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* → *have*)
4. Remove line breaks
5. Replace digits with *<digit>*
6. Remove double whitespaces
7. Reduce words to their word stem (e.g. *lines* → *line*)
8. Lemmatize words (e.g. *better* → *good*)

Finally the unique words used as word dictionary were filtered by removing words occuring in more than 50% of the questions and words occuring 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.

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

```
    start = f.tell()
    f.seek(chunk_size, 1)
    s = f.readline()
    yield start, f.tell() - start
    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

```
# Seek to chunk start bytes
f.seek(from_bytes)

# Read end of chunk until end of line
chunk = f.read(size)

# Split in lines (Removing the last newline)
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):

```
indptr, indices, data, tags = [0], [], [], []
for input_indices in chunk_to_indices(chunk):
  for idx in input_indices:
    indices.append(idx)
    data.append(1)
  indptr.append(len(indices))

X = csr_matrix(
  (data, indices, indptr),
  shape=(len(indptr) - 1, word_count)
)
```

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

procedure KMeansClustering(X, K)
  # Initialize cluster centers
  for $k = 0$ to $K - 1$ **do**
    $\mu_k \leftarrow$ random point in X
  # Run iterations
  while $iter < max\_iter$ **do**
    # Update cluster means
    $\mu_{\text{old}} = \mu$
    for $k = 0$ to $K - 1$ **do**
      $C_k \leftarrow \{\text{Points in } X \text{ closest to } \mu_k\}$
      $\mu_k \leftarrow \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$
    # Check convergence criteria
    $norm \leftarrow \|\mu - \mu_{\text{old}}\|$
    if $norm < \epsilon$ **then**
      break

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

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

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

procedure KMeansClusteringDistributed(X, K)
  # Initialize cluster centers
  for $k = 0$ to $K - 1$ **do**
    $\mu_k \leftarrow$ random point in X
  # Run iterations
  while *iter* $<$ *max_iter* **do**
    # Initialize shared cluster sums and cluster point counts.
    for $k = 0$ to $K - 1$ **do**
      $C\mathrm{sum}_k \leftarrow \mathbf{0}$
      $C\mathrm{count}_k \leftarrow 0$
    # Process each chunk in a distributed manner
    for all Chunks X_{chunk} in X **do**
      for $k = 0$ to $K - 1$ **do**
        $C_k \leftarrow \{$Points in X_{chunk} closest to $\mu_k\}$
        $C\mathrm{sum}_k \leftarrow C\mathrm{sum}_k + \sum_{x_i \in C_k} x_i$
        $C\mathrm{count}_k \leftarrow C\mathrm{count}_k + |C_k|$
    # Gather results and update cluster means
    $\mu_{\mathrm{old}} = \mu$
    for $k = 0$ to $K - 1$ **do**
      $\mu_k \leftarrow \frac{C\mathrm{sum}_k}{C\mathrm{count}_k}$
    # Check convergence criteria
    norm $\leftarrow \|\mu - \mu_{\mathrm{old}}\|$
    if *norm* $< \epsilon$ **then**
      break

4.3.3 Implementation

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

```
cluster_sums   = {k: np.zeros((1, word_count)) for k in
    range(0, K)}
cluster_counts = {k: 0 for k in range(0, K)}

for chunk in chunks:

  # Load chunk lines to sparse matrix
  X = chunk_to_sparse_mat(chunk)

  # Get closest cluster indices
  max_idx = sparse_matrix_to_cluster_indices(X, mu)

  # Assign points to clusters
  mu_subs = collections.defaultdict(list)
  for i, k in enumerate(max_idx):
    mu_subs[k].append(X[i].toarray())

  # Compute sub-means
  for k in range(0, K):
```

```
    mu_sub = mu_subs[k]
    if len(mu_sub) == 0:    continue
    cluster_sums[k] += np.asarray(mu_sub).mean(axis=0)
    cluster_counts[k] += 1

# Save old means
mu_old = np.array(mu, copy=True)

# Update means
for k in range(0, K):
  count = cluster_counts[k]
  if count == 0:  continue
  mu[k] = cluster_sums[k] / cluster_counts[k]

# Check convergence criteria
mu_norm = np.linalg.norm(mu - mu_old)

if mu_norm < epsilon:
  print('Converged after %d iterations' % (iteration+1))
  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

procedure DecisionTreeEnsembleDistributed(X, Y, K)
    # Process each chunk in a distributed manner
    for all Chunks $(X_{\text{chunk}}, Y_{\text{chunk}})$ in (X, Y) **do**
        $T \leftarrow$ Decision tree trained on $(X_{\text{chunk}}, Y_{\text{chunk}})$
        # 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):

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.externals import joblib

for chunk in chunks:

  # Convert to sparse matrix
  X, Y = chunk_to_sparse_mat(chunk)

  # Train decision tree
  clf = DecisionTreeClassifier(
      splitter='best',
      max_features='auto',
      max_depth=None,
  )

  # Fit data
  clf.fit(X, Y)

  # Save trained classifier
  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:

```
import multiprocessing

# Initialize shared variable manager
manager = multiprocessing.Manager()
lock = multiprocessing.Lock()

for chunks in list_of_chunks:
  p = multiprocessing.Process(
    target=process_chunks,
    kwargs={
      'chunks': chunks,
      ...
      'lock': lock
    }
  )
  processes.append(p)

# Start processes
for p in processes:
    p.start()
```

```

# Wait for processes to finish
for p in processes:
    p.join()

# Use results
# ...
```

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 multiprocess lock works

```
with lock:
  shared_counter += 1
```

5 Results

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

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

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: 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	$\approx$ 24hours	$\approx$ 1min
Decision tree ensemble	$\approx$ 10min	$\approx$ 2hours

Table 2: Approximate run-times for the models run one 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

```
import re
import Stemmer
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()

# Precompile regular expressions
reg_links  = re.compile(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+')
re_digits  = re.compile(r'\b\d+\b')
re_spaces  = re.compile(r'\s{2,}')

reg_symbols = re.compile(r'[^A-Za-z0-9(),!?\'\`]')
reg_symb_1 = re.compile(r',')
reg_symb_2 = re.compile(r'!')
reg_symb_3 = re.compile(r'\(')
reg_symb_4 = re.compile(r'\)')
reg_symb_5 = re.compile(r'\?')
reg_symb_6 = re.compile(r'\'')

reg_suf_1 = re.compile(r'\'s')
reg_suf_2 = re.compile(r'\'ve')
reg_suf_3 = re.compile(r'n\'t')
reg_suf_4 = re.compile(r'\'re')
reg_suf_5 = re.compile(r'\'d')
reg_suf_6 = re.compile(r'\'ll')

stemmer = Stemmer.Stemmer('english')
word_to_stem = {}
def stem_word(word):
    if not word in word_to_stem:
        word_to_stem[word] = stemmer.stemWord(word)
    return word_to_stem[word]

word_to_lemma = {}
def lemmatize_word(word):
    if not word in word_to_lemma:
        word_to_lemma[word] = lemmatizer.lemmatize(word)
    return word_to_lemma[word]

def clean_string(text):
  # Replace links with link identifier
  text = reg_links.sub('<link>', text)

  # Remove certain symbols
  text = reg_symbols.sub(' ', text)

  # Remove suffix from words
  text = reg_suf_1.sub(' ', text)
  text = reg_suf_2.sub(' ', text)
```

```
    text = reg_suf_3.sub(' ', text)
    text = reg_suf_4.sub(' ', text)
    text = reg_suf_5.sub(' ', text)
    text = reg_suf_6.sub(' ', text)

    # Remove "'" from string
    text = reg_symb_6.sub('', text)

    # Replace breaks with spaces
    text = text.replace('<br />', ' ')
    text = text.replace('\r\n', ' ')
    text = text.replace('\r', ' ')
    text = text.replace('\n', ' ')

    # Pad symbols with spaces on both sides
    text = reg_symb_1.sub(' , ', text)
    text = reg_symb_2.sub(' ! ', text)
    text = reg_symb_3.sub(' ( ', text)
    text = reg_symb_4.sub(' ) ', text)
    text = reg_symb_5.sub(' ? ', text)

    # Replace digits with 'DIGIT'
    text = re_digits.sub('<DIGIT>', text)

    # Remove double whitespaces
    text = re_spaces.sub(' ', text)
    text = text.strip()

    # Convert to lowercase
    text = text.lower()

    # Stem each word
    text = ' '.join(stem_word(word) for word in text.split(' '))

    # Lemmatize each word
    text = ' '.join(lemmatize_word(word) for word in text.split(' '))
```

A.2 Distributed K-means

```
import math
import collections
import numpy as np
import multiprocessing
import time

import helpers
import config

# Read tags
tags, tag2idx, tag_count = helpers.read_tags()

# Read words
```

```
words, word2idx, word_count = helpers.read_words()

# Clusters
K = tag_count

# Initialize cluster centers
mu = np.random.rand(K, word_count)

# Get chunks
chunk_reader = helpers.ChunkReader(post_filename=config.
    paths.TRAIN_DATA_IDX, chunk_size=config.data.CHUNK_SIZE)
    # TODO: Change
chunks = [chunk for chunk in chunk_reader]
chunk_count = len(chunks)

# Split chunks across processes
n = math.ceil(chunk_count / config.algorithm.PROCESS_COUNT)
chunks_split = []
for i in range(0, len(chunks), n):
  chunks_split.append(chunks[i:i+n])

# Initialize shared variable manager
manager = multiprocessing.Manager()
lock = multiprocessing.Lock()

# Define function to run in parallel
def process_chunks(chunks, word_count, K, mu, cluster_sums,
    cluster_counts, lock):
  for chunk in chunks:

    # Convert to sparse matrix
    X, _ = helpers.chunk_to_sparse_mat(chunk, word_count)

    if X is None:   continue

    # Get closest cluster indices
    max_idx = helpers.sparse_matrix_to_cluster_indices(X, mu
    )

    mu_subs = collections.defaultdict(list)
    for i, k in enumerate(max_idx):
      mu_subs[k].append(X[i].toarray())

    # Compute sub-means
    for k in range(0, K):
      mu_sub = mu_subs[k]
      if len(mu_sub) == 0:    continue

      with lock:
        cluster_sums[k] = cluster_sums[k] + np.asarray(
    mu_sub, dtype=np.float32).mean(axis=0)
        cluster_counts[k] += 1


```

```
for iteration in range(0, config.algorithm.MAX_ITER):
  start = time.time()

  cluster_sums = manager.dict({k: np.zeros((1, word_count))
    for k in range(0, K)})
  cluster_counts = manager.dict({k: 0 for k in range(0, K)})

  # Init processes
  processes = []
  for i, chunk_list in enumerate(chunks_split):
    p = multiprocessing.Process(target=process_chunks,
    kwargs={
      'chunks': chunk_list,
      'word_count': word_count,
      'K': K,
      'mu': mu,
      'cluster_sums': cluster_sums,
      'cluster_counts': cluster_counts,
      'lock': lock
    })
    processes.append(p)

  # Start processes
  for p in processes:
    p.start()

  #print('Started %d processes' % (len(processes)))

  # Wait for processes to finish
  for p in processes:
    p.join()

  # Save old means
  mu_old = np.array(mu, copy=True)

  # Update means
  for k in range(0, K):
    count = cluster_counts[k]
    if count == 0:  continue
    mu[k] = cluster_sums[k] / cluster_counts[k]

  # Check convergence criteria
  mu_norm = np.linalg.norm(mu - mu_old)

  print('Iteration %d took: %.4fs' % (iteration + 1, time.
    time() - start))

  if mu_norm < config.algorithm.EPSILON:
    print('Converged after %d iterations' % (iteration+1))
    break


# Determine cluster tags
cluster_tag_counts = {k: {tag: 0 for tag in range(0, K)} for
```

```
    k in range(0, K)}
for chunk in chunks:

  # Convert to sparse matrix
  X, tags = helpers.chunk_to_sparse_mat(chunk, word_count)

  if X is None:   continue

  # Get closest cluster indices
  max_idx = helpers.sparse_matrix_to_cluster_indices(X, mu)

  # Count cluster tags
  for i, k in enumerate(max_idx):
    for tag_idx in tags[i]:
      cluster_tag_counts[k][tag_idx] += 1

# Assign tags to clusters
tags_labelled = []
cluster2tag = {}
for k, tag_counts in cluster_tag_counts.items():
  tag_counts_sorted = sorted(tag_counts.items(), key=lambda
    x: x[1], reverse=True)
  for tag, count in tag_counts_sorted:
    if tag not in tags_labelled:
      cluster2tag[k] = tag
      tags_labelled.append(tag)
      break

# Save cluster tags dict
config.data.save_cluster_tags(cluster_tags=cluster2tag)

# Save means
with open(config.paths.MU, 'wb') as f:
  np.save(f, mu)
```

A.3 Distributed decision trees ensemble algorithm

```
import os
import math
import collections
import numpy as np
import multiprocessing
from sklearn.externals import joblib

from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

import helpers
import config

# Create models folder
if not os.path.exists(config.paths.MODELS_FOLDER):
    os.makedirs(config.paths.MODELS_FOLDER)

```

```
# Read tags
tags, tag2idx, tag_count = helpers.read_tags()

# Read words
words, word2idx, word_count = helpers.read_words()

# Get chunks
chunk_reader = helpers.ChunkReader(post_filename=config.paths.TRAIN_DATA_IDX, chunk_size=config.data.CHUNK_SIZE_TREES) # TODO: Change
chunks = [chunk for chunk in chunk_reader]
chunk_count = len(chunks)

# Filesize total
bytes_total = sum(chunks[-1])

# Split chunks across processes
n = math.ceil(chunk_count / config.algorithm.PROCESS_COUNT)
chunks_split = []
for i in range(0, len(chunks), n):
  chunks_split.append(chunks[i:i+n])

# Initialize shared variable manager
manager = multiprocessing.Manager()
lock = multiprocessing.Lock()

# Define function to run in parallel
def process_chunks(chunks, word_count, tag_count, clf_folder, classifier_filenames, bytes_processed, bytes_total, lock):
  for chunk in chunks:

    # Convert to sparse matrix
    X, target_indices = helpers.chunk_to_sparse_mat(chunk, word_count)

    if X is None:   continue

    # Create target vector from target indices
    Y = np.zeros((len(target_indices), tag_count))
    for i, indices in enumerate(target_indices):
      Y[i,indices] = 1

    # Train decision tree
    clf = DecisionTreeClassifier(
      splitter='best',
      max_features='auto',
      max_depth=None,
    )

    # Fit data
    clf.fit(X.toarray(), Y)

    # Save trained classifier
```

```
    classifier_filename = os.path.join(clf_folder, 'clf-%s-%s.pkl' % chunk)
    joblib.dump(clf, classifier_filename)

    # Add classifier name to file
    with lock:
      classifier_filenames.append(classifier_filename)
      bytes_processed.value += chunk[1]
      print('Processed: %d/%d' % (bytes_processed.value, bytes_total))


classifier_filenames = manager.list([])
bytes_processed = manager.Value('i', 0)

# Init processes
processes = []
for i, chunk_list in enumerate(chunks_split):
  p = multiprocessing.Process(target=process_chunks, kwargs={
    'chunks': chunk_list,
    'word_count': word_count,
    'tag_count': tag_count,
    'clf_folder': config.paths.MODELS_FOLDER,
    'classifier_filenames': classifier_filenames,
    'bytes_processed': bytes_processed,
    'bytes_total': bytes_total,
    'lock': lock
  })
  processes.append(p)

# Start processes
for p in processes:
  p.start()

# Wait for processes to finish
for p in processes:
  p.join()

# Save classifier filenames to file
config.data.save_classifier_filenames(classifier_filenames)
```

A.4 Helper functions

```
import os
import re
import math
import numpy as np

from xml.etree import ElementTree as ET
from scipy.sparse import csr_matrix
from scipy.sparse.linalg import norm as sparse_norm
from sklearn.metrics.pairwise import cosine_similarity
#from sklearn.preprocessing import normalize
```

```

import Stemmer

from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()

import config

stemmer = Stemmer.Stemmer('english')
word_to_stem = {}
def stem_word(word):
  if not word in word_to_stem:
    word_to_stem[word] = stemmer.stemWord(word)
  return word_to_stem[word]

word_to_lemma = {}
def lemmatize_word(word):
  if not word in word_to_lemma:
    word_to_lemma[word] = lemmatizer.lemmatize(word)
  return word_to_lemma[word]


def chunk_to_sparse_mat(chunk, word_count):
  with open(config.paths.TRAIN_DATA_IDX, 'r') as f:
    indptr = [0]
    indices = []
    data = []
    has_data = False
    tags = []
    for i, (input_indices, target_indices) in enumerate(
    chunk_to_indices(chunk, f)):
      for idx in input_indices:
        indices.append(idx)
        data.append(1)
      indptr.append(len(indices))
      tags.append(list(target_indices))
      has_data = True

    if has_data:
      X = csr_matrix((data, indices, indptr), dtype=np.
    float32, shape=(len(indptr) - 1, word_count))

      return X, tags
    else:
      return None, tags

def sparse_matrix_to_cluster_indices(X, mu):
  # Compute cosine similarities
  cos_sims = cosine_similarity(X, mu, dense_output=True)
  max_idx = cos_sims.argmax(axis=1)

  return max_idx

def sparse_matrix_to_sorted_cluster_indices(X, mu):
```

```
  # Compute cosine similarities
  cos_sims = cosine_similarity(X, mu, dense_output=True)
  sorted_idx = cos_sims.argsort(axis=1)[:,::-1]

  return sorted_idx


def chunk_to_indices(chunk, f):
  # Seek to chunk start bytes
  f.seek(chunk[0])

  # Read end of chunk until end of line
  chunk_decoded = f.read(chunk[1])

  # Split in lines (Removing the last newline)
  lines = chunk_decoded.rstrip('\n').split('\n')

  for line in lines:
    line_splitted = line.split(',')
    if len(line_splitted) == 2:
      input_indices  = map(int, filter(lambda x: len(x) > 0, line_splitted[0].split(' ')))
      target_indices = map(int, filter(lambda x: len(x) > 0, line_splitted[1].split(' ')))
      yield input_indices, target_indices


def get_file_size(filename):
  st = os.stat(filename)
  return st.st_size

def hash_word(word, hashing_dim):
  return sum(ord(a) for a in word) % hashing_dim

def hash_sentence(sentence, hashing_dim):
  vec = np.zeros(hashing_dim).astype('uint32')
  for word in sentence.split(' '):
    vec[hash_word(word, hashing_dim)] += 1
  return vec

def encode_tags(tags, tags_count):
  target = np.zeros(tags_count)
  for tag in tags:
    idx = tag2idx.get(tag, -1)
    if idx > -1:
      target[idx] = 1
  return target.astype('uint8')


def read_tags():
  with open(config.paths.TAGS, 'r') as f:
    tags = set([tag.rstrip('\n') for tag in f])
  tags = list(sorted(tags))
```

```
  tag_count = len(tags)
  tag2idx = {}
  for i, tag in enumerate(tags):
    tag2idx[tag] = i

  return tags, tag2idx, tag_count

def read_words():
  with open(config.paths.WORDS, 'r') as f:
    words = set([word.rstrip('\n') for word in f])
  words = list(sorted(words))

  word_count = len(words)
  word2idx = {}
  for i, word in enumerate(words):
    word2idx[word] = i

  return words, word2idx, word_count


class ChunkReader:
  def __init__(self, post_filename, chunk_size=1024*1024):
    self.post_filename = post_filename
    self.chunk_size = chunk_size

  def __iter__(self):
    with open(self.post_filename, 'rb') as f:
      while True:
        start = f.tell()
        f.seek(self.chunk_size, 1)
        s = f.readline()
        yield start, f.tell() - start
        if not s:   break

  def process_chunk(self, chunk):
    with open(self.post_filename, 'rb') as f:

      # Seek to chunk start bytes
      f.seek(chunk[0])

      # Read end of chunk until end of line end decode it
      chunk_decoded = f.read(chunk[1]).decode('utf-8')

      ## Split in lines (Removing the last newline)
      lines = chunk_decoded.rstrip('\n').split('\n')

      for line in lines:
        # Split in title, body and tags
        lines_splitted = line.split(config.text.delimitter)
        if len(lines_splitted) == 3:
          yield line.split(config.text.delimitter)


# Precompile regular expressions
```

```
reg_links  = re.compile(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+')
re_digits  = re.compile(r'\b\d+\b')
re_spaces  = re.compile(r'\s{2,}')

reg_symbols = re.compile(r'[^A-Za-z0-9(),!?\'\`]')
reg_symb_1 = re.compile(r',')
reg_symb_2 = re.compile(r'!')
reg_symb_3 = re.compile(r'\(')
reg_symb_4 = re.compile(r'\)')
reg_symb_5 = re.compile(r'\?')
reg_symb_6 = re.compile(r'\'')

reg_suf_1 = re.compile(r'\'s')
reg_suf_2 = re.compile(r'\'ve')
reg_suf_3 = re.compile(r'n\'t')
reg_suf_4 = re.compile(r'\'re')
reg_suf_5 = re.compile(r'\'d')
reg_suf_6 = re.compile(r'\'ll')

def clean_string(text):
  # Replace links with link identifier
  text = reg_links.sub('<link>', text)

  # Remove certain symbols
  text = reg_symbols.sub(' ', text)

  # Remove suffix from words
  text = reg_suf_1.sub(' ', text)
  text = reg_suf_2.sub(' ', text)
  text = reg_suf_3.sub(' ', text)
  text = reg_suf_4.sub(' ', text)
  text = reg_suf_5.sub(' ', text)
  text = reg_suf_6.sub(' ', text)

  # Remove "'" from string
  text = reg_symb_6.sub('', text)

  # Replace breaks with spaces
  text = text.replace('<br />', ' ')
  text = text.replace('\r\n', ' ')
  text = text.replace('\r', ' ')
  text = text.replace('\n', ' ')

  # Pad symbols with spaces on both sides
  text = reg_symb_1.sub(' , ', text)
  text = reg_symb_2.sub(' ! ', text)
  text = reg_symb_3.sub(' ( ', text)
  text = reg_symb_4.sub(' ) ', text)
  text = reg_symb_5.sub(' ? ', text)

  # Replace digits with 'DIGIT'
  text = re_digits.sub('<DIGIT>', text)
```

```
  # Remove double whitespaces
  text = re_spaces.sub(' ', text)
  text = text.strip()

  # Convert to lowercase
  text = text.lower()

  # Stem each word
  text = ' '.join(stem_word(word) for word in text.split(' '))

  # Lemmatize each word
  text = ' '.join(lemmatize_word(word) for word in text.split(' '))

  return text


def get_tags():
  xml_parser = ET.iterparse(config.paths.TAGS_DUMP)
  for i, (_, element) in enumerate(xml_parser):
    if 'TagName' in element.attrib:
      yield {
          'name': element.attrib['TagName'],
          'count': int(element.attrib['Count'])
      }
    element.clear()

def get_top_N_tags(N, include_counts=False):
  tags = [tag for tag in get_tags()]
  tags = sorted(tags, key=lambda tag: tag['count'], reverse=True)
  tags = tags[0:N]
  if include_counts:
    return tags
  else:
    return [tag['name'] for tag in tags]


def get_posts(max_posts=math.inf):
  tag_regex = re.compile(r'(<[^<>]*>)')
  xml_parser = ET.iterparse(config.paths.POST_DUMP)
  for i, (_, element) in enumerate(xml_parser):
    if 'Tags' in element.attrib:
      title = element.attrib.get('Title', '') # Not all have title
      body = element.attrib['Body']
      tags = [tag[1:-1] for tag in tag_regex.findall(element.attrib['Tags'])]

      yield {
        'title': title,
        'body': body,
        'tags': tags
```

```
        }

        if i > max_posts:    break
      element.clear()


def get_posts_filtered(tags, **kwargs):
  tags = set(tags)
  for post in get_posts(**kwargs):
    if next(filter(tags.__contains__, post['tags']), None)
     is not None:
        yield post


if __name__ == '__main__':

  chunk_reader = ChunkReader(post_filename=config.paths.POST
    , chunk_size=1024)
  for chunk in chunk_reader:
    print(chunk)
```

A.5 Config file

```
import os
import pickle
import multiprocessing

FILEPATH = os.path.dirname(os.path.abspath(__file__))
class paths:
  DATA_FOLDER = os.path.join(FILEPATH, 'data')

  # Posts
  POST = os.path.join(DATA_FOLDER, 'posts.csv')
  POST_DUMP = '/Volumes/Seagate EXP/datasets/stackoverflow-
    data-dump/stackoverflow/stackoverflow.com-Posts'

  # Tags
  TAGS = os.path.join(DATA_FOLDER, 'tags.csv')
  TAGS_DUMP = '/Volumes/Seagate EXP/datasets/stackoverflow-
    data-dump/stackoverflow/stackoverflow.com-Tags'

  # Words
  WORDS = os.path.join(DATA_FOLDER, 'words.csv')

  # Meta data
  META = os.path.join(DATA_FOLDER, 'meta.pkl')

  # Input/target indices
  TRAIN_DATA_IDX = os.path.join(DATA_FOLDER, 'train-data-
    indices.csv')
  TEST_DATA_IDX  = os.path.join(DATA_FOLDER, 'test-data-
    indices.csv')

  # Mean numpy array
```

```
  MU = os.path.join(DATA_FOLDER, 'means.dat')

  # Cluster tags dict
  CLUSTER_TAGS = os.path.join(DATA_FOLDER, 'cluster-tags.pkl')

  # Evaluations
  PRECISION_AT_K = os.path.join(DATA_FOLDER, 'precision.csv')

  # Classifiers folder
  MODELS_FOLDER = os.path.join(FILEPATH, 'models')

  # Classifier filename
  CLASSIFIERS = os.path.join(DATA_FOLDER, 'classifiers.csv')


class data:
  TEST_FRACTION = 0.33

  CHUNK_SIZE = 1 * 1024 ** 2 # 1MB
  CHUNK_SIZE_TREES = 2 * 1024 ** 2 # 2MB

  @classmethod
  def save_cluster_tags(cls, cluster_tags):
    with open(paths.CLUSTER_TAGS, 'wb') as f:
      pickle.dump(cluster_tags, f)

  @classmethod
  def load_cluster_tags(cls):
    with open(paths.CLUSTER_TAGS, 'rb') as f:
      return pickle.load(f)

  @classmethod
  def save_classifier_filenames(cls, classifier_filenames):
    with open(paths.CLASSIFIERS, 'w') as f:
      for filename in classifier_filenames:
        f.write('%s\n' % (filename))


  @classmethod
  def load_classifier_filenames(cls):
    with open(paths.CLASSIFIERS, 'r') as f:
      filenames = [filename.rstrip('\n') for filename in f]
    return filenames


class algorithm:
  # Convergence criteria
  MAX_ITER = 1000
  EPSILON = 1e-10

  # Number of processes to use in parallel
  # TODO: Maybe use 2 * cpu_count (Hyperthreading)
```

```
  PROCESS_COUNT = int(os.environ.get('PROCESS_COUNT',
    multiprocessing.cpu_count()))
  #PROCESS_COUNT = int(os.environ.get('PROCESS_COUNT',
    multiprocessing.cpu_count() * 2))




class text:
  delimitter = '#MY_CUSTOM_COMMA#'

  @classmethod
  def get_text_count(cls):
    meta_data = cls.load_meta_data()
    return meta_data['text_count']

  @classmethod
  def save_meta_data(cls, text_count):
    meta_data = {
      'text_count': text_count
    }
    with open(paths.META, 'wb') as f:
      pickle.dump(meta_data, f)

  @classmethod
  def load_meta_data(cls):
    with open(paths.META, 'rb') as f:
      return pickle.load(f)
```

A.6 Preprocess file

```
import os
import math
import collections
from nltk.corpus import stopwords

import helpers
import config


if __name__ == '__main__':

  if 'MAX_POSTS' in os.environ:
    MAX_POSTS = int(os.environ['MAX_POSTS'])
  else:
    MAX_POSTS = math.inf

  # Create data folder
  if not os.path.exists(config.paths.DATA_FOLDER):
    os.makedirs(config.paths.DATA_FOLDER)

  # Get tags
  tags = helpers.get_top_N_tags(N=20)

```

```
# Save top tags to file
with open(config.paths.TAGS, 'w') as f:
  for tag in tags:
    f.write('%s\n' % (tag))

# Create word counter
word_counter = collections.Counter()

# Save posts to file
config.paths.POST
text_count = 0
word_count = 0
with open(config.paths.POST, 'w') as f:
  for post in helpers.get_posts_filtered(tags, max_posts=MAX_POSTS):
    title = helpers.clean_string(post['title'])
    body = helpers.clean_string(post['body'])
    tags = ' '.join(post['tags'])

    for text in [title, body]:
      for word in text.split():
        word_counter[word] += 1
        word_count += 1

    line = config.text.delimitter.join([title, body, tags])

    f.write('%s\n' % (line))
    text_count += 1

# Save meta data
config.text.save_meta_data(text_count=text_count)

# Create dictionary of words to use in Bag of words
# Only take words occuring atleast 0.1% times and not occuring
# in more than 50% of the texts
#min_count = 10
min_count = text_count // 1000.0
#min_count = 2 * text_count // 100.0
max_count = text_count // 2.0

# Get english stop words
stop_words = stopwords.words('english')

with open(config.paths.WORDS, 'w') as f:
  for word, count in word_counter.items():
    if count < min_count:  continue
    if count > max_count:  continue
    if word in stop_words:  continue
    f.write('%s\n' % (word))
```

A.7 Transform file

```python
import helpers
import config
from sklearn.model_selection import train_test_split


# Read tags
tags, tag2idx, tag_count = helpers.read_tags()

# Read words
words, word2idx, word_count = helpers.read_words()

# Get number of texts in data
text_count = config.text.get_text_count()

# Read chunks
chunk_reader = helpers.ChunkReader(post_filename=config.paths.POST, chunk_size=config.data.CHUNK_SIZE) # TODO: Change
all_chunks = [chunk for chunk in chunk_reader]

# Split chunks in training and test
chunks_train, chunks_test = train_test_split(all_chunks, test_size=config.data.TEST_FRACTION)

for chunks, target_filename in [
  (chunks_train, config.paths.TRAIN_DATA_IDX),
  (chunks_test,  config.paths.TEST_DATA_IDX),
]:

  with open(config.paths.POST, 'rb') as f, open(target_filename, 'w') as f_indices:
    for chunk in chunks:

      # Seek to chunk start bytes
      f.seek(chunk[0])

      # Read end of chunk until end of line end decode it
      chunk_decoded = f.read(chunk[1]).decode('utf-8')

      ## Split in lines (Removing the last newline)
      lines = chunk_decoded.rstrip('\n').split('\n')

      for line in lines:
        # Split in title, body and tags
        lines_splitted = line.split(config.text.delimitter)
        if len(lines_splitted) == 3:
          title, body, tags = line.split(config.text.delimitter)
          text = '%s %s' % (title, body)
          input_vec = []
          for word in text.split():
            idx = word2idx.get(word, None)
            if idx is not None:
              input_vec.append(idx)
```

```
          target_vec = []
          for tag in tags.split():
            idx = tag2idx.get(tag, None)
            if idx is not None:
              target_vec.append(idx)

          input_str  = ' '.join(map(str, input_vec))
          target_str = ' '.join(map(str, target_vec))

          f_indices.write('%s,%s\n' % (input_str, target_str))
```

A.8 Evaluate file

```
import csv
import numpy as np

import helpers
import config

# Read tags
tags, tag2idx, tag_count = helpers.read_tags()

# Read words
words, word2idx, word_count = helpers.read_words()

# Load means
with open(config.paths.MU, 'rb') as f:
  mu = np.load(f)

# Get chunks
chunk_reader = helpers.ChunkReader(post_filename=config.paths.TEST_DATA_IDX, chunk_size=config.data.CHUNK_SIZE)
chunks = [chunk for chunk in chunk_reader]

# Load cluster tags dict
cluster2tag = config.data.load_cluster_tags()

with open(config.paths.TEST_DATA_IDX, 'r') as f:

  # Count number of true retrieved tags in 'top k'
  true_counts_at_k = {k: 0 for k in range(0, tag_count)}
  total_tag_counts = 0
  for chunk in chunks:

    # Convert to sparse matrix
    X, y_tags = helpers.chunk_to_sparse_mat(chunk, word_count)

    # Get closest cluster indices
    sorted_idx = helpers.sparse_matrix_to_sorted_cluster_indices(X, mu)

```

```
    # Count true retrieved tags
    for i, closest_indices in enumerate(sorted_idx):
      true_tags = [cluster2tag[idx] for idx in y_tags[i]]
      total_tag_counts += len(true_tags)
      for k in range(0, tag_count):
        tag_predictions = [cluster2tag[cluster] for cluster
    in closest_indices[0:k+1]]
        for tag in true_tags:
          if tag in tag_predictions:
            true_counts_at_k[k] += 1

  # Compute precision at k (P@K)
  precision = {k: true_counts_at_k[k] / total_tag_counts for
     k in range(0, tag_count)}
  for k, val in precision.items():
    print('P@%d:\t%.4f' % (k+1, val))

  # Save precision at k
  with open(config.paths.PRECISION_AT_K, 'w') as f:
    writer = csv.writer(f)
    writer.writerow([
      'P@%d' % (k+1) for k in range(0, tag_count)
    ])
    writer.writerow([
      precision[k] for k in range(0, tag_count)
    ])
```

A.9 Evaluate trees file

```
import csv
import numpy as np

from sklearn.externals import joblib

import helpers
import config

# Read tags
tags, tag2idx, tag_count = helpers.read_tags()

# Read words
words, word2idx, word_count = helpers.read_words()

# Get chunks
chunk_reader = helpers.ChunkReader(post_filename=config.
    paths.TEST_DATA_IDX, chunk_size=config.data.
    CHUNK_SIZE_TREES)
chunks = [chunk for chunk in chunk_reader]

# Load classifier filenames
classifier_filenames = config.data.load_classifier_filenames
    ()

# Load classifiers
```

```
classifiers = [joblib.load(filename) for filename in classifier_filenames]

with open(config.paths.TEST_DATA_IDX, 'r') as f:

  # Count number of true retrieved tags in 'top k'
  true_counts_at_k = {k: 0 for k in range(0, tag_count)}
  total_tag_counts = 0
  for chunk in chunks:

    # Convert to sparse matrix
    X, y_tags = helpers.chunk_to_sparse_mat(chunk, word_count)

    # Predict tag probabilities
    clf_class_probs = []
    for clf in classifiers:
      probs = clf.predict_proba(X)

      # Extract class probabilities
      class_probs = np.asarray([1.0 - prob[:,0] for prob in probs]).T
      clf_class_probs.append(class_probs)

    # Compute mean class probabilities across classifiers
    clf_class_probs = np.asarray(clf_class_probs)
    clf_class_probs = clf_class_probs.mean(axis=0)

    # Sort by highest probability
    sorted_class_indices = clf_class_probs.argsort(axis=1)[:,::-1]

    # Count true retrieved tags
    for i, closest_indices in enumerate(sorted_class_indices):
      true_tags = y_tags[i]
      total_tag_counts += len(true_tags)
      for k in range(0, tag_count):
        for tag in true_tags:
          if tag in closest_indices[0:k+1]:
            true_counts_at_k[k] += 1


  # Compute precision at k (P@K)
  precision = {k: true_counts_at_k[k] / total_tag_counts for k in range(0, tag_count)}
  for k, val in precision.items():
    print('P@%d:\t%.4f' % (k+1, val))

  # Save precision at k
  with open(config.paths.PRECISION_AT_K, 'w') as f:
    writer = csv.writer(f)
    writer.writerow([
      'P@%d' % (k+1) for k in range(0, tag_count)
```

```
    ])
    writer.writerow([
      precision[k] for k in range(0, tag_count)
    ])
```

References

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