Project: spaCy

2015-10-04T16:22:26Z aeneaswiener 53comments

Use in Apache Spark / English() object cannot be pickled

Web url: https://github.com/explosion/spaCy/issues/125

API url: https://api.github.com/repos/explosion/spaCy/issues/125

For spaCy to work out of the box with [Apache Spark](http://spark.apache.org) the language modles need to be pickled so that they can be initialised on the master node and then sent to the workers.

This currently doesn't work with plain pickle, failing as follows:

```

>>> from \_\_future\_\_ import unicode\_literals, print\_function

>>> from spacy.en import English

>>> import pickle

>>> nlp = English()

>>> nlpp = pickle.dumps(nlp)

Traceback (most recent call last):

[...]

TypeError: can't pickle Vocab objects

```

Apache Spark ships with a package called [cloudpickle](https://github.com/cloudpipe/cloudpickle) which is meant to support a wider set of Python constructs, but serialisation with cloudpickle also fails resulting in a segmentation fault:

```

>>> from pyspark import cloudpickle

>>> pickled\_nlp = cloudpickle.dumps(nlp)

>>> nlpp = pickle.dumps(nlp)

>>> nlpp('test text')

Segmentation fault

```

By default Apache Spark uses pickle, but can be told to use cloudpickle instead.

Currently a feasable workaround is lazy loading of the language models on the worker nodes:

```

global nlp

def lazyloaded\_nlp(s):

global nlp

try:

return nlp(s)

except:

nlp = English()

return nlp(s)

```

The above works. Nevertheless, I wonder if it would be possible to make the English() object pickleable? If not too difficult from your end, having the language models pickleable would provide a better out of box experience for Apache Spark users.

-------------------------------------------------------------------------

2015-10-06T12:03:19Z honnibal

I've spent a little time looking into this now.

The workflow that's a little bit tricky to support is something like this:

- Create an `English()` instance

- Change the state of some binary data, e.g. modify the lexicon

- Send it to workers, with new state preserved

Now, when I say "a little bit tricky"...If this is a requirement, we can do it. It'll mean writing out all the state to binary data strings, shipping ~1gb to each worker, and then loading from the strings. The patch will touch every class, and it might be fiddly, especially to keep efficiency nice. But there's no real problem.

The question is whether this work-flow is really important. I would've thought that the better way to do things was to divide the documents in the master node, and then send a reference to a function like this:

``` python

def do\_work(batch\_of\_texts):

nlp = English()

for text in texts:

doc = nlp(text)

# Stuff

distribute(texts, do\_work, n\_workers=10)

```

Does PySpark not work this way?

-------------------------------------------------------------------------

2015-10-06T15:46:04Z aeneaswiener

Yes the way you suggest is similar to what I am doing now with a lazy loaded `English()` object. I mainly created this issue in case someone else is running into similar problems when trying spaCy on Spark, as the error messages raised by Spark when failing to pickle the language models are not very helpful (workers just crash because of segmentation fault). This issue can be closed from my end.

-------------------------------------------------------------------------

2015-10-06T15:51:01Z honnibal

Thanks.

Let's leave it open for now and think about it a bit more.

-------------------------------------------------------------------------

2015-10-08T01:02:29Z chrisdubois

@honnibal When you say "writing out all the state", can you clarify what is involved here? Which objects inside of `nlp`'s state are easy to serialize and which ones aren't?

-------------------------------------------------------------------------

2015-10-08T01:29:00Z honnibal

Good question. A quick summary:

| Class | Instance | Changes if | Frequently changes? | Size | Description |

| --- | --- | --- | --- | --- | --- |

| `StringStore` | `nlp.vocab.strings` | Any unseen token is processed | Yes | 9mb | Mapping table of strings to integers. Serialized as list of strings. |

| `LexemeC\*` | `nlp.vocab.\_by\_orth` | User writes Lexeme properties | Maybe | 80mb | Pre-computed attributes for every lexical type in the vocabulary. |

| `numpy.ndarray` | `nlp.vocab.vectors` | User updates word vectors | Maybe | 200mb to 1gb | Word vectors. |

| `Matcher` | `nlp.matcher` | User adds to gazetteer | Yes | <1mb | User-custom entity recognition behaviour. |

| `Packer` | `nlp.vocab.serializer` | Shouldn't, but must check. | Think no? | Small | I think the serializer reads data off the vocab. But need to double check. |

| `thinc.LinearModel` | `nlp.parser.model` | Training | No | 490mb | Parser's statistical model. Immutable in normal use. |

| `thinc.LinearModel` | `nlp.entity.model` | Training | No | 38mb | NER statistical model. Immutable in normal use. |

| `thinc.LinearModel` | `nlp.tagger.model` | Training | No | 12mb | POS statistical model. Immutable in normal use. |

Really everything could be serialized, because everything knows how to write itself to disk and load itself back, usually in a binary format. One super simple way to do the serialization would be to dump to a directory, tar it, send the bits, untar, load. I'm sure we can do better than that, though,

In standard use, the only things likely to change are within `Vocab`.

If nothing is written to the NLP class, then serialization with Pickle is super simple: we can just tell it to load with the arguments originally passed to the constructor.

-------------------------------------------------------------------------

2015-10-10T00:56:51Z jkbradley

Just to chime in: Based on my experience with Spark, it sounds like loading English locally at each worker is best. Does this summary of pros/cons sound about right?

Pros

- Less communication (so faster)

- If the English object will be used on workers multiple times, it'd be important to cache it to prevent re-loading.

- Simpler to implement

- This seems like the biggest argument---the cost of dev time. Also, serialization might have to be updated if the English object gets updated.

- Less memory used on driver

- It's much better to do heavy work on workers than on the driver. If an executor dies, things are often recoverable, but more may be lost when the driver dies.

Cons

- Need to ensure configuration/setup of English is identical across workers

- This was the part I was not sure about: Is there any setup required, or is it sufficient to just load the English object at each worker? If there is some setup, then it'd be worth discussing what needs to be communicated for each worker to do the same setup.

-------------------------------------------------------------------------

2015-10-10T01:08:53Z honnibal

Thanks.

Okay so. To be clear, the \*\*wrong\*\* way to do this is to batch the texts into jobs yourself, and then map over the jobs, right? Instead if the task is per text, we should let Spark map over the texts, and it's our/user's job to make that work.

It's also wrong to let Spark see the English instance as a shared variable. It needs to be private to the workers, because we really don't want the workers to transfer it back to the driver.

In terms of the trade-offs, my perspective is that we're happy to take on implementation complexity. The library is very willing to do work so that users don't have to.

They key priority is that the library should make the right thing easy. Whatever our recommended workflow is, it should be the obvious and least-effort thing to do. But if there's an important trade-off to make, we don't want to have a silent default that just picks an option, and then the user gets a bad result and has to go back and figure out what went wrong.

I worry that adding this pickling capability will lead users down the wrong track. It seems to me that it makes the whole thing transparent, when actually there's a meaningful decision here, that maybe the user needs to make. Masking that decision isn't necessarily a good service to them.

-------------------------------------------------------------------------

2015-10-10T19:50:29Z jkbradley

> To be clear, the wrong way to do this is to batch the texts into jobs yourself, and then map over the jobs, right? Instead if the task is per text, we should let Spark map over the texts, and it's our/user's job to make that work.

If it's a pure map operation, then it is better to let Spark handle the map, if only because it can then handle distributing the work and providing some fault tolerance.

> It's also wrong to let Spark see the English instance as a shared variable. It needs to be private to the workers, because we really don't want the workers to transfer it back to the driver.

It's not wrong, but it is not ideal since 1GB is a fairly big object. It'd be nice to design an API which encouraged minimal communication: either (a) load it on the driver and broadcast it (transfer only once) or (b) load it on each worker (once). The API could potentially handle this under the hood.

#

If you're not worried about development effort, then actually serialization sounds like the best option. But you could hide it from the user. Here's what I'm thinking:

- Goal: Have users use the same English object on the driver or in Spark jobs, and not worry about communicating the big object.

- Implementation A: Simpler, but requires user to be aware of local vs. distributed operations

- When the user wants to do a distributed operation, they call `nlp(myRDDofTexts)` on the driver, and English handles the map under the hood.

- Under the hood, English could broadcast itself (requiring pickling), or just load itself locally on each worker.

- Implementation B: More complex, but user can be oblivious

- Within English, make the big objects transient (i.e., specify that via `\_\_getstate\_\_` for pickling).

- Store those big objects in broadcast variables (initialized on the driver in the constructor), which are then read when needed on the workers.

Both of these options require pickling, but B requires somewhat more complex pickling.

After thinking more about it, I'd recommend broadcasting English (unless you think it will grow far beyond 1GB in the future). That will be easier for handling English's settings or parameters.

#

> I worry that adding this pickling capability will lead users down the wrong track.

I think it's OK to hide things from the user. As far as I can tell, the main things the user needs to be aware of are (a) whether any options they set will be used both on the driver and on workers and (b) when a data object is local vs. distributed.

-------------------------------------------------------------------------

2015-10-11T01:07:18Z honnibal

Okay, thanks. Is this how it works on sk-learn, Theano, etc?

-------------------------------------------------------------------------

2015-10-12T05:45:52Z honnibal

Can I expect tempfiles and tempdirs to work? Is the file system shared between the driver and all children?

-------------------------------------------------------------------------

2015-10-12T09:01:43Z honnibal

Okay so `cloudpickle.dump(nlp, file\_)` is working, in a branch (`attrs`). I'll do some further testing with Spark before pushing it up to master.

I'm concerned that 100% correctness seems difficult to achieve here. The only \_known\_ problem with the current implementation arises when you call the `.train()` methods prior to pickling. But, I haven't done much to test against various subtle problems that might occur. There's a lot of potential state, and guaranteeing that 100% of it is saved is difficult. The current implementation also saves to disk, using tempfiles. This might prove to be problematic.

Once we're playing with the current implementation, we can see whether another approach might be better.

-------------------------------------------------------------------------

2015-10-18T08:46:48Z honnibal

Just pushed v0.95. Everything should now pickle.

The only known issue is that you shouldn't begin training a model and then pickle it part way through training. Please keep an eye out for other issues, and report them :)

-------------------------------------------------------------------------

2015-10-21T22:00:10Z chrisdubois

I'm trying this out with the following snippet

```

import cloudpickle

nlp = English()

p = cloudpickle.CloudPickler(open('tmp', 'w'))

p.dump(nlp)

p.close()

```

I get the following:

```

TypeError: can't pickle Tokenizer objects

```

It seems like the pickler tries to pickle a Vocab, Tokenizer, Tagger, Parser, and Matcher object. When it gets to the [Tokenizer](https://github.com/honnibal/spaCy/blob/master/spacy/tokenizer.pyx#L22), it doesn't seem to have a doesn't seem to have a `\_\_reduce\_\_` implemented. I don't see Tokenizer in the above list of "state objects that need pickling"; do we not expect the pickler to find the Tokenizer object?

Thanks in advance for any hints.

-------------------------------------------------------------------------

2015-10-22T09:14:33Z honnibal

This is odd ‚Äî I definitely thought I'd done this. And I don't get how my tests passed if the Tokenizer doesn't pickle. I'll figure this out, thanks.

-------------------------------------------------------------------------

2015-10-22T18:09:17Z chrisdubois

FWIW: tests/test\_pickle.py passes for me as well. There seems to be something different between `p.dump(object)` and `p.dump(object, file)` that I don't understand.

Also, my test code above wasn't quite correct nor complete. It should be:

```

import spacy

from spacy.en import English

nlp = English()

import cloudpickle

f = open('tmp', 'wb')

p = cloudpickle.CloudPickler(f)

p.dump(nlp)

f.close()

```

-------------------------------------------------------------------------

2015-10-24T06:09:45Z chrisdubois

I made a PR #149 that appears to pickle the tokenizer a bit better, but if I try and query for vectors from the deserialized model I get

```

> raise ValueError(

E ValueError: Word vectors set to length 0. This may be because the data is not installed. If you haven't already, run

E python -m spacy.en.download all

E to install the data.

spacy/tokens/token.pyx:149: ValueError

========================================================================================================= 1 failed in 30.63 seconds ==========================================================================================================

```

even though I'm certain I've downloaded data.

Then I found [spacy/vocab.pyx:108](https://github.com/honnibal/spaCy/blob/master/spacy/vocab.pyx#L108) which says `# TODO: Dump vectors`. Any suggestions on what needs to be done to save the vectors?

-------------------------------------------------------------------------

2015-10-26T01:33:54Z honnibal

Fixed ‚Äî thanks.

-------------------------------------------------------------------------

2016-01-19T14:05:22Z honnibal

Reopening this. The current pickling implementation was only supposed to be an exploratory kludge. However, I didn't leave a TODO and the status of it got lost.

`Vocab.\_\_reduce\_\_` currently writes state to temp files, which are then not cleaned up. Pickling therefore fills the disk, and really only pretends to work.

The root of the problem is that a number of spaCy classes carry large binary data structures. Common usage is to load this data and consider it immutable, however you can write to these models, e.g. to change the word vectors, and pickle should not silently dump these changes. On the other hand it's harsh to assume we always need to write out the state. This would mean that users who follow the pattern of keeping the data immutable have to write out ~1gb of data to pickle the models. This makes average usage of Spark etc really problematic.

We could do this implicitly with copy-on-write semantics, but I don't think it's great to invoke some method where it may or may not write out 1gb of data to disk, depending on the entire execution history of the program.

We could have a more explicit version of copy-on-write, where all the classes track whether they've been changed, and then the models should refuse to be pickled if the state is unclean. Users would then explicitly save the state after they change it. I think this is a recipe for having long-running processes suddenly die, though. Mostly Python is designed around the assumption that things can either be pickled or they can't. It's surprising to find that your pickle works, sometimes, depending on state, but then your long-running process dies because you didn't meet the assumed invariant. And then next time you run, you get an error in that other place in your code where the classes get pickled.

I've been thinking for a while that context managers are the idiomatic standard for dealing with this problem. The idea would be that if you want to write to any of this loaded data, you have to open it within a context manager, so that the changes are explicitly scoped, and you explicitly decide whether you want to save the changes or dump them.

Ignoring the naming of everything, this might look like:

``` python

from spacy.en import English

nlp = English()

# Open a pre-trained model, do some more training, and save the changes

with nlp.entity.update\_model(file\_or\_path\_or\_etc):

for doc, labels in my\_training\_data:

nlp.entity.train(doc, labels)

# Change the vector of 'submarines' to be the vector formed

# by "spaceships - space + ocean"

# When the context manager exits, revert the changes

with nlp.vocab.update\_lexicon(revert=True):

submarines = nlp.vocab[u'submarines']

spaceships = nlp.vocab[u'spaceships']

space = nlp.vocab[u'space']

ocean = nlp.vocab[u'ocean']

submarines.vector = (spaceships.vector - space.vector) + ocean.vector

```

-------------------------------------------------------------------------

2016-01-19T20:04:16Z aborsu

Having a bit of experience in spark, using Spacy with spark will always be tricky as the stringstore will evolve differently in all the workers so a string converted to an int will not necessarily be equivalent to the same int on another worker. So any shuffling of Spacy data across nodes will result in strange things.

Knowing this, serializing the English instance is of little interest. To avoid dependency hell (making sure the same version of spacy and it's data is installed on each node), a much better approach would be to broadcast the library and the data models as broadcast objec (which takes less space and is sent as a torrent file)t, un-serialize them in a temp directory and lazy load them from there.

-------------------------------------------------------------------------

2016-01-19T20:12:04Z honnibal

That's a problem for some usages, yes. But the tokenizer is over 100x faster than the rest of the library, so you can simply tokenize all the text in a single process, to initialize the `StringStore` before you send out the batches.

There will also be use-cases where you only care that the activity on each shard is internally consistent. For instance, if you want to just recognise entities and store them in textual form, you don't care that the `StringStore` instances can diverge between your shards.

We think it's important to support Spark as best we can. We can at least pickle our data correctly :)

-------------------------------------------------------------------------

2016-01-19T20:22:44Z aborsu

I agree with both things you said, you could indeed tokenize all texts in one process before sending them out, but if you need to use Spark (and I mean need not want to because it's cool), chances are that this would still be blocking as all the data would have to go through a single node.

And for your second use case, I still think that bundling the Spacy data and classes then lazy loading them would be easier than adding serialization to all your classes.

(I'm saying that because before playing with Spacy on apache-storm, I painfully implemented a serializable nlp pipeline in scala on Spark.)

Extract from the [spark 1.6.0 manual](http://spark.apache.org/docs/latest/submitting-applications.html)

```

For Python, you can use the --py-files argument of spark-submit to

add .py, .zip or .egg files to be distributed with your application.

If you depend on multiple Python files we recommend packaging them

into a .zip or .egg.

```

\*\*edit\*\* Also, Spacy uses compiled C-libraries that need to be installed on all the nodes. So installing Spacy on all the machines and just packaging the data to be lazy-loaded seems the easiest solution to me. [confer](http://blog.cloudera.com/blog/2015/09/how-to-prepare-your-apache-hadoop-cluster-for-pyspark-jobs/)

-------------------------------------------------------------------------

2016-01-19T20:34:05Z honnibal

Hmm. I do see your argument.

-------------------------------------------------------------------------

2016-02-10T16:00:40Z honnibal

With the new improvements in loading time (down to 15s from about 100s), and the addition of multi-threading, I'm not nearly as keen on supporting in-memory pickling as I was before. I think @aborsu is right: the problem of serializing such that pickle can work is basically the same as the problem of loading from disk, so there's no reason to believe unpickling the classes would be faster than just loading them as new objects.

The important thing is to have a workflow for users that want to use spaCy with a map/reduce framework like Spark. I think supporting the pickle protocol is not necessarily the best way to do that. Thoughts?

-------------------------------------------------------------------------

2016-02-18T19:32:39Z mikepb

I also had issues with pickling with [SFrame](https://github.com/dato-code/SFrame), which behaves similarly to how Spark has been described to work here.

To avoid pickling and unpickling data that could just be read from disk (on a single machine), I've been using the following `Pickleless` proxy object.

``` py

class Pickleless(object):

def \_\_init\_\_(self, cls, \*args, \*\*kwargs):

object.\_\_setattr\_\_(self, '\_\_init', (cls, args, kwargs))

def \_\_o(self):

try:

o = self.\_\_dict\_\_['\_\_instance']

except KeyError:

cls, args, kwargs = self.\_\_dict\_\_['\_\_init']

o = cls(\*args, \*\*kwargs)

object.\_\_setattr\_\_(self, '\_\_instance', o)

return o

def \_\_getstate\_\_(self):

return self.\_\_dict\_\_['\_\_init']

def \_\_setstate\_\_(self, state):

cls, args, kwargs = state

Pickleless.\_\_init\_\_(self, cls, \*args, \*\*kwargs)

def \_\_call\_\_(self, \*args, \*\*kwargs):

return self.\_\_o()(\*args, \*\*kwargs)

def \_\_getattr\_\_(self, name):

return getattr(self.\_\_o(), name)

def \_\_setattr\_\_(self, name, value):

setattr(self.\_\_o(), name, value)

\_English = Pickleless(spacy.en.English)

def nlp(text, \*\*kwargs):

return \_English(\_decode(text), \*\*kwargs)

```

I had originally implemented the proxy using thread-local storage, but this works just as well for my purposes.

Just my 2¬¢...

-------------------------------------------------------------------------

2016-03-10T21:18:32Z gushecht

Hi guys,

Relative noob here looking for a recommendation.

My goal: POS tag the English language Wikipedia dump (approx 23GB, maybe 3B words).

Can someone tell me either:

A) DON'T DO IT, go for something smaller; or

B) A distillation of the above discussion into the current recommended workflow? Unfortunately, I can't quite follow the conversation.

P.S. Thanks Matthew for spaCy! I'm using that and following your sense2vec implementation, with a few changes, to attempt the [word clustering that is mentioned on Google Code word2vec page](https://code.google.com/archive/p/word2vec/).

-------------------------------------------------------------------------

2016-03-10T21:31:00Z mikepb

I was planning on doing that until... I ran out of memory. I was using the scikit-learn word count module, so YMMV. They also have a good implementation of word2vec, but it unfortunately the scikit pipeline does not always use iterators and would need quite a bit of memory (>24GB including paging when I tried a \_much\_ smaller dataset).

I recommend starting with something smaller, such as using only the Wikipedia page titles, to test out your pipeline. Debugging would go more quickly too :+1:

If you are on a Mac, you can enable multithreading per the instructions in #267 .

This conversation is about distributing spacy over multiple machines. The problem is not that it doesn't work, but that the distributed computing frameworks mentioned in this thread pickle and send over the spacy data models. That could amount to several GB of data. Instead, workarounds have been posted to ensure that spacy loads its models from local disk on each machine.

The workaround also means that you won't be able to share spacy states among machine, including custom dictionaries.

Hope this helps!

Cheers.

-------------------------------------------------------------------------

2016-03-10T21:32:53Z honnibal

Tagging Wikipedia should be no problem. I haven't benchmarked the tagger in a while, but I think it should be running at over 100k tokens a second per process, and it doesn't use much memory. On a high compute node you should be done in a couple of hours.

I've run the full pipeline on over ten times that much data, and I don't use Spark --- just simple multiprocessing, on a node with lots of cores.

Example script: https://github.com/spacy-io/spaCy/blob/master/examples/parallel\_parse.py

-------------------------------------------------------------------------

2016-03-10T21:35:08Z gushecht

Thanks both of you. I was thinking Spark because of the benchmarking on this [blog post](https://spacy.io/blog/part-of-speech-POS-tagger-in-python) suggested I'd be at it for about half a year (although I also may have misinterpreted the results).

Will continue onwards and let you know about my results when I get them!

-------------------------------------------------------------------------

2016-03-10T21:35:31Z mikepb

Second what @honnibal said. The issues I ran into were outside of spacy in doing the word count. You could try the [HashingVectorizer](http://scikit-learn.org/stable/modules/generated/sklearn.feature\_extraction.text.HashingVectorizer.html#sklearn.feature\_extraction.text.HashingVectorizer) in sklearn, though I had issues with that too.

-------------------------------------------------------------------------

2016-03-11T01:06:14Z gushecht

Hey @honnibal, before going onto AWS, I wanted to start small. I took your merge\_text.py from the sense2vec implementation and trimmed it down. Running it on a pretty small input is taking a while. Can you see something that I'm doing wrong? Updated script and data file attached.

I'm on a 4 core 2011 MacBook Pro with 8GB RAM, so it shouldn't be the hardware.

[Archive.zip](https://github.com/spacy-io/spaCy/files/168294/Archive.zip)

Thanks!

-------------------------------------------------------------------------

2016-03-11T01:36:16Z mikepb

Seems to me that Parallel is doing the pickle thing mentioned in this thread.

This means that each worker process will take a while to load all the data models. On my MacBook Pro, each Python worker process uses between 800MB and 1.5GB. I have 16GB physical memory and an SSD, so I don't notice paging as much. Apple didn't make SSD MBP in 2011, so it's probably going to be much slower for you. I stopped your script after spending a few minutes writing this post.

The overhead of pickling and unpickling the spacy data models is significant. As spacy itself is very fast, it would take huge batch sizes to make it worth it to use this multiprocessing model.

Instead, it would be significantly faster to use a single Python process and spacy's `nlp.pipe(docs, n\_threads=-1)` multithreading API. On Linux, this is supported natively. On Mac, you'll need to do a little more work (#267). I modified your script to not use Parallel and it ran to completion in seconds, single-threaded.

-------------------------------------------------------------------------

2016-03-12T02:56:55Z rw

Would a different serialization format help here? Pickle is heavyweight.

There are many to choose from.

I maintain the Python port of FlatBuffers, which is a serialization format

optimized for minimal memory usage. Happy to answer any questions.

On Mar 10, 2016 5:36 PM, "mikepb" notifications@github.com wrote:

> Seems to me that Parallel is doing the pickle thing mentioned in this

> thread.

>

> This means that each worker process will take a while to load all the data

> models. On my MacBook Pro, each Python worker process uses between 800MB

> and 1.5GB. I have 16GB physical memory and an SSD, so I don't notice paging

> as much. Apple didn't make SSD MBP in 2011, so it's probably going to be

> much slower for you. I stopped your script after spending a few minutes

> writing this post.

>

> The overhead of pickling and unpickling the spacy data models is

> significant. As spacy itself is very fast, it would take huge batch sizes

> to make it worth it to use this multiprocessing model.

>

> Instead, it would be significantly faster to use a single Python process

> and spacy's nlp.pipe(docs, n\_threads=-1) multithreading API. On Linux,

> this is supported natively. On Mac, you'll need to do a little more work (

> #267 https://github.com/spacy-io/spaCy/issues/267). I modified your

> script to not use Parallel and it ran to completion in seconds,

> single-threaded.

>

> ‚Äî

> Reply to this email directly or view it on GitHub

> https://github.com/spacy-io/spaCy/issues/125#issuecomment-195136000.

-------------------------------------------------------------------------

2016-04-07T23:16:47Z sjjpo2002

I ran into this issue too. I'm trying to do some work on the text from Wiki Dump. Using a Spark map function do\_work. As suggested here when I move English() inside the function it works but when it's outside it doesn't. Python crashes. But instantiating English() in every function call makes it super slow and practically useless.

```

`def do\_work(batch\_of\_texts):

nlp = English()

for text in texts:

doc = nlp(text)`

```

-------------------------------------------------------------------------

2016-04-07T23:20:52Z mikepb

I've had success using the `Pickeless` workaround in https://github.com/spacy-io/spaCy/issues/125#issuecomment-185881231

The wrapper class essentially tells Python not to pickle the English. Instead, the Pickleless English are reloaded and cached for each process, when defined as a global variable.

Remember that custom vocabulary will not transfer between Python processes using this method.

-------------------------------------------------------------------------

2016-04-07T23:24:30Z gushecht

I'm sure there's a better way to do this, but what I ended up doing was writing a script to process my corpus in parallel on one machine, instead of using Spark. Each separate process had its own instantiation of the English object (so it was memory intensive) and a portion of the corpus.

-------------------------------------------------------------------------

2016-04-08T00:31:01Z sjjpo2002

I don't know how but the Pickeless workaround does work. Python doesn't crash anymore but the process is still slow that I would guess it's normal for the large text that I'm dealing with

-------------------------------------------------------------------------

2016-04-08T01:46:51Z honnibal

I really need to write a blog post/tutorial on working with large jobs. I think at the moment this is quite confusing. I'm grateful to @mikepb , @aborsu and others for their contributions to the discussion here.

@sjjpo2002 : As a quick note, this doesn't change the deeper issues, but your code should be faster if you do:

``` python

def do\_work(batch\_of\_texts):

nlp = English()

for doc in nlp.pipe(batch\_of\_texts, n\_threads=-1):

# Do stuff.

```

This allows spaCy to make use of multi-threading for the most expensive parts of the pipeline, the parser and named entity recogniser. In future more of the pipeline will be multi-threaded, so I'd recommend using this construction even if you're just using the tokenizer and tagger, which are currently single-threaded (they're very fast though --- and load much quicker.)

A much better improvement will be gained if you can manage to reuse the same `nlp` object across more text, i.e. if you can increase the size of your batches of documents.

To summarize the deeper issue as I understand it: the ultimate dream is to have `reduce(analyse\_data, map(process\_texts, texts))` "Just Work", without having to deal with the details of how work is allocated to physical machines, let alone individual processes or threads.

Unfortunately, with spaCy (and probably with lots of other situations), the constants really matter. Loading the spaCy models takes about as much time as processing 1,000 documents (at ~200 tokens each). This means you have to somehow manually manage the chunk size. You have to somehow help Spark schedule the work in larger increments. If you have your data already split out into large files, e.g. by month, then it might be enough to have each process work on one month of data. In this situation, we can make `reduce(analyse\_data, map(process\_month, months))` work transparently, which is nice. But if what you have is a collection of texts, not a collection of larger archives, you have to think about the details a bit more.

My main advice would be to not assume that Spark, Hadoop etc are automatically better than using a single machine and using multiple processes with spaCy's `.pipe()` method for multithreading. In the worst case, if you had a chunk size of 1, you could use a thousand node cluster and have your job complete slower than it would on a Macbook Air. So, if your process is slow, it might be worth taking a closer look and checking that you're at least doing around 10,000 tokens per thread per second. If you're doing an order of magnitude less than that, then either the problem is outside of spaCy, or the work isn't being scheduled effectively.

-------------------------------------------------------------------------

2016-04-08T06:20:33Z sjjpo2002

Thanks @honnibal for your advice. I've been trying to do NLP with Spark with NLTK and Stanford and etc. But their objects are big and not serializable. I really like Spacy's simple and all in one approach. It's close integration with Spark will be a great advantage for many researchers. As you mentioned that would be heaven to see this code just work:

``` python

from operator import add

from pyspark import SparkContext, SparkConf

from nltk.corpus import stopwords

from spacy.en import English

wiki\_title\_words = set(article\_titles)

english\_vocab = set(w.lower() for w in nltk.corpus.words.words())

stop\_words = set(stopwords.words("english"))

nlp = English()

def lemmatize\_filter(s):

candidates = nlp(s)

result = []

for candidate in candidates:

if (candidate.lemma\_ not in stop\_words) and \

(candidate.pos\_ != u'PUNCT') and \

(candidate.lemma\_ in english\_vocab or \

candidate.lemma\_ in wiki\_title\_words):

result.append(candidate.lemma\_)

return result

sc = SparkContext()

rawData = sc.textFile(r"wiki\artile\_per\_line.txt") # title \t article\_body

flatten\_words = rawData.flatMap(lambda x: lemmatize\_filter(x.split('\t')[1]))

words\_reduced = flatten\_words.map(lambda x: (x, 1)).reduceByKey(add)

print "Number of distinct lemmas in the whole Wikipedia: ", words\_reduced.count()

```

So far it looks like using Spacy pipe without Spark is the fastest option to do the job. I will feed Spark with processed data for the rest of the work.

-------------------------------------------------------------------------

2016-04-08T22:01:33Z sjjpo2002

It might be off topic but @mikepb mentioned with `nlp.pipe([batch\_docs], n\_thread=-1` we can see a lot of improvement in speed. But I'm not gaining speed using pipe. I suspect that I have to enable multi-threading using OpenMP. But I couldn't find any instructions to do that on Windows.

-------------------------------------------------------------------------

2016-04-08T22:05:51Z mikepb

Yes, you would need OpenMP to enable multithreading. It's tricky to get right:

- See #267 and #266 for info on using OpenMP on OS X.

It's wonderful when it works though!

I believe Microsoft has implemented OpenMP in Visual Studio. I'm wholly unfamiliar with Windows development...

-------------------------------------------------------------------------

2016-04-09T05:43:43Z henningpeters

@mikepb Normally, you don't have to do anything special on Windows, it's trickier on OS X because Apple's default clang doesn't support OpenMP, yet. MSVC does, though.

-------------------------------------------------------------------------

2016-04-10T00:54:20Z sjjpo2002

@henningpeters I managed to compile and run pipe with OpenMP support. It looks beautiful when it utilizes all CPU cores. OpenMP support is by default disabled for MSVC. I added the compiler option to the setup file and committed the changes to the master branch. I'm surprised nobody is using Windows it seems. But with that switch enabled next releases will work properly on Windows

@honnibal What you mentioned about some parts of the pipeline multi-threaded made a lot more sense after I managed to use pipe in multi-threaded scenario. But it also seems single-threaded parts of the pipeline take almost the same time as the multi-threaded parts. That would make it a lot faster if those single threads also support multithreaded. I attach my CPU usage here. It's clear the single threads play some role in the time it takes to finish the job.

![pipe\_performance](https://cloud.githubusercontent.com/assets/8649062/14407459/644783d0-fe7e-11e5-9340-f6b06b44433f.jpg)

-------------------------------------------------------------------------

2016-04-10T02:07:39Z mikepb

Tricky :+1:

> On Apr 9, 2016, at 5:54 PM, SJ notifications@github.com wrote:

>

> @henningpeters https://github.com/henningpeters I managed to compile and run pipe with OpenMP support. It looks beautiful when it utilizes all CPU cores. OpenMP support is by default disabled for MSVC. I added the compiler option to the setup file and committed the changes to the master branch. I'm surprised nobody is using Windows it seems. But with that switch enabled next releases will work properly on Windows

>

> ‚Äî

> You are receiving this because you were mentioned.

> Reply to this email directly or view it on GitHub https://github.com/spacy-io/spaCy/issues/125#issuecomment-207892428

-------------------------------------------------------------------------

2016-04-10T07:49:51Z henningpeters

Oh, seems we overlooked msvc's openmp option. Thanks @sjjpo2002 for your PR! Just merged it.

-------------------------------------------------------------------------

2016-04-12T07:10:19Z henningpeters

It seems the `/openmp` option is not available for some(?) Visual Studio express editions. That means many/most(?) Windows users won't be able to install spaCy anymore from source for free. There seems to exist a workaround (http://stackoverflow.com/questions/865686/openmp-in-visual-studio-2005-standard), but it's not something I'd like to put in our getting-started install guide. Hence, I think we should make OpenMP support on Windows optional for now.

-------------------------------------------------------------------------

2016-04-12T18:10:06Z sjjpo2002

Microsoft Visual C++ Compiler for Python is [free ](https://www.microsoft.com/en-us/download/details.aspx?id=44266)for eveyone and it supports openmp.

-------------------------------------------------------------------------

2016-04-12T18:15:20Z henningpeters

Thanks, do you know whether this also works for Python 3.4/3.5? We need to have a solution in place that works as seamless as possible across all environments.

-------------------------------------------------------------------------

2016-04-12T18:46:35Z sjjpo2002

I never tried it on Python 3.x. But a quick check for /openmp [here](https://msdn.microsoft.com/en-us/library/fw509c3b%28v=vs.100%29.aspx) it is supported on VS 2010 ans 2015. For Python 3.4 using VS 2010 and for 3.5 VS 2015 are [suggested](http://stackoverflow.com/questions/29909330/microsoft-visual-c-compiler-for-python-3-4).

But, I agree that it should be tested on all platforms. Maybe I do it over the weekend.

-------------------------------------------------------------------------

2016-04-12T19:07:37Z henningpeters

My reading about it so far is that MS only provides support for OpenMP in VS professional editions or higher. Here's the error output from VS 2010 express edition: https://ci.spacy.io/builders/spacy-win64-py34-64-install/builds/98/steps/shell\_2/logs/stdio.

It might be possible to make it work as I suggested above, but so far I haven't seen a simple install instruction I would like to keep as default for spaCy all users.

-------------------------------------------------------------------------

2016-06-08T05:43:48Z aliabbasjp

I think this is relevant:

https://github.com/spacy-io/spaCy/issues/413

-------------------------------------------------------------------------

2017-05-07T16:21:20Z ines

Closing this and making #1045 the master issue. Work in progress for spaCy v2.0!

-------------------------------------------------------------------------

2017-11-18T11:02:03Z easimadi

Dears,

Is this issue resolved with the release of spacy 2.0.

How can I use spacy in Spark?

Thanks for your help.

ea.

-------------------------------------------------------------------------

2018-05-08T06:55:49Z lock[bot]

This thread has been automatically locked since there has not been any recent activity after it was closed. Please open a new issue for related bugs.

-------------------------------------------------------------------------