Project: spaCy

2016-03-10T17:34:10Z ELind77 37comments

Streaming Data Memory Growth

Web url: https://github.com/explosion/spaCy/issues/285

API url: https://api.github.com/repos/explosion/spaCy/issues/285

I have been using spacy for streaming data (twitter and news stories mostly) and I believe that the fundamental design of the vocab/StringStore in spacy is problematic for streaming processing. When used for batch jobs the additional memory overhead of storing a new lexeme struct for each new word form encountered in parsing is negligible compared to the speed gains, and because most text conforms to the assumption that vocabulary size grows logarithmically as the total number of tokens grows linearly this is usually a safe bet. But for streaming text, especially for social media where new terms are invented by the minute (hashtags and URLs in particular) this assumption no longer holds and the spacy vocabulary storage represents a dynamic element in what should be a completely static production deployment.

In order to test this assumption, I took one million tweets and performed a rudimentary analysis using the resources module in python to get the maximum memory used by the program at regular intervals during processing. I first performed some minor preprocessing to remove newlines from the data so that it could be read line by line so that it wouldn't all be kept in memory, then I ran spacy with all models set to false, only the tokenizer loaded. I then did the same thing again after removing all URLs, hashtags, and twitter mentions from the data , and then filtering all empty strings (this resulted in a 1.4% data loss in terms of total tweets processed but that's fairly minor).

The final result was that spacy used an additional 278.6 MB after tokenizing the raw tweets and 60.99 MB of additional memory when tokenizing the pre-processed tweets. This result confirms my hypothesis but also shows that the memory increase really isn't all that significant (especially at the relatively low volume that I am currently processing). But it still points to a potential flaw in the design of the library.

My suggestion/request in the near term would be to have an option to make the vocabulary read only so that users who want to be able to leave spacy alone to do streaming data processing don't need to worry about changing memory requirements. In the long term, I think that an optimal solution would be to add some functionality for a timeout on vocabulary entries that aren't loaded at initialization. E.g. if this lexeme hasn't been accessed for the last \_n\_ seconds, delete it from the StringStore. And \_n\_ would be user configurable.

My code and results are available here: https://github.com/ELind77/spacy\_memory\_growth

Thanks again for continuing to develop such a great library!

-- Eric

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

2016-03-10T21:36:04Z honnibal

I really need to fix this issue. Thanks.

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

2016-03-15T19:18:30Z syllog1sm

I think this patch from a few weeks ago might have fixed this. Unfortunately the patch wasn't pushed to master --- I've only just merged it now.

The fix is: https://github.com/spacy-io/spaCy/commit/141639ea3a8fc65f006d16594ab207b3a1c8c7d5

Could you test this by `pip install https://github.com/spacy-io/spaCy/archive/master.zip` ?

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

2016-03-21T21:23:40Z ELind77

Sorry this has taken a while. I'll test again today/tomorrow and get back to you.

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

2016-03-26T00:25:04Z ELind77

I performed the same tests again, both installing from the zip you posted above and installing directly from master (commit 9cd21ad5b5aa664642a2e17925cd7b39eacb9aa9) and got nearly identical results to my previous trials.

If you believe that this is the cause of some kind of memory leak, I think we should really take a look at my testing script and update it as it's very rudimentary and I'm far from an expert profiler. However, I don't think that this is a leak. As I said in my original post, I think that this is just part of how spacy works. When parsing things like social media where there are many tokens that occur only once (e.g. links) storing them in the StringStore causes memory bloat. In your comments on https://github.com/spacy-io/spaCy/issues/172 you proposed a batch-processing generator that uses, and then throws away a tokenizer object for each batch in order to help find OOV tokens. I think that's a fine approach, and could even be done a bit more quickly by asynchronously loading the new English() instance and replacing the old one when the new is ready, but that still leads to quite the slow down.

If your feeling is that spacy is really meant for batch processing and that I should use mini-batches if I want to approximate streaming, I can do that. Spacy is still far superior to anything else out there in my opinion, but it would be nice if I could use it with the expectation of roughly constant space complexity.

-- Eric

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

2016-03-29T03:52:16Z honnibal

To clarify a little bit, the current release version has three known places that could be growing in memory use.

1. The `StringStore`

2. A cache in the `Tokenizer`

3. The `Vocab`, for tokens that are part of `prefix`, `infix` and `suffix` patterns.

The patch I asked you to try out addresses 3. We can also easily address 2. Addressing 1 is hard, because we currently intern all the strings, which is a much easier policy to implement than something more subtle.

Can you report the lengths of the `StringStore` class in your two benchmark cases? There's currently no Python API for inspecting the size of the tokenizer's cache, so it's easiest to do this by elimination.

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

2016-08-09T15:42:48Z ptully

Hi @honnibal, why do you think addressing 1 is so hard? What about FIFO queue or similar, or something like @ELind77 suggested like:

> functionality for a timeout on vocabulary entries that aren't loaded at initialization

Do you have any more information on this issue since it cropped up a few months ago? I notice the same type of memory issues on my systems that analyze streaming Twitter data - note I've not yet narrowed it down to spacy yet but my first cursory look found this ticket to be the most relevant possibility

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

2016-08-09T15:44:20Z ptully

Also curious if this issue is already solved already, I will test updating my version (currently 0.100.6) to see if that helps at all

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

2016-08-16T18:43:57Z natb1

Hi @honnibal, I have had similar issues in my streaming application. Basically memory grows at a logarithmic-ish pace. We have to deal with it as though it were a memory leak and periodically re-initialize the code.

I ran the benchmark you requested above - collecting metrics on the length of the `StringStore` as memory usage grows. Here are the results:

<img width="555" alt="screen shot 2016-08-16 at 2 24 25 pm" src="https://cloud.githubusercontent.com/assets/1669062/17711263/c23cb262-63be-11e6-9aaf-96e9763a00e0.png">

Here is the code I used to create the metrics. It basically ran until I ran out of memory on a 4G box. https://github.com/natb1/spaCy/blob/memory-benchmark/spacy/tests/benchmark/test\_memory.py

I'd be glad to help implement some strategies to address this problem if you could help me isolate the issue and/or suggest some approaches.

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

2016-08-23T20:24:17Z tomtung

Same problem here. Would also be glad to help.

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

2016-08-25T13:36:45Z ptully

pinging @henningpeters given recent announcement on spaCy homepage

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

2016-08-25T15:03:04Z honnibal

To clarify the current behaviour a little: `StringStore` is currently interning \_all\_ strings seen. I agree that this should be changed. I'll discuss the design decision here, so that we can consider the trade-offs.

I'll start from the beginning: why intern the strings? Two main reasons:

1) String-to-int mapping

2) Save memory to represent lots of documents at once.

We can't do without 1 entirely ‚Äî it's too fundamental to how spaCy is working, and we definitely don't want to be making lots of string comparisons. Comparing by integer value is pretty important.

Consideration 2 is very useful, but it's only really a saving if strings occur multiple times. Certainly, for strings that occur once, there's no advantage. And it's also bad to have unbounded memory use on the streaming process.

So the solution we want to get to is one where a limited number of somewhat common strings are interned in the common vocab. However, we still need to map \_all\_ strings, even rare ones, to integers. We also want the string-to-int table to be consistent, even for rare strings.

Here's the bit of code where the memory growth is occuring:

https://github.com/spacy-io/spaCy/blob/master/spacy/strings.pyx#L147

The purpose here is to resolve a string to an integer. We first hash the string. Now, this is an integer representation --- so why not just use the hash? The problem is we also want the inverse mapping. We therefore store the string, causing the memory growth.

If we insist that all integers can always be mapped back to strings, there's no solution. We have to accept the memory growth. But if we can accept that these strings pass out of date, so that they're around for a while and then they're not, the situation should be manageable.

A simple way to achieve this is to extend the `StringStore` so that the mapping is split in two. There's the main intern area, which holds a fixed number of strings, hopefully the common ones. But there's also a rolling buffer, in which strings are interned, and then later freed. This could be a LRU cache, or even something simpler. Efficiency is not really a problem here: only a small percentage of the encountered tokens will be triggering this logic, so we don't have to make it blazing fast, and it's easy to make sure we operate on contiguous buffers.

A slightly more tricky solution is to do some reference counting. The idea here would be for the `Doc` object to register interest in all OOV strings it owns. When the ref count of an OOV string drops to 0, it's freed. This way, if you keep a `Doc` object in memory, you know that the string lookup will always be well behaved ‚Äî but if you're letting the `Doc` objects pass out of scope, your memory won't be growing.

I think for both solutions, we should use the hash of the string as the integer representation for OOV strings. This means that at least the string-to-int mapping will stay consistent, even if strings are passing out of memory. The only way to have a problem here is if you hold onto the integer representation, release all of the documents, and later want to recover the string. In this situation, you'll be out of luck --- but we'll at least know to use an OOV symbol when you try to look up your string.

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

2016-08-28T02:18:34Z tomtung

Simply hashing oov tokens sounds good enough to me. As long as we know the

indices of the fist and last characters of the token in the input text, so

that we can look it up if we need to, I don't find saving the token in

string store particularly necessary.

On Thu, Aug 25, 2016, 8:03 AM Matthew Honnibal notifications@github.com

wrote:

> To clarify the current behaviour a little: StringStore is currently

> interning \_all\_ strings seen. I agree that this should be changed. I'll

> discuss the design decision here, so that we can consider the trade-offs.

>

> I'll start from the beginning: why intern the strings? Two main reasons:

>

> 1) String-to-int mapping

>

> 2) Save memory to represent lots of documents at once.

>

> We can't do without 1 entirely ‚Äî it's too fundamental to how spaCy is

> working, and we definitely don't want to be making lots of string

> comparisons. Comparing by integer value is pretty important.

>

> Consideration 2 is very useful, but it's only really a saving if strings

> occur multiple times. Certainly, for strings that occur once, there's no

> advantage. And it's also bad to have unbounded memory use on the streaming

> process.

>

> So the solution we want to get to is one where a limited number of

> somewhat common strings are interned in the common vocab. However, we still

> need to map \_all\_ strings, even rare ones, to integers. We also want the

> string-to-int table to be consistent, even for rare strings.

>

> Here's the bit of code where the memory growth is occuring:

>

> https://github.com/spacy-io/spaCy/blob/master/spacy/strings.pyx#L147

>

> The purpose here is to resolve a string to an integer. We first hash the

> string. Now, this is an integer representation --- so why not just use the

> hash? The problem is we also want the inverse mapping. We therefore store

> the string, causing the memory growth.

>

> If we insist that all integers can always be mapped back to strings,

> there's no solution. We have to accept the memory growth. But if we can

> accept that these strings pass out of date, so that they're around for a

> while and then they're not, the situation should be manageable.

>

> A simple way to achieve this is to extend the StringStore so that the

> mapping is split in two. There's the main intern area, which holds a fixed

> number of strings, hopefully the common ones. But there's also a rolling

> buffer, in which strings are interned, and then later freed. This could be

> a LRU cache, or even something simpler. Efficiency is not really a problem

> here: only a small percentage of the encountered tokens will be triggering

> this logic, so we don't have to make it blazing fast, and it's easy to make

> sure we operate on contiguous buffers.

>

> A slightly more tricky solution is to do some reference counting. The idea

> here would be for the Doc object to register interest in all OOV strings

> it owns. When the ref count of an OOV string drops to 0, it's freed. This

> way, if you keep a Doc object in memory, you know that the string lookup

> will always be well behaved ‚Äî but if you're letting the Doc objects pass

> out of scope, your memory won't be growing.

>

> I think for both solutions, we should use the hash of the string as the

> integer representation for OOV strings. This means that at least the

> string-to-int mapping will stay consistent, even if strings are passing out

> of memory. The only way to have a problem here is if you hold onto the

> integer representation, release all of the documents, and later want to

> recover the string. In this situation, you'll be out of luck --- but we'll

> at least know to use an OOV symbol when you try to look up your string.

>

> ‚Äî

> You are receiving this because you are subscribed to this thread.

> Reply to this email directly, view it on GitHub

> https://github.com/spacy-io/spaCy/issues/285#issuecomment-242420597, or mute

> the thread

> https://github.com/notifications/unsubscribe-auth/AAfUutsOJ17JS8IRWp2pZNtc6BmNbExsks5qja6tgaJpZM4Ht9oH

> .

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

2016-08-28T08:56:24Z honnibal

The input text is currently not saved/represented on the document at the moment. Instead, we guarantee that the `orth` attribute faithfully retains the slice for each token, so that we just have to join the `orth` attributes and check whether the token has a trailing space.

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

2016-08-28T09:02:27Z tomtung

Yeah I understand. The point I was making was that, since the caller of the `Language` object has the full input text string anyways, it shouldn't be a big problem to deal with the slight inconvenience of having to look up the original substring of OOV tokens.

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

2016-08-28T09:37:55Z honnibal

I think we're in agreement here. However, I think it's important that we either assume the string is unavailable, or save it ourselves.

Saving the string on the document isn't a huge waste of memory, and it only impacts the API in a few places (e.g., `doc.from\_array`, deserialisation, etc). So if we want the user to be slicing into the string ever, we should probably switch to saving it.

Here's a design that achieves something like the reference counting:

- Add a `oov\_stores` member to `StringStore`, which will be a sequence of `StringStore` instances.

- Already in `Vocab.get`, we accept a `Pool` argument, that represents the allocation pool that will own the memory for the created `LexemeC` struct. This allows `Doc` objects to own their OOV lexemes. We need to extend this such that the document also owns the strings. Relevant code in `Vocab`: https://github.com/spacy-io/spaCy/blob/master/spacy/vocab.pyx#L149 (called by `Vocab.get()`, called by `Tokenizer.\_attach\_tokens()`)

- I suggest using `id(mem)` as a way of selecting the appropriate child oov store. This will allow us to have a method `get\_oov\_string(string, store\_id)` that can be called from the `Doc`, `Token` etc instances.

- We then define a `Doc.\_\_dealloc\_\_` method, which is the Cython way of adding a destructor. In `Doc.\_\_dealloc\_\_`, we tell the `StringStore` to drop the oov store associated with the `Doc` object.

- The `StringStore` remains a single source of truth for the string-to-integer mapping. When decoding an integer, we can search for it in all the OOV stores. This makes it easier to prevent integers from being "stranded".

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

2016-08-30T03:20:32Z tomtung

This sounds great! Although probably due to lack of context and familiarity to the code base, I personally would still prefer some simpler approach that can keep the `StringStore` immutable. e.g. use hashing to map OOV tokens to ints, keep reference to the text string in the `Doc` object, and obtain `orth` strings by indexing on it. Maybe this immutability can help parallelizing other parts of the pipeline too.

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

2016-08-30T08:29:36Z honnibal

Well, I think you could say "the trap is set": the existing design is such that the strings have to be globally available.

Recall that we're allowing transport to/from numpy arrays. This means we're expecting to be able to unpack an array of ints and understand some of them as strings, without ties to a particular `Doc` object. This is the mechanism being used for deserialization.

We could hack through this by writing down the OOV strings in the global store only when we pack into an array. But I hope we can all agree that this is just digging ourselves a deeper hole. I would be very unhappy if I tried to pack an array myself in the obvious way, and I found that the library's version of this was quietly writing to global state, and without this write my method failed, but only on OOV words, so not on my test data!

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

2016-08-30T19:54:20Z tomtung

> We could hack through this by writing down the OOV strings in the global store only when we pack into an array. But I hope we can all agree that this is just digging ourselves a deeper hole.

Yeah this sounds terrifying.

> Recall that we're allowing transport to/from numpy arrays. This means we're expecting to be able to unpack an array of ints and understand some of them as strings, without ties to a particular Doc object. This is the mechanism being used for deserialization.

I might be totally wrong, but I expect the feature of converting to/from numpy to only be used internally? The array doesn't seem to work across different `Language` instances if there're OOV tokens, which kind of defeats the purpose of serialization for normal users. So maybe we don't need to worry about breaking user code that uses it?

I guess what I was proposing entails always including the original text as part of (de)serialization. This might be too much refactor work, in which case what you mentioned also sounds great :)

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

2016-09-30T08:16:17Z honnibal

Implemented üéâ

Need to update other modules to reflect the change, and do testing.

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

2016-09-30T19:02:28Z honnibal

Hmm. I don't want to rush this, because it touches a lot of files, but I also don't want to block the v1.0.0 release, which is otherwise ready. So unfortunately I have to move this out of the milestone. I'll probably get back to it week after next.

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

2016-10-24T12:02:10Z honnibal

New plan ‚Äî let's at least get a good workaround in place, where the user will do some manual management of when the strings will be freed. This should be enough to keep you all productive, while we try to plan out a prettier, 'automagical' solution/wrapper around this. The freeze/flush behaviour is off by default, so it shouldn't disrupt anyone. @tomtung ‚Äî I think this is the sort of solution you were looking for, since this makes it a bit easier to control things manually.

Summary:

- New `freeze` keyword argument to `StringStore.\_\_init\_\_`

- New `.set\_frozen(bool)` method on `StringStore`, controlling whether to start handling new strings as OOV

- New `.flush\_oov()` method on `StringStore`, indicating that the current batch of OOV strings should be flushed away, and the memory freed.

Example (untested):

``` python

nlp = spacy.load('en')

nlp.vocab.strings.set\_frozen(True)

for doc in nlp.pipe(texts, batch\_size=5000, n\_threads=2):

do\_my\_stuff(doc)

nlp.vocab.strings.flush\_oov()

```

`.flush\_oov()` should be super cheap, so don't stress about trying to call it as late as possible. Call it whenever convenient.

The OOV strings are encoded using the hash of the byte string. This means that you'll get consistent integer encodings between flushings. However, if you're holding an integer ID for an OOV string, and you flush the OOVs and try to decode the integer, you'll get an `IndexError`. Hopefully, this is logical.

I've pushed the solution to the branch `issue285`. Since the patch is fully backwards compatible, I should be able to push it to PyPi later today ‚Äî I just wanted to make sure everything is looking okay, and get some feedback.

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

2016-10-29T01:49:33Z alldefector

> However, if you're holding an integer ID for an OOV string, and you flush the OOVs and try to decode the integer, you'll get an IndexError.

If one serializes a Doc with an OOV word, the above is bound to happen. Since serialization is the only way to reuse parsing results in a data pipeline and most real-world docs would have OOV words, this problem is pretty critical.

When serializing a Doc (with `to\_bytes`), would it make sense to include the relevant OOV entries? That way, we can deserialize a Doc with only the standard vocab.

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

2016-10-29T02:33:00Z alldefector

I'd be happy to try to take a crack at it, but things like the use of `HuffmanCodec` in `Packer` make it pretty involved... (actually, given a doc, are we guaranteed to get the same packing result if the vocab grows?)

Perhaps offer a way for self-contained serialization that doesn't depend on any vocab altogether? (Or only depending on a small set of symbols that are future proof)

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

2016-10-29T02:35:28Z honnibal

The serialiser backs off to a character codec for OOV words.

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

2016-10-29T02:40:20Z honnibal

Incidentally I have regrets about the serialiser. I think I got carried away...

I don't even remember how much bigger a `(text, numpy\_array)` tuple would be. Does anyone want to run a benchmark?

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

2016-11-22T23:01:18Z ELind77

I'm so glad that this has received so much thought and attention!

@honnibal could we get an update on the current status of this and your thoughts on how best to proceed?

Your suggestion of splitting the string store seems most in line with my thoughts on this. If that is still the way you are thinking of going with this and you're thinking of using multiple string stores for OOV words, I'd also just like to put out there that it might be a good idea to use some kind of data structure for storing them other than an array, especially if there are a lot of them. If the integer ids of the multiple StringStores are guaranteed to never overlap a BST might be a good candidate, if they can overlap though you might need to go with something a bit different, like a UnionFind.

-- Eric

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

2017-06-16T21:25:06Z rulai-huajunzeng

#589 issue still exists. So the workaround doesn't really work. This is one of the blocking issues for us now. Will a more stable fix be available in next 1.x releases?

Thanks a lot for the work!

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

2017-07-07T04:31:15Z ELind77

Hey,

I just took a look at the StringStore class in main and saw that some work has been done on this. I still need to play with is a bit to see how it works but this looks really great. https://github.com/explosion/spaCy/commits/master/spacy/strings.pyx

Thank you so much @honnibal !

-- Eric

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

2017-08-11T12:04:04Z azar923

Hi @honnibal

First of all, thank you for this great tool, we use it as part of NLP in our product. However, our case is very high-load system with streaming data (hundreds of thousands emails per day). And we are experiencing the same problem as was discussed here - growth of StringStore causes tremendous memory growth over time, so it really blocks usage of spaCy without fear of crashing the whole system because of OOM. The only workaround we came up with is to reload nlp object each N processed content items and force garbage collector to free memory of deleted object. However, it seem not always working way - sometimes it frees all the memory, and sometimes not. So my questions are as follows:

1) Is it planned to deal with this issue somehow? From what I see, in version 2.0 the problem still exists.

2) If it is such a fundamental way how spaCy works, maybe, there are some more clever workarounds to prevent such memory leaks?

Thanks in advance.

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

2017-08-11T13:17:16Z honnibal

@azar923 Did you try the `set\_frozen(True)` mitigation above?

The situation around this is much improved in spaCy 2, because the string-to-integer mapping no longer depends on the `StringStore` state --- it's just a hash value. This makes everything much easier. First, the `StringStore` is smaller per unit, but more importantly, if you're streaming documents through, we can restore the original string store every N documents without causing any problems.

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

2017-08-11T14:36:21Z azar923

@honnibal thank you for quick answer,

Yes, I tried set\_frozen(True) but experienced the same issue https://github.com/explosion/spaCy/issues/589

I use 1.6 now, did not try older versions yet because of some performance degradation in one-thread mode, which is critical for us now.

"we can restore the original string store every N documents without causing any problems" - sorry, did not catch that, how is it restored every N documents?

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

2017-08-11T15:38:40Z honnibal

The restoration idea would look like this:

```python

backup\_strings\_data = nlp.vocab.strings.to\_bytes()

backup\_strings = StringStore().from\_bytes(backup\_strings\_data)

for i, doc in enumerate(nlp.pipe):

yield doc

for word in doc:

backup\_strings.add(word.text)

if i % 1000 == 999:

nlp.vocab.strings = backup\_strings

backup\_strings = StringStore().from\_bytes(backup\_strings\_data)

```

This would ensure that strings stay available from only the last 1000 documents. It works by keeping two copies of the `StringStore`: the active one, and the backup. The backup tracks the active store for 1000 documents, and then takes over. We then start a new backup from the original strings data, which adds entries for the next 1000 documents, so that when it takes over, those recent documents' strings will be available.

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

2017-08-11T16:42:43Z azar923

@honnibal Great, thanks very much for these improvements. Will look forward to 2.0 release to try. For now workaround with reloading / collecting nlp object works quite ok in production.

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

2017-08-31T02:36:30Z vmandke

@honnibal I'm also facing the same issue, (spacy 1.5.0). I know this is hackish, however, would resetting the \_map and setting size to 0, or resetting the StringStore itself after a certain critical size is reached could cause any problems? Currently using spacy to get the POS tags. (from sentence subtree etc)

Is there a way to reset the StringStore without reloading the model again ? The workaround of using set\_frozen does not work.

Ref: [v1.5.0\_source](https://github.com/explosion/spaCy/blob/v1.5.0/spacy/strings.pyx)

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

2017-09-04T09:58:50Z oroszgy

I am experimenting with this workaround with the 1.x version. So far it is working well.

```python

from concurrent.futures import ThreadPoolExecutor

class RestartingEnglish:

RESTART\_CALLS = 1000000

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

self.nr\_calls = 0

self.\_init\_args = args

self.\_init\_kwargs = kwargs

self.nlp = self.\_init\_nlp()

self.fut\_nlp = None

self.exec = ThreadPoolExecutor(max\_workers=1)

def \_init\_nlp(self):

return English(\*self.\_init\_args, \*\*self.\_init\_kwargs)

def \_restart\_nlp(self):

self.nr\_calls += 1

if self.nr\_calls >= self.RESTART\_CALLS:

if self.fut\_nlp == None:

print("Getting new NLP", self.nr\_calls)

self.fut\_nlp = self.exec.submit(self.\_init\_nlp)

else:

if self.fut\_nlp.done():

print("Got new NLP", self.nr\_calls)

self.nlp = self.fut\_nlp.result()

self.fut\_nlp = None

self.nr\_calls = 0

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

doc = self.nlp(\*args, \*\*kwargs)

self.\_restart\_nlp()

return doc

```

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

2017-10-16T18:20:36Z honnibal

Fixed! (!!!!)

:tada: :tada: :tada:

Please see https://github.com/explosion/spaCy/pull/1424

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

2018-05-08T14:27:37Z 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.

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

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