10_Data_Scaling

May 6, 2024

Programming Techniques in Computational Linguistics II – FS24

1 Lecture 10: Data Scaling

- Midterm results are online
- Last exercies (worth 3 points!) released today
- 2023 final exam is available on OLAT

- Individual Course Evaluation (ILE), open until May 24th!
- Link to survey: ????

1.1 Topics

- Repetition: Generators
- Sequential Text Processing
- Sequential Parsing of XML
- Sequential Parsing of JSON
- Parallelism

1.2 Learning Objectives

After today's lecture you:

- know how to efficiently handle large text files without loading the entire content into memory.
- can utilize the `lxml` package to efficiently manage large xml files.
- can utilize the `ijson` package to efficiently manage large json files.

1.3 Motivation

Big Data: Data that are so large that traditional methods of processing do not work.

Examples in NLP:

- GPT-3 was trained on 300 billion words (570GB)
- more than 270,000 hours of videos are uploaded to Youtube every day
- 2.7 billion web pages in Common Crawl Archive from April 2024 (~386 TB)

2 Repetition: Generators

- Generators are itertators.

- `yield` is a keyword used to create a generator function.
- Generator functions are highly memory efcient, as the function is only executed when caller iterates over the object.

2.1 Generator Expressions (Repetition)

Generator expressions are an alternative to list comprehensions in two cases:

- When the iterator returns an infinite stream (e.g. fibonacci numbers)
- When the iterator handles a large amount of data.

Generator expressions are surrounded by parentheses (`()`) while list comprehensions are surrounded by square brackets (`[]`).

2.2 Fibonacci Generator (Repetition)

[ ]:
```
from typing import Iterable
def fibonacci() -> Iterable[int]:
    x, y = 0, 1
    while True:
        yield y
        x, y = y, x + y
fib_generator = fibonacci()
# fib_squarer = (x ** 2 for x in fib_generator)
```

[ ]:
```
next(fib_generator)
```

3 Sequential Text Processing

Task: Perform sentence segmentation for a very large file.

Naive solution:

[ ]:
```
# Get an English sentence splitter.
import nltk.data
punkt = nltk.data.load('tokenizers/punkt/english.pickle')

with open('large_corpus.txt') as f:
    text = f.read()
    sentences = punkt.tokenize(text)

with open('large_corpus.sents.txt', 'w') as f:
    for sentence in sentences:
        f.write(sentence.replace('\n', ' ') + '\n')
```

[ ]:
```
!head -n 20 large_corpus.txt
```

large_corpus.txt used in the example: Moby Dick by Herman Melville

- ~2 million tokens, ~9000 sentences, file size: 1.2 MB

- No problem for a modern computer

Using a 540 MB file:

Recorded memory usage with memory profiler

3.0.1 Problems

- `f.read()`: The complete file is in memory.
- As a rule, a Python string needs considerably more space than the corresponding file
- `PunktSentenceTokenizer.tokenize()` internally creates two additional copies of the full text.

→ Data representation can easily use up four times the original file size

3.1 Use generators

First use case: use method `span_tokenize()`

[ ]:
```
spans = punkt.span_tokenize(text)
```

[ ]:
```
next(spans)
```

[ ]:
```
spans = punkt.span_tokenize(text)
sentences = (text[start:end] for start, end in spans)
```

[ ]:
```
next(sentences)
```

→ avoids a copy (list), instead reads in chunks on demand

Second use case for generators: line-wise processing of the input file

[ ]:
```
def iter_sentences(stream):
    '''Iterate over sentences from a text stream.'''
    for line in stream:
        for start, end in punkt.span_tokenize(line):
            yield line[start:end]

with open('large_corpus.txt') as f:
    sentences = iter_sentences(f)

with open('large_corpus.sents.txt', 'w') as f:
    for sentence in sentences:
        f.write(sentence.replace('\n', ' ') + '\n')
        f.write('\n')
```

New problem: Input file is already closed when it is supposed to be iterated over.

Procedure:

1. Calling the function `iter_sentences` creates a generator object which uses the opened file. The instructions in `iter_sentences` have not been executed yet!

2. End of the **with** statement: the file is closed.
3. **for** loop over the generator object:
    - only now the file contents are iterated
    - the file object still exists but has been closed!

Solution: keep both files open in parallel.

[ ]:
```
def iter_sentences(stream):
    '''Iterate over sentences from a text stream.'''
    for line in stream:
        for start, end in punkt.span_tokenize(line):
            yield line[start:end]

with open('large_corpus.txt') as inp, open('large_corpus.sents.txt', 'w') as␣
 ↪outp:
    sentences = iter_sentences(inp)
    for sentence in sentences:
        outp.write(sentence.replace('\n', ' ') + '\n')
```

Goal achieved!

- Input text is read in line by line and then «forgotten» again
- Intermediate results are also «forgotten» after each write operation
- Execution jumps in the code from `iter_sentences` to the next yield statement, and then back to the main for loop

[ ]:
```
!head large_corpus.sents.txt
```

Further problem: Sentences with a line break are cut apart.

Solution: Iterate paragraph by paragraph

→ We replace the built-in functionality (line-wise iteration) by a custom generator function

[ ]:
```
def iter_sentences(stream):
    '''Iterate over sentences from a text stream.'''
    for par in iter_paragraphs(stream):
        for start, end in punkt.span_tokenize(par):
            yield par[start:end]
```

3.2 Accumulating lines into paragraphs

Assumption: Paragraphs are divided by blank lines

[ ]:
```
def iter_paragraphs(stream):
    '''Iterate over paragraphs separated by a blank line.'''
    current = ''
    for line in stream:
        if line.isspace(): # blank line -- paragraph ended if current:
            yield current
```

```
            current = ''
        else:
            current += line
    # Don't forget the last one.
    if current:
        yield current
```

[ ]:
```
with open('large_corpus.txt') as inp:
    paragraphs = iter_paragraphs(inp)
    print(next(paragraphs))
    print("***")
    print(next(paragraphs))
    print("***")
    print(next(paragraphs))
    print("***")
```

Do you see any potential problems with this approach?

→ what if there are no blank lines (paragraphs)?

3.3 Fixed-size chunking

Improvement: Arbitrary portioning (independent of blank lines)

From the python docs:

> To read a file's contents, call `f.read(size)` [...] When `size` is omitted or negative, the entire contents of the file will be read and returned; **it's your problem if the file is twice as large as your machine's memory**. Otherwise, at most `size` characters (in text mode) or `size` bytes (in binary mode) are read and returned. If the end of the file has been reached, `f.read()` will return an empty string.

[ ]:
```
def iter_chunks(stream, chunksize=1000):
    '''Iterate over chunks of fixed size.'''
    while True:
        chunk = stream.read(chunksize)
        if not chunk:
            # End of file reached.
            break
        yield chunk.replace('\n', ' ')
```

[ ]:
```
with open('large_corpus.txt') as stream:
    chunks = iter_chunks(stream, 50)
    for _ in range(5):
        print(next(chunks) + "\n***")
```

Text is cut at arbitrary lines – last sentence is probably incomplete

→ carry over to next chunk

[ ]:
```python
def iter_sentences(stream):
    '''Iterate over sentences from a text stream.'''
    remainder = ''
    for chunk in iter_chunks(stream):
        # Add remainder from the previous chunk.
        chunk = remainder + chunk
        *spans, last = punkt.span_tokenize(chunk)
        for start, end in spans:
            yield chunk[start:end]
        # Keep the last sentence -- it might be continued in the next chunk.
        remainder = chunk[last[0]:]
    # Remember to yield the very last remainder.
    if remainder:
        yield remainder
```

[ ]:
```python
chunk = "Call me Ishmael. Some years ago-never mind how lon"

*spans, last = punkt.span_tokenize(chunk)
print(f"spans: {spans}, last: {last}")
```

[ ]:
```python
# yield the spans in start
for start, end in spans:
    print(chunk[start:end])
```

[ ]:
```python
# carry over last span
remainder = chunk[last[0]:]
remainder
```

3.4 Overview of the implementation

Pipeline with two generators

- `iter_chunks` cuts the input into equally sized portions.
- `iter_sentences` segments the portions into sentences.
  - The last sentences may be incomplete and are thus carried over to the next portion

The elements are called in reverse:

- Main loop requests next sentences
- Sentence generator requests next chunk if needed
- Chunk generator reads in the next text portion from the open file

[ ]:
```python
with open('large_corpus.txt') as inp, open('large_corpus.sents.txt', 'w') as␣
↪outp:
    sentences = iter_sentences(inp)
    for sentence in sentences:
        outp.write(sentence + '\n')
```

[ ]:
```python
!head large_corpus.sents.txt
```

3.5 Performance

Compare to naive solution:

3.6 Summary

- Input and output files are open at the same time
- Input is read in chunk-wise, processed, and results are written to the output file right away → Less memory needed
- Generator functions are ideal for this as they logically structure the code

3.7 Klicker Quiz

https://app.klicker.uzh.ch/join/lfische

4 Sequential Parsing of XML

Data: PubMed abstracts (MEDLINE)

- Collection of abstracts and bibliographical information
- biomedical scientific publications
- currently more than 34 million entries
- freely available: dump of about 1200 XML files with 30000 abstracts each

Task: Extract title, year of publication and author's names for all entries

4.1 First Idea

[ ]:
```
def extract(fn: str) -> list[tuple]:
    '''Get author names, years, and titles from a Medline XML.'''
    meta = []
    tree = ET.parse(fn)
    for article in tree.iterfind('.//PubmedArticle'):
        title = article.findtext('.//ArticleTitle')
        year = article.findtext('.//DateRevised/Year')
        authors = ', '.join(
            name.text
            for name in article.iterfind('.//AuthorList/Author/LastName'))
        meta.append((authors, year, title))
    return meta
```

4.2 Replace List with Generator

[ ]:
```
def extract(fn: str) -> list[tuple]:
    '''Get author names, years, and titles from a Medline XML.'''
    tree = ET.parse(fn)
    for article in tree.iterfind('.//PubmedArticle'):
        title = article.findtext('.//ArticleTitle')
        year = article.findtext('.//DateRevised/Year')
```

```
        authors = ', '.join(
            name.text
            for name in article.iterfind('.//AuthorList/Author/LastName'))
        yield (authors, year, title)
```

4.3 Performance

Check out slide version to see the performance graphs!

4.3.1 Measurement

Processing file 1 of 1166 (file size : 185 MB):

t : 5.2s m_{max} : 1.45 G

4.3.2 Extrapolation

1166 * 5.2s = 1h 41 min

4.4 Parallelization

- With 8G, we can run a maximum of 5 parallel processes (e.g. by running the program with 5 different input files at the same time) → 20 minutes in the best case

4.5 Limiting Factors

- Python structures need a multiple of the memory compared to the file itself
- `ET.parse()` reads the full file into memory
- built-in library `xml.etree.ElementTree` is not the most efficient solution

4.6 Optimized library: `lxml`

- lxml offers a Python interface for the very efficient C libraries `libxml` and `libxslt`.
- Third party library: may have to be installed
- API has been modeled after `xml.etree.ElementTree`

Replace the import statement

`import xml.etree.ElementTree as ET`

with

`import lxml.etree as ET`

4.7 Use Sequential Parsing

Instead of parsing the entire file at the start, use `lxml.etree.iterparse()`.

[ ]:
```
from typing import Iterable
```

[ ]:
```
def extract(fn: str) -> Iterable[tuple]:
    '''Get author names, years, and titles from a Medline XML.'''
```

```
    for _, article in ET.iterparse(fn, tag='PubmedArticle'):
        title = article.findtext('.//ArticleTitle')
        year = article.findtext('.//DateRevised/Year')
        authors = ', '.join(
            name.text
            for name in article.iterfind('.//AuthorList/Author/LastName'))
        yield authors, year, title
```

4.8 Performance

	xml	lxml
t	5.2s	2.1s
m_{max}	1.45G	1.60G

4.8.1 Extrapolation

1166 * 2.1s = 40 min

With 5 processes in parallel: 8 min

4.9 Analysis

- Memory usage increases steadily, peak is slightly higher than with `xml` library
- It seems that the processed nodes are still kept in memory

4.10 Solution

- Explicitly «empty» the nodes in every loop iteration → method `Element.clear()`
- Delete the empty nodes, their memory requirement still adds up

[ ]:
```
def extract(fn: str) -> Iterable[tuple]:
    '''Get author names, years, and titles from a Medline XML.'''
    for _, article in ET.iterparse(fn, tag='PubmedArticle'):
        title = article.findtext('.//ArticleTitle')
        year = article.findtext('.//DateRevised/Year')
        authors = ', '.join(
            name.text
            for name in article.iterfind('.//AuthorList/Author/LastName'))
        # clear node and delete previous sister nodes
        article.clear()
        while article.getprevious() is not None:
            del article.getparent()[0]
        yield authors, year, title
```

4.11 Performance

	xml	lxml	lxml + clear()
t	5.2s	2.1s	1.9s
m_{max}	1.45G	1.60G	13.6M

4.11.1 Extrapolation

1166 * 1.9s = 37 min

Number of parallel processes is no longer limited by memory usage (next bottleneck: CPU cores, I/O, ...)

4.12 Experiment: Process multiple files

```
# data is a folder containing 20 xml files
for file in os.listdir("data"):
    if file.endswith(".xml"):
        for metadata in extract("data/"+file):
            print(metadata)
```

4.13 Performance

`lxml`: crashes!

Now that the `extract` function is optimized, let's write the metadata to an XML file.

`metadata.xml`:

```
<?xml version='1.0' encoding='utf-8'?>
<root><article>
  <authors>Tcherdakoff</authors>
  <year>2013</year>
  <title>[Beta-blockers and arterial hypertension in the pregnant woman].</title>
</article>
<article>
  <authors>Dubois, Petitcolas</authors>
  <year>2013</year>
  <title>[beta-blockers and high risk pregnancies. Viewpoint of the nephrologist and the obste
</article>
    ...
</root>
```

4.14 Writing to an XML file

Classical way:

```
root = ET.Element("root")
for file in files:
    for authors, year, title in extract(file):
        article = ET.SubElement(root, "article")
        ET.SubElement(article, "authors").text = authors
```

```
        ET.SubElement(article, "year").text = year
        ET.SubElement(article, "title").text = title
tree = ET.ElementTree(root)
tree.write("metadata.xml", pretty_print=True)
```

Memory efficient way:

```
with ET.xmlfile("metadata.xml", encoding='utf-8') as xf:
    with xf.element('root'):
        for file in files:
            for authors, year, title in extract(file):
                el = ET.Element('article')
                ET.SubElement(el, 'authors').text = authors
                ET.SubElement(el, 'year').text = year
                ET.SubElement(el, 'title').text = title
                xf.write(el, pretty_print=True)
                el.clear()
```

Check if file is valid with

```
xmllint --noout metadata.xml
```

4.15 Performance

Parsing 20 XML files with fully optimized `extract` function.

4.16 Conclusions

- Observe memory consumption of processes (no fancy profiler needed, task manager (windows) or activity monitor (mac) is sufficient to see memory spikes).
- Sequential processing includes parsing input files, writing to output files and any steps in betweeen!

5 Sequential parsing of JSON

`example.json`:

```
[
{
  "task": "EE",
  "source": "CASIE",
  "instruction": "{\"instruction\": \"You are an expert in event extraction. Please extract eve
  "output": "{\"ransom\": [{\"trigger\": \"the ransomware attack\", \"arguments\": {\"victim\"
},
{
  "task": "EE",
  "source": "CASIE",
  "instruction": "{\"instruction\": \"You are an expert in event extraction. Please extract eve
  "output": "{\"data breach\": [], \"ransom\": [], \"patch vulnerability\": [{\"trigger\": \"ha
},
```

```
...
]
```

What if file is too large to parse with `json.loads(file)`?

Classical way:

```python
def extract(json_file):
    '''Get sources from a JSON file.'''
    with open(json_file) as f:
        data = json.load(f)

    for item in data:
        source = item['source']
        yield source
```

Memory efficient way:

```python
import ijson

def extract(json_file):
    '''Get sources from a JSON file.'''
    for x in ijson.items(open(json_file), 'item.source'):
        yield x
```

See ijson documentation

5.1 Performance

6 Parallelism

If we use a for-loop to process a large number of files sequentially, this will often be too slow depending on how long our program takes to process an individual file.

Also: Running a program on one CPU core to process a large number of files sequentially is a waste of time if there are multiple CPUs / cores available.

We want to be able to process code blocks in parallel.

6.1 Call the python script muliple times at once

```bash
python optimized_extraction.py --input file_1.xml --output metadata_1.xml &
python optimized_extraction.py --input file_2.xml --output metadata_2.xml &
python optimized_extraction.py --input file_3.xml --output metadata_3.xml &
...
python optimized_extraction.py --input file_n.xml --output metadata_n.xml &
```

ToDo: Find optimal number of processes!

6.2 Use `multiprocessing` to run parallel processes within the script

The `Pool` class allows to submit tasks to a fixed number of processes managed by the pool.

```python
from multiprocessing import Pool

def extract(fn: str) -> Iterable[tuple]:
    ... # Process a single file in some way

if __name__ == '__main__':
    files = ['data/'+file for file in os.listdir("data") if file.endswith(".xml.gz")]
    with Pool(processes=5) as pool:  # Reuse the same 5 processes
        for metadata in pool.map(extract, files):
            # write metadata to file
            ...
```

6.3 Multiprocessing for PubMed abstracts

Process 1000 Files with a `Pool` of 5 processes.

	1 process	5 processes
t	26m	7min
m_{max}	32.5M	172.2M

In practice, 5 processes are not 5 times as fast, and need more than 5 times of the memory!

7 Take Home Messages

- Process large files sequentially if possible
- Use generators instead of lists
- Use sequential functionality of libraries
  - iterate line by line over text files
  - iterate chunk-wise over input (use read method with an integer as argument)
  - `lxml.etree.iterparse()` + `Element.clear()` for sequential XML parsing
  - `ijson` for parsing large json files