(')
16 reg_symb_4 = re.compile(r'\)')
17 reg_symb_5 = re.compile(r'\?')
18 reg_symb_6 = re.compile(r'\'')
19
20 reg_suf_1 = re.compile(r'\s')
21 reg_suf_2 = re.compile(r'\ve')
22 reg_suf_3 = re.compile(r'n\t')
23 reg_suf_4 = re.compile(r'\re')
24 reg_suf_5 = re.compile(r'\d')
25 reg_suf_6 = re.compile(r'\ll')
26
27 stemmer = Stemmer.Stemmer('english')
28 word_to_stem = {}
29 def stem_word(word):
30     if not word in word_to_stem:
31         word_to_stem[word] = stemmer.stemWord(word)
32     return word_to_stem[word]
33
34 word_to_lemma = {}
35 def lemmatize_word(word):
36     if not word in word_to_lemma:
37         word_to_lemma[word] = lemmatizer.lemmatize(word)
38     return word_to_lemma[word]
39
40 def clean_string(text):
41     # Replace links with link identifier
42     text = reg_links.sub('<link>', text)
43
44     # Remove certain symbols
45     text = reg_symbols.sub(' ', text)
46
47     # Remove suffix from words
48     text = reg_suf_1.sub(' ', text)
49     text = reg_suf_2.sub(' ', text)

```

```
49 text = reg_suf_3.sub(' ', text)
50 text = reg_suf_4.sub(' ', text)
51 text = reg_suf_5.sub(' ', text)
52 text = reg_suf_6.sub(' ', text)
53
54 # Remove "\"" from string
55 text = reg_symb_6.sub(' ', text)
56
57 # Replace breaks with spaces
58 text = text.replace('<br />', ' ')
59 text = text.replace('\r\n', ' ')
60 text = text.replace('\r', ' ')
61 text = text.replace('\n', ' ')
62
63 # Pad symbols with spaces on both sides
64 text = reg_symb_1.sub(' ', ' ', text)
65 text = reg_symb_2.sub(' ! ', text)
66 text = reg_symb_3.sub(' ( ', text)
67 text = reg_symb_4.sub(' ) ', text)
68 text = reg_symb_5.sub(' ? ', text)
69
70 # Replace digits with 'DIGIT'
71 text = re_digits.sub('<DIGIT>', text)
72
73 # Remove double whitespaces
74 text = re_spaces.sub(' ', text)
75 text = text.strip()
76
77 # Convert to lowercase
78 text = text.lower()
79
80 # Stem each word
81 text = ' '.join(stem_word(word) for word in text.split(' '))
82
83 # Lemmatize each word
84 text = ' '.join(lemmatize_word(word) for word in text.
    split(' '))
```

A.2 Distributed K-means

```
1 import math
2 import collections
3 import numpy as np
4 import multiprocessing
5 import time
6
7 import helpers
8 import config
9
10 # Read tags
11 tags, tag2idx, tag_count = helpers.read_tags()
12
13 # Read words
```

```
14 words, word2idx, word_count = helpers.read_words()
15
16 # Clusters
17 K = tag_count
18
19 # Initialize cluster centers
20 mu = np.random.rand(K, word_count)
21
22 # Get chunks
23 chunk_reader = helpers.ChunkReader(post_filename=config.
    paths.TRAIN_DATA_IDX, chunk_size=config.data.CHUNK_SIZE)
    # TODO: Change
24 chunks = [chunk for chunk in chunk_reader]
25 chunk_count = len(chunks)
26
27 # Split chunks across processes
28 n = math.ceil(chunk_count / config.algorithm.PROCESS_COUNT)
29 chunks_split = []
30 for i in range(0, len(chunks), n):
31     chunks_split.append(chunks[i:i+n])
32
33 # Initialize shared variable manager
34 manager = multiprocessing.Manager()
35 lock = multiprocessing.Lock()
36
37 # Define function to run in parallel
38 def process_chunks(chunks, word_count, K, mu, cluster_sums,
    cluster_counts, lock):
39     for chunk in chunks:
40
41         # Convert to sparse matrix
42         X, _ = helpers.chunk_to_sparse_mat(chunk, word_count)
43
44         if X is None: continue
45
46         # Get closest cluster indices
47         max_idx = helpers.sparse_matrix_to_cluster_indices(X, mu
    )
48
49         mu_subs = collections.defaultdict(list)
50         for i, k in enumerate(max_idx):
51             mu_subs[k].append(X[i].toarray())
52
53         # Compute sub-means
54         for k in range(0, K):
55             mu_sub = mu_subs[k]
56             if len(mu_sub) == 0: continue
57
58             with lock:
59                 cluster_sums[k] = cluster_sums[k] + np.asarray(
    mu_sub, dtype=np.float32).mean(axis=0)
60                 cluster_counts[k] += 1
61
62
```

```

63 for iteration in range(0, config.algorithm.MAX_ITER):
64     start = time.time()
65
66     cluster_sums = manager.dict({k: np.zeros((1, word_count))
67                                 for k in range(0, K)})
68     cluster_counts = manager.dict({k: 0 for k in range(0, K)})
69
70     # Init processes
71     processes = []
72     for i, chunk_list in enumerate(chunks_split):
73         p = multiprocessing.Process(target=process_chunks,
74                                   kwargs={
75                                       'chunks': chunk_list,
76                                       'word_count': word_count,
77                                       'K': K,
78                                       'mu': mu,
79                                       'cluster_sums': cluster_sums,
80                                       'cluster_counts': cluster_counts,
81                                       'lock': lock
82                                   })
83         processes.append(p)
84
85     # Start processes
86     for p in processes:
87         p.start()
88
89     #print('Started %d processes' % (len(processes)))
90
91     # Wait for processes to finish
92     for p in processes:
93         p.join()
94
95     # Save old means
96     mu_old = np.array(mu, copy=True)
97
98     # Update means
99     for k in range(0, K):
100         count = cluster_counts[k]
101         if count == 0: continue
102         mu[k] = cluster_sums[k] / cluster_counts[k]
103
104     # Check convergence criteria
105     mu_norm = np.linalg.norm(mu - mu_old)
106
107     print('Iteration %d took: %.4fs' % (iteration + 1, time.
108                                         time() - start))
109
110     if mu_norm < config.algorithm.EPSILON:
111         print('Converged after %d iterations' % (iteration+1))
112         break
113
114     # Determine cluster tags
115     cluster_tag_counts = {k: {tag: 0 for tag in range(0, K)} for

```

```
    k in range(0, K)}
114 for chunk in chunks:
115
116     # Convert to sparse matrix
117     X, tags = helpers.chunk_to_sparse_mat(chunk, word_count)
118
119     if X is None:     continue
120
121     # Get closest cluster indices
122     max_idx = helpers.sparse_matrix_to_cluster_indices(X, mu)
123
124     # Count cluster tags
125     for i, k in enumerate(max_idx):
126         for tag_idx in tags[i]:
127             cluster_tag_counts[k][tag_idx] += 1
128
129     # Assign tags to clusters
130     tags_labelled = []
131     cluster2tag = {}
132     for k, tag_counts in cluster_tag_counts.items():
133         tag_counts_sorted = sorted(tag_counts.items(), key=lambda
134             x: x[1], reverse=True)
135         for tag, count in tag_counts_sorted:
136             if tag not in tags_labelled:
137                 cluster2tag[k] = tag
138                 tags_labelled.append(tag)
139                 break
140
141     # Save cluster tags dict
142     config.data.save_cluster_tags(cluster_tags=cluster2tag)
143
144     # Save means
145     with open(config.paths.MU, 'wb') as f:
146         np.save(f, mu)
```

A.3 Distributed decision trees ensemble algorithm

```
1 import os
2 import math
3 import collections
4 import numpy as np
5 import multiprocessing
6 from sklearn.externals import joblib
7
8 from sklearn.ensemble import RandomForestClassifier
9 from sklearn.tree import DecisionTreeClassifier
10
11 import helpers
12 import config
13
14 # Create models folder
15 if not os.path.exists(config.paths.MODELS_FOLDER):
16     os.makedirs(config.paths.MODELS_FOLDER)
17
```

```
18 # Read tags
19 tags, tag2idx, tag_count = helpers.read_tags()
20
21 # Read words
22 words, word2idx, word_count = helpers.read_words()
23
24 # Get chunks
25 chunk_reader = helpers.ChunkReader(post_filename=config.
    paths.TRAIN_DATA_IDX, chunk_size=config.data.
    CHUNK_SIZE_TREES) # TODO: Change
26 chunks = [chunk for chunk in chunk_reader]
27 chunk_count = len(chunks)
28
29 # Filesize total
30 bytes_total = sum(chunks[-1])
31
32 # Split chunks across processes
33 n = math.ceil(chunk_count / config.algorithm.PROCESS_COUNT)
34 chunks_split = []
35 for i in range(0, len(chunks), n):
36     chunks_split.append(chunks[i:i+n])
37
38 # Initialize shared variable manager
39 manager = multiprocessing.Manager()
40 lock = multiprocessing.Lock()
41
42 # Define function to run in parallel
43 def process_chunks(chunks, word_count, tag_count, clf_folder
    , classifier_filenames, bytes_processed, bytes_total,
    lock):
44     for chunk in chunks:
45
46         # Convert to sparse matrix
47         X, target_indices = helpers.chunk_to_sparse_mat(chunk,
            word_count)
48
49         if X is None: continue
50
51         # Create target vector from target indices
52         Y = np.zeros((len(target_indices), tag_count))
53         for i, indices in enumerate(target_indices):
54             Y[i,indices] = 1
55
56         # Train decision tree
57         clf = DecisionTreeClassifier(
58             splitter='best',
59             max_features='auto',
60             max_depth=None,
61         )
62
63         # Fit data
64         clf.fit(X.toarray(), Y)
65
66         # Save trained classifier
```

```

67     classifier_filename = os.path.join(clf_folder, 'clf-%s-%s.pkl' % chunk)
68     joblib.dump(clf, classifier_filename)
69
70     # Add classifier name to file
71     with lock:
72         classifier_filenames.append(classifier_filename)
73         bytes_processed.value += chunk[1]
74         print('Processed: %d/%d' % (bytes_processed.value, bytes_total))
75
76
77 classifier_filenames = manager.list([])
78 bytes_processed = manager.Value('i', 0)
79
80 # Init processes
81 processes = []
82 for i, chunk_list in enumerate(chunks_split):
83     p = multiprocessing.Process(target=process_chunks, kwargs
84     ={
85         'chunks': chunk_list,
86         'word_count': word_count,
87         'tag_count': tag_count,
88         'clf_folder': config.paths.MODELS_FOLDER,
89         'classifier_filenames': classifier_filenames,
90         'bytes_processed': bytes_processed,
91         'bytes_total': bytes_total,
92         'lock': lock
93     })
94     processes.append(p)
95
96 # Start processes
97 for p in processes:
98     p.start()
99
100 # Wait for processes to finish
101 for p in processes:
102     p.join()
103
104 # Save classifier filenames to file
105 config.data.save_classifier_filenames(classifier_filenames)

```

A.4 Helper functions

```

1 import os
2 import re
3 import math
4 import numpy as np
5
6 from xml.etree import ElementTree as ET
7 from scipy.sparse import csr_matrix
8 from scipy.sparse.linalg import norm as sparse_norm
9 from sklearn.metrics.pairwise import cosine_similarity
10 #from sklearn.preprocessing import normalize

```



```
11
12 import Stemmer
13
14 from nltk.stem import WordNetLemmatizer
15 lemmatizer = WordNetLemmatizer()
16
17 import config
18
19 stemmer = Stemmer.Stemmer('english')
20 word_to_stem = {}
21 def stem_word(word):
22     if not word in word_to_stem:
23         word_to_stem[word] = stemmer.stemWord(word)
24     return word_to_stem[word]
25
26 word_to_lemma = {}
27 def lemmatize_word(word):
28     if not word in word_to_lemma:
29         word_to_lemma[word] = lemmatizer.lemmatize(word)
30     return word_to_lemma[word]
31
32
33 def chunk_to_sparse_mat(chunk, word_count):
34     with open(config.paths.TRAIN_DATA_IDX, 'r') as f:
35         indptr = [0]
36         indices = []
37         data = []
38         has_data = False
39         tags = []
40         for i, (input_indices, target_indices) in enumerate(
            chunk_to_indices(chunk, f)):
41             for idx in input_indices:
42                 indices.append(idx)
43                 data.append(1)
44             indptr.append(len(indices))
45             tags.append(list(target_indices))
46             has_data = True
47
48         if has_data:
49             X = csr_matrix((data, indices, indptr), dtype=np.
                float32, shape=(len(indptr) - 1, word_count))
50
51             return X, tags
52         else:
53             return None, tags
54
55 def sparse_matrix_to_cluster_indices(X, mu):
56     # Compute cosine similarities
57     cos_sims = cosine_similarity(X, mu, dense_output=True)
58     max_idx = cos_sims.argmax(axis=1)
59
60     return max_idx
61
62 def sparse_matrix_to_sorted_cluster_indices(X, mu):
```

```

63     # Compute cosine similarities
64     cos_sims = cosine_similarity(X, mu, dense_output=True)
65     sorted_idx = cos_sims.argsort(axis=1)[:,:-1]
66
67     return sorted_idx
68
69
70 def chunk_to_indices(chunk, f):
71     # Seek to chunk start bytes
72     f.seek(chunk[0])
73
74     # Read end of chunk until end of line
75     chunk_decoded = f.read(chunk[1])
76
77     # Split in lines (Removing the last newline)
78     lines = chunk_decoded.rstrip('\n').split('\n')
79
80     for line in lines:
81         lineSplitted = line.split(',')
82         if len(lineSplitted) == 2:
83             input_indices = map(int, filter(lambda x: len(x) > 0,
84             lineSplitted[0].split(' ')))
85             target_indices = map(int, filter(lambda x: len(x) > 0,
86             lineSplitted[1].split(' ')))
87             yield input_indices, target_indices
88
89
90 def get_file_size(filename):
91     st = os.stat(filename)
92     return st.st_size
93
94
95 def hash_word(word, hashing_dim):
96     return sum(ord(a) for a in word) % hashing_dim
97
98
99 def hash_sentence(sentence, hashing_dim):
100     vec = np.zeros(hashing_dim).astype('uint32')
101     for word in sentence.split(' '):
102         vec[hash_word(word, hashing_dim)] += 1
103     return vec
104
105
106 def encode_tags(tags, tags_count):
107     target = np.zeros(tags_count)
108     for tag in tags:
109         idx = tag2idx.get(tag, -1)
110         if idx > -1:
111             target[idx] = 1
112     return target.astype('uint8')
113
114
115 def read_tags():
116     with open(config.paths.TAGS, 'r') as f:
117         tags = set([tag.rstrip('\n') for tag in f])
118         tags = list(sorted(tags))

```

```

115     tag_count = len(tags)
116     tag2idx = {}
117     for i, tag in enumerate(tags):
118         tag2idx[tag] = i
119
120     return tags, tag2idx, tag_count
121
122 def read_words():
123     with open(config.paths.WORDS, 'r') as f:
124         words = set([word.rstrip('\n') for word in f])
125         words = list(sorted(words))
126
127     word_count = len(words)
128     word2idx = {}
129     for i, word in enumerate(words):
130         word2idx[word] = i
131
132     return words, word2idx, word_count
133
134
135 class ChunkReader:
136     def __init__(self, post_filename, chunk_size=1024*1024):
137         self.post_filename = post_filename
138         self.chunk_size = chunk_size
139
140     def __iter__(self):
141         with open(self.post_filename, 'rb') as f:
142             while True:
143                 start = f.tell()
144                 f.seek(self.chunk_size, 1)
145                 s = f.readline()
146                 yield start, f.tell() - start
147                 if not s: break
148
149     def process_chunk(self, chunk):
150         with open(self.post_filename, 'rb') as f:
151
152             # Seek to chunk start bytes
153             f.seek(chunk[0])
154
155             # Read end of chunk until end of line end decode it
156             chunk_decoded = f.read(chunk[1]).decode('utf-8')
157
158             ## Split in lines (Removing the last newline)
159             lines = chunk_decoded.rstrip('\n').split('\n')
160
161             for line in lines:
162                 # Split in title, body and tags
163                 lines_splitted = line.split(config.text.delimiter)
164                 if len(lines_splitted) == 3:
165                     yield line.split(config.text.delimiter)
166
167
168 # Precompile regular expressions

```

```

169 reg_links = re.compile(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@
    .&+]|[*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+')
170 reg_digits = re.compile(r'\b\d+\b')
171 reg_spaces = re.compile(r'\s{2,}')
172
173 reg_symbols = re.compile(r'^A-Za-z0-9(),!?\`\'')
174 reg_symb_1 = re.compile(r',')
175 reg_symb_2 = re.compile(r'!')
176 reg_symb_3 = re.compile(r'\(')
177 reg_symb_4 = re.compile(r'\)')
178 reg_symb_5 = re.compile(r'\'')
179 reg_symb_6 = re.compile(r'\`')
180
181 reg_suf_1 = re.compile(r'\`s')
182 reg_suf_2 = re.compile(r'\`ve')
183 reg_suf_3 = re.compile(r'\`t')
184 reg_suf_4 = re.compile(r'\`re')
185 reg_suf_5 = re.compile(r'\`d')
186 reg_suf_6 = re.compile(r'\`ll')
187
188 def clean_string(text):
189     # Replace links with link identifier
190     text = reg_links.sub('<link>', text)
191
192     # Remove certain symbols
193     text = reg_symbols.sub(' ', text)
194
195     # Remove suffix from words
196     text = reg_suf_1.sub(' ', text)
197     text = reg_suf_2.sub(' ', text)
198     text = reg_suf_3.sub(' ', text)
199     text = reg_suf_4.sub(' ', text)
200     text = reg_suf_5.sub(' ', text)
201     text = reg_suf_6.sub(' ', text)
202
203     # Remove '"' from string
204     text = reg_symb_6.sub(' ', text)
205
206     # Replace breaks with spaces
207     text = text.replace('<br />', ' ')
208     text = text.replace('\r\n', ' ')
209     text = text.replace('\r', ' ')
210     text = text.replace('\n', ' ')
211
212     # Pad symbols with spaces on both sides
213     text = reg_symb_1.sub(' , ', text)
214     text = reg_symb_2.sub(' ! ', text)
215     text = reg_symb_3.sub(' ( ', text)
216     text = reg_symb_4.sub(' ) ', text)
217     text = reg_symb_5.sub(' \' ', text)
218
219     # Replace digits with 'DIGIT'
220     text = reg_digits.sub('<DIGIT>', text)
221

```

```

222 # Remove double whitespaces
223 text = re_spaces.sub(' ', text)
224 text = text.strip()
225
226 # Convert to lowercase
227 text = text.lower()
228
229 # Stem each word
230 text = ' '.join(stem_word(word) for word in text.split(' '))
231
232 # Lemmatize each word
233 text = ' '.join(lemmatize_word(word) for word in text.
234                 split(' '))
235
236 return text
237
238 def get_tags():
239     xml_parser = ET.iterparse(config.paths.TAGS_DUMP)
240     for i, (_, element) in enumerate(xml_parser):
241         if 'TagName' in element.attrib:
242             yield {
243                 'name': element.attrib['TagName'],
244                 'count': int(element.attrib['Count'])
245             }
246         element.clear()
247
248 def get_top_N_tags(N, include_counts=False):
249     tags = [tag for tag in get_tags()]
250     tags = sorted(tags, key=lambda tag: tag['count'], reverse=
251                  True)
252     tags = tags[0:N]
253     if include_counts:
254         return tags
255     else:
256         return [tag['name'] for tag in tags]
257
258 def get_posts(max_posts=math.inf):
259     tag_regex = re.compile(r'(<[^\>]*>)'
260     xml_parser = ET.iterparse(config.paths.POST_DUMP)
261     for i, (_, element) in enumerate(xml_parser):
262         if 'Tags' in element.attrib:
263             title = element.attrib.get('Title', '') # Not all have
264                 title
265             body = element.attrib['Body']
266             tags = [tag[1:-1] for tag in tag_regex.findall(element
267                 .attrib['Tags'])]
268
269             yield {
270                 'title': title,

```

```
271     }
272
273     if i > max_posts: break
274     element.clear()
275
276
277 def get_posts_filtered(tags, **kwargs):
278     tags = set(tags)
279     for post in get_posts(**kwargs):
280         if next(filter(tags.__contains__, post['tags']), None)
281             is not None:
282                 yield post
283
284 if __name__ == '__main__':
285
286     chunk_reader = ChunkReader(post_filename=config.paths.POST
287                                , chunk_size=1024)
288     for chunk in chunk_reader:
289         print(chunk)
```

A.5 Config file

```
1 import os
2 import pickle
3 import multiprocessing
4
5 FILEPATH = os.path.dirname(os.path.abspath(__file__))
6 class paths:
7     DATA_FOLDER = os.path.join(FILEPATH, 'data')
8
9     # Posts
10     POST = os.path.join(DATA_FOLDER, 'posts.csv')
11     POST_DUMP = '/Volumes/Seagate EXP/datasets/stackoverflow-
12               data-dump/stackoverflow/stackoverflow.com-Posts'
13
14     # Tags
15     TAGS = os.path.join(DATA_FOLDER, 'tags.csv')
16     TAGS_DUMP = '/Volumes/Seagate EXP/datasets/stackoverflow-
17               data-dump/stackoverflow/stackoverflow.com-Tags'
18
19     # Words
20     WORDS = os.path.join(DATA_FOLDER, 'words.csv')
21
22     # Meta data
23     META = os.path.join(DATA_FOLDER, 'meta.pkl')
24
25     # Input/target indices
26     TRAIN_DATA_IDX = os.path.join(DATA_FOLDER, 'train-data-
27               indices.csv')
28     TEST_DATA_IDX = os.path.join(DATA_FOLDER, 'test-data-
29               indices.csv')
30
31     # Mean numpy array
```

```

28 MU = os.path.join(DATA_FOLDER, 'means.dat')
29
30 # Cluster tags dict
31 CLUSTER_TAGS = os.path.join(DATA_FOLDER, 'cluster-tags.pkl
    ')
32
33 # Evaluations
34 PRECISION_AT_K = os.path.join(DATA_FOLDER, 'precision.csv'
    )
35
36 # Classifiers folder
37 MODELS_FOLDER = os.path.join(FILEPATH, 'models')
38
39 # Classifier filename
40 CLASSIFIERS = os.path.join(DATA_FOLDER, 'classifiers.csv')
41
42
43 class data:
44     TEST_FRACTION = 0.33
45
46     CHUNK_SIZE = 1 * 1024 ** 2 # 1MB
47     CHUNK_SIZE_TREES = 2 * 1024 ** 2 # 2MB
48
49     @classmethod
50     def save_cluster_tags(cls, cluster_tags):
51         with open(paths.CLUSTER_TAGS, 'wb') as f:
52             pickle.dump(cluster_tags, f)
53
54     @classmethod
55     def load_cluster_tags(cls):
56         with open(paths.CLUSTER_TAGS, 'rb') as f:
57             return pickle.load(f)
58
59     @classmethod
60     def save_classifier_filenames(cls, classifier_filenames):
61         with open(paths.CLASSIFIERS, 'w') as f:
62             for filename in classifier_filenames:
63                 f.write('%s\n' % (filename))
64
65
66     @classmethod
67     def load_classifier_filenames(cls):
68         with open(paths.CLASSIFIERS, 'r') as f:
69             filenames = [filename.rstrip('\n') for filename in f]
70             return filenames
71
72
73 class algorithm:
74     # Convergence criteria
75     MAX_ITER = 1000
76     EPSILON = 1e-10
77
78     # Number of processes to use in parallel
79     # TODO: Maybe use 2 * cpu_count (Hyperthreading)

```

```
80 PROCESS_COUNT = int(os.environ.get('PROCESS_COUNT',
multiprocessing.cpu_count()))
81 #PROCESS_COUNT = int(os.environ.get('PROCESS_COUNT',
multiprocessing.cpu_count() * 2))
82
83
84
85
86 class text:
87     delimiter = '#MY_CUSTOM_COMMA#'
88
89     @classmethod
90     def get_text_count(cls):
91         meta_data = cls.load_meta_data()
92         return meta_data['text_count']
93
94     @classmethod
95     def save_meta_data(cls, text_count):
96         meta_data = {
97             'text_count': text_count
98         }
99         with open(paths.META, 'wb') as f:
100             pickle.dump(meta_data, f)
101
102     @classmethod
103     def load_meta_data(cls):
104         with open(paths.META, 'rb') as f:
105             return pickle.load(f)
```

A.6 Preprocess file

```
1 import os
2 import math
3 import collections
4 from nltk.corpus import stopwords
5
6 import helpers
7 import config
8
9
10 if __name__ == '__main__':
11
12     if 'MAX_POSTS' in os.environ:
13         MAX_POSTS = int(os.environ['MAX_POSTS'])
14     else:
15         MAX_POSTS = math.inf
16
17     # Create data folder
18     if not os.path.exists(config.paths.DATA_FOLDER):
19         os.makedirs(config.paths.DATA_FOLDER)
20
21     # Get tags
22     tags = helpers.get_top_N_tags(N=20)
23
```



```
24 # Save top tags to file
25 with open(config.paths.TAGS, 'w') as f:
26     for tag in tags:
27         f.write('%s\n' % (tag))
28
29 # Create word counter
30 word_counter = collections.Counter()
31
32 # Save posts to file
33 config.paths.POST
34 text_count = 0
35 word_count = 0
36 with open(config.paths.POST, 'w') as f:
37     for post in helpers.get_posts_filtered(tags, max_posts=
38         MAX_POSTS):
39         title = helpers.clean_string(post['title'])
40         body = helpers.clean_string(post['body'])
41         tags = ' '.join(post['tags'])
42
43         for text in [title, body]:
44             for word in text.split():
45                 word_counter[word] += 1
46                 word_count += 1
47
48         line = config.text.delimiter.join([title, body, tags
49             ])
50
51         f.write('%s\n' % (line))
52         text_count += 1
53
54 # Save meta data
55 config.text.save_meta_data(text_count=text_count)
56
57 # Create dictionary of words to use in Bag of words
58 # Only take words occurring atleast 0.1% times and not
59 # occurring
60 # in more than 50% of the texts
61 #min_count = 10
62 min_count = text_count // 1000.0
63 #min_count = 2 * text_count // 100.0
64 max_count = text_count // 2.0
65
66 # Get english stop words
67 stop_words = stopwords.words('english')
68
69 with open(config.paths.WORDS, 'w') as f:
70     for word, count in word_counter.items():
71         if count < min_count: continue
72         if count > max_count: continue
73         if word in stop_words: continue
74         f.write('%s\n' % (word))
```

A.7 Transform file

```
1 import helpers
2 import config
3 from sklearn.model_selection import train_test_split
4
5
6 # Read tags
7 tags, tag2idx, tag_count = helpers.read_tags()
8
9 # Read words
10 words, word2idx, word_count = helpers.read_words()
11
12 # Get number of texts in data
13 text_count = config.text.get_text_count()
14
15 # Read chunks
16 chunk_reader = helpers.ChunkReader(post_filename=config.
    paths.POST, chunk_size=config.data.CHUNK_SIZE) # TODO:
    Change
17 all_chunks = [chunk for chunk in chunk_reader]
18
19 # Split chunks in training and test
20 chunks_train, chunks_test = train_test_split(all_chunks,
    test_size=config.data.TEST_FRACTION)
21
22 for chunks, target_filename in [
23     (chunks_train, config.paths.TRAIN_DATA_IDX),
24     (chunks_test, config.paths.TEST_DATA_IDX),
25 ]:
26
27     with open(config.paths.POST, 'rb') as f, open(
        target_filename, 'w') as f_indices:
28         for chunk in chunks:
29
30             # Seek to chunk start bytes
31             f.seek(chunk[0])
32
33             # Read end of chunk until end of line end decode it
34             chunk_decoded = f.read(chunk[1]).decode('utf-8')
35
36             ## Split in lines (Removing the last newline)
37             lines = chunk_decoded.rstrip('\n').split('\n')
38
39             for line in lines:
40                 # Split in title, body and tags
41                 lines_splitted = line.split(config.text.delimiter)
42                 if len(lines_splitted) == 3:
43                     title, body, tags = line.split(config.text.
                        delimiter)
44                     text = '%s %s' % (title, body)
45                     input_vec = []
46                     for word in text.split():
47                         idx = word2idx.get(word, None)
48                         if idx is not None:
49                             input_vec.append(idx)
```

```
50
51         target_vec = []
52         for tag in tags.split():
53             idx = tag2idx.get(tag, None)
54             if idx is not None:
55                 target_vec.append(idx)
56
57         input_str = ' '.join(map(str, input_vec))
58         target_str = ' '.join(map(str, target_vec))
59
60         f_indices.write('%s,%s\n' % (input_str, target_str
    ))
```

A.8 Evaluate file

```
1 import csv
2 import numpy as np
3
4 import helpers
5 import config
6
7 # Read tags
8 tags, tag2idx, tag_count = helpers.read_tags()
9
10 # Read words
11 words, word2idx, word_count = helpers.read_words()
12
13 # Load means
14 with open(config.paths.MU, 'rb') as f:
15     mu = np.load(f)
16
17 # Get chunks
18 chunk_reader = helpers.ChunkReader(post_filename=config.
19     paths.TEST_DATA_IDX, chunk_size=config.data.CHUNK_SIZE)
20 chunks = [chunk for chunk in chunk_reader]
21
22 # Load cluster tags dict
23 cluster2tag = config.data.load_cluster_tags()
24
25 with open(config.paths.TEST_DATA_IDX, 'r') as f:
26
27     # Count number of true retrieved tags in 'top k'
28     true_counts_at_k = {k: 0 for k in range(0, tag_count)}
29     total_tag_counts = 0
30     for chunk in chunks:
31
32         # Convert to sparse matrix
33         X, y_tags = helpers.chunk_to_sparse_mat(chunk,
34             word_count)
35
36         # Get closest cluster indices
37         sorted_idx = helpers.
38             sparse_matrix_to_sorted_cluster_indices(X, mu)
```

```

37     # Count true retrieved tags
38     for i, closest_indices in enumerate(sorted_idx):
39         true_tags = [cluster2tag[idx] for idx in y_tags[i]]
40         total_tag_counts += len(true_tags)
41         for k in range(0, tag_count):
42             tag_predictions = [cluster2tag[cluster] for cluster
43 in closest_indices[0:k+1]]
44             for tag in true_tags:
45                 if tag in tag_predictions:
46                     true_counts_at_k[k] += 1
47
48     # Compute precision at k (P@K)
49     precision = {k: true_counts_at_k[k] / total_tag_counts for
50 k in range(0, tag_count)}
51     for k, val in precision.items():
52         print('P@d:\t%.4f' % (k+1, val))
53
54     # Save precision at k
55     with open(config.paths.PRECISION_AT_K, 'w') as f:
56         writer = csv.writer(f)
57         writer.writerow([
58             'P@d' % (k+1) for k in range(0, tag_count)
59         ])
60         writer.writerow([
61             precision[k] for k in range(0, tag_count)
62         ])

```

A.9 Evaluate trees file

```

1  import csv
2  import numpy as np
3
4  from sklearn.externals import joblib
5
6  import helpers
7  import config
8
9  # Read tags
10 tags, tag2idx, tag_count = helpers.read_tags()
11
12 # Read words
13 words, word2idx, word_count = helpers.read_words()
14
15 # Get chunks
16 chunk_reader = helpers.ChunkReader(post_filename=config.
17     paths.TEST_DATA_IDX, chunk_size=config.data.
18     CHUNK_SIZE_TREES)
19 chunks = [chunk for chunk in chunk_reader]
20
21 # Load classifier filenames
22 classifier_filenames = config.data.load_classifier_filenames
23     ()
24
25 # Load classifiers

```

```

23 classifiers = [joblib.load(filename) for filename in
24                 classifier_filenames]
25
26 with open(config.paths.TEST_DATA_IDX, 'r') as f:
27     # Count number of true retrieved tags in 'top k'
28     true_counts_at_k = {k: 0 for k in range(0, tag_count)}
29     total_tag_counts = 0
30     for chunk in chunks:
31
32         # Convert to sparse matrix
33         X, y_tags = helpers.chunk_to_sparse_mat(chunk,
34                                                  word_count)
35
36         # Predict tag probabilities
37         clf_class_probs = []
38         for clf in classifiers:
39             probs = clf.predict_proba(X)
40
41             # Extract class probabilities
42             class_probs = np.asarray([1.0 - prob[:,0] for prob in
43                                     probs]).T
44             clf_class_probs.append(class_probs)
45
46         # Compute mean class probabilities across classifiers
47         clf_class_probs = np.asarray(clf_class_probs)
48         clf_class_probs = clf_class_probs.mean(axis=0)
49
50         # Sort by highest probability
51         sorted_class_indices = clf_class_probs.argsort(axis=1)
52        [:,::-1]
53
54         # Count true retrieved tags
55         for i, closest_indices in enumerate(sorted_class_indices):
56             true_tags = y_tags[i]
57             total_tag_counts += len(true_tags)
58             for k in range(0, tag_count):
59                 for tag in true_tags:
60                     if tag in closest_indices[0:k+1]:
61                         true_counts_at_k[k] += 1
62
63         # Compute precision at k (P@K)
64         precision = {k: true_counts_at_k[k] / total_tag_counts for
65                     k in range(0, tag_count)}
66         for k, val in precision.items():
67             print('P@d:\t%.4f' % (k+1, val))
68
69         # Save precision at k
70         with open(config.paths.PRECISION_AT_K, 'w') as f:
71             writer = csv.writer(f)
72             writer.writerow([
73                 'P@d' % (k+1) for k in range(0, tag_count)

```

```
71 |     ])  
72 |     writer.writerow([  
73 |         precision[k] for k in range(0, tag_count)  
74 |     ])
```

References

- [1] L. Nocedal and S. J. Wright, *Numerical Optimization*, ser. Springer Series in Operations Research and Financial Engineering. Springer, 2006, ISBN: 9780387303031.