sng Documentation

Release 0.3

Alexander Engelhardt

Dec 05, 2018

CONTENTS:

The `sng` package (short for "Startup Name Generator") is hosted on GitHub. This is its documentation.

CHAPTER

ONE

INTRODUCTION

1.1 Summary

This package takes a wordlist, then trains a model that can automatically generate name suggestions for things like companies or software. You feed it a text corpus with a certain theme, e.g. a Celtic text, and it then outputs similar sounding suggestions. An example call to a trained model looks like this:

```
>>> cfg = sng.Config(suffix=' Labs')
>>> gen = sng.Generator(wordlist_file='my_wordlist.txt')
>>> gen.fit()
>>> gen.simulate(n=4)

['Ercos Software', 'Riuri Software', 'Palia Software',
 'Critim Software']
```

The package source is available on GitHub.

I also gave a lightning talk presenting the basic idea, it's available on Youtube.

1.2 Supplied wordlists

The package comes with sample word lists in German, English, and French, and more "exotic" corpora of Pokemon names, death metal song lyrics, and J.R.R. Tolkien's Black Speech, the language of Mordor. Below, I'll briefly describe them and also show some randomly sampled output for the (totally not cherry-picked) generated words. These corpora are available in the wordlists subdirectory.

- German. The first chapter of Robinson Crusoe in German
- English. Alice in Wonderland
- Greek A short list of Greek words (in the latin alphabet)
- Gallic (source 1): A list of Gallic words. (source 2): Selected Gallic song lyrics from the band Eluveitie
- Latin: The first book of Ovid's Metamorphoses
- French: The French Wikipedia entry for France
- Behemoth: English song lyrics by the death metal band Behemoth. Sampled words will have be more occult themed
- The Black Speech: JRR Tolkien's language of the Orcs
- Lorem Ipsum: The classic lorem ipsum text
- Pokemon: A list of 900 Pokemon. Your company will sound like one of them, then!

1.2.1 Celtic

My main target was a Celtic sounding name. Therefore, I first created a corpus of two parts (browse it here): first, a Gallic dictionary, and second, selected song lyrics by the swiss band Eluveitie. They write in the Gaulish language, which reads very pleasantly and makes for good startup names in my opinion:

```
Lucia
Reuoriosi
Iacca
Helvetia
Eburo
Ectros
Uxopeilos
Etacos
Neuniamins
Nhellos
```

1.2.2 Pokemon

I also wanted to feed the model a list of all Pokemon, and then generate a list of new Pokemon-themed names:

```
Grubbin
Agsharon
Oricorina
Erskeur
Electrode
Ervivare
Unfeon
Whinx
Onterdas
Cagbanitl
```

1.2.3 Tolkien's Black Speech

J.R.R. Tolkien's Black Speech, the language of the Orcs, was a just-for-fun experiment (wordlist here). It would be too outlandish for a company name, but nonetheless an interesting sounding corpus:

```
Aratani
Arau
Ushtarak
Ishi
Kakok
Ulig
Ruga
Arau
Lakan
Udaneg
```

1.2.4 Death metal lyrics

As a metal fan, I also wanted to see what happens if the training data becomes song lyrics. I used lyrics by the Polish death metal band Behemoth, because the songs are filled with occult-sounding words (see the wordlist):

```
Artered
Unlieling
Undewfions
Archon
Unleash
Architer
Archaror
Lament
Unionih
Lacerate
```

You can add anything from "Enterprises" to "Labs" as a suffix to your company name. I found a long list of possible suffixes here.

1.3 Background

1.3.1 My need for automatic company names

Recently, an associate and I started work on founding a software development company. The one thing we struggled most with was to come up with a good name. It has to sound good, be memorable, and the domain should still be available. Both of us like certain themes, e.g. words from Celtic languages. Sadly, most actual celtic words were already in use. We'd come up with a nice name every one or two days, only to find out that there's an HR company and a ski model with that exact name.

We needed a larger number of candidate names, and manual selection took too long. I came up with an idea for a solution: Create a neural network and have it generate new, artificial words that hopefully are not yet in use by other companies. You'd feed it a corpus of sample words in a certain style you like. For example, Celtic songs, or a Greek dictionary, or even a list of Pokemon. If you train the model on the character-level text, it should pick up the peculiarities of the text (the "language") and then be able to sample new similar sounding words.

A famous blog post by Andrej Karpathy provided me with the necessary knowledge and the confidence that this is a realistic idea. In his post, he uses recurrent neural networks (RNNs) to generate Shakespeare text, Wikipedia articles, and (sadly, non-functioning) source code. Thus, my goal of generating single words should not be a big problem.

CHAPTER

TWO

INSTALLATION

2.1 Install from GitHub

Clone the repository and install the package:

```
git clone https://github.com/AlexEngelhardt/startup-name-generator.git
cd startup-name-generator
python setup.py install
```

I think this also works:

```
pip install --upgrade git+git://github.com/AlexEngelhardt/startup-name-generator.git
```

2.2 Install from PyPI

Just issue:

```
pip install sng
```

I am still working on making this package available on PyPI though.

CHAPTER

THREE

QUICKSTART

```
In [1]: %load_ext autoreload
        %autoreload 2

In [2]: # While the sng package is not installed, add the package's path
        # (the parent directory) to the library path:

        import os
        import sys
        sys.path.insert(0, os.path.abspath('../../'))

In [3]: import sng

Using TensorFlow backend.
```

3.1 Prepare and train the model

Create a Config object to set your own preferences regarding training or simulation:

```
In [4]: cfg = sng.Config(
            epochs=50
        )
        cfg.to_dict()

Out[4]: {'batch_size': 64,
         'debug': True,
         'epochs': 50,
         'hidden_dim': 50,
         'max_word_len': 12,
         'min_word_len': 4,
         'n_layers': 2,
         'suffix': '',
         'temperature': 1.0,
         'verbose': True}
```

Choose from one of these builtin wordlists to get started quickly:

```
In [5]: sng.show_builtin_wordlists()

Out[5]: ['gallic.txt',
         'english.txt',
         'behemoth.txt',
         'lorem-ipsum.txt',
         'greek.txt',
         'black-speech.txt',
         'german.txt',
         'french.txt',
```

```
         'latin.txt',
         'pokemon.txt']
```

We'll load the latin wordlist and look at a few sample words:

```
In [6]: latin = sng.load_builtin_wordlist('latin.txt')

In [7]: latin[:5]

Out[7]: ['in', 'nova', 'fert', 'animus', 'mutatas']
```

Initialize and fit the word generator:

```
In [8]: gen = sng.Generator(wordlist=latin, config=cfg)

2973 words

24 characters, including the \n:
['\n', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',

First two sample words:
['detque\n', 'concordia\n']

In [9]: gen.fit()

epoch 0 words: Rcqeslrot, Hqvqiie, Ntyfuamiie, Fvgseafueiaa, loss: 1.5849
epoch 10 words: Uutsque, Duluas, Epeenunt, Omanetas, loss: 1.2277
epoch 20 words: Eibosque, Iuritas, Xoncut, Omniditus, loss: 1.1217
epoch 30 words: Lirus, Timone, Lilosidum, Oggo, loss: 1.0706
epoch 40 words: Unga, Oricuva, Umnaras, Untit, loss: 1.029

In [10]: gen.simulate(n=4)

Out[10]: ['Baleve', 'Remiduitum', 'Urbam', 'Ugnue']

In [11]: gen.config.suffix = ' Software'

In [12]: gen.simulate(n=4)

Out[12]: ['Otimanae Software', 'Repte Software', 'Redeps Software', 'Urque Software']
```

3.2 Save and load the model for later

```
In [14]: gen.save('my_model', overwrite=True)
```

Then:

```
In [15]: gen2 = sng.Generator.load('my_model')

2973 words

24 characters, including the \n:
['\n', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',

First two sample words:
['demittere\n', 'sonanti\n']

In [16]: gen2.simulate(n=4)

Out[16]: ['Redent Software', 'Ulde Software', 'Uxsit Software', 'Ortitum Software']
```

CHAPTER

FOUR

DISCUSSION

4.1 Data preprocessing

For input data, I just built a corpus by using raw, copy-pasted text that sometimes included numbers and other symbols. A preprocessing was definitely necessary. I first stripped out all non-letter characters (keeping language-specific letters such as German umlauts). Then, I split the text up in words and reduced the corpus to keep only unique words, i.e. one copy of each word. I figured this step was reasonable since I did not want the model to learn the most common words, but instead to get an understanding of the entire corpus' structure.

After this, most text corpora ended up as a list of 1000 to 2000 words.

4.2 The RNN architecture

The question which type of neural network to use was easily answered. Recurrent neural networks can model language particularly well, and were the appropriate type for this task of word generation.

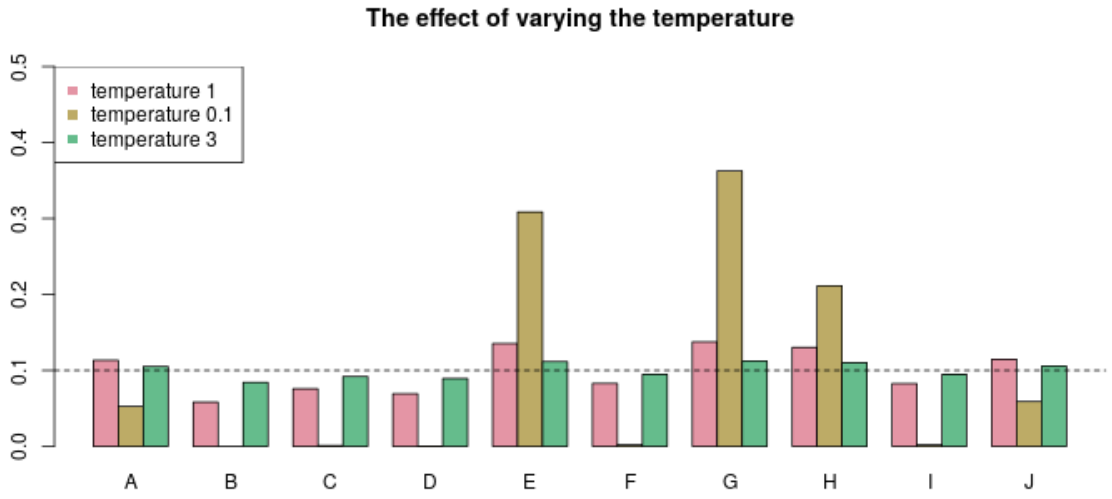
However, to my knowledge, finding the 'perfect' RNN architecture is still somewhat of a black art. Questions like how many layers, how many units, and how many epochs have no definite answer, but rely on experience, intuition, and sometimes just brute force.

I wanted a model that was as complex as necessary, but as simple as possible. This would save training time. After some experiments, I settled for a two-layer LSTM 50 units each, training it for 500 epochs and a batch size of 64 words. The words this model outputs sound good enough that I didn't put any more energy in fine-tuning the architecture.

4.3 Sampling Temperature

The RNN generates a new name character by character. In particular, at any given step, it does not just output a character, but the distribution for the next character. This allows us to pick the letter with the highest probability, or sample from the provided distribution.

A nice touch I found is to vary the temperature of the sampling procedure. The temperature is a parameter that adapts the weights to sample from. The "standard" temperature 1 does not change the weights. For a low temperature, trending towards zero, the sampling becomes less random and the letter corresponding to the maximum weight is chosen almost always. The other extreme, a large temperature trending towards infinity, will adjust the weights to a uniform distribution, representing total randomness. You can lower the temperature to get more conservative samples, or raise it to generate more random words. For actual text sampling, a temperature below 1 might be appropriate, but since I wanted new words, a higher temperature seemed better.



In the image above, imagine we want to sample one letter from A, B, ..., J. Your RNN might output the weights represented by the red bars. You'd slightly favor A, E, G, H, and J there. Now if you transform these weights with a very cold temperature (see the yellow-ish bars), your model gets more conservative, sticking to the argmax letter(s). In this case, you'd most likely get one letter of E, G, and H. If you lower the temperature even further, your sampling will always return the argmax letter, in this case, a G.

Alternatively, you can raise the temperature. In the image above, I plotted green bars, representing a transformation applied with a temperature of 3. You can still see the same preferences for E, G, and H, but the magnitude of the differences is much lower now, resulting in a more random sampling, and consecutively, in more random names. The extreme choice of a temperature approaching infinity would result in a totally random sampling, which then would make all your RNN training useless, of course. There is a sweet spot for the temperature somewhere, which you have to discover by trial-and-error.

CHAPTER

FIVE

MODULES

5.1 Config

The Config module. It defines the Config class.

class sng.Config.**Config**(***kwargs*)

Configuration options for model training and name generation

****kwargs:** Keyword arguments that will overwrite the default config options.

To create a Config object that results in simulating names between 6 and 10 letters:

```
cfg = sng.Config(
    min_word_len=6,
    max_word_len=10
)
```

To quickly inspect all values:

```
cfg.to_dict()
```

batch_size = None

int: The batch size for training the RNN

debug = None

bool: If true, methods will add some additional attributes to a Generator object's `debug` dict.

epochs = None

int: How many epochs to train the RNN for?

hidden_dim = None

int: Number of hidden units per LSTM layer

max_word_len = None

int: How long should simulated words be maximum?

min_word_len = None

int: How long should simulated words be at least?

n_layers = None

int: How many LSTM layers in the model?

suffix = None

str: A suffix to append to the suggested names.

Choose e.g. " Software" (with a leading space!) to see how your company name would look with the word Software at the end.

temperature = None
float: Sampling temperature. Lower values are "colder", i.e. sampling probabilities will be more conservative.

to_dict()
Convert Config object to dictionary.

verbose = None
bool: If true, prints helpful messages on what is happening.

5.2 Generator

class sng.Generator.**Generator**(*config=<sng.Config.Config object>*, *wordlist_file=None*, *wordlist=None*)
Main class that holds the config, wordlist, and the trained model.

config [sng.Config, optional] A Config instance specifying training and simulation parameters. If not supplied, a default configuration will be created.

wordlist_file [str] Path to a textfile holding the text corpus you want to use.

wordlist [list of strings] Alternatively to `wordlist_file`, you can provide the already processed wordlist, a list of (ideally unique) strings.

config [sng.Config] The Config object supplied, or a default object if none was supplied at initialization.

wordlist [list of strings] A processed list of unique words, each ending in a newline. This is the input to the neural network.

You can create a word generator like this:

```
import sng
cfg = sng.Config()

# Folder for pre-installed wordlists:
wordlist_folder = os.path.join(
    os.path.dirname(os.path.abspath(sng.__file__)), 'wordlists')
sample_wordlist = os.path.join(wordlist_folder, 'latin.txt')

# Create a Generator object with some wordlist:
gen = sng.Generator(wordlist_file=sample_wordlist, config=cfg)

# Train the model:
gen.fit()

# Get a few name suggestions:
gen.simulate(n=5)
```

fit()
Fit the model. Adds the 'model' attribute to itself.

classmethod load(*directory*)
Create a Generator object from a stored folder.

directory [str] Folder where you used Generator.save() to store the contents in.

save(*directory*, *overwrite=False*)
Save the model into a folder.

directory [str] The folder to store the generator in. Should be non-existing.

overwrite [bool] If True, the folder contents will be overwritten if it already exists. Not recommended, though.

simulate(*n=10*, *temperature=None*, *min_word_len=None*, *max_word_len=None*)
Use the trained model to simulate a few name suggestions.

n [int] The number of name suggestions to simulate

temperature [float or None] Sampling temperature. Lower values are "colder", i.e. sampling probabilities will be more conservative. If None, will use the value specified in self.config.

min_word_len [int or None] Minimum word length of the simulated names. If None, will use the value specified in self.config.

max_word_len [int or None] Maximum word length of the simulated names. If None, will use the value specified in self.config.

5.3 Wordlists

sng.wordlists.wordlists.**load_builtin_wordlist**(*name*)
Load and process one of the wordlists that ship with the sng package.

name [str] A file name of one of the files in the wordlists/ directory. Call *show_builtin_wordlists()* to see a list of available names. Choose one of these.

A wordlist. Literally, a list of words in the text corpus. It's not yet preprocessed, so there are still duplicates etc. in there. This is taken care of by *sng.Generator*'s __init__ method.

sng.wordlists.wordlists.**show_builtin_wordlists**()
Returns a list of all builtin wordlists' filenames.

Use one of them as an argument to *load_builtin_wordlist()* and get back a ready-to-go wordlist.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX