Introduction

This exercise will cover XML and JSON validation. For the next four weeks you will be using oXygen (https://www.oxygenxml.com/xml_editor/software_archive_editor.html). AN XML/JSON development IDE. You should have received a license key for it by mail. Before starting, make sure oXygen is installed and working on your computer.

At the end of this exercise sheet you will find some additional Spark exercises.

1. XML

1.1 Create your own XML

1. Copy the text of the introduction above (including the title until 'Spark exercises') and paste it into oXygen as plain text. Create a possible XML document, having the same context and including formatting (title, sections, style, links, etc.). Make sure your XML is well-formed and save it as doc1.xml.

2. Copy the same text into Microsoft Word or OpenOffice and save it XML (both programs allow to export as .xml if you use Save as...).

Questions

1. Compare the two XML. What differences do you notice?
2. Is your XML valid?
3. Is this data structured, unstructured or semi-structured?

Solution

There is no unique way to create an XML encoding the text above and its structure. This is one possible solution:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<document lang="en">
  <title font-size='15px' font-style='bold'>Introduction</title>
  <paragraph font-size='12px'>This exercise will cover XML and JSON validation. For the next four weeks you will be using oXygen, an XML/JSON development IDE. You should have received a license key for it by mail. Before starting, make sure oXygen is installed and working on your computer.</paragraph>
  <paragraph font-size='12px'>At the end of this exercise sheet you will find some additional Spark exercises.</paragraph>
</document>
```

1. The differences depend on which program you used to generate export the document. The file exported by Word or OpenOffice probably contains much more information, including the font family, page layout, paragraph layout, etc.
2. The question cannot be answered unless you defined a schema for your data format.
3. This data is semi-structured: there is some structure, but not all of the content is structured as in a flat database.

1.2 Well-formedness

Correct the following XML documents to be well-formed. Try first to "parse" it in mind, the use oXygen to check.

1.
<catalog>
<!-- Start book list -->
<Book id='bk101'>
  <author>&copyright; Gambardella, Matthew</author>
  <title>XML Developer's Guide</title>
  <genre>Computer</genre>
  <price>44.95€</price>
  <publish_date version='hard' version='soft'>2000-10-01</publish_date>
  <description lang=en>An in-depth look at creating applications with XML <for dummies>.</description>
  <xml_parse>true</xml_parse>
</book>
</catalog>

2.

<head><h:title>Book Review</title></head>
<body>
  <xdc:bookreview>
    <xdc:title>XML: A Primer</xdc:title>
    <_table _style='container'>
      <h:tr align="#center">
        <h:td>Author</h:span>St. Laurent & Tom Faron</h:td></h:tr>
      <h:tr align="#left">
        <h:td><xdc:author>Simon St. Laurent</xdc:author></h:td>
        <h:td><xdc:price>31.98</xdc:price></h:td>
        <h:td><xdc:#pages>352</xdc:#pages></h:td>
        <h:td><xdc:_date>1998/01</xdc:_date></h:td>
        <h:td><xdc:-comment>Love it</xdc:-comment></h:td>
      </h:tr>
    </_table>
  </xdc:bookreview>
</body>
</h:library>

Solution

Document 1 has the following problems:

1. the quotes in XML must always be simple quotes or double quotes, but not "Word-style" quotes ( ` or ");
2. the book start tag does not correspond to the Book end tag;
3. the catalog tag is not closed correctly;
4. the entity &cright; is not defined in XML. You have to define it explicitly;
5. you cannot have the < or > sign inside attributes. Use &lt; or &gt; instead (defined by XML). Also it is advised to use &gt; for the </ symbol;
6. attribute version in publish_date is duplicated, this is forbidden;
7. comments <!-- --> cannot include the characters --;
8. the lang attribute should be quoted;
9. XML names beginning with xml are reserved by the W3C. Their usage should be avoided (except if it is as specified as the W3C, e.g. xml:space, xml:lang, xmlns...).

Here is the corrected document:
Document 2 has the following problems:

1. `<h:title>` opening tag does not match the closing tag `</title>`;
2. body uses an empty tag when opening tag is required instead;
3. in `<xdc:bookreview>` the namespace `_xdc` is not defined;
4. the `<h:span>` element containing the author name should be closed before closing its parent;
5. the `&` in the author text field should be escaped;
6. `<xdc:#pages>` is not a valid tag name;
7. `<xdc:_comment>` is not a valid tag name.

Here is the corrected document:

```xml
<?xml version="1.0" encoding="utf-16"?>
    <head><h:title>Book Review</h:title></head>
    <body>
        <xdc:bookreview>
            <xdc:title>XML: A Primer</xdc:title>
            <table style='container'>
                <tr align="#center">
                    <td><h:span>St. Laurent &amp; Tom Faron</h:span></td>
                </tr>
                <tr align="#left">
                    <td>Simon St. Laurent</td>
                    <td>31.98</td>
                    <td>352</td>
                    <td>1998/01</td>
                    <td>Love it</td>
                </tr>
            </table>
        </xdc:bookreview>
    </body>
</h:library>
```
1.3 XML Names

Which of the following are valid XML Names?

1. `<_bar/>`
2. `<Xmlelement/>`
3. `<Foo/>`
4. `<foo123/>`
5. `<foo_123/>`
6. `<foo-123/>`
7. `<foo#123/>`
8. `<foo.123/>`
9. `<-123/>`
10. `<123foo/>`
11. `<doctype/>`
12. `<bigĐäta/>`

Solution

1, 3, 4, 5, 6, 8, 11 and 12 are valid names. Remember:

1. Element names are case-sensitive.
2. Element names must start with a letter or underscore.
3. Element names cannot start with the letters xml (or XML, or Xml, etc).
4. Element names can contain letters, digits, hyphens, underscores, and periods.
5. Element names cannot contain spaces.

1.4 Predefined entities

XML has only 5 predefined entities. Connect each escape code to the corresponding value.

1. `&lt;`   `>`
2. `&amp;`  `"`
3. `&gt;`   `'
4. `&quot;` &`
5. `&apos;` `<`

Solution

1. `&lt;`   `<`
2. `&amp;`  `&`
3. `&gt;`   `>`
4. `&quot;` `"`
5. `&apos;` `'`
1.5 HTML vs XHTML

Is the following correct HTML? Is it correct XML (XHTML)?

```html
<html>
  <head>
    <title>Untitled</title>
  </head>
  Dear jane <br/>
  <p>You are invited at the weekly meeting</p>
  <p>Yours sincerely, <br/></p>
  John
</html>
```

**Solution** This will be shown correctly in most browsers. However, it is not well-formed XML: the br and p tags are not closed. The following would be well-formed XML:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <head>
    <title>Untitled</title>
  </head>
  <body>
    Dear jane
    <p>You are invited at the weekly meeting</p>
    <p>Yours sincerely, </p>
    John
  </body>
</html>
```

But XHTML is more than just XML: it also has to have a certain structure (this is called to be “valid”). Among others, the tags have to live in the XHTML namespace:

```xml
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Untitled</title>
  </head>
  <body>
    <p>Dear jane <br/></p>
    <p>You are invited at the weekly meeting</p>
    <p>Yours sincerely, <br/></p>
    John
  </body>
</html>
```
1.6 XML Namespaces

1. Correct the mistakes in the following (not well-formed) XML document.

```xml
<?xml version="1.0"?>
<!DOCTYPE "eth" SYSTEM "eth.dtd">  
<eth xmlns="http://www.ethz.ch"
     xmlns:xmldb="http://www.foo.ethz.ch"
     date="11.11.2006"
     xmldb:date="12.11.2006">
  <date>13.11.2006</date>
  <president number="1">Empty</president>
  <Rektor>Name 2</Rektor>
</eth>
</Doc>
```

Solution

1. The document has the following mistakes: `eth` has a wrong closing tag, so has `Rektor`, a `/` is missing for the president tag, the `DOCTYPE` should be introduced with `!` and the root element should not be quoted.

```xml
<?xml version="1.0"?>
<!DOCTYPE eth SYSTEM "eth.dtd">
<eth xmlns="http://www.ethz.ch"
     xmlns:xmldb="http://www.foo.ethz.ch"
     date="11.11.2006"
     xmldb:date="12.11.2006">
  <date>13.11.2006</date>
  <president number="1">Empty</president>
  <Rektor>Name 2</Rektor>
</eth>
```

2. `eth` is in the namespace `http://www.ethz.ch`
   
   `xmlns:xmldb="http://www.dbis.ethz.ch"
     date="11.11.2006"`
produces a date that does not inherit the namespace from the root element, so it is in no namespace. Unlike elements, an attribute with no prefix is in no namespace, even if there is a default namespace.

   ```xml
   <date>13.11.2006</date>
   <president number="1">Empty</president>
   <Rektor>Name 2</Rektor>
   ```
   
   This attribute is in the `http://www.foo.ethz.ch (http://www.foo.ethz.ch)` namespace - it is allowed to have two attributes with the same local name if their namespaces are different.

   ```xml
   <date>13.11.2006</date>
   <president number="1">Empty</president>
   <Rektor>Name 2</Rektor>
   ```
   
   All children elements are in the namespace of the root, i.e. `http://www.ethz.ch (http://www.ethz.ch)`, but the number attribute is in no namespace.

2. JSON

2.1 JSON Values

JSON data is written as name/value pairs. List the 6 possible value types of JSON.

Solution

1. A number (integer or floating point)
2. A string (in double quotes)
3. A Boolean (true or false)
4. An array (in square brackets)
5. An object (in curly braces)
6. null
2.2 Well-formedness

Correct the following JSON document to be well-formed. Try first to "parse" it in mind, then use oXygen to check.

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  age: 25,
  "isRetired",
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100",
    "isverified": "true"
  },
  'phoneNumbers': [
  {
    "type": [['home']],
    "@number": "212 555-1234"
  },
  {
    "type": [['office']],
    "@number": "646 555-4567"
  },
  {
    "type": [['mobile']],[
    "@number": "123 456-7890"
  }
  ],
  "children": [],
  "settings": {},
  "spouse": Null
}
```

Solution

1. age name must be quoted
2. isRetired must have a value
3. address object must be followed by a comma
4. is verified and phoneNumbers should be double quoted
5. Null is not a valid value.

Using whitespaces and non-ascii characters for key names is allowed although not recommended. Mixing proper boolean values and strings used as boolean values (ie. "true") is also considered a bad practice.
3. Back to Spark

In this exercise you will implement a simple query engine over the Gutenberg dataset using Spark. The Gutenberg dataset (http://web.eecs.umich.edu/~lahiri/gutenberg_dataset.html) consists of 3036 free ebooks. The goal of this exercise is to develop a search engine to find the most relevant books given a text query.

3.1 Get the data

You can download the dataset from https://anconamstoragewest.blob.core.windows.net/public/gutenberg_books.tar.gz. Create a Spark cluster as explained in the exercise of last week. When ready, you can upload the dataset in its Blob Storage container. You can either 1) connect via SSH, wget the dataset, decompress it and upload it to HDFS, or 2) download the dataset on your computer, decompress it locally and upload the dataset folder to the Blob Storage using Azure Storage Explorer (http://storageexplorer.com/).

Create a new notebook on the Spark Jupyter server (https://<cluster-name>.azurehdinsight.net/jupyter).

3.2 Understand the model

We will use the tf-idf (https://en.wikipedia.org/wiki/Tf%E2%80%93idf) statistic to determine the relative importance of the words that each book contains. tf-idf is the product of two statistics, term frequency (tf) and inverse document frequency (idf).

Given a word t, a document d (in this case a book) and the collection of all documents D we can define \( tf(t, d) \) as the number of times t appears in d. This gives us some information about the content of a document but because some terms (eg. "the") are so common, term frequency will tend to incorrectly emphasize documents which happen to use the word "the" more frequently, without giving enough weight to the more meaningful terms.

The inverse document frequency \( idf(t, D) \) is a measure of how much information the word provides, that is, whether the term is common or rare across all documents. It can be computed as:

\[
idf(t, D) = \log\left( \frac{|D|}{\{ d \in D : t \in d \}} \right)
\]

where \( |D| \) is the total number of documents and the denominator represents how many documents contain the word t at least once. However, this would cause a division-by-zero exception if the user query a word that never appear in the dataset. A better formulation would be:

\[
idf(t, D) = \max(0, \log\left( \frac{|D|}{1 + |\{ d \in D : t \in d \}|} \right))
\]

Then, the \( tdidf(t, d, D) \) is calculated as follows:

\[
tdidf(t, d, D) = tf(t, d) \times idf(t, D)
\]

A high weight in tfidf is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents.

Finally we need a ranking function: one of the simplest ranking functions is computed by summing the tf-idf for each query term. You can then sort the results and and print some of those with the best score.

3.3 Implement tf-idf

Once you have upload the dataset folder into HDFS you can read the files from PySpark using the following line:

```python
In [ ]:

# sc is automatically defined as SparkContext
# docs will be an RDD in the format [(docPath, text), (...)]
docs = sc.wholeTextFiles('wasb:///path-to-gutenberg-folder/*.txt', minPartitions=100)
n_docs = docs.count()
print ('Num of document: %d' % n_docs)
print ('Num of partitions: %d' % docs.getNumPartitions())
```

Develop a Spark program that can be lunchy by calling a query() function providing a query as the one below:
In [ ]:
```
# query() takes a query string as input and outputs an RDD containing a sorted list of tuples.
# each tuple will be in the format (book_path, total_tdidf_score)
# we use .take(2) as action to limit the number of results we print in this notebook
results = query('derivation of the lorentz transformation').take(2)

# print tfidf score together with book filename, line by line
for r in results:
    print ('%.2f  %s' % (r[1], r[0].split('/')[1]))
```

Hints
- Start developing your program using a subset of the entire dataset. It might take some time to run your DAG on 3036 books.
- Think about which RDD can be cached (http://spark.apache.org/docs/latest/quick-start.html#caching). The first run might take some time but subsequent queries should be solved in a few seconds.

Question
1. List the words in the dataset with idf score of 0.
2. What are 2 best-matching books with the query *derivation of the lorentz transformation*?
3. What about the query *island and pirates*?

Solution
1. The words with idf of 0 are those words that appear in all documents (or all minus one given with the formula above), meaning that they carry no valuable information for the ranking and they will be effectively ignored:
   - all
   - and
   - from
   - not
   - a
   - for
   - the
   - it
   - that
   - in
   - ...

2. For the first query above, the main results should be related to *Albert Einstein*:
   - [414.34] - Albert_Einstein_Relativity_The_Special_and_General_Theory.txt
   - [86.22] - Albert_Einstein_Sidelights_on_Relativity.txt

3. For the second query:
   - [589.54] - Frank_Richard_Stockton_Kate_Bonnet.txt
   - [576.66] - Frank_Richard_Stockton_Buccaneers_and_Pirates_of_Our_Coasts.txt
from math import log
import re

docs = sc.wholeTextFiles('wasb:///inputdir/*.txt')
n_docs = docs.count()
print('Num of document: %d % n_docs)
print('Num of partitions: %d % docs.getNumPartitions())

def tokenize(s):
    # Note that these map and filter are normal Python functions (not Spark actions)
    tokens = map(lambda w: w.strip(), re.split('\W+', s.lower()))
    return filter(lambda t: len(t) > 0, tokens)

    # [(docId, text)] --> [(docId, tokens)] --> [[(docId, word), count]
    doc_words = docs.flatMapValues(tokenize).map(lambda x: (x[0], x[1]).reduceByKey(lambda a, b: a + b)
    # [(docId, token), count] --> [(token, docId)]
    inv_doc_words = doc_words.map(lambda pair: pair[0][::-1])

    # tf: [(docId, token), count] --> [(token, (docId, count))]
    tf = doc_words.map(lambda pair: (pair[0][1], pair[0][0], pair[1])).persist()
    # idf: [(token, docId)] --> [(token, idf(token))]
    idf = inv_doc_words.mapValues(lambda _ : 1).reduceByKey(lambda a, b: a + b) 
    .mapValues(lambda x: max(0, log(n_docs/(1.0 + x)))).persist()

    # tfidf [(token, (docId, tf), idf])
    tfidf = tf.join(idf).persist()

    # Filter and reduce tfidf RDD given a query
    def query(q):
        return tfidf.filter(lambda x: x[0] in q.lower().split())
            .values() 
            .map(lambda x: (x[0][0], x[0][1]*x[1])) 
            .reduceByKey(lambda a, b: a + b) 
            .sortBy(lambda x: -x[1])

    # Then use the following code to run a query
    results = query('derivation of the lorentz transformation').take(2)

    for r in results:
        print('%.2f - %s' % (r[1], r[0].split('/'[-1]))
Update: let’s make it faster

A blind, step-by-steps implementation of the tf-idf definition turns out to be very inefficient. We know discuss some possible improvements to the solutions above. As you know, every time data has to go across the cluster network, parallel algorithms drastically degrade in performance. As a general design principle, you should always try to minimize network traffic (eg. shuffling operations).

Here are some ideas for the tf-idf algorithm:

1 - Both tf and idf metrics can be computed in a single pass through the data. In fact, we can compute a tf pair RDD where, for each entry, the key is a word and the value is a list of documents where the word appears, together with the frequency for each document (ie. (word, [(docId, tf), ...]). Then we can use a simple python function to compute the tf-idf on the fly whenever needed (see also complete solution below):

```python
def ComputeTfIdf(n_docs, doc_list, tf):
    idf = max(0, log(n_docs/(1.0 + len(doc_list))))
    return idf * tf
```

2 - Similarly to what we did with the in-mapper combiner in Exercise 05, we can optimize the word count for each document by doing this operation in memory. Notice that, as discussed for MapReduce, you need to make sure that your temporary data will fit in memory. For this specific problem, a word count has a memory complexity of $O(\text{#unique-words})$ which is safe even if the pessimistic scenario of a document containing only unique words (since the biggest document is 12 MB).

```python
from collections import Counter

tfs_per_doc = docs.flatMap(lambda (docId, text):
    [(term, (docId, tf)) for term, tf in Counter(tokenize(text)).items()])
```

3 - We can further reduce the network traffic by replacing the path of a document with an shorter unique id. Notice that to get the same final result, you would also need to do the inverse mapping. The Spark function `zipWithUniqueId()` comes in handy for this:

```python
named_docs = sc.wholeTextFiles('wasb:///allbooks/*.txt')
named_docs_with_id = named_docs.zipWithUniqueId()
```

4 - Spark does not optimize automatically the number of partitions for a job. If we know the number of working nodes in a cluster, together with how many tasks each node can run in parallel, a reasonable number of partitions would be a small multiple of the product between these two. Our cluster has 5 nodes, each of which can run 3 tasks at the same time, a number of partitions some factor larger than 15. Factors between 5 and 20 are usually good.

```python
n_partitions = 100
```

Here is the improved code:

```python
from math import log
from collections import Counter
import re

# For better performance, we use “several” times more tasks than can be processed at the same time. Our cluster has 5 nodes,
# each of which can run 3 tasks at the same time, a number of partitions some factor larger than 15. Factors between 5 and 20
# are usually good.
# Load list of documents
# [(name, content)]
named_docs = sc.wholeTextFiles('wasb:///allbooks/*.txt', minPartitions=n_partitions)

# Enhance each document with a unique ID
# [((filename, content), id)] --> [((filename, content), id)]
named_docs_with_id = named_docs.zipWithUniqueId()

# Use the IDs instead of the filenames for higher speed
# [((filename, content), id)] --> [id, content]
docs = named_docs_with_id.map(lambda ((name, content), did): (did, content))

# Count the documents
n_docs = docs.count()
```
"Number of documents: %d\ n_docs"
"Number of partitions: %d\ docs.getNumPartitions()"

def tokenize(s):
    # Note that these map and filter are normal Python functions (not Spark actions)
    tokens = map(lambda w: w.strip(), re.split(\'\W+\', s.lower()))
    return filter(lambda t: len(t) > 0, tokens)

# Do a word count for each document
# [(docId, text)] --> [(term, (docId, tf))]
    tfs_per_doc = docs.flatMap(lambda (docId, text):
        [(term, (docId, tf)) for term, tf in Counter(tokenize(text)).items()])

# Group the TFs of different documents by term
# [(term, (docId, tf))] --> [term, list(docId, tf)]
    tfs_per_term = tfs_per_doc.groupByKey(n_partitions).persist()

# Now, given a search term, all information necessary to compute TFIDF
# is present in the corresponding value of tp_per_term

def ComputeTfIdf(n_docs, doc_list, tf):
    idf = max(0, log(n_docs/(1.0 + len(doc_list))))
    return idf * tf

def query(q, n_results):
    query_terms = tokenize(q)

    # Look up the TF lists for the query terms
    tfs_per_query_term = tfs_per_term.filter(lambda (term, _): term in query_terms)

    # Compute TFIDFs for all (query_term, docId) pairs
    # [term, list(docId, tf)] --> [docId, tfidf]
    query_tfidf = tfs_per_query_term.flatMap(lambda (term, doc_list):
        lambda (term, doc_list): map(  
            lambda (docId, tf): (docId, ComputeTfIdf(n_docs, doc_list, tf)),  
            doc_list)  
        )

    # Sum up TFIDFS for each document
    relevance_per_doc = query_tfidf.reduceByKey(lambda a, b: a + b)

    # Return most relevant documents
    return relevance_per_doc.takeOrdered(n_results, lambda (docId, relevance): -relevance)

# Then use the following code to run a query
#results = query('pirates and islands', 2)
results = query('derivation of the lorentz transformation', 2)
for (docId, relevance) in results:
    print ('[%.2f] - %s' % (relevance, docId))