Predicting Stackoverflow tags
02807 Final project

November 29, 2016

1 Introduction

In this project a model predicting tags associating a given text will be constructed. The problem will be handled as a classification problem, hence a classifier will be trained.

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

Initially only the top 20 frequent tags and posts containing these tags will be used for training and prediction.

Finally the model will be evaluated using all tags and posts.

During implementation and debugging a subset of the data will be used.

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 Methods

In this section the different methods and steps in the process will be explained.

3.1 Preprocessing

The preprocessing step regards the transforming of questions in an `XML` file to processed questions in a `.csv` file. This 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 really 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.

3.2 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

```python
# 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')
```

3.3 K-means clustering

3.3.1 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
        μ_k ← random point in X
    # Run iterations
    while iter < max_iter do
        # Update cluster means
        μ_old = μ
        for k = 0 to K − 1 do
            C_k ← {Points in X closest to μ_k}
            μ_k ← (1/|C_k|) Σ_{x_i ∈ C_k} x_i
        # Check convergence criteria
        norm ← ||μ − μ_old||
        if norm < ϵ then
            break
```

3.3.2 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**
      $C\text{sum}_k \leftarrow \mathbf{0}$
      $C\text{count}_k \leftarrow 0$
    # Process each chunk in a distributed manner
    for all Chunks X_{chunk} in X **do**
      for $k = 0$ to $K - 1$ **do**
        $C_k \leftarrow \{\text{Points in } X_{\text{chunk}} \text{ closest to } \mu_k\}$
        $C\text{sum}_k \leftarrow C\text{sum}_k + \sum_{x_i \in C_k} x_i$
        $C\text{count}_k \leftarrow C\text{count}_k + |C_k|$
    # Gather results and update cluster means
    $\mu_{\text{old}} = \mu$
    for $k = 0$ to $K - 1$ **do**
      $\mu_k \leftarrow \frac{C\text{sum}_k}{C\text{count}_k}$
    # Check convergence criteria
    $norm \leftarrow \|\mu - \mu_{\text{old}}\|$
    if $norm < \epsilon$ **then**
      break

3.3.3 Implementation

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

```python
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
```

3.4 Parallel processing

4 Results

5 Discussion

6 Conclusion

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
```

```python
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)
```

Include helper functions and config