Project Work for NLP
Poetry Generation using the Divine Comedy

Francesco Farinola

January 2022

1 Introduction

In this project work, we are going to exploit and test different techniques to generate text using as corpus the Divine Comedy, and subsequently we will evaluate them.

The automatic generation of poetry involves several levels of language like phonetics, lexical choice, syntax and semantics and usually demands a considerable amount of input knowledge. Although, the Divine Comedy is composed of only 14,233 lines that are divided into three cantiche (singular cantica) – Inferno (Hell), Purgatorio (Purgatory), and Paradiso (Paradise) – each consisting of 33 cantos, we may assume that the results we are going to obtain will not be enough consistent and satisfactory.

2 Text cleaning

After loading the corpus of the Divine Comedy, some generic operations to clean the text are made in order to process it correctly.

Since spacing in the loaded corpus may be misleading, there are been defined different regex functions to correct the spacing such as:

- `clean_start`: which replaces the two starting whitespaces with one only.
- `remove_double_whitespaces`: which replaces two generic whitespaces with only one.
- `clean_newline`: which replaces the newline symbol with the same newline symbol but with correct spacing, namely `"\n"` with `" \n "`.
- `change_apostrophe`: which changes the kind of apostrophe in order to make a correct tokenization later.

Other regex functions are used to clean the text by uncommon symbols and punctation:

- `remove_punctuation`: to remove non-alphabetic symbols such as parenthesis, hyphens, dashes, punctuation marks, dots, double dots, etc.
- `replace_uncommon_symbols`: since modern Italian language is different from the old one and generating text with particular accents may be misleading, characters with particular accents are replaced with the same one without the accent according to this table:

Previous char	ä	é	ë	Ë	ï	Ï	ó	ö	ü
Replaced char	a	è	è	E	i	I	ò	o	u

3 Markov Chains text generation

Markov chains are simple stochastic models with a finite set of states that transition from one to the next. These sets of state transitions are determined by a probability distribution. In contrast, in a system, the probability distribution of the next state may be affected by the system's past history. Because of this simplifying assumption, Markov chains can be easily applied in a variety of domains, including text generation. Essentially, every word in our corpus is 'connected' to every other word with varying probabilities using Markov Chains.

This model can be easily implemented thanks to the `markovify` Python library. We simply initialize the model with the function:

`markovify.NewlineText(corpus)` and passing as argument the whole corpus.

Then we may generate a sentence calling the model function:

`model.make_short_sentence(number_of_chars)` and passing as argument the number of characters to generate.

Example. *Generation of 3 triplets with 50 characters each.*

Se cagion altra al mio veder si brami
parea del loco mio
fin che la perse
come virtute e canoscenza
vicino al fin d'esto sentiero
E a tal da cui si legge che l'angelica natura
naturalmente fu sì cortese
Nasce per quello a parte
Noi ricidemmo il cerchio di Giuda

4 Char-level text generation

A Language Model can be trained to generate text character-by-character. In this case, each of the input and output tokens is a character. Moreover, Language Model outputs a probability distribution over the character set.

Since we want to reduce computational and memory costs, and make the whole process easy to be managed, we will **lower all the capital letters** in order to halve the number of output classes. The number of output classes (`VOCAB_SIZE`) is equal to 35, which is relatively low.

Also, in order to fit the corpus data in the `Tensorflow` model we are going to build, we will use specific dictionaries (`chars2idx, idx2chars`) that map each character to an integer (and viceversa) and **use an encoded input**.

4.1 Input pipeline preparation

After encoding the input into integers, we prepare the data to be fed into the `Tensorflow` model by creating a `tf.data.Dataset` object in order to combat bottlenecks or latency and eliminate errors.

To do so, we will apply different functions in sequence:

- `from_tensor_slices`: creates a Dataset whose elements are slices of the given tensors. The given tensors are sliced along their first dimension. This operation preserves the structure of the input tensors, removing the first dimension of each tensor and using it as the dataset dimension;
- Apply `batch(SEQUENCE_LENGTH+1, drop_remainder=True)` since we have a sequence of integers representing characters which is much longer so it is much helpful if we divide this into many sequences of `SEQUENCE_LENGTH+1`. `drop_remainder=True` whether the last batch should be dropped in the case it has fewer than `batch_size` elements;
- `map` function: for each sequence created before with `from_tensor_slices` applies the function `split_input_target` which returns the input and output sequences for the model. i.e. We have a sequence *"Nel mezzo del"*, respectively the input text will be *"Nel mezzo de"* and the output will be *"el mezzo del"*
- `shuffle`: namely shuffles the sequences
- `batch(BATCH_SIZE, drop_remainder=True)` namely creates batches of length `BATCH_SIZE` with sequences shuffled.

The size used for `SEQUENCE_LENGTH` is equal to 100, while using a `BATCH_SIZE` of 128.

The output of this pipeline would be namely but encoded with integers:

Input sequence: *"nel mezzo del cammin di nostra vita \n mi ritrovai pe"*
Output sequence: *"el mezzo del cammin di nostra vita \n mi ritrovai per "*

4.2 Model implementation and training

The model implemented is a `Sequential` keras model which consists of:

- `Embedding` layer: which takes as arguments `input_dim=VOCAB_SIZE`, `output_dim=EMBEDDING_SIZE`, `batch_input_shape=[BATCH_SIZE, None]`
- Two `GRU` subsequent layers: the first one with `UNITS * 2` cells and the second one with `UNITS` cells. Both layers have set `return_sequences=True` in order to concatenate them with the following layers. Furthermore, the two GRUs are `stateful=True`.

  This flag is used to have truncated back-propagation through time: the gradient is propagated through the hidden states of the GRU across the time dimension in the batch and then, in the next batch, the last hidden states are used as input states for the GRU. This allows the GRU to use longer context at training time while constraining the number of steps back for the gradient computation. This can be exploited in Language Models when the training set is a list of sequences and the batches are created so that each sequence in a batch is the continuation of the sequence at the same position in the previous batch. This allows having document-level when computing predictions.
- `Dense` layer: with `VOCAB_SIZE` cells in order to obtain as output the probability of the next character.

The model is compiled using the `Adam` optimizer and the `sparse_categorical_cross_entropy` with `from_logits=True` as loss function.

The parameters chosen for the model are: `VOCAB_SIZE=35`, `EMBEDDING_DIM=300`, `UNITS=500`.

The model is finally trained for 50 epochs using an early stopping with patience equal to 5.

4.3 Temperature sampling

To sample from the created model, we need to create a new inference model using the same architecture but with `BATCH_SIZE=1`, load the weights of the trained model and define the new input shape.

To perform the temperature sampling, the function `char_level_temperature_sampling` takes in input:

- the model;
- the input string, i.e. "nel mezzo del cammin di nostra vita" as the context to generate the next characters
- the number of characters to generate;
- the temperature, which defines how distant the predictions are far from the dataset (the smaller, the more similar to the input dataset)

After encoding the input string which defines the context, we reset the states of the model to make an independent call from the fit method, and then start to generate the next characters with iterative calls to the model.

Only in the first call we pass the entire input string since the model is stateful and we do not need to pass the whole sequence on recurrent calls. So as, when recurrently calling the model, we simply pass the last character ID.

5 Seq2Seq text generation

Sequence to Sequence (seq2seq) models is a special class of Recurrent Neural Network architectures that we typically use (but not restricted) to solve complex Language problems. The most common architecture used to build Seq2Seq models is Encoder-Decoder architecture, which is the one we are going to build.

5.1 Input preparation

The kind of model we are going to build requires, as input, sequences of tokens of fixed size. To do so, we perform the text tokenization with **SpaCy** using the model `it_core_news_lg`. This model includes also the pre-trained Italian word embeddings that we are going to use to train the model.

We inspect the sequences length to define a proper maximum sequence length for the model. We will set two maximum sequence lengths: one for the encoder and one for the decoder. The decoder will have 1 token more since its input will include the `<SOS>` token and the output the `<EOS>` token.

Given the following table, i decided to discard all the sequences with length below 4 and greater than 11:

Length	≤3	4	5	6	7	8	9	10	11	≥12
# lines	112	182	1046	2802	3991	3351	1784	722	254	89

By doing so, the `MAX_SEQ_LEN_ENCODER` is set to 11 while the `MAX_SEQ_LEN_DECODER` to 12 as a tradeoff in order to discard few and unnecessary long sequences that would increase the model complexity and training time.

Since we are working with the old Italian language, some terms might not be recognized and classified as out-of-vocabulary words when computing word embeddings. To avoid this, we will use the library **SymSpellPy** with with a custom Italian vocabulary.

The function `compute_embeddings` computes an embedding matrix using the pre-trained Italian word embeddings of SpaCy. If a word is not found in the SpaCy vocabulary we lookup for a similar word using SymSpellpy's lookup function with `max_edit_distance=1` and search again in the Spacy vocabulary. If we find a word embedding, we assign it to the original word assuming it would have a similar vector, otherwise we assign a random vector. i.e. "intrai" assigned the word embedding of "entrai" or "mpediva" with "impediva" assigned.

`<PAD>` token will have a zero vector while "`\n` ", `<EOS>`, `<SOS>` a random vector.

The `encode_dataset` function transforms the sequences of the dataset into encoded ones using the word2idx dictionary.

In an Encoder/Decoder model designed for text generation we have two parts which receive and produce different inputs and outputs respectively.

Assuming the special tokens `<PAD> = 0`, `<SOS> = 1`, `<EOS> = 2`.

The Encoder has the task to produce a hidden representation of the sentence: so, it will receive as input the encoded sentence padded without any additional token (i.e. sentence to be fed = "Nel mezzo del cammin di nostra vita", input to be fed to the encoder = [3 4 5 6 7 8 9 0 0 0 0]) and produce through an RNN the states that will give a better context representation and will be passed to the decoder.

The Decoder instead will be fed with inputs and outputs:

- the input is in the form: `<SOS>` + encoded sentence = [1 3 4 5 6 7 8 9 0 0 0 0]
- the output is in the form: encoded sentence + `<EOS>` = [3 4 5 6 7 8 9 2 0 0 0 0]

`preprocess_encoder_input` encodes the input of the encoder as previously said, so we simply pad the sequence.

`preprocess_decoder_input` produces the decoder input and output are previously said.

5.2 Encoder/Decoder implementation and training

We define the classes for Encoder and Decoder:

- **Encoder**: after passing the input to the Embedding layer, we go through two stacked GRUs and pass the final state to the Decoder.
- **Decoder**: is made of two stacked GRUs whose states are initialized with the output states of the Encoder final layer. The final layer of the decoder is a Dense Layer with softmax function to get the categorical probabilities for each token to be the next one. Along with the probabilities, also the final state of the final GRU layer is returned, which will be used as context in the text generation phase.

The Embedding layer has set `trainable=True` to improve the word embeddings during training since the dataset is not so large.

NB: We cannot use Bidirectional GRU in the Decoder part since we should not know anything about the next tokens and we are predicting the probabilities of the next tokens we can work only in one direction.

All the GRUs layers have set `return_sequences=True` in order to stack them and also `return_state=True` to remember the context, so to not make independent calls on layers. Furthermore, a dropout of 0.2 is applied to each of the GRU layers.

The parameters of the model are: `UNITS=500`, `EMBEDDING_DIM=300`, `VOCAB_SIZE=13532`.

The model is compiled using the `Nadam` optimizer and sparse categorical cross entropy as loss function. Finally the model is trained for 15 epochs using a `BATCH_SIZE` of 32.

5.3 Sampling methods tested

The generic procedure for sampling using a Seq2Seq model consists of different phases. First, an input string defined by us is fed to the encoder to get a context. Then, we first call the Decoder using the output state of the Encoder and the start of sequence token `<SOS>` only, choose the next sampled token according to any of the sampling techniques and iteratively call the Decoder with the last sampled token. When the desired length of sequence is reached or when a end of sequence token `<EOS>` is sampled we stop the generation and proceed to the next sequence. The different sampling methods tested are:

- **Greedy search:** the token with the maximum probability to be the next word is chosen.
- **Temperature sampling:** probabilities are reweighted to a certain "temperature", apply the softmax function and sample at random the next token according to the reweighted distribution.
- **Top-k sampling:** the K most likely next words are filtered and the probability mass is redistributed among only those K next words.
- **Beam Search:** Beam search reduces the risk of missing hidden high probability token sequences by keeping the most likely num_beams of hypotheses at each time step and eventually choosing the hypothesis that has the overall highest scores. In this case, scores are computed as $1 - log(probability)$

6 Evaluation and results

The samples obtained by the different methods used were evaluated by using two different metrics: BLEU score and METEOR score, both provided by NLTK.

BLEU score is the most popular metric for NLG tasks and is a precision focused metric that calculates n-gram overlap of the reference and generated texts. METEOR, instead, works on word alignments. It computes one to one mapping of words in generated and reference texts taking into account also synonyms/mispelled words using WordNet or porter stemmer. Finally, it computes an F-score based on these mappings.

To calculate both scores for the whole generated text, for each sentence generated by the models we calculate the score w.r.t. each row in the Divine Comedy corpus and then average them.

Results

All the texts generated had as input seed: "Nel mezzo del cammin di nostra vita"

6.1 Char-level temperature sampling

400 characters with temperature=0.7

mi piacea per mille giaci
o diva quanto pon mente a la spiga
ch'ogn' erba si conosce per lo seme d'alta terra già col piè morrocco
io era già da quell' ombre partito
e seguitava l'orme d'i sante
qual sovra 'l ventre e qual sovra le spalle
omai sarebbe li suoi regi ancora
nati per me de l'etterno consiglio
cade vertù ne l'acqua e ne la pianta

BLEU score: 0.6057; **METEOR score:** 0.7250

6.2 Seq2Seq greedy search

dinanzi amendue cortese cortese quanto caggia
dinanzi amendue cortese meco riguardando
dinanzi amendue cortese meco riguardando
dinanzi amendue cortese meco riguardando
dinanzi amendue cortese meco riguardando

BLEU score: 0.0658 **METEOR score:** 0.1466

6.3 Seq2Seq temperature sampling

with temperature=1

dinanzi amendue primavera dove dovessi avvisar mia conoscenza
seco tornerai maraviglia anzi senta ma cara fidanza
seco simiglianza mo centauro dimostrato questo de ferza
maraviglia operare differente ladroneccio inveggiar che risponde
maraviglia seguitando Da tema ria cetra avria così

BLEU score: 0.0819 **METEOR score:** 0.2122

6.4 Seq2Seq top-k sampling

with k=10

giuso Miserere rinova Quando tu Però deduca li
dinanzi Miserere Magno anzi
rimirando rimirando Miserere e Bèatrice virtute
piangendo Miserere Miserere quando Bèatrice Bèatrice quando disii
gridando Allor tacendo tacendo tacendo indugio Non facci

BLEU score: 0.0895 **METEOR score:** 0.2579

6.5 Seq2Seq beam search

with `num_beams=2` (increasing it to 3 would increase exponentially the computational time) and `num_words=6`

pianto appresso amendue disio giuso dinanzi
maraviglia animal maraviglia animal maraviglia animal
popol popol maraviglia animal maraviglia animal
maraviglia animal maraviglia animal maraviglia animal
popol popol maraviglia animal maraviglia animal

BLEU score: 0.0494 **METEOR score:** 0.1241

7 Discussion

From the results obtained, we can see that the strategy working better is the char-level one.

About the Seq2Seq model, several attempts have been made to improve the results: GRU layers have been substituted with LSTMs and also, stateless layers have been tried. Also, the Encoder structure was changed to a Bidirectional LSTM to see if the internal representation of the context was a fault related to the generation phase. Differents parameters for the training of the model have been tried. This includes changing the dropout, increasing the number of cells of GRU/LSTM layers, changing batch size.

Word embeddings were also calculated using a random strategy (setting all the vectors with random values and training them with the model), or by disabling the training of word vectors in the Embedding layer.

Finally, i tried also to add Bahdanau and Luong Attention (AdditiveAttention and Attention keras layers) and added normalized term frequency as input to the encoder and decoder (both and individually - 3 tries) to see if unfrequent terms are handled better.

None of this attempts really improved significantly the results obtained by the Seq2Seq model, given that using the char-level model achieves really good ones.

We can assume that working on word-level for this kind of task might be a really difficult task and that it could be a better choice to focus on char-based approaches by applying for example some constraints inside the training loop, improving char embeddings or build a more complex architecture.