Automatic Generation of Rap Lyrics

Sotirios Lamprinidis
plf240

Supervisor: Manex Aguirrezabal Zabaleta

A thesis presented to the Faculty of Humanities
in partial fulfilment of the requirements for the degree of
Master of Science in IT & Cognition

University of Copenhagen
Denmark
31 July 2018

Με πληρώνει η ρίμα, έτσι ζω
Οι φίλοι μου, μου λένε πως δεν είναι σωστό
Νομίζουν ότι κάνω ραπ μόνο για να πληρωθώ
Ρίμα για χρήμα, ραπάρω μόνο για να πληρωθώ

Παιδί Θαύμα, 1998

Automatic Generation of Rap Lyrics

Sotirios Lamprinidis

Abstract

In the present dissertation we examine how we can automatically generate creative content, in particular rap lyrics. We show that while having a lot in common with poetry generation, rap lyrics pose their own challenges to researchers, stemming mostly from their free, unconstrained and diverse form. To address the problem, we employ traditional natural language processing techniques, such as n-gram models, but also look into recent advances in the field of deep learning and neural networks. We assemble an array of different model and heuristics and evaluate using both quantitative and qualitative methods.

Contents

Chapter 1

Introduction

1.1 Motivation

Poetry, using ordinary language to produce an aesthetically pleasing form of literature by taking advantage of rythmic and auditory qualities, has emerged alongside the first civilizations (e.g. the Egyptian epic love poems), if not predated them. Let's introduce an example, one of the oldest known English nursery rhymes, *pat-a-cake*:

> Pat-a-cake, pat-a-cake baker's man
>
> Bake me a cake as fast as you can
>
> Pat it and prick it and mark it with 'B'
>
> Put it in the oven for baby and me

This delightful little children's poem utilizes the linguistic device known as *rhyme* to create a pleasant rhythm matching the endings of every two lines, marked with different colors above. In fact, a *rhyme* in English is also synonym with a *"A short poem in which the sound of the word or syllable at the end of each line corresponds with that at the end of another"* [2] like that nursery rhyme.

A rhyme can be defined as a recurrence of identical or related sounds matching two or more words. Identical sounding rhymes, or more specifically when two words match perfectly starting from the last stressed vowel to the coda (last consonant) are also known as **pure** or **perfect** rhymes and can be further classified by syllable count into single, double, triple, quadruple and so forth. In our example, we have the perfect rhymes *man* ∼ *can*, *'B'* ∼ *me* and *bake* ∼ *cake* (single pure rhymes). Dale (1998) did extensive work to describe all the possible kinds of rhyme appearing on english poetry works. A couple more complex examples are *baby* ∼ *and me* and *pat it* ∼ *mark it*, which are known as **assonance** rhymes, meaning it is only the vowels that match. There is also an example of an **uneven rhyme**, where the two words form a near-perfect rhyme but have different number of syllables or different stress position (*cake* ∼ *baker*). Sometimes a single word is involved into different rhyme schemes, for example if we read the poem in the following way:

> Pat-a-cake, pat-a-cake baker's man
>
> Bake me a cake as fast as you can

[2] https://en.oxforddictionaries.com/definition/rhyme, Last accessed: 20–07–2018

Pat it and prick it and mark it with "B"

Put it in the oven for baby and me

pat also forms an assonance rhyme with *man*, can and *fast*, and we can further see what Dale identifies as a **pararhyme**, when all the consonants have similar sounds as in *pat it* ∼ *put it*, and a simple **consonance** rhyme, where some of the consonants match as in *prick it* ∼ *mark it*.

Despite being one of the most simple poems one can find, one can see the complex way phonemes and sounds interact to create pleasing rythms. But apart from the rhyming, the poem gives us insight into 18th century British life, where *mark it with a "B"* refers to the common practice of marking the bread and sending it to the baker to bake it, at a time where it was a luxury for a family to own an oven. Working-class practices and feelings, this time more dramatic and with a much darker tint, come to life using rhyming in *The Chimney Sweeper: When my mother died I was very young* by William Blake:

When my mother died I was very young,

And my father sold me while yet my tongue

Could scarcely cry "'weep! 'weep! 'weep! 'weep!"

So your chimneys I sweep & in soot I sleep.

Here, another kind of rhyme is introduced, the **syllable** rhyme (*yet my* ∼ *very*) where an arbitrary number of syllables can match, whether it is only the consonants or only the vowels. Our example can so be classified as a single syllable assonance rhyme (*ye_* ∼ *ve_*).

But poetry was not always about rhyming. In fact while ancient Greek and Hebrew poets actually knew how to rhyme and did use it, it was not common (Slavitt et al., 1999; LaSor et al., 1996), and some argue that Arab influences among other played an important role in bringing rhyming to the European continent (Menocal, 2004). Arguably, the most complex rhymes available today are also related to non-Western cultures (Rose, 1994), specifically African Americans, and can be found in hip-hop music, specifically in the form of vocal delivery known as *rapping*.

Rap Lyrics

The definition of rap in the Oxford dictionary is "*A type of popular music of US black origin in which words are recited rapidly and rhythmically over an instrumental backing*" [1]. Rap music has a "*strong rythm in which the words are spoken, not sung*" [2]. Of course, style and rythm plays an important role in poetry, but it is apparent that rythm and rhyming is the cornerstone of the lyrical part of rap music. This is not to say, though, that rap music lacks the expression of feeling and emotions through language, as in the case of poetic works. University of Calgary in Canada currently offers a *Rap Linguistics* course [3], and the professor Darin Flynn teaching the course stresses the importance of studying rap music as it *" it gives us a window into how people really talk in working-class environments, in both black and white communities."* (McCoy, 2014). He also stresses the craftsmanship needed to create rap lyrics in terms of grammar, sounds and metaphors

[1] https://en.oxforddictionaries.com/definition/rap Last accessed on 20–07–2018

[2] https://dictionary.cambridge.org/dictionary/english/rap Last accessed: 20–07–2018

[3] https://www.ucalgary.ca/dflynn/rap. Accessed 14–06–2018

The excerpt of William Blake's *The Chimney Sweeper: When my mother died I was very young* we presented above contains 33 words and about 22 of them rhyme in some way, thus we consider it to have a rhyme density of of $22/33 \approx 0.66$. Fast forward two centuries to 2004 *MF DOOM*'s *Figaro*:

> The rest is empty with no brain but the clever nerd
> The best emcee with no chain ya ever heard
> Take it from the Tec-9 holder
> They've bit but don't know their neck shine from Shinola
> [...]
> Not enough tracks
> Hot enough black
> It's too hot to handle, you got blue sandals
> Who shot ya? Ooh got you new spots to vandal
> Do not stand still, boast yo' skills
> Close but no krills, toast for po' nils, post no bills
> Coast to coast Joe Shmoe's flows ill, go chill
> Not suppposed to overdose No-Doz pills

Some phrases, words and syllables are part of different parallel rhyme schemes, so we choose two ways to mark rhymes: color words and colored underlines. The colors are not preserved across methods. It is instantly evident that almost all words in this excerpt rhyme. All words participate in at least one rhyming scheme, making it one of the denser hip-hop verses with respect to rhyme content, and it comes to no surprise that *MF DOOM* is frequently referred to as "your favorite rapper's favorite rapper" [1]

In the second part, DOOM builds upon the motif of the vowels /oʊ/ and /ɪ/, and creates some remarkable assonance chain rhymes:

go chill

/oʊ/-/ɪ/

post no bills

/oʊ/-/oʊ/-/ɪ/

close but no krills

/oʊ/-/ʌ/-/oʊ/-/ɪ/

toast for po nils

/oʊ/-/ʉ/-/oʊ/-/ɪ/

coast to coast Joe Smhoe's flows ill

/oʊ/-/u/-/oʊ/-/oʊ/-/oʊ/-/oʊ/-/ɪ/

not supposed to overdose No-Doz pills

/ɑ/-/ə.oʊ/-/u/-/oʊ.ə.oʊ/-/oʊ/-/oʊ/-/ɪ/

[1] https://pigeonsandplanes.com/in-depth/2014/03/rappers-doom/mos-def-3, Last accessed: 22–07–2018

Comparing rap lyrics such as the excerpt above with classic poetic forms that follow specific stress and rhyming patterns, reveals little in common. Au contraire, rap lyrics often do not follow norms and patterns, changing the rhyming schemes often in a single line, and the point is that the lyricism of rap lyrics transcends towards a radically different realm.

But rapping is not always about rhyming. Kendrick Lamar, became the first hip-hop artist to win the Pulitzer prize for Music in 2018 [1] and is known for creating musical motifs through his writing, as we can see in the following excerpt from *Alright*:

1 **I** don't **think** ab**out** it, **I** dep**os**it **ever**y **oth**er **ze**ro
2 **Thin**kin' **of** my **part**ner **put** the **can**dy, **paint** it **on** the **reg**al
3 **Dig**gin' **in** my **pock**et **ain't** a **pro**fit, **big** en**ough** to **feed** you
4 **Ev**ery**day** my **lo**gic, **get** an**oth**er **dol**lar **just** to **keep** you

5 **In** the **pres**ence **of** your **chi**co, **ah**

6 **I** don't **talk** ab**out** it, **be** ab**out** it, **ever**y **day** I **see** cool
7 **If** I **got** it **then** you **know** you **got** it, **Heav**en, **I** can **reac**h you

8 **Pet** dog, **pet** dog, **pet** dog, **my** dog **that**'s all
9 **Pick** back **and** chat **I** shut **the** back **for** y'all
10 **I** rap, **I'm** black, **on** track **so** rest **ass**ured
11 **My** rights, **my** wrongs **are** right **till** I'm **right** with **God**

Here, Lamar creates a verse with a relatively conservative meter that could be categorized as a *trochaic octameter*. If we label stressed syllables with "***DUM***" and unstressed ones with "*da*", a trochaic foot consists of a stressed syllable followed by an unstressed one, as in "***DUM***-*da*". The term *octameter* indicates that there are eight such feet on each line, as in:

DUM-da-**DUM**-da-**DUM**-da-**DUM**-da-**DUM**-da-**DUM**-da-**DUM**-da-**DUM**-da

Naturally, Kendrick Lamar is not the first to use this meter. Known examples include *Edgar Allan Poe*'s *The Raven* ("**Once** up**on** a **mid**night **drea**ry, **while** I **pond**ered, **weak** and **wea**ry") [2], *Alfred Tennyson*'s *Locksley Hall* ("**Com**rades, **leave** me **here** a **li**ttle, **while** as **yet** 't is **ear**ly **morn**") [3] and *Robert Browning*'s *A Toccata of Galuppi's* ("**Oh** Gal**up**pi, **Ba**ldass**ar**o, **this** is **ve**ry **sad** to **find**") [4], to name a few.

Yet, the artiste utilizes the full liberty rap forms favor and abruptly ends the octameter with a pentameter in line 5 ending with the stressed exclamation "ah", just to forthwith resume the octameter for two lines. Finally, the pentameter rebounds for the last four lines, with the last line (11) is also a pentameter, but Lamar throws in an extra word which falls out of the meter, choosing specifically the word *God* for this.

Problem Statement

From the prior we can see that rap lyrics build on conventions cultured and refined over the history of human and the use of language, nevertheless promoting lyricism to more unrestricted and creative forms. Boden (1998) argues that creativity poses a fated challenge

[1] http://www.wweek.com/music/2018/05/03/the-pros-and-cons-of-kendrick-lamars-pulitzer-prize/ , Last accessed: 20–06–2018
[2] https://www.poetryfoundation.org/poems/48860/the-raven, Last accessed: 22–07–2018
[3] https://www.poetryfoundation.org/poems/45362/locksley-hall, Last accessed: 22–07–2018
[4] https://www.poetryfoundation.org/poems/43777/a-toccata-of-galuppis, Last accessed: 22–07–2018

for AI and that research into creative AI models can yield fruitful results in two ways: both in commercial applications and helping cognitive scientists to understand what human creativity actually consists of. Regarding the former, not only ghostwriting (the act of writing lyrics for other artists without taking credit) is widespread in hip-hop music [1], but there exist also commercial services that offer on-demand lyrics online that anybody can buy, with prices ranging from 24$ for a 8-bar verse to 5000$ for a full album [2] making the need for, at least semi-, automatic rap lyrics generation imperative.

Additionally, it is evident from the aforementioned that in order to create rap lyrics it takes significant lyrical skill, rhyme matching and creativity combining words and rythms into patterns, therefore we believe it forms an ideal territory to address the question: how can we computationally generate something complex and human-like using state-of-the-art computational linguistics research findings. We call this task **Automatic Generation of Rap Lyrics**. The term *automatic* here is of chief importance: we are interested in unconditionally modelling and generating rap language without depending on any particular style, metre, vocabulary or any other priors. We evaluate our results using both quantitative and qualitative methods.

1.2 Thesis Outline

Next, in sections 2.1 to 2.2, we present and discuss various methods that have been tried and can be used for our task, and what the current state-of-the-art is. We do not focus only on song lyrics but give an overview regarding automatic poetry generation and language models, which we think are tasks similar to ours. Our objective is to present the literature but also raise issues and consider limitations specific to our approach. We proceed to formally present the concepts, models and techniques we use on in section 2.3

Moving to the next chapter, in section 3.1 we present how we selected our samples and how we acquired the dataset, the filtering and preprocessing steps we employed. In section 3.2 we present the implementation details of our models, which fall into two tiers: **n-gram language models** and **neural network language models**. In sections 3.3 and 3.4 we define our experimental and evaluation procedure, respectively

In chapters 4 and 5 we present, analyze and discuss our results, also giving some possible directions for future work.

[1] https://en.wikipedia.org/wiki/Ghostwriter#Music, Last accessed: 20–07–2018
[2] http://www.rap-rebirth.com, Last accessed: 20–07–2018

Chapter 2

Previous Work

Studying how computers can generate meaningful natural language the same way individuals swimmingly do is not an new idea in academia. It was 1952 when the famous computer scientist and mathematician Alan Turing proposed reducing the question *"Can machines think?"* to a statistical survey of whether humans can be *persuaded* by machines that they are humans after having a short conversation (Turing, 1950). Another well-known early example of a practical application is the computer program ELIZA (Weizenbaum, 1966), trying to guess the context from a conversation with a human and reply with psychoanalyst-like questions. We proceed to give an overview of the history of language modelling, poetry generation and rap lyrics generation.

2.1 Background

2.1.1 Statistical Language Modelling

Language, in a broad sense, involves any system that consists of symbols and ways of manipulating them in order to create arbitrary sequences. Computers can easily deal with some kind of languages, e.g. formal languages such as mathematics or programming languages. Natural language on the other hand, the kind of language that has involved through millennia and in radically different cultures, is not devised in the same way formal languages do, instead human languages just appear and evolve. The meaning of words and syntax often change in different geographical and temporal contexts, and additionally there are hundreds of thousands of terms that are often ambiguous to the extent of being incomprehensible when approaching it as a formal, strict, rule-based construct.

Therefore, having a computer able to decide confidently if a sentence is a well-formed one in a given language is of crucial importance to linguists, with extensions to "*applications such as machine-translation and automatic speech recognition*" (Goldberg, 2017, p. 105), "*document classification and routing, optical character recognition, information retrieval, handwriting recognition, spelling correction and more*" (Rosenfeld, 2000), so accordingly "*language modeling plays a central role in natural-language processing, AI, and machine-learning research*" (Goldberg, 2017, p. 105). The first successful approaches come from the early 80's, and there have been vast improvements (Rosenfeld, 2000), with neural network approaches dominating the field in recent years (Melis et al., 2017). We present the problem formally, as well as why it is valuable for our own research, in section 2.3.1.

Word- and Character-based Models

When we mention language models in an academic context, it is always the case that we mean word-level models, that is to say models that take word sequences as input and output word probabilities. This is a sober supposition in consideration of the traditionally limited context of n-gram language modelling, with the longest dependencies being in the order of magnitude of 1, having prevailing setups counting bigrams or trigrams. Neural networks, however, can effectively model very long dependencies compared to traditional methods, especially with newer architectures such as Long short-term memory units (see section 2.3.2.2) that are theoretically capable of learning infinite long dependencies.

Meanwhile, individual chunks of a word can carry significant information, especially for morphologically rich languages where individual morphemes in a word can carry significant and independent meaning that the model can pick up and exploit to extract information through different lexemes and lemmas, assuredly lost when considering each word as a separate individual token. As a consequence, there has been an increasing interest in the natural language processing community towards developing methods that take advantage of character-level features. Indeed, there have been publications bearing upon such features, both using individual characters as input together with outputting word probabilities (Kim et al., 2016) and outputting character probabilities (Graves, 2013), achieving competitive or state of the art results with much fewer parameters, as instead of a vocabulary counting to hundreds of thousands of words these models entail vocabularies consisting of individual characters often under a hundred tokens. Mixed approaches also exist that combine word features with character features, achieving state of the art results (Miyamoto and Cho, 2016) and also producing very promising results in poetry generation (Xie et al., 2017). Last but not least, character-level models can effectively model not only natural language but also source code in a given programming language and even markup code such as LaTeX (Karpathy, 2015).

2.1.2 Approaches in poetry generation

Early computer enthusiasts were apparently the first to experiment in hobby projects with poetry generation, with one of the earliest related publications being the book *Virtual Muse: Experiments in Computer Poetry* by Hartman (1996) (Manurung et al., 2000b).

Gervás (2002) discriminates between 4 types of approaches:

- Template-based methods, where the system basically fills in a template extracted from an already existing poem
- Generate and test methods, where the system generates string stochastically and evaluates them to maximize phonetic, structural and semantic related metrics
- Evolutionary methods that try to model the real creative process behind a human author and usually involve generating a population of drafts and evolving them using a fitness function
- Case-based reasoning approaches, where we usually have a database of verses or lines and the task is retrieving a relevant line and slightly changing some aspects.

It is worth noting that neural network approaches are mentioned only in the evaluation part of evolutionary-based approaches, fact that is easily explainable by the publishing date of Gervás (2002), as it was not before the late 00's that neural networks started dominating in essentially all fields of machine learning, e.g. image recognition (Graves et al., 2009;

Krizhevsky et al., 2012; CireşAn et al., 2012) at times beating even human performance [1], speech recognition (Hinton et al., 2012; Hannun et al., 2014) & generation (Van Den Oord et al., 2016) and, related to our task, language modelling (Mikolov et al., 2010, 2011; Kim et al., 2016; Miyamoto and Cho, 2016). There are of course neural network approaches to poetry or rap generation, which we are going to present in the next section.

Manurung et al. (2000b) is one of the first papers that address the task of automatic poetry generation. They employ a stochastic process where they perform various transformations and evaluating phonetics, syntax, and semantics to settle on the best poem, a process they call *Stochastic Hillclimbing*. Despite the stochastic factor, they still use a hand-crafted grammar as well as pre-defined sets of semantically similar words, facts that make this approach very limiting for our task, where rhythms and flows vary even in a single artist's works and evolve much more organically. Despite it being one of the first attempts, we can see some recurring themes: the authors stress the difficulty of evaluating such systems, and this is a not easy to overcome difficulty all approaches face. We also note the extensive use of predefined templates and hand-crafted rules to produce the resulting poem.

In his thesis (Manurung, 2004), the same author proposes to look at the problem as a state space search problem and define the goal state as this that satisfies three specific properties. The first is *meaningfulness*, as in any poem (or any purposeful text, actually) must convey some meaning under some interpretation. The second, *grammaticality*, states that a poem must be syntactically correct and can be seen as a Chomskian constraint satisfied by every possible sentence for every possible grammar. Finally, he introduces the notion of *poeticness* which he reduces to phonetic features such as rhythm (stress) patterns and rhyming. He argues that for any natural language construct to be classified as a poem, it should satisfy all three constraints and that almost all approaches (in 2004, that is) do not effectively satisfy them. At this point we should mark a departure from our own approach, as rap music is greatly characterized by casual dialects frequently violating formal syntactic rules, e.g. the line "Alls my life I has to fight" as well as the very title of *Kendrick Lamar's Alright*. With these three properties as the departure point, he infers 4 possible systems:

- Word salad: systems that just concatenate random words together, much like monkeys typing on a typewriter [2]
- Template and grammar-based: words are selected out of a vocabulary that fit in specific templates, satisfying only the property of grammaticality
- Form-aware: systems that follow metrical rules for producing poetic forms that have a predefined number of syllables or rhyming patterns, e.g. sonnets. Manurung argues that this kind of systems do not satisfy the meaningfulness property.
- Poetry generation: Systems that satisfy all the 3 properties, grammaticality, meaningfulness and poeticness

Manurung's solution, dubbed *McGonnagal* and being an improved version of Manurung et al. (2000b) (though already formulated earlier in Manurung et al. (2000a)), uses an evolutionary approach that applies iteratively an evaluation and an evolution phase over a set of candidates (population) and satisfies all 3 properties, making it a poetry generation system. While it is a significant step forward compared to previous attempts, the evaluation part

[1] http://www.kurzweilai.net/how-bio-inspired-deep-learning-keeps-winning-competitions, Last accessed: 20–07–2018

[2] https://en.wikipedia.org/wiki/Infinite_monkey_theorem, Last accessed: 20–07–2018

still depends on user-given stress and rhyme patterns as well as a semantic target, making it infutile for our task.

Oliveira (2009) gives an overview of the field after Manurung (2004), based on his taxonomies and properties, and adds two more systems to the list of those that can be classified as poetry generation systems, ASPERA (Gervás, 2001) and COLIBRI (Díaz-Agudo et al., 2002). These two approaches are quite similar to each other and are based on Case-Based Reasoning (CBR) processes, which informally try to solve a new problem (generating a new poem in this case) using experience from past problems. Unfortunately, both of the systems suffer from the same limitations as previous approaches, e.g. in the ASPERA system the user should enter both parameters such as length of the poem, rhyme structure and mood, as also an intended message to convert to a poem, making it a prose-to-stanza translator.

In Greene et al. (2010) the authors first train Finite-State Transducers (FST) to recognize stress patterns in text in a semi-unsupervised manner and to transform any text to iambic meter. The system works by taking a user-supplied template, generating some candidate sequences using a trigram model and then ranking them using the trained FST to produce a poem. This paper is an example on how such research can be used in other similar applications, as here the authors use the model to translate Italian poems to English whilst realizing specified rhythm patterns. The contribution of the paper is that it offers a probabilistic technique for extracting stress patterns from words depending on the context, as in poems a lot of times the stress pattern does not follow the dictionary stress patterns, as well as there are a lot of archaic pronunciations and unknown words.

Finally, we would like to mention the work of Ghazvininejad et al. (2016), which follows a modular approach with interconnected tasks to generate sonnets in iambic pentameter meter given a user-defined topic. The system starts by extracting the vocabulary from a corpus including semantic information, stress patterns and rhymes, building every possible path that satisfies the given meter and rhyme constraints. Thereupon, the information is passed to a Finite-State Acceptor, which is a variant of a Finite-State Machine that outputs a binary target for acceptance or rejection of the string. The noteworthy feature of their system, and a notable deficiency of ours (see section 5.1.1), is the ability of the model to combine different words in the same rhyme scheme, as a single word can rhyme with multiple words, for example "*people happy* $\sim$ *Cincinnati*", taken from an generated poem they include.

Other systems that do not seem straightforwardly suitable for our task but can nevertheless provide useful techniques and insights into poetry generation include Colton et al. (2012); Oliveira (2012, 2015); Toivanen et al. (2013); Gervás (2013).

2.1.3 Approaches in rap lyrics generation

The earliest approach in generating rap lyrics that we are aware of is by Wu et al. (2013). The architecture, labeled *FREESTYLE*, is based on stochastic inversion transduction, a technique assuming the language has a context-free grammar, that has application in machine translation systems. The basic mode of operation is parsing the symbols of one language and using transduction rules on non-terminal tokens until all tokens are terminals. For that reason, their model by design takes a single line and answers back with another line that rhymes with the given one. They use human evaluation and while they collect an imposing dataset of 260,000 verses, only about 60% of the generated responses are rated acceptable and about 30% are rated good.

The next publication, regarding specifically rap lyrics, is by Malmi et al. (2016). Their approach can be categorized as hybrid, as it involves both detecting the best next line given the previous one and generating new lyrics. Their technique involves two machine learning

techniques to embed each line in a dense vector, representing the rhyming and rythmical qualities of it, and then rank them according to similarity. Generating new lyrics is done by querying the dataset for the closest match. The dominant drawback is that the system cannot generate new content, but merely combines existing lines into new stanzas.

2.2 State of the Art

Markov chains for generating lyrics with style

In regard to the aforementioned approaches on poetry generation and apart from the demand for predefined templates or standard stress patterns in the stanzas, most of these systems are associated with another issue. Specifically, generate-and-test, as well as evolutionary approaches, are unbounded in terms of execution time, in other words it can take forever to generate one sequence, at least in theory. Au contraire, n-gram models / Markov chains can generate content in real-time but they do not guarantee the desired form of the output. Pachet and Roy (2011) undertake this issue by combining Markov chains with a constraint satisfaction problem (CSP). The original paper uses constrained Markov chains to generate music, specifically chord progressions, given user constraints, in bounded time. Pachet et al. (2001) show that as long as the constraints are unary (binding only to a single variable), the original Markov process M can be transformed into a constrained process $\tilde{M}$, generating exactly the desirable sequences while the probabilities are the same, scaled up to a constant factor. Simply stated, the authors manage to constrain the Markov process without introducing long-range dependencies, as that would violate the Markov property of memorylessness.

This is the starting point for Barbieri et al. (2012), where they employ this technique to generate lyrics in the style of Bob Dylan. They focus on 3 aspects of poetic forms: rhyme and meter, syntactic correctness and semantic relatedness. They impose a hard rhyming constraint after transcribing phonetically the words in the vocabulary using the CMU pronouncing dictionary (Rudnicky, 2014), and consider two words rhyming if the last stressed syllable matches. Similarly, they extract the stress for each word as well as meter for lines in Bob Dylan's work, creating a set of rythmic templates. Syntactical correctness is ensured by using part-of-speech (POS) templates which are also extracted from the corpus, keeping only the patterns that appear at least two times. Semantic relatedness, which is arguably the most difficult part, is realized using a predefined mapping of all n words that are semantically related to every w word in the vocabulary. The constraint is then applied randomly to pairs of words, to create a seemingly semantic coherent verse.

The advantage of their approach lies in the computational efficiency, the satisfaction of desirable properties such as rhyming, rhythm, syntax and semantics as well as the ability to imitate closely the style of a specific lyricist.

Using an LSTM for Automatic Lyric Generation

The most relevant experimentation to our own venture that we are aware of is done in Potash et al. (2015), where they address the problem of generating rap lyrics in the style of an existing artist, the concept that we introduced in 1.1 as *ghostwriting*. Another similarity is that we are also interested in generating lyrics in an unsupervised manner, i.e. do not want to provide any hard-based rules nor priors concerning rhythm schemes, rhyming schemes and semantic content. A notable deviation, though, is that we are not interested in writing lyrics in any specific artist's style. The authors use a relatively plain LSTM architecture

trained on 219 verses by the rapper *Fabolous*. As a baseline, they employ a simple n-gram model, and they evaluate with respect to how similar the lyrics are to the original artist (without plagiarising) using a similarity algorithm from the literature, as well as rap metrics as extracted by the methods described in Hirjee and Brown (2010b), which we employ too. They show that an LSTM outperforms n-gram methods in modelling the rapper's style, but most of the line endings are direct plagiarizations showing that the model just overfits on the dataset, which is reasonable since they do neither mention using any regularization technique nor a hidden validation set to stop training when the model starts to overfit. The authors resume their work in Potash et al. (2016), where they introduce a dataset of 12687 verses by 13 artists, manually annotated for similarities between artist, to extend their previous attempt. The paper offers a novel automatic evaluation approach in order to assess style similarity versus plagiarism.

Shakespearean Sonnets using RNN

The most promising to date use of neural networks to produce text with metre can be found in Xie et al. (2017). The authors offer an impressive assemblage of recurrent neural network architectures and techniques, including dense word embeddings, a word-LSTM model (section 2.3.2.2) similar to Potash et al. (2015) plus a word-GRU (section 2.3.2.3) model, their character level equivalents (inspired by the emblematic blog post by Karpathy (2015)), as well as mixed Convolutional-Recurrent and word-character models (inspired by Kim et al. (2016); Miyamoto and Cho (2016)). They suggest some useful hyperparameters that we also partly adopt and use both dropout and early stopping when the perplexity on a validation set stops improving. As a consequence, the produced texts are much better than Potash et al. (2015) and Potash et al. (2016) in generating original content, with their best model producing less than 5% plagiarized content counted in overlapping trigrams between the model output and the training dataset.

They conclude that subword (character) features allow models to reach lower perplexity levels, as well as they improve the quality of the generated poems in the qualitative analysis they perform. According to the authors, while they improve the coherence of the produced text, their models do not address adequately the rhyming issue, which they propose to tackle using networks with attention.

DeepRhyme

Neural networks represent the cutting edge in, and probably the future of, machine learning research, as we have oftentimes mentioned so far. It is of great interest, however, how sometimes breakthroughs happen in a non-academic context by empirical experimentation. We already mentioned the work of Karpathy (2015), where in a blog post back in 2015 he presented a character-level RNN architecture inspired by Graves (2013) and used it to model Shakespearean sonnets, Linux source code, Wikipedia including markdown and XML source and Algebraic geometry source code in LaTeX, providing also an insight on why and how these models work (although later he collaborated with other researchers to publish a paper on it (Karpathy et al., 2015)). Despite being a mere blog post, as of 20–07–2018 it has more citations in Google Scholar than Barbieri et al. (2012); Potash et al. (2015, 2016); Malmi et al. (2016); Xie et al. (2017) combined.

In the spirit of this, interesting and extremely relevant approach in generating rap lyrics is presented in Jones (2016) that at least is as valuable as the previous works presented. The author uses an LSTM trained on generating a verse backwards, i.e. starting from the last line and the last word of the line and moving towards the beginning of the verse, which he constrains as for every two line endings to rhyme. He also constrains the line length produced,

forcing the model to output exactly 10 words on each line. He includes experiments with an AA rhyming scheme (rhyming every two lines) as well as an ABAB rhyming schemes with the rhyming lines alternating. He unfortunately does not include any evaluation method, but the article contains many ideas we also draw upon in the present work.

2.3 Models

We proceed to introduce some basic theory behind the models and techniques we employ as well as some mathematical formalisms.

2.3.1 N-gram language modelling

N-gram models are one of the first approaches to solve the language modelling problem and are part of every computational linguistics studies' core curriculum. While trivial in their conception they often provide non-trivial results and have undergone vast improvements over the years, e.g. with smoothing techniques.

In n-gram language models the probability of a sequence S is expressed as the product of the probabilities of all words, with each word's probability conditioned on the previous $n-1$ words, where n is the number of tokens in n-gram tokenization (Chen and Goodman, 1999). Formally:

$$P(w_1, \ldots, w_l) = \prod_{i=1}^{l} P(w|w_{1-i}) \approx \prod_{i=1}^{l} P(w|w_{i-n+1}^{i-1})$$

Where x_i^j denotes all the words from i to j.

2.3.1.1 Back-off n-gram models

The intuition behind back-off n-gram models is that not all words are equally probable to appear, given specific contexts, for example it is trivial to guess which word follows the phrase *I took the dog for a* ____[1]. Given a sizeable corpus, we can then proceed to calculate the probabilities of each word given a specific context, and so we are able to iteratively estimate the probability of a given sentence appearing in the modelled language.

This is expressed in the following equation:

$$\begin{aligned} P_{ML}(w_m|w_{m-1},\ \ldots, w_{m-n}) &= \frac{c(w_m,\ \ldots, w_{m-n})/N_s}{c(w_{m-1},\ \ldots, w_{m-n})/N_s} \\ &= \frac{c(w_m,\ \ldots, w_{m-n})}{c(w_{m-1},\ \ldots, w_{m-n})} \end{aligned} \tag{2.1}$$

Where $c(S)$ is the counts of the words in a sentence S and N_s is number of total words in our training set. The probability we get in eq. (2.1) is called the *maximum likelihood* estimate (Chen and Goodman, 1999). For example, the phrase *The Red* is more often followed by the word *Cross* than any other word, or even as is, so the maximum likelihood gets assigned a high value.

[1] hint: *walk*

Let's consider for a minute what happens if the word sequence is completely novel for the model: maybe the word *dog* is an epithet I am using to refer to my loyal bicycle that has never let me down, and I just wanted to say am taking *dog for a ride*. The count of this 4-gram is zero, as the model has never seen this exact combination of words, making the maximum likelihood estimation zero: $P_{ML} = c(\text{dog for a ride}) = 0$. Instead of the model giving up, it is a better idea to return instead the count for an order lower, i,e, $c(\text{for a ride})$, which is something the model probably can handle better. Reducing the word order when we have zero counts for a specific sequence is what is called a *back-off n-gram* model, as we fall-back in lower orders of n, all the way to the probability of a single individual token.

While it looks like given enough training data such a model would give us good results, in practice n-gram models can become very complex (as in many parameters) as the vocabulary size and the order increases. An even more serious objection, though, is as argued by Chomsky (2002) using the famous example *colorless green ideas sleep furiously*, that grammar and syntax indeed *"project the finite and somewhat accidental corpus of observed utterances to a set (presumably infinite) of grammatical occurrences"*, so having observed all the possible combinations of grammatically correct sentences is impossible, and probably not how language works in humans.

2.3.1.2 Smoothing

The solution to backing off is *smoothing*. The idea is that we should assign a probability even if we never seen a word given the specific context, with the most simple way being to pretend that we've seen each combination at least $\delta = 1$ time even if the n-gram is unknown to the model, transforming eq. (2.1) to:

$$P_{ML} = \frac{c(w_m,\ \ldots, w_{m-n}) + \delta}{c(w_{m-1},\ \ldots, w_{m-n}) + \delta|V|} \tag{2.2}$$

in probably the oldest smoothing technique, first appeared in 1920 (Chen and Goodman, 1999) and is known as additive, or Laplace smoothing. The smoothing techniques we experiment with are presented in section 3.2.

2.3.1.3 Constrained Markov Chains

Markov chains are equivalent to n-gram models though used mostly as a generative model in what is known as a Markov process. A Markov process is a stochastic process satisfying the Markov property that states that the probability of an event depends only on the state of the previous event, making it effectively a bigram model when applied to language modelling, although there exist also a variant known as Markov chains with memory, meaning they can look back more than one state. A n-gram model (without back-off) of order n is identical to a Markov chain with memory of order n.

Another concept in mathematics is a *constraint satisfaction problem*, in which we apply a finite set of constraints over the possible states. For the sake of example, let's say our model has just generated the following lines, taken from *Nas*'s *N.Y. State of Mind*, and at this phase the model is predicting the last word of the next line

> *Bullet holes left in my peepholes*
> *I'm suited up in street* _____

and we have the following transition matrix M

$$\begin{array}{c} \\ street \\ nice \\ clothes \end{array} \begin{array}{c} \begin{array}{ccc} street & nice & clothes \end{array} \\ \begin{pmatrix} 0.03 & 0.35 & 0.62 \\ 0.29 & 0.18 & 0.53 \\ 0.69 & 0.28 & 0.03 \end{pmatrix} \end{array}$$

but we also want the word to rhyme with the previous line ending, so we constrain the possible state transitions to only the ones that rhyme, zeroing out the ones that do not

$$\hat{M}^{(1)} = \begin{pmatrix} 0 & 0 & 0.62 \\ 0.29 & 0.18 & 0.53 \\ 0.69 & 0.28 & 0.03 \end{pmatrix}$$

the probabilities, however, do not sum to 1 and as a results we loose the stochasticity property of the model, so we need to divide by the sum

$$\hat{M}_{i,j} = \frac{M_{i,j}}{\sum_{j=1}^{n=3} M_{i,j}} \Rightarrow \hat{M} = \begin{pmatrix} 0 & 0 & 1 \\ 0.29 & 0.18 & 0.53 \\ 0.69 & 0.28 & 0.03 \end{pmatrix}$$

However, Pachet et al. (2001) show that normalizing individually each matrix in this fashion, by dividing each row with its sum, skews the statistical distribution of the original model, i.e. the model will output different probability ratios for possible sequences under the two models. The proposed solution is to generate the sequence right-to-left (backwards) instead of left-to-right and propagate the perturbation in the matrices caused by the previous normalization steps, which practically translates to multiplying the each element j, k at time-step i with the previous sum of row j, at time-step $i + 1$ before taking the sum.

2.3.2 Recurrent Neural Networks (RNN)

A recurrent neural network is a subclass of neural networks, where each node is conditioned on the previous time-steps of a sequence. This marks a difference from feed-forward neural networks, as RNNs have access to their internal state "memory" when processing a sequence, and as such the model can learn features that feed-forward neural network intrinsically cannot.

We start with presenting the most simple RNN variant called Elman network (also referred informally as vanilla RNN) and continue to the LSTM and GRU variants.

2.3.2.1 Elman networks as a language model

The is the simplest version of an RNN and was introduced by Elman (1990). In section 2.1.1 we presented how a n-gram language model maps seen bigrams to word probabilities. In neural network instead of chunking the input into n-grams and backing off to lower orders of n when we have a sentence smaller than the model order or when we have an unseen n-gram. Neural networks mediated the need for lower orders and variable length sequences by design, as we neither make any assumption about the length of the input sequence neither need to constrain the window they can look back in time (though this is often done for practical reasons).

We start with a sequence s of arbitrary length m, made up of words w that belong to a pre-defined vocabulary V: $s = (w_0, w_1, \ldots, w_m) | w \in V$ and we are interested in outputting the probability distribution $P(w) | \forall w \in V$. The words are assumed to be in a 1-of-k format, represented as a vector $\mathbf{v}$ of zeros and of length $|\mathbf{v}| = |V|$, where $\mathbf{v}_i = 1$ with i being equal to the index of our word in the vocabulary. We start by converting every 1-of-k representation to a dense vector matching the input dimensionality d of our network by taking the dot product with the equivalent parameter $\underset{|V| \times d}{W_e}$, $w\dot{W}_e = w', |w'| = d$ to obtain the sequence $s' = (w'_0, w'_1, \ldots w'_m)$, where W_e is a parameter to be optimized during the training process. The equation for getting the next word probabilities

$$h_m = \tanh(W_h\, w'_m + U_h\, h_{m-1} + b_h)$$
$$y_m = \text{sigmoid}(W_y\, h_m + b_y)$$

where y_m is the probability mass function of word w_{m+1} represented as an 1-of-k vector and W_x, U_x and b_x are parameters to be learned. The sigmoid function is used to convert the model outputs to a probability mass function, but in deep architectures k layers deep, the sigmoid is replaced by the hyperbolic tangent function in $y^i_m, \forall i < k$ and the output is passed as the input to the next layer replacing w'_m. We also need to redefine h_m as

$$h_m = \begin{cases} \sigma_h(W_h\, w'_m + U_h\, h_{m-1} + b_h), & \text{if} x \geq \\ 0, & \text{otherwise} \end{cases}$$

since don't have any prior hidden state h_{m-1} when propagating the first word w'_0.

Despite the time-domain architecture of RNNs, it is customary to unroll the network in time while keeping the parameters shared and use the backpropagation algorithm to find the gradient with respect to time, a technique called Backpropagation Through Time (BPTT) (Robinson and Fallside, 1987). However, since backpropagation relies on derivation and the chain rule $((f \circ g)' = (f' \circ g) \circ g')$ and the activation functions regularly used have derivatives in the range $(0, 1)$, each consecutive multiplication minimizes the gradient. As a result, the gradient decreases exponentially as the number of layers and time distance increases, with the effect that nodes further away from the final node train very slowly, or not at all. This is what is called the *vanishing gradient problem*, notably affecting recurrent and deep networks that use hyperbolic tangent or sigmoid activations, with alternatives to backpropagation having being considered (Pascanu et al., 2013; Hochreiter, 1998). Nowdays, the problem is almost exclusively dealt with by using more advanced architectures (Jozefowicz et al., 2015), which we proceed to present.

2.3.2.2 Long Short-Term Memory (LSTM)

Probably the most popular and effective recurrent neural network variant, introduced by Hochreiter and Schmidhuber (1997). The idea is to augment the vanilla RNN model with three gates: an input gate i that controls how much of the new input is learned, an output gate o that controls how much of the output is passed on to the next timestep, and a forget gate f that controls how much of the previous state passes on to forward timesteps, implementing a mechanism of memory in the model. The LSTM architecture is one of the most successful neural network architectures and have been commercially exploited on many applications, e.g. in *Google*'s voice recognition (Beaufays, 2015; Sak et al., 2015) and *Translate* (Wu et al.,

2016) and in *Apple iPhone*'s text Auto-completion [1]. Additionally, LSTMs have been very successful in pattern recognition contests since their introduction, winning e.g. the 2009 ICDAR [2] [3] handwriting competition (Graves et al., 2009), as well as achieving state-of-the-art results on well-established dataset, e.g. in speech recognition (Graves et al., 2013).

The forward pass of an LSTM is given in the following equations

$$
\begin{aligned}
i_t &= \sigma_g(W_i\, x_t + U_i\, h_{t-1} + b_i) \\
f_t &= \sigma_g(W_f\, x_t + U_f\, h_{t-1} + b_f) \\
o_t &= \sigma_g(W_o\, x_t + U_o\, h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c\, x_t + U_c\, x_{t-1} + b_c) \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

where W_x, U_x, b_x are free parameters to be learned, f_t, o_t, i_t is the input, forget and output gates respectively, c_t is the cell memory and h_t the hidden activation/output at time step t. Note that a LSTM unit at a single timestep receives three inputs, the input x_t, the previous cell memory c_{t-1} as well as the previous hidden state h_{t-1}. The $\circ$ symbol denotes the Hadamard product, the σ_g is the sigmoid function, and the σ_c, σ_h are usually hyperbolic tangent functions. Since the Hadamard product multiplies each element i, j of two matrices and the gate activation is the sigmoid function, the network is able to learn exactly what information is needed to pass on or be forgotten, effectively eliminating the vanishing gradient problem.

2.3.2.3 Gated Recurrent Units (GRU)

Gated recurrent units were first introduced by Cho et al. (2014) as a computationally cheaper alternative to LSTM, and although it has fewer parameters it can still compete with LSTM, depending on the task (Chung et al., 2014). A single GRU cell has two gates, the update gate z_t and the reset gate r_t at timestep t, and the equations for the forward pass are as follows

$$
\begin{aligned}
z_t &= \sigma_g(W_z\, x_t + U_z\, h_{t-1} + b_z) \\
r_t &= \sigma_g(W_r\, x_t + U_r\, h_{t-1} + b_r) \\
h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \sigma_h(W_h\, x_t + U_h\, (r_t \circ h_{t-1}) + b_h)
\end{aligned}
$$

where W_x, U_x, b_x are free parameters to be learned and in similar fashion to the LSTM cell, the σ_g is the sigmoid function, and the σ_h is the hyperbolic tangent function. We can see that the gate z controls the amount of input and negatively controls the magnitude of the last hidden state the network receives.

2.3.2.4 Word Embeddings

In order for any statistical method to be able to process words, they first need to be expressed in a numeric format. The naive approach is to assign each unique word in the vocabulary a unique integer identifier, so given the phrase "*The quick brown fox jumps over the lazy dog*"

[1] https://www.theinformation.com/articles/apples-machines-can-learn-too, Last accessed in 21–07–2018

[2] International Conference on Document Analysis and Recognition

[3] http://www.cvc.uab.es/icdar2009/competitions.html Last accessed in 21–07–2018

we proceed to create our vocabulary V as follows: *the* $\mapsto 0$, *quick* $\mapsto 1$, *brown* $\mapsto 2$, *fox* $\mapsto 3$, *jumped* $\mapsto 4$, *over* $\mapsto 5$, *the* $\mapsto 0$ (note that we assign the same integer, since we have already seen this word), *lazy* $\mapsto 6$, *dog* $\mapsto 7$. Now, for example, if we analyze a labeled corpus to predict if the entity represented by a word is an animal or not, given enough training data the model will learn that the integers 3 and 7 correspond to the 'animal' entity. This, however, is not reflected on the ordering of the words, as the closest words to the word *fox* are $2 \mapsto$ *brown* and $4 \mapsto$ *jumped*, words obviously completely irrelevant. In fact, we normally represent these integers as what we call one-hot vectors, which is a vector of size $|V|$ where all the values are zeros except for the index of the word we want to represent, so for the word *over* that would be $\mathbf{v}_{\text{over}} = [0, 0, 0, 0, 1, 0, 0]$, and this is why this kind of representation is referred to as a *sparse* representation, since for any sizeable vocabulary the overwhelming majority of the vector elements are zeros. If we consider the last statement again, another shortcoming becomes apparent: for a vocabulary consisting of 80000 words, which is not at all uncommon, each word is represented by a vector of length 80000.

One of the most exciting developments in natural language processing is also the one that solves this particular problem, and is know as word embeddings, with a popular implementation being *word2vec* (Mikolov et al., 2013) and, simply put, is a technique based on neural networks to transform the aforementioned high-dimensional sparse vectors into a lower dimension, dense vectors that actually represent more efficiently each word. Starting with the high-dimensional sparse encodings, we train the model on a corpus using methods that allow the model to learn which words are similar to each other. There are two proposed models in Mikolov et al. (2013): Continuous bag-of-words and Skip-gram. The common element of the two models is that we have a matrix M of dimensionality $|V| \times D$, where $|V|$ is the size of our vocabulary and, subsequently, the dimensionality of the one-hot encoded word vectors and D the desired dimensionality of the produced dense embeddings, so that taking the dot product of any one-hot vector

$$\forall\ \mathbf{v}_{\text{one-hot}} \in V \mid \mathbf{v} \cdot M = \hat{\mathbf{v}} \tag{2.3}$$

gives us the desired dense embedding of that word. We also define the hyperparameter w as the *window* that the model can sample words from, both forward (words following the target word) and backward (words preceding the target word).

In the continuous bag-of-words variant, after we embed each word in accordance with eq. (2.3), we take the w preceding as well as the w following words from our target words, sum them up, and then try to predict our target word. So, in our miniscule example and for a window of $w = 2$, we sum the dense representations of *the*, *quick*, *fox*, *jumped* and try to output the dense vector matching the word *brown*. We iterate for all the possible combinations of our corpus and modify the matrix M using backpropagation to minimize the difference between our predicted vector and the ground truth one. In the skip-gram model we begin with a similar procedure to the continuous bag-of-words model, but proceed in the opposite way: given the word *brown* we minimize the prediction error of the words *the*, *quick*, *fox* and *jumped*. Then, the goal is to predict the context given the target word. It follows that after training on a large enough corpus the model will effectively learn that *brown* is something appearing often in the same context with *fox* or *dog*, effectively mapping *fox* closer to *dog* than to *quick*, which was the closest word to *fox* using a one-hot embedding.

Here we would like to mention an efficiency problem that we believe addressing it contributed significantly to the momentum these methods have gained. When trying to predict a word w (or *words* in the skip-gram model) given the context c we need to output a probability mass distribution summing up to exactly 1, and the way this is always done in machine learning when we have more than two classes (in which case we can use the sigmoid function)

is by applying the softmax function that normalizes a vector to sum up to 1, converting it to a valid probability distribution. In word2vec this is done approximately:

$$p(w|c) = \frac{\exp(h^\intercal \hat{\mathbf{v}})}{\sum_{w_i \in V} \exp(h^\intercal \hat{\mathbf{v}}_i)}$$

where h is the hidden state embedding the context and $\hat{\mathbf{v}}_i$ the dense representation of the word w_i. Calculating the sum in the denominator is dominates the computational complexity as we need to take the inner product of the matrix h with every word $w_i \in V$, and this needs to happen for every training example. The complexity is bounded by the size of the context, which is admittedly small and the size of our vocabulary, which can get really large. The two proposed solutions are *hierarchical softmax* and *negative sampling*.

In hierarchical softmax, a solution inspired by binary tree search, instead of considering every single word separately we transform the problem into a binary search and output instead if the target word is in the first or second part of the vocabulary and so on, limiting the complexity to $O(\log_2(|V|))$. The negative sampling is even more simple. Again, instead of calculating the product for every single word in our vocabulary we select only n words from our vocabulary, typically in the range 5∼20 (and even smaller as the vocabulary becomes larger), and update the parameters only for these words. Given a large enough vocabulary, it approximates the full softmax good enough.

Back to where we left, the potential of word2vec is in fact really big. First of all, the model effectively learns synonyms, so the closest vectors to the word *dog* will be something along the lines of *canine*, *doggy*, *puppy* etc. More interestingly, such a setup allows as to do vector mathematics, as in the classic example $\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}$. As a matter of fact, the authors claim that well-trained word2vec models on adequate-sized datasets can learn semantic relationships such as $\mathbf{v}_{\text{Athens}} - \mathbf{v}_{\text{Greece}} + \mathbf{v}_{\text{Oslo}} \approx \mathbf{v}_{\text{Norway}}$, $\mathbf{v}_{\text{brother}} - \mathbf{v}_{\text{sister}} + \mathbf{v}_{\text{grandson}} \approx \mathbf{v}_{\text{granddaughter}}$, and syntactic relationships such as $\mathbf{v}_{\text{apparent}} - \mathbf{v}_{\text{apparently}} + \mathbf{v}_{\text{rapid}} \approx \mathbf{v}_{\text{rapidly}}$ and $\mathbf{v}_{\text{Switzerland}} - \mathbf{v}_{\text{Swiss}} + \mathbf{v}_{\text{Cambodia}} \approx \mathbf{v}_{\text{Cambodian}}$. To give a picture of the publication's pervasiveness, almost all neural network architectures for natural processing mentioned in this dissertation use dense word embeddings.

At this point we would like to mention an alternative to word2vec from Stanford University called *GloVe* (Pennington et al., 2014) (standing for **Glo**bal **Ve**ctors) which, instead of training iteratively in a neural-network manner, considers the co-occurrence of words in a pure statistical manner and constructs the transformation matrix analytically. While word2vec is considerably more popular that GloVe (7098 vs 4647 citiations according to Google Scholar as of 20–07–2018), they both work very well in practice, modelling effectively word similarities as well as semantic and synactic relationships.

Chapter 3

Materials and Methods

3.1 Data

3.1.1 Dataset

Our goal with respect to the dataset is to assemble a collection of rap lyrics representative of the finest contemporary lyricists, so we resort in the longest running hip-hop magazine [2] *The Source*. Issue #254, issued on June 26th 2012, features a list of the top 50 lyricist of all time, according to the editors [3], which we use as our main source. We futhermore include *Kendrick Lamar* and *MF DOOM*, as they are renowned for their novel contributions in rap music [4], as well as *Drake* for holding several Billboard magazine records [5] regarding his commercial success. We augment the dataset with the hip-hop groups or acts some artists are involved in, specifically *Grandmaster Flash* $\mapsto$ *Grandmaster Flash and the Furious Five*, *André 3000* $\mapsto$ *OutKast*, *GZA, Method Man* $\mapsto$ *Wu-Tang Clan*, *Guru* $\mapsto$ *GangStarr*, *Ice Cube* $\mapsto$ *N.W.A.*, *Prodigy* $\mapsto$ *Mobb Deep*, and *MF DOOM* $\mapsto$ *King Geedorah*. The full artist list with individual number of verses for each artist is presented in table 3.1.

We choose to retrieve the lyrics from lyrics website *Genius*, as it offers a API we could use programmatically to query for individual artists, it automatically includes artist collaborations, we empirically found the quality of the transcribed lyrics superior to other web services and finally a lot of sections were annotated with the type (verse or chorus) as well as the verse artist. We skipped all tracks that their title included words hinting that it is a non-lyrics or duplicated instance, i.e. tour dates, remix, instrumental, skit, intro, outro, edit, mix, version, a cappella, dub, translation, acceptance speech, interlude, tracklist, artwork, snippet, demo, q&a, interview, cover, live performance, lyric video, demo and credits. A common annotation format, and one of our incentives to use the specific website is as follows (warning: explicit content):

> *[Intro: The Notorious B.I.G.]*
>
> *Uhh, check it out, uhh*

[2] https://www.nytimes.com/2003/01/29/arts/war-of-the-words-at-hip-hop-magazines.html, Last accessed: 20–07–2018

[3] Scans available at https://genius.com/discussions/8591-The-source-top-50-lyricists-magazine-scans, Last accessed: 20–07–2018

[4] https://www.youtube.com/watch?v=QWveXdj6oZU, Last accessed: 20–07–2008

[5] https://www.billboard.com/articles/columns/chart-beat/7736706/drake-breaks-hot-100-records-most-hits-solo-artists-more-life-songs, Last accessed: 20–07–2008

I steps in, where the Mo's and the hoes at bay-bee?!

Fuck all that pretty shit

(...)

[Verse 3: The Notorious B.I.G.]

Damn, it feel good to see people up on it (uh)

Flipped two keys in two weeks and didn't flaunt it (uh-huh)

My brain is haunted, with mean dreams

GS's with BB's on it, supreme schemes

(...)

[Chorus: Kelly Price]

Just some ghetto boys

Living in these ghetto streets

And everyday they gotta fight to stay alive

It's just reality - It's just reality

(...)

Given the lyrics follow this convention, we extract all the lines that follow *[Verse X: {Artist}]* lines, given the artist matches the one we queried for, effectively discarding guest verses from other artists. In cases when the format did not include any artist name (as in *[Verse X]*, plain *[Verse]* or no annotations at all, we keep all the verses included. Finally, we use a Python port of Google's `langdetect` language detection library (Mimino666, 2017) to detect and discard non-English verses and transcribe all the non-standard characters (e.g. é ŏ, etc.) using the `unidecode` library (Šolc, 2009).

On the implementation part, we use the Python library `requests` [1] to send and receive HTTP requests and `BeatifulSoup` [2] to parse the HTML code.

At this stage we end up with 57507 verses.

3.1.2 Filtering

As described in section 3.1.1, we observed a lot of duplicate verses as well as choruses, tracklists and other non-relevant content. On that account, we expanded on our filtering methods to elect which lyrics were commissioned for our final dataset. We would like to note that after applying the follwing filtering steps we end up with 75% of the original downloaded dataset, and so we consider it imperative to describe all of the individual filters in adequate detail. The filters we used are as follows:

Empty verses

Beacuse of the text preprocessing steps described in section 3.1.4 we end up with 68 empty verses which we proceed to do away with.

Exact duplicates

We removed 3863 clear-cut duplicate verses.

[1] https://www.tablix.org/~avian/blog/archives/2009/01/unicode_transliteration_in_python/, Last accessed: 12–06–2018

[2] https://www.crummy.com/software/BeautifulSoup/bs4/doc/, Last accessed: 12–06–2018

Artist	Number of verses
2Pac	1139
50 Cent	893
Big Daddy Kane	341
Big L	395
Big Pun	437
Black Thought	387
Bun B	1514
Busta Rhymes	896
Canibus	900
Chuck D	389
Common	573
DMX	866
De La Soul	240
Drake	823
Eminem	1303
Eric B. & Rakim	119
Fabolous	815
GZA	707
Grandmaster Melle Mel	162
Guru	602
Ice Cube	684
JAY-Z	2156
Jadakiss	693
KRS-One	1307
Kanye West	911
Kendrick Lamar	951
Kool G Rap	331
LL Cool J	725
Lauryn Hill	202
Lil Wayne	2856
Lil' Kim	633
Ludacris	537
Lupe Fiasco	725
MF DOOM	432
Method Man	697
Mobb Deep	1045
N.W.A	233
Nas	1978
OutKast	398
Pharoahe Monch	275
Q-Tip	922
Queen Latifah	217
Redman	561
Rick Ross	1116
Royce Da 5'9"	981
Scarface	555
Slick Rick	217
Snoop Dogg	2169
Styles P	719
T.I.	2188
Talib Kweli	792
The Notorious B.I.G.	342
Vince Staples	267
Wu-Tang Clan	506
Yasiin Bey	236

Table 3.1: Verses per artist in all 3 dataset splits

Expand long verses

For each verse that is longer than 32 lines, we split the verse in two using as an anchor the middle. In this fashion we create 2620 new verses.

Tracklists

As we mentioned in section 3.1.1, there were some entries that were lists of songs for an album. We use a regular to expression to detect the lines l_N that start with one or more digits followed by a space and discard all verses that satisfy the condition $\frac{|l_N|}{|l_A|} > 0.5$, where l_A stands for all the lines in the verse. Using this heuristic, we discard 40 verses, as this example titled *Illmatic 10th Anniversary Platinum Edition Album Art* by *Nas*:

> *disc 1*
>
> *1 the genesis*
>
> *2 n.y. state of mind*
>
> *3 life's a bitch*
>
> *4 the world is yours*
>
> *5 halftime*
>
> *6 memory lane (sittin' in da park)*
>
> *7 one love*
>
> *8 one time 4 your mind*
>
> *9 represent*
>
> *10 it ain't hard to tell*

Choruses

We discard 105 verses that start with the characters *chorus*.

Short verses

We discard all verses with less than 4 lines. We attribute small verses to transcriptions that use heavily newline characters, as in this excerpt from *Kayne West*'s *Anything*:

> *i mean wow you know man*

but we discover also news-like stories instead of vereses, as in this one titled *Kanye West Talks Fashion and Surprise Concerts in Paris*:

> *paris fashion has a new odd couple: bernard arnault and kanye west*

This is our filter causing the most discards, 7005 samlples

Invariant verses

We discovered samples that contained repeated words to a great extend, and consequently choose to discard 3218 that satisfy the conditon $\frac{N_{\text{unique}}}{N_{\text{total}}}$ where N_x is the word count for unique and total words respectively. We attribute samples with low word diversity to choruses or interludes, such as this dropped sample from *Snoop Dogg*'s *Smooth*:

hes been watching you so smooth
longbeach is on the move smooth
hes been watching you been watching you, so smooth
longbeach is on the move

but also to excessive use of exclamations, as in this example found in *Jay-Z*'s *Big Pimpin'*:

everybody in here just bounce
just bounce
just bounce
just bounce
uh uh uh uh
i like, ok let's go
uh uh
yeah yeah yeah yeah
uh uh uh
yeah yeah yeah yeah
uh uh uh

Too short / too long lines

We noticed samples that had their average line length was too small or too long. This can be attributed to erroneous transcriptions, as in this example found in *Styles P*'s *Ghost Dilla (The Red Freestyle)* (Note that the excerpt is a single line):

> *i'm too high to act normal, g to act formal, the game will transform you, watch for the word from niggas that word bond you, watch where you headed cause niggas will turn on you, classy beamer wagon, i'm playing the firm on you,yeah nature and cormega, plastic paper and pall bearers, pain in the street but we know that the lord hear us*

We found also non-lyrical content, as in this sample listing *Rick Ross*'s tattoos titled *Rick Ross Tattoos*:

Stomach
Barack Obama
Rosary
Statue Of Liberty
Richard Pryor
Isaac Hayes
'Chill Will'
100 Mill
Skull with cross
'Teflon Don'
5 Star G

To handle such cases, we extract all line lengths and calculate the 1st percentile P_1, the 99th percentile P_{99} and keep samples that satisfy the condition $P_1 < \mu(L_i) < P_{99}$, where L_i is the list of all line lengths for the verse i. leading to the disposal of 936 samples.

Non-uniform line length

Regardless the unrestrained forms rap lyrics often assume, we maintain that a verse has a more or less uniform line length. As such, we extract the line lengths for every individual verse and discard the ones that do not satisfy the condition $\sigma(L_{\text{char}}) > 20$, where L_{char} is the list of line lengths expressed in character counts, i.e. we discard every verse that has a line length standard deviation higher than 20 characters includng spaces. Such verses are often bridges or interludes, as in *Lil' Wayne*'s *Grown Man*:

damn, look, that
oh, that right there?
that's the sunset girl
get your visor, where those chanel shades i get you, you always losing things
and i'm always buying new things, i come a long way huh, remember?
remember that block, look at you, you think you all grown

Duplicated verses

Notwithstanding eliminating exact duplicates, we discovered during some early experimentation that even simple n-gram models achieved very low perplexity on our hidden validations set. We attribute this to duplicates that are not exactly the same, differing only on a couple of lines or words. We ascribe this to multiple performances of the same song, as in live or unplugged together with guest verses and sampling from other songs. In order to come to grips with such cases, we used Python's sequence comparison module `difflib` [1] to get rid of samples that had a matching ratio higher than 0.8 (the Python documentation mentions *"As a rule of thumb, a ratio() value over 0.6 means the sequences are close matches"*). Rough duplicates extracted with the described technique amount to 1597 verses.

3.1.3 Dataset Splits

We split the dataset into 3 sets, one set designated 'training' to train or models on, one set 'validation' to compare the various methods and hyper-parameters in that we get the best results and a final 'testing' set to present our final results on. To make sure each split maintains the same ratio of artists, we use the scikit-learn machine learning framework Pedregosa et al. (2011), specifically the `StratifiedShuffleSplit` class from the `model_selection` module which we used to get the folds that try to preserve the same percentage of samples for each class. Although this is not guaranteed, it is stated that "*...this is very likely for sizeable datasets.*"

After filtering and splitting we end up with 43058 verses, specifically 34446 verses on the train split and 4306 on each of the validation and test split. Some statistics for the different splits of our dataset can be found in table 3.2.

For storing the data, we used the pandas (McKinney) data analysis library to create one `DataFrame` per split, each with one row per verse and three columns: `artist` containing the artist's name for the corresponding verse, `track` containing the song's title and `lyrics`

[1] https://docs.python.org/3.6/library/difflib.html, Last accessed: 20–07–2018

containing the actual verse. The three dataframes were then saved using the `.to_csv` method, disabling the index column. We perceive three advantages when using the pandas library, and specifically storing and loading our datasets as a dataframe: compatibility, as the file is still a plain csv file that can be parsed manually, ease of use, as loading is as simple as `import pandas as pd; df = pd.read_csv(dataset)` and automatic handling of newline characters present in verses, as manualy parsing those files might create confusion regarding which of these lines separate the verses and which ones actually belong to the content.

Statistic	Training set	Validation set	Testing set
N verses	34446	4306	4306
Lines per verse	14.70 (6.78)	14.69 (6.85)	14.63 (6.77)
Words per verse	124.05 (61.47)	123.60 (61.84)	123.67 (61.37)
Words per line	8.35 (1.56)	8.33 (1.59)	8.35 (1.56)
Characters per line	40.52 (7.63)	40.41 (7.75)	40.57 (7.65)
Verses per artist	628.29 (458.78)	78.29 (57.38)	78.29 (57.44)

Table 3.2: Mean statistics and standard deviations for the different splits of our dataset

3.1.4 Preprocessing

Normalization is a vital part of any language processing or information retrieval task involving textual content. Following time-honored conventions in natural language processing (Christopher et al., 2008), we get under way by lowercasing all text and removing all punctutation. The one and only exception to this rule is the apostrophe (') character, which we treat as an alphanumeric character. As a result, all compounds including the apostrophe character are treated as a single token, as in e.g. *don't* or *i'm*. Diacritics such as é were already converted to their standard English equivalents when downloading the lyrics from the web, as described in section 3.1.1.

After filtering non-verse samples, we found the dataset still incommensurate to our experiments mainly beacuse of arbitrary notation practices. We noticed some recurring patterns that while might make sense for a human reader interested in interpreting a verse still might contravene with a model's ability to model the desired features of lyrical content.

Overdubs / other sounds annotations

Firstly, rap songs often contain voice overdubs repeating the previous words or adding an extra phrase, as in this example from *KRS-One*'s *Step Into a World*:

> *Comentating (say what?) illustrating (yeah)*
> *Descriptions given, adjective expert (I hear you)*

A similar pattern is annotation of sounds in the song, as in this excerpt from *Lil' Wayne*'s *Grown Man*:

> *Then I tell her never mind and we do it one more time [laugh]*

We used a regular expression to remove altogether the content between parentheses, brackets and curly brackets, as well as the parentheses, brackets and curly brackets themselves.

Repeated characters

Another pattern we observed is stressing the word pronuncuation by repeating a single character when a stress, streching or change in the singing style occurs, as in *Ghostface Killah*'s verse in *Wu-Tang Clan*'s *The Heart Gently Weeps* (warning: explicit content):

> *You know you're bootyyyyyy*
>
> *You pulled your toolie, out on meeeee... motherfucker*

Here the artist sings melodically the signified words instead of reciting them rhythmically as rappers normally do, and this is what the annotator is try to convey. However that only adds a unique word that depends on how long the key was pressed without any special meaning. We used a regular expression again to replace all consecutive occurences of more than 3 identical characters with the single character.

3.1.5 Vocabulary

Word-level models

We assemble our vocabulary using all words in the training corpus. While many natural language processing applications limit the vocabulary to tokens that appear at least on N samples, we found out that tokens that appear only once constitute about 45% of the total token count, suggesting that any model trained with limited vocabulary will output the token corresponding to the out of vocabulary words very often, and this could skew the quality of the output. Thus, we choose to include all words in the vocabulary, with the final word count being 70080 words. We nevertheless add a token representing out of vocabulary words (`<unk>`) so we can calculate the modelling effectiveness of our models on the validation and testing set. Finally, we include two special tokens, `<sos>` for start-of-sequence, that we insert before each verse and `<eos>`, which we append at the end of each verse.

Character-level models

For the character-based models, as well as models that incorporate subword features, the vocabulary consists of the 26 lowercase letters of the English alphabet (a–z) and digit characters (0–9), plus the apostrophe character, newline, and the special `<sos>` and `<eos>` tokens.‘

3.1.6 Rhyming Words, Rhyme and Stress Templates

Since we are also interested in experimenting with constrained models, inspired by Barbieri et al. (2012) and Jones (2016), we need to know beforehand which words rhyme and what rhyming and stress patterns we would like to use. To the best of our knowledge, the state of the art in rhyme detection in a rap lyrics context is described in Hirjee and Brown (2009). The authors use techniques inspired from research in biology, specifically amino acid sequencing, to detect internal and imperfect rhymes using a stochastic method, improving on previously used methods. They also show that their method produces metrics capable of capturing many aspects of an artist's style and prove their point by classifying artists only by their extracted rhyming statistics (Hirjee and Brown, 2010a). They subsequently released a Java implementation of their method (Hirjee and Brown, 2010b) freely available online. To allow for rapid prototyping and experimentation, we used the *Jython* [1] Python implementation that runs on top of the Java virtual machine and allows calling java objects natively from

[1] http://www.jython.org/

Python code. We used the source code as our documentation on how the classes were used on the original GUI and translated the logic of the relevant parts, while still using the original objects from the compiled `.jar` file. To build the (Java) project, it was required to use the Apache Ant build system [1]. Although the Java integration greatly facilitated and speeded up our development process, we note for readers interested in Jython that only Python version 2 is is supported, and the relatively old version 2.7.0 versus 2.7.15 for the latest version of the reference CPython implementation. Additionally there is very limited support for popular natural language processing and numerical computing libraries such as gensim, nltk, numpy and scipy, so our use was limited to extracting the rhyme and stress patterns and nothing more.

Nevertheless, integrating Hirjee and Brown's rhyming detection and analysis classes in our scripts running in pure Python proved of vital importance for our quantitative evaluation of our models. For each verse on our train dataset we iterate through every pair of lines and extract the rhyme and stress pattern, but with the following limitations:

- We keep only rhyme pairs that have the same number of words per rhyme
- In multi-word rhymes, we consider the separate words as separate rhymes
- We join common rhymes under the same rhyme index
- We discard rhymes that appear only once in the two pairs combined
- We discard pairs that their last words are not rhyming with each other
- We discard pairs that include a line without any rhymes
- We discard pairs that appear only once

Additionally, we discovered that stress patterns are much more unique than rhyme patterns, i.e. we found the same patterns a lot of times over different artists, while stress patterns rarely appeared more than a time. As a results, and because we want the two to match in quantity, after extracting rhyme patterns that appear at least to times we select randomly a stress pattern that matches the specific rhyme pattern and associate it with it, discarding the rest. We finally end up with 10436 unique rhyme/stress pairs.

Additionally, for each word in our vocabulary we extract the last stressed syllable, or the last syllable if the word does not contain any stressed syllable, all the vowels up to the last stressed syllable or all the vowels if the word does not contain any stressed syllable and the stress pattern of the word.

Rhyme definition

Drawing inspiration from Barbieri et al. (2012), where they consider two words rhyming if the last stressed vowel matches, we restrain the definition a little more and consider two words rhyming with another if all the vowels starting from the last until the last stressed, or all vowels in case of an unstressed word, match.

3.2 Model Implementation

Back-off n-gram

[1] https://ant.apache.org/, Last accesed: 20–07–2018

For the back-off n-gram models we use Stanford SRI's language model implementation [1] (Stolcke, 2002). The models were trained using the provided command-line Linux utilities. For generating texts, we use a Python wrapper [2] to run the C++ code directly from our Python scripts, allowing for rapid prototyping and experimentation, as well as sharing common code components regarding rhyming and stress constraints between the models. The version we use is 1.7.2, which is the latest available as of 20–07–2018.

We choose to train the different model combinations using a Python script and spawning the SRiLM process using the `subprocess` module. The motivation behind this decision is that we can load the texts using pandas from our dataset files described in section 3.1.6, apply any necessary preprocessing steps and then pass the texts using the Linux standard input to the SRiLM process, logging each step and any possible errors, as well as validating using our validation set and generating some samples from the trained model using a single script.

The SRiLM's `ngram-count` utility used to train n-gram models expect each sample to be in a single line, so we preprocessed each verse to replace newlines with the `<n>` token. Additionally, the start-of-sequence and end-of-sequence `<sos>` and `<eos>` tokens are automatically added by SRiLM as `<s>` and `</s>` so we made sure of to translate from one another when generating new samples. Since we do not prune any words based on frequency our training set has no occurrences of the `<unk>` token, and this can cause problems [3] as for example assigning too high a probability to the unknown token, so we choose to build a closed-vocabulary model, ignoring every unknown word. Because we are interested on constraining end rhymes, we choose to feed the sequences in reverse, i.e. starting with the last word on the last line all our way up to the first word in the first line, as done in Barbieri et al. (2012); Jones (2016) where they also employ rhyme constraints. Finally, during generation of unconstrained text, we noticed the model could not reliably output lines of reasonable length, so we use a technique called distillation to alter the probability outputs of the model Hinton et al. (2015) to produce a "harder" probability distribution, i.e. increase further the probability of selecting the most probable words. The process is controlled by a temperature hyperparameter, with temperature $= 1.0$ leaving the distribution unaffected, temperature > 1.0 softening up the distribution by lowering the probabilities of probable words and making the probabilities of relatively lower probability words higher, and temperature < 1.0 making the distribution harder as we described. During generation of content using constraints, we get the log-probability for each word in the vocabulary that satisfies the constraint for the specific place in the line, if any, and sample the next token after applying a softmax function to make the output probabilities sum to 1.

Training of n-gram models was done in the server of the Centre for Language Technology, Department of Nordic Studies and Linguistics of the University of Copenhagen. The server we had access is based on an Intel Xeon E3–1246 v3 processor @ 3.5 GHz consisting of 4 physical / 8 logical cores and 32 GB RAM. The SRiLM toolkit was compiled from source with support for SSE2 instructions for faster performance. Generation of verses was done on our own machine.

Constrained Markov Chains

For the constrained Markov chains, we base our implementation on the authors' own

[1] Source available at http://www.speech.sri.com/projects/srilm/ Last accessed in 25–06–2018

[2] https://github.com/desilinguist/swig-srilm, Last accessed: 20–07–2018

[3] https://mailman.speech.sri.com/pipermail/srilm-user/2007q4/000543.html, Last accessed 21–07–2018

implementation in JavaScript [1]. We transcribed the JavaScript code in Python, following the same logic for the most part. For counting the n-gram combinations, though, we wrote our own script that creates the transition matrices in JSON format. Due to the pure Python implementation using dictionaries, we found the model to be quite ineffective and slow when using a lot of samples, therefore we limit the set we extract the transition matrices from to a random subset equal to the 1/10 of the dataset.

Extracting the transition matrices was done on our machine but inference was done on the Center for Language Technology server described in the back-off n-gram implementation.

RNNs

The LSTM and GRU models were presented in detail in section 2.3.2 and were implemented pretty straightforwardly from scratch using PyTorch (Paszke et al., 2017). Our implementation takes advantage of PyTorch features that allow batch processing during training and inference. Specifically, we handle sequences of varying length by using padding and the suitable PyTorch utilities [2] in order to skip inference calculations as well as loss propagation for the zeroed out parts. This approach also takes advantage of the GPUs we had available by running many samples in parallel instead of feeding one sample at a time.

For the Gated LSTM model, we use a single layer bi-directional LSTM model of hidden size 100 for each direction that takes as input a word as sequences of characters. Bi-directional LSTMs combine two separate networks, commonly referred as the *forward* and *backward* pass, as the forward pass takes the word as it appears, i,e. $\{\mathrm{w}, \mathrm{o}, \mathrm{r}, \mathrm{d}\}$, while the backward pass receives the word in the opposite order, as in $\{\mathrm{d}, \mathrm{r}, \mathrm{o}, \mathrm{w}\}$. This is the way it was originally introduced in Miyamoto and Cho (2016) and while both passes process the whole word, the forward pass focuses on features present in the latter part of the word, after having seen the antecedent part and the backward pass respectively focuses on antecedent features after having processed the latter part of the word. The last hidden state of each pass, i.e. the forward pass after having processed the last letter and the backward pass after having processed the first letter, are concatenated in $\hat{h}, |\hat{h}| = 200$ and transformed to match the dimensionality of the word embeddings $x_{\text{char}} = W_c\,\hat{h}$ using the trainable matrix W_c of dimensionality $200 \times |x_{\text{word}}|$. The gate is conditioned on the word embedding using another trainable vector u_g of dimensionality $|u_g| = |x_{\text{word}}|$ plus a bias term b_g as in $g = \text{sigmoid}(u_g^\intercal\, x_{\text{word}} + b_g)$. To get the input for the network we add the word and character embedding using the learned gate to regulate each embedding's contribution: $x_{\text{in}} = (1 - g)\, x_{\text{word}} + g\, x_{\text{char}}$.

In a similar fashion to the back-off n-gram models and for reasons already expressed, we also train our neural network models to predict the sequence in reverse. During generation of text we also experimented with a temperature hyper-parameter to skew the output probability distribution as we did with the back-off n-gram models, and while we found lower temperatures producing denser rhyming output, it also increases a lot the plagiarised trigram count, so we choose to sample from the original output distribution.

Training and inference was done on University of South Denmark's ABACUS 2.0 supercomputer, with the node we had access to being based on 2 Intel Xeon E5–2680 v3 processors @ 2.5 GHz with 12 physical / 24 logical cores each, 64 GB RAM in total plus 2 Nvidia K40 GPUs, each GPU coming with 12 GB RAM.

Word Embeddings

[1] https://github.com/gabrielebarbieri/redylan, Last accessed: 20–07–2018

[2] https://pytorch.org/docs/stable/nn.html#packedsequence, Last accessed: 20–07–2018

For training word embedding we used the gensim library [1] (Rehurek and Sojka, 2010). However, because there is no way to control how gensim tokenizes the words, we first tokenize them with the same method used for training our other models and feed the input as lists of separate strings, so the generated vectors and vocabulary matches the one used on our recurrent neural network models.

Training and evaluation was done on our own machine.

3.3 Experiments

3.3.1 Preliminary Experimentation

We start off by running combinations of n-gram and RNN models in order to find the configuration that best models our dataset and proceed to select the best model from each of the following tiers with their respective hyper-parameters.

Back-off n-gram models

- Order: $[3, 4, 5, 6, 7, 8, 9]$
- Smoothing:
  - Witten-Bell discounting (Witten and Bell, 1991) *(WB)*
  - Ristad's natural discounting law (Ristad, 1995) *(RN)*
  - Kneser-Nay method (Kneser and Ney, 1995) *(KN)*
  - Chen and Goodman's modified Kneser-Nay method (Chen and Goodman, 1999) *(CG)*

The models are evaluated using the perplexity on the validation set.

Constrained Markov models

The only hyper-parameter to tune regarding the constrained Markov models is the model order. However, we have no method of quantitatively evaluating the models' ability to model text (the authors of the original paper propose none, either). We found empirically, however, that a model of order 2 does not produce meaningful texts in a lot of cases and conversely, as the order increases it becomes increasingly difficult for the model to satisfy the constraints, and as such the trigram model is at the same time the minimum and maximum order that can be practically exploited. A model of order 3 is also what Barbieri, Pachet, Roy, and Degli Esposti use.

Word & character RNNs

We fix the dropout probability to $d = 0.5$, as suggested by Xie et al. (2017); Karpathy (2015) and validate for the following hyper-parameters:

- Hidden units: $[512, 1024]$
- Number of layers: $[1, 2, 3]$
- Cell type: $[\text{GRU}, \text{LSTM}]$
- Embeddings: $[\text{random}, \text{pretrained}]$ (Only for the word model and for the best model found for the hyper-parameters above)

[1] https://radimrehurek.com/gensim/index.html, Last accessed: 20–07–2018

For optimization we choose to use Stochastic Gradient Descent with learning rate $\alpha = 0.1$ and momentum $\mu_{\text{pre-set}} = 0.99$ after empirical experiments on a small subset of our dataset. We initially set the momentum to $\mu = 0.5$ so that learning stabilizes and gradually increase it using the formula $\mu_i = \min(\frac{i\mu_{i-1}}{3}, \mu_{\text{pre-set}})$ where i is the number of epochs, effectively using the pre-set value from the third epoch onwards.

We monitor the validation loss after each epoch and half the learning rate every time it stops decreasing, stopping training when $\alpha < 10^{-6}$. We select the model achieving the lowest validation loss, regardless of the epoch.

Gated LSTM

For the Gated LSTM model, we select the best hyper-parameters we found on our RNN preliminary experiment.

Word Embeddings

We set the dimensionality D to the best value found in the Word & character RNN preliminary evaluation and validate for the following hyper-parameters:

- Method: [continuous bag-of-words, skip-gram]
- Softmax: [plain, hierarchical, negative sampling]

When the method is continuous bag-of-words we set the window size to 5, and when the method is skip-gram to 10, as suggested in the original paper (Mikolov et al., 2013). For selecting the best model we compile a short list of common slang hip-hop such as *hood* (neighborhood), *mc* or *emcee* (rapper), *tec* (a type of semi-automatic pistol), each with a list of synonyms or semantically related words and sum the rankings each model outputs for the target words. Additionally we evaluate the rankings for some common semantic and syntactic operations inspired from the original paper, like *woman* - *king* + *man* = *queen*. As anticipated, we choose the hyper-parameters that give the lowest summed ranking score.

3.3.2 Heuristic Techniques

We apply different heuristics to generate 200 verses for each heuristic and model combination, consisting of 3 line pairs, when constraints are applicable, or else until the start of sequence token.

Unconstrained

Acting as a baseline we generate unconstrained samples from the back-off and RNN models, in the same fashion as Potash et al. (2015, 2016); Xie et al. (2017).

Constrained - without templates

The first constrained heuristic does not make use of the extracted templates. The most simple case we designate `last`, similar to the one presented in Jones (2016), we force the model to rhyme the end words of each line pairs, without any constraint on the generated words stresses. Additionally, we include a heuristic designated `follow`, where we let the model generates a single line and apart from rhyming the end words we additionally constrain the model so that the next line follows the same stress pattern of the generated line.

Constrained - with templates

In template mode, we use the extracted rhyme and stress templates described in section 3.1.6 to zero out the probabilities for each word not satisfying the rhyme and stress constraint.

To sum up, we evaluate these 4 heuristics:

- `unconstrained`
- `last`
- `follow`
- `template`

3.4 Evaluation

3.4.1 Quantitative

As we mentioned numerous times through, evaluating creative computer systems have always been a problem, as if we had a way to reliably match the quality of human judgment there would be a high chance that the very problem of having computers generating creative content would be solved. This is not to mean, though that we are helpless, as we can at least partly evaluate the quality of our models, in terms of resemblance to the content we are trying to model (perplexity), individual desirable qualities of the model (rhyming) and originality of the generated output (plagiarism).

Our first proposed metric, perplexity, is nothing new in language modelling. Perplexity, roughly stated, measures how accurately a language model models the domain language. More specifically, [TODO:this]. Thus, as a first step we compare the perplexities of our different models and assume, at least theoretically, that the one with the lowest perplexity on a set the model has no knowledge of will produce the most realistic content.

The second metric has to do with measuring the rhyming ability of a model. As we previously described in section 3.1.6, Hirjee and Brown (2009, 2010b) develop an automated method to extract features related to the quantity and quality of the rhymes in a text. Perhaps the single most important feature we can extract using their framework is *rhyme density*, which is the ratio of rhyming words to all words in a verse. The metric has been used by other approaches on analyzing rap lyrics (Malmi et al., 2016) and has been shown by Hirjee and Brown (2010a) to be enough to differentiate between rap and other genres' lyrics, e.g. rock. It can also help researchers into analyzing trends in hip-hop music, as in the same paper a correlation shown between newer release and denser verses, i.e. with a high higher rhyme density. We believe that it can give us an insight into which model has learned more effectively how to rhyme, probably the essence of rap lyrics.

Finally, we are interested in generating original content, and not only plagiarising on existing rhymes, lines and stanzas. Thus, we adopt the method used by Xie et al. (2017) to detect plagiarism between our generated verses and the dataset our models we train on. The idea is to extract all the trigrams from the original dataset as well as our generated verses, and calculate the percentage of our verses' trigrams relative to the original dataset's trigrams. However, when manually testing with a couple of verses not present in our datasets (from artists not even included in our dataset), we noticed that it was hard to go below the 30% plagiarism level, even for those. We proceeded to manually check the most popular trigrams and noticed combinations that is only natural to occur in rap lyrics. For example, here are the 10 most common trigrams, with their occurrences in the train dataset:

1. up in the, 1258 occurrences

2. you know what, 739 occurrences
3. and if you, 637 occurrences
4. what the fuck, 605 occurrences
5. in the back, 583 occurrences
6. you know the, 562 occurrences
7. me and my, 531 occurrences
8. back to the, 510 occurrences
9. in the hood, 501 occurrences
10. in the club, 495 occurrences

As it would be unfair for a model to be considered that it plagiarizes just because it generates trigrams like these, we ignore the trigrams that appear in more than half of the documents. A lower percentage of common trigrams indicates that the model has effectively learned the underlying mechanics of rhythmical texts and that it does not just combine existing phrases and rhyme combinations.

All three metrics, perplexity, rhyme density and percentage of plagiarised trigrams can be applied to all of our models, except from calculating the perplexity of the constrained Markov model, for reasons we mention in section 3.3.

3.4.2 Qualitative

When judging creative content, quantitative evaluation provides (less than) half of the story. We consider it very important to have actual humans judging our generated verses, as this particular form of content is addressed to humans. We decide not to use any original content from the dataset on our evaluation procedure, i.e. in a task that participants are asked to judge if the content is originating from human author or a computer algorithm, as we consider this would set the bar too high. What we would like on the other hand is to get an insight on how all the techniques we apply actually influence the final result, with a two fold contribution: which of them are actually mature enough to use in production of such systems, but also which seem worth pursuing and developing further with the hope of yielding fruitful results in the future.

We draw inspiration from Manurung (2004), which mentions that for a natural language construct to be classified as poem it should satisfy three properties: grammaticality, meaningfulness and poeticness, a concept we elaborate in section 2.1.2. Xie et al. (2017) also use three concepts for human evaluators to judge the quality of their generated Shakespearean sonnets: coherence, poeticness and meter/rhyme. Unfortunately they do not elaborate further on what instructions the participants were given, or what their interpretation of these three scales are. It seems to us however that there is a loose mapping between theirs and Manurung's concepts. The first property, grammaticality, refers to the syntactical correctness of the text, which seems reasonable to match to Xie et al.'s coherence scale. Meaningfulness, as we understand it, refers to the fact that the text should convey a higher-level meaning of some kind instead of just connecting grammatically correct but nonsensical phrases. This can be loosely matched to the poeticness scale of Xie et al.. What is left is poeticness on Manurung's side and meter/rhyme on the other side. Despite the obvious name mismatch, this

is probably the clearest-cut correlations as it is defined explicitly in terms of stress patterns, rythm and rhyming on the former. We choose to use the former naming scheme, grammaticality, meaningfulness and poeticness, as we consider it is more elaborate and specific on what the judgment is based on, and keep the 3 to 5 scale used on the latter.

Chapter 4

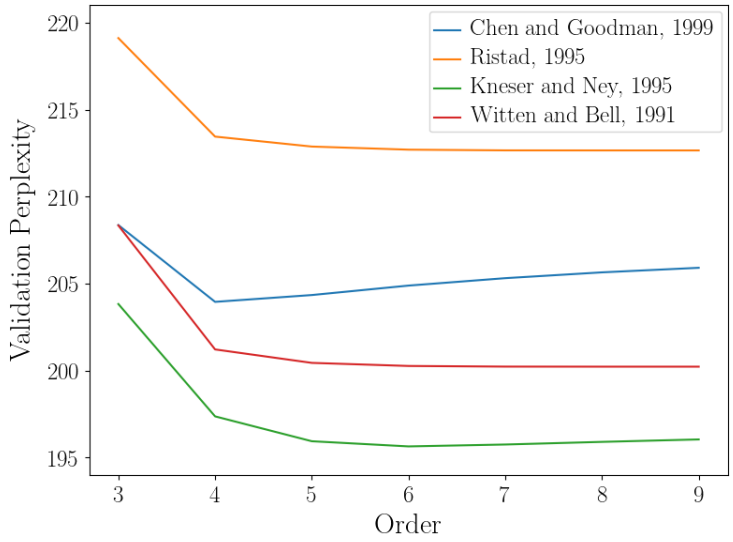
Results

4.1 Preliminary Experimentation

4.1.1 Back-off n-gram

In our experiments with back-off n-grams we found the unmodified Kneser-Ney discount Kneser and Ney (1995) performing significantly better than the other smoothing methods on all orders, as demonstrated in fig. 4.1. The lowest perplexity we got on the validation set was 195.634, using a model of order 6.

Figure 4.1: Back-off validation results for different orders and discounts

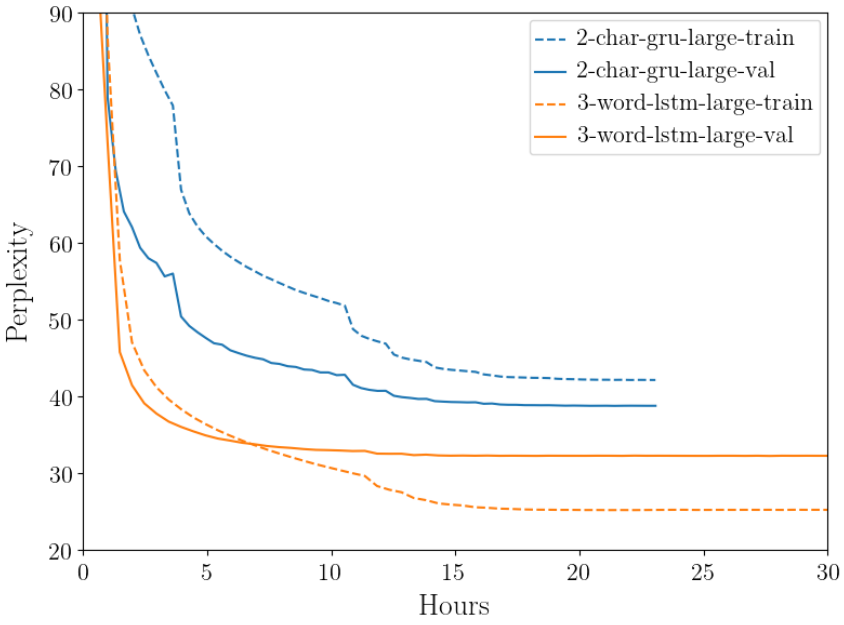


4.1.2 RNNs

General observations

As demonstrated by Xie et al. (2017), long short-term memory models seem to have an edge on verse and rhyme modelling tasks over gated recurrent units, and this is also reflected upon our results, with the long short-term memory model reaching a perplexity of 32.2 on the validation set, versus 38.7 for the best performing gated recurrent unit, as seen in fig. 4.2, where we plot the best long short-term memory model together with the best gated recurrent unit model. There are two things worth noting here: first, the gated recurrent unit model is a character-level model and does not seem to benefit from deep architectures, as the best model has 2 layers versus 3 for all the cases of long short-term memory models, illustrating its limits in learning deeper internal representations, probably stemming from its constant try to counterbalance learning new pieces of information at the expense of forgetting learned ones. Our second finding is that the gated recurrent unit model, due to its much fewer parameters, takes less time, to train. The average time per epoch for the gated recurrent unit model was 19.8 minutes versus 39 minutes for the long short-term memory model. To the model's defense, in fig. 4.3, where we plot all the character-level gated recurrent unit models versus their word-level counterparts, we can see that the character-level models indeed perform better than its word-level counterparts, proving gated recurrent models are better and more efficient at less complex tasks.

Figure 4.2: Best LSTM vs best GRU model



This does not hold true for long short-term memory models that perform better on the word level as demonstrated in the equivalent fig. 4.4. We can however see in both architectures that character-level models display a remarkable resistance to overfitting, achieving better performance on the validation than the training set throughout the training procedure, in contrast to word-level models that start to overfit quite early in their training time span.

In spite of gated recurrent units being easier to compute, at the end of the day long short-term memory units outperform them on every scenario, even in character-level and shallower 2-layer models. We conclude the first leg of our experiments selecting the 3-layer and 1024-unit model for both the character- and word-level tasks and continue with our experiments

Figure 4.3: GRU: Word- vs, Character-level

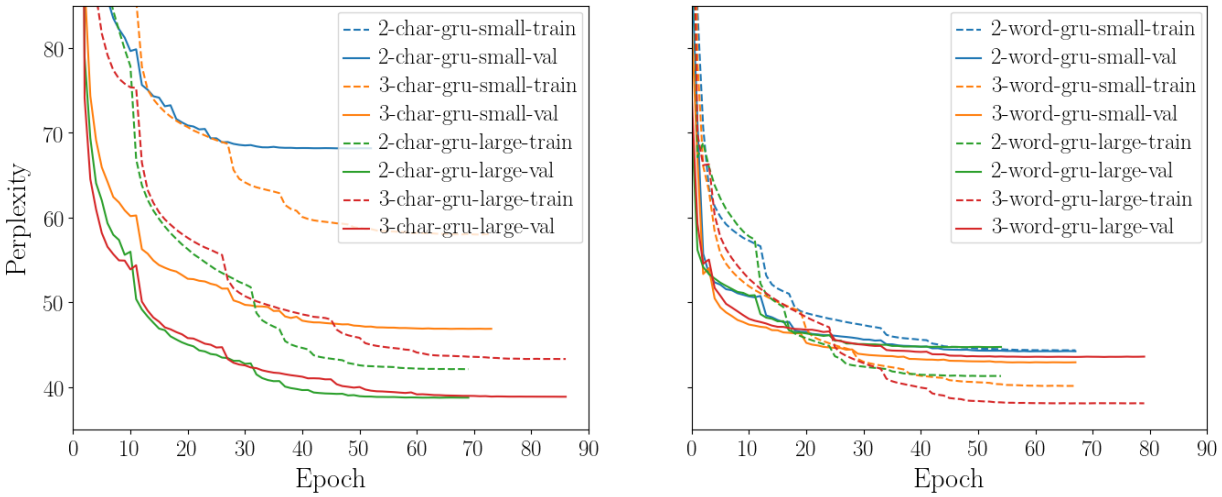
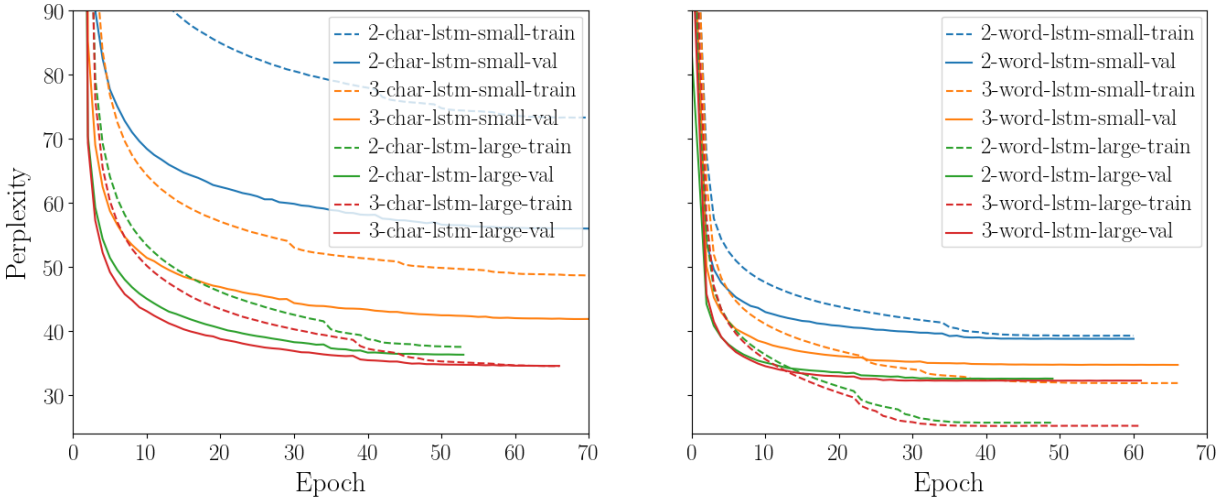


Figure 4.4: LSTM: word vs char

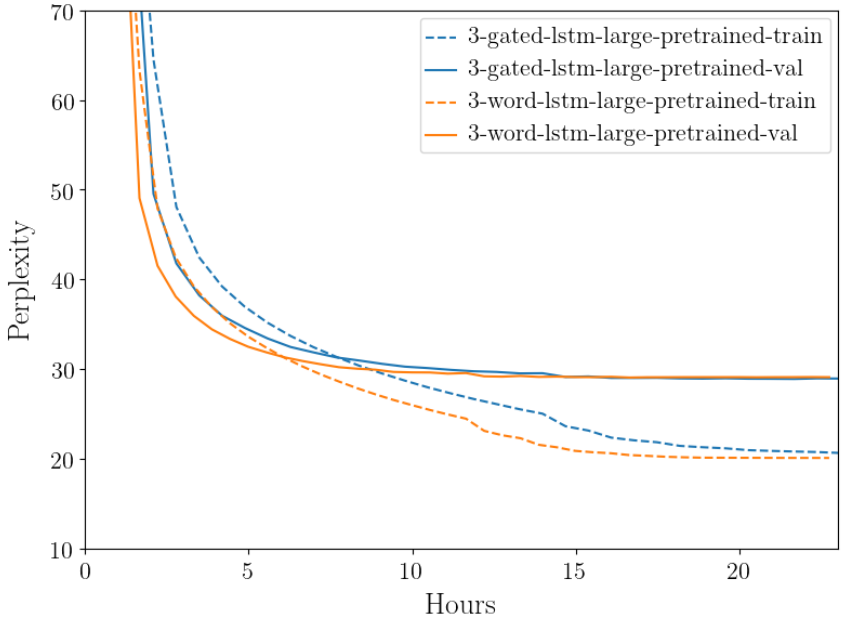


using the gated LSTM variant.

Gated LSTM

The gated LSTM that combines word- with character-level features symbolizes what it is like to do cutting-edge research. On the relevant sonnet-generating task (Xie et al., 2017), the author reports an almost 10-fold increase over the word-level long short-term memory model with the same number of units. On the other hand, while the original paper presenting the model (Miyamoto and Cho, 2016) achieves state of the art on well-established language modelling datasets, it does so by combining the two features with a gate tuned via a hyper-parameter, with the adaptive variant used by Xie et al. improving only subtly on previous methods. It comes as a disappointment that our results align with the original paper; specifically while the model gained some edge when evaluated against the validation set during training (fig. 4.5), it was eventually surpassed by the word-character model in the test set, as we will shown in section 4.2. Some speculations on the causes of this outcome are given in chapter 5.

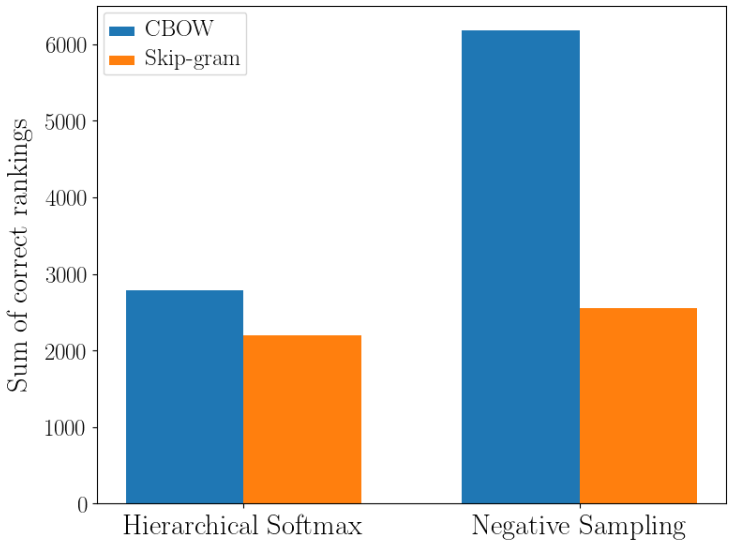
Figure 4.5: Gated vs Plain LSTM



4.1.3 Word Embeddings

We found the skip-gram word embedding model performing consistently better than the continuous bag-of-words model, though the divergence is minimized when using hierarchical softmax. Our findings are in accordance with what is known about the two models' strong and weak points, according to the original paper's author: "*Skip-gram: works well with small amount of the training data, represents well even rare words or phrases[,] CBOW: several times faster to train than the skip-gram, slightly better accuracy for the frequent*

Figure 4.6: Word embeddings validation results



words" [1], since we both have a lot of rare words and a relatively small dataset. Using the pre-trained embeddings instead of random ones yields significant improvements for our word-level RNN.In fig. 4.7 we present the training curves for the best model with random and pre-trained embeddings, respectively.

Chosen models

After our preliminary experimentation we choose the following models from each tier, based on their superior performance on the validation set:

- Word-level RNN: 3-layer, 1024-unit long short-term memory model with pretrained embeddings based on a skip-gram model trained with hierarchical softmax
- Character-level RNN: 3-layer, 1024-unit long short-term memory model
- Gated LSTM: Using the same pretrained embeddings as the word-level RNN
- Back-off n-gram: Model of order 6 with unmodified Kneser-Ney discount

4.2 Quantitative Evaluation

Perplexity

The perplexities for the selected models on the test dataset are presented in section 4.2. The low perplexity scores reveal that the neural networks have succeeded in modelling the language present in our verses. The results on the validation set match closely the ones on

[1] https://groups.google.com/forum/#!searchin/word2vec-toolkit/c-bow/word2vec-toolkit/NLvYXU99cAM/E5ld8LcDxlAJ, Last accessed: 22–07–2018

Figure 4.7: Random vs Word2Vec embeddings

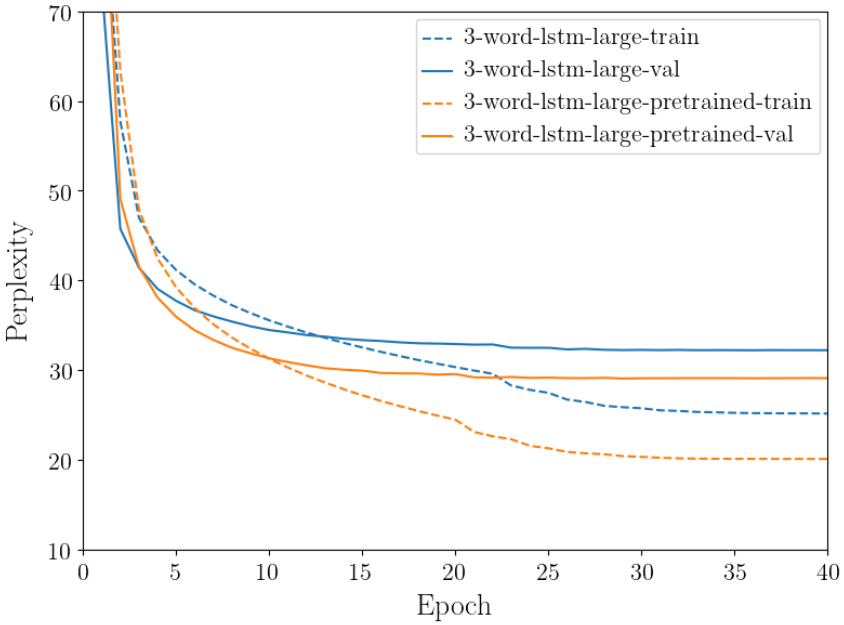
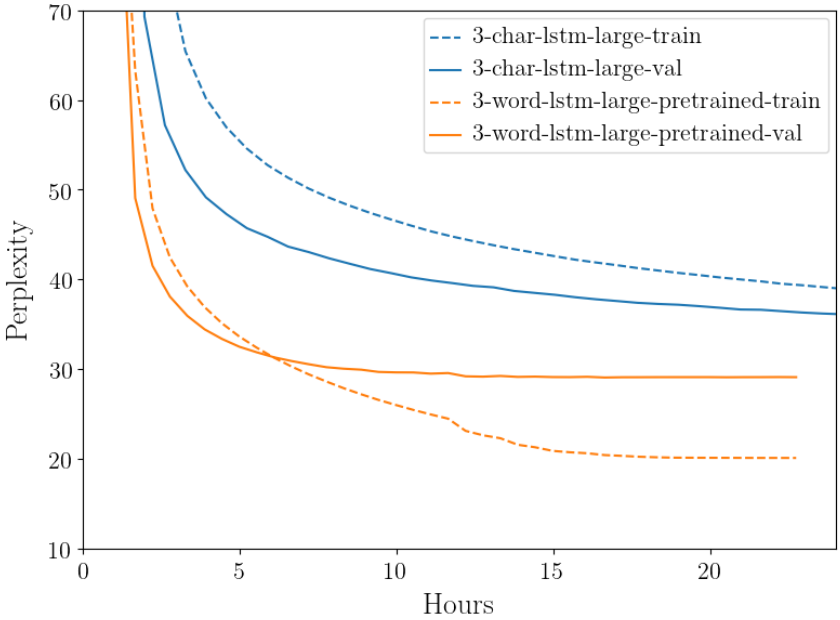


Figure 4.8: The best word and character models

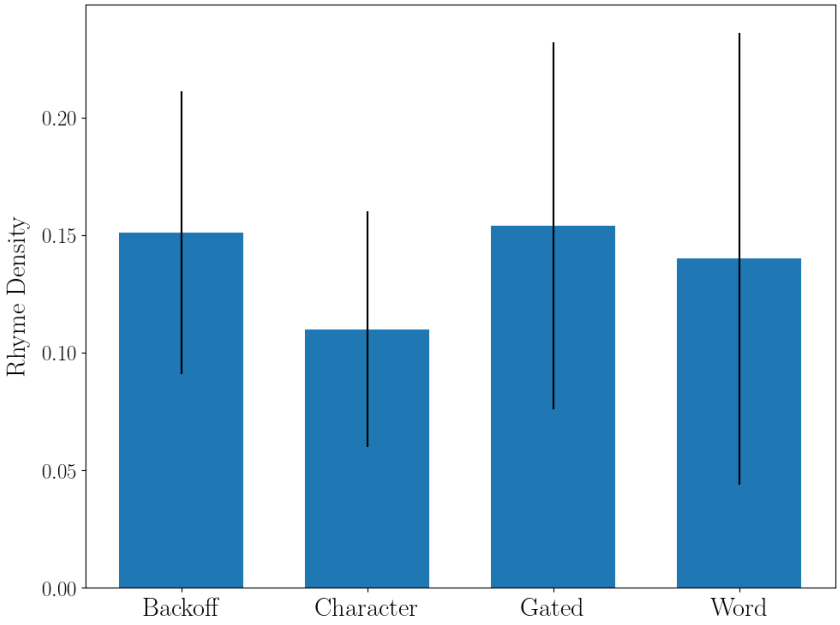


Model	Test perplexity
Back-off n-gram	196.993
Character-level RNN	34.289
Gated LSTM	29.161
Word-level RNN	**28.964**

the testing dataset, indicating we did not have any significant bias on our selection of hyper-parameters. These observations, however, do little to reveal to what extend we succeeded in our task, generating rap lyrics.

Rhyming

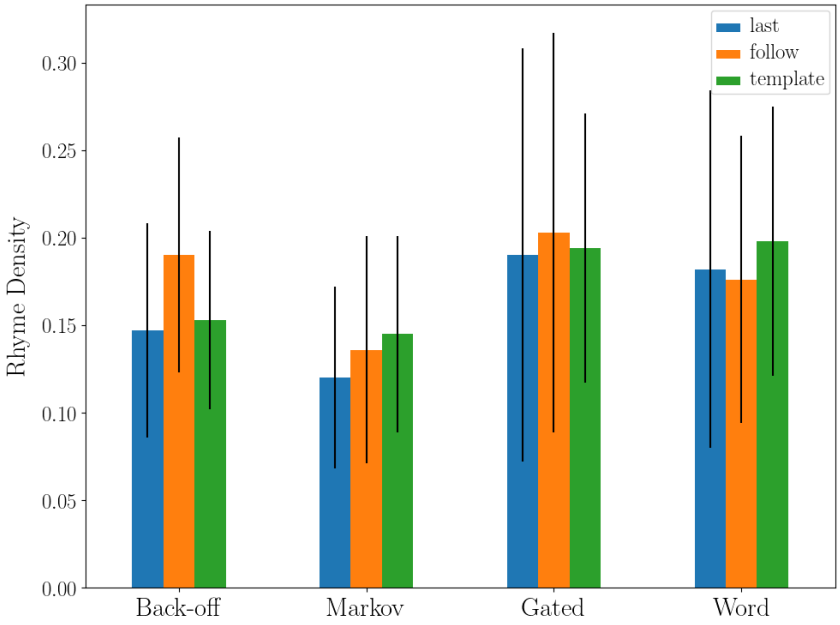
Figure 4.9: Rhyme density for unconstrained methods



In fig. 4.9 we present the rhyming performance of our models when generating verses without any constraint applied, judged by perhaps the most relevant metric, rhyme density. The back-off n-gram and the gated LSTM model look similar with respect to the density of the rhymes when producing texts unconditionally. The character-level RNN falls a little behind but also demonstrates a less variable behaviour, demonstrating a higher degree of robustness in its produced texts. The word-level RNN produced the most dense rhymes but its average generated text is not on par with the back-off n-gram and gated LSTM models.

In fig. 4.10 we can see the performance of the constrained models using different heuristic methods. Starting with the back-off n-gram model, we observe it benefits the least from forcing it to rhyme every two line pairs and using rhyme & stress templates. However, when following the stress pattern it itself produced it is able to step up a level and reliably produce denser rhyming texts. Using the templates also reduces its variance, which is reasonable given that the specific model has it difficult with reliably producing uniform lines with respect to

Figure 4.10: Rhyme density for constrained methods



length. Moving on to the constrained Markov model, each successive heuristics seems to improve the model's rhyming capabilities, but it still cannot match any of the other models, even when using their worst heuristic.

Regarding our neural networks, the gated LSTM displays some behaviour that could be characterized as unstable. While in the unconstrained task it had less variability in its output compared to the word-level RNN, applying the constraints seems to have a de-stabilizing effect. However, the same pattern as the back-off model applies here, using pre-defined templates somehow mediates the high variance. The picture we get from the word-level LSTM is similar, with the sole difference that it seems to get along a little better than the other models with the templates, increasing the lower boundary of its rhyme density variance.

Plagiarism

Of course, no benefit can be consider real unless the model is shown to produce novel rhymes. In fig. 4.11 we see the plagiarism metric for the unconstrained version of our models. The gated LSTM unfortunately looks like it has some serious problems coming up with fresh content. The rest of the models plagiarize to the same degree, which we do not consider so bad, as rap music contains a lot of slang and commonly used phrases, that 20% of common trigrams can also be called a "tribute to the original artist". However, the back-off model produces the most original content by far.

Moving on to the constrained variants, the real winner here is the template heuristic, maintaining the mediating effect demonstrated on the previous part on all models except the constrained Markov model. Since the latter does not use any backing off or smoothing, it is only natural that most of the time the sequence satisfying the constraint is exactly the sequence the template was extracted from. It is also worth noting that while both the neural network models do not plagiarize more when constrained, the back-off model, the least plagiarizing model in the unconstrained task, plagiarizes much more when any constrained

Figure 4.11: Plagiarized trigrams for unconstrained methods

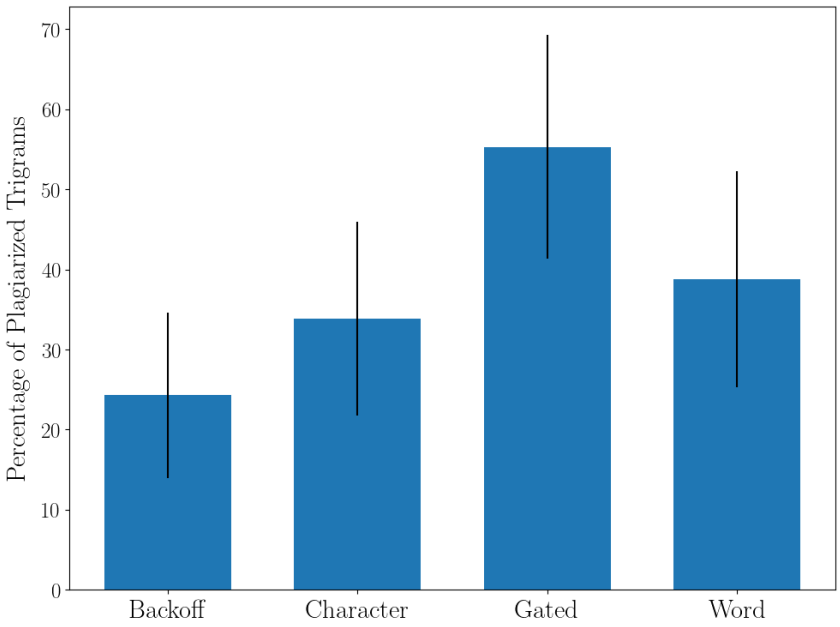
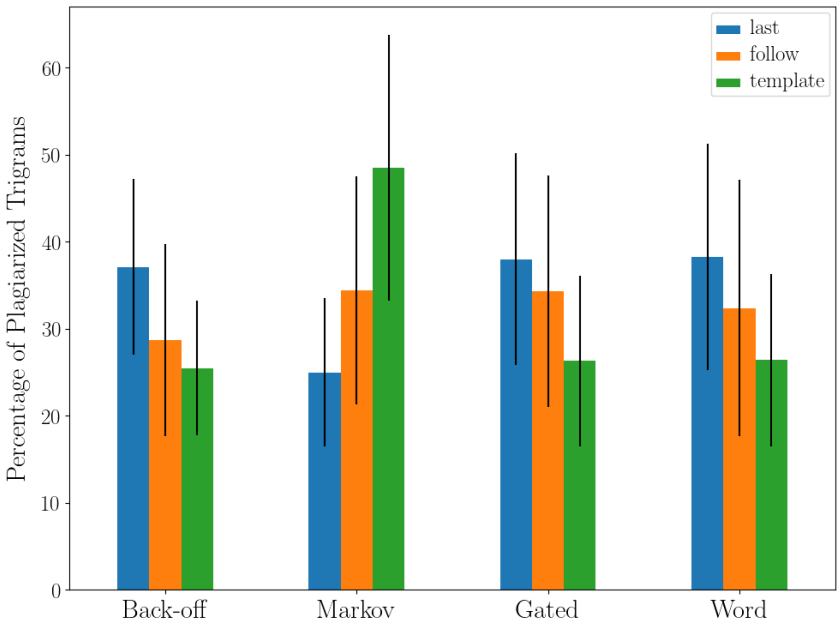


Figure 4.12: Plagiarized trigrams for constrained methods



is applied.

Conclusion

From this short quantitative analysis we can see a subtle domination of the neural networks using templates. The effect is maximized in the case of the word-level RNN model, where applying the template constraint both improves the rhyme density as well as the novelty of the produced output.

4.3 Qualitative Evaluation

We asked 9 friends to evaluate our generated verses using the metrics and scale we defined in section 3.4.2, specifically *grammaticality*, *meaningfulness* and *rhyme / meter*. All participants were between 27 and 32 years old, 2 of them would characterize themselves as hip-hop fans, and none of them would say that she is negatively preoccupied against hip-hop / rap music. None of them was a native English speaker, but all of them – being / having been international students – have a solid understanding of the English language. All the combinations are presented in table 4.1. For every combination each participant was presented with 2 random verses, judging a total. For every combination each participant was presented with 2 random samples, judging a total of 32 stanzas.

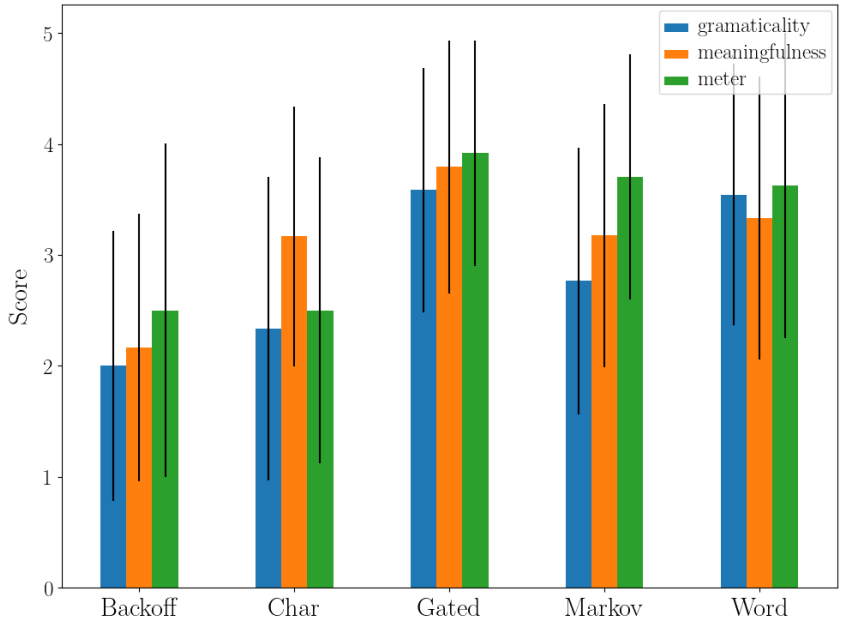
Model	Heuristic
Back-off n-gram	Unconstrained Last word rhyme Last word rhyme & Follow stress Template
Character-level RNN	Unconstrained
Word-level RNN	Unconstrained Last word rhyme Last word rhyme & Follow stress Template
Gated LSTM	Unconstrained Last word rhyme Last word rhyme & Follow stress Template
Constrained Markov Model	Last word rhyme Last word rhyme & Follow stress Template

Table 4.1: All the model and template combinations for the qualitative evaluation

The experiment took place on our location and our machine. The participants were briefed shortly on what the task is about and it was made explicit that all the samples they were about to evaluate are computer generated, as we did not want to invoke a suspicious attitude that they would be participating in a Turing-like test where we would try to persuade them that our computer-generated verses were generated by humans. For each presented stanza the participant typed in the keyboard a number from 1 to 5 followed by Enter for each metric

respectively. If the participant made an erroneous decision / changed her mind, the option of pressing Ctrl + C was given to discard all the metrics for the current stanza. Out of range values were discarded. The results were aggregated into two figures, one grouped by model(fig. 4.13) and one grouped by template(fig. 4.14). One a couple of occasions a kind of informal interview emerged, giving us insights on limitations and future work of our models and templates which we draw upon in chapter 5.

Figure 4.13: Qualitative evaluation aggregated by model



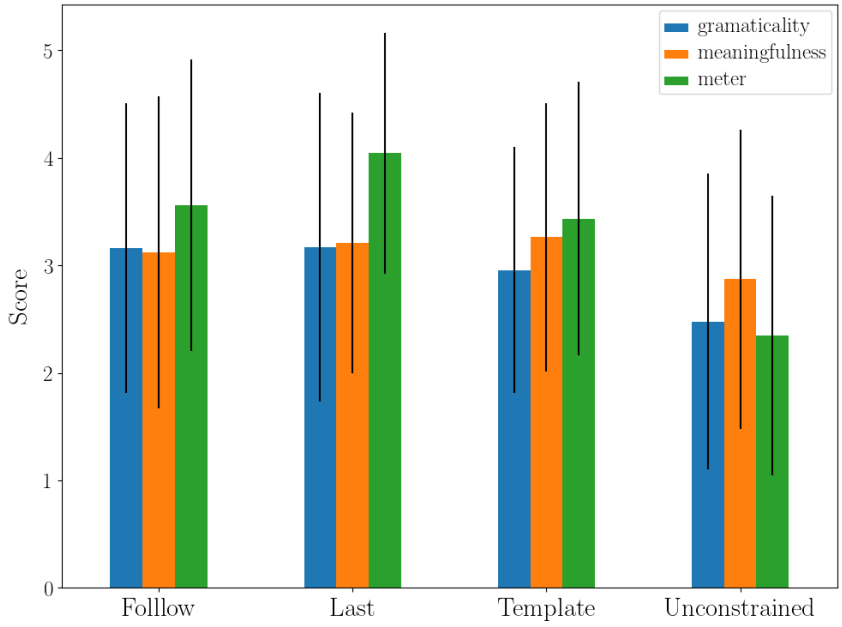
Model comparison

A quick reading of fig. 4.13 reveals a very interesting picture with a lot of space for reflection. First of all we notice a very clean pattern: the easiest property for all models (excluding the character-level RNN) to satisfy is the meter. We do not know if we should attribute it to a bias on the participants, seeing patterns where there are actually none, or whether it is actually easy for the models to satisfy the humans idea of what good rhyming and good meter means. Given that the results are aggregated over all the constraint, we go with the latter explanation, as this explains also the divergence of the character-level RNN from th observed pattern: it is the only model generating only unconstrained text. In general the gated LSTM seems to produce the most pleasing overall texts, something that supports Xie et al.'s findings. The constrained Markov model and the back-off n-gram have a similar distribution over the three qualities, revealing their common origins, as they are equivalent mathematically. The word-character RNN displays the most unique pattern, satisfying the grammaticality and meter properties but failing to communicate to the participants any remarkable underlying meaning. Here, the clear winner is the gated LSTM model, satisfying all the properties better than all the other models and with less variance.

Heuristic comparison

fig. 4.14 is equally revealing. We notice first that unconstrained generation produces

Figure 4.14: Qualitative evaluation aggregated by heuristic



output that the best it has to offer is meaning. It really comes as surprise that using any constrain not only improves the meter, which is something to expect, but also seems to improve the meaningfulness property. We believe that there is a correlation between meter, grammaticality and meaningfulness, in the sense that even the meaning is roughly equivalent, a more balanced output will increased the perceived meaning of the text, while if a reader struggles to pick up on the rhyme or the meter pattern she has a hard time actually focusing on the meaning. Also, our previous hypothesis about the character-level rnn seems to be at least partially true as the unconstrained mode displays roughly the same pattern as the character-level RNN on the model-aggregated plot.

Comparing the last with the follow heuristic, it seems that our participants judge meter mainly by the end rhymes, as when forcing word stress constraints we do not get any perceived improvement on the rythmic qualities. It could also mean, though, that the model is forced too hard to satisfy both the ending rhyme constraint and the stress pattern throughout the line that the overall rhythmical quality deteriorates.

The pre-defined template heuristic gives the most uniform improvement, as all three perceived metrics improve without particularly sacrificing any of them.

Conclusion

The take-home message from our qualitative analysis is that templates actually influence to a great degree the perceived qualities of the generated stanzas. While there is no doubt that the RNN models are very powerful at modelling natural language and just by themselves produce output that is meaningful to the human evaluators, it is the constraints that gives the edge needed to assume a more artistic form, resembling the rap lyrics we are interested to generate.

Chapter 5

Discussion and Future Work

5.1 Discussion

We finish our starting section of the dissertation by stating what we wanted to do: automatically generate rap lyrics, so let's start our finish section by asking the question: did we achieve it?

Well, despite using various heuristic methods for enhancing the rhyming abilities of the models, we require no additional input from the user, and so we claim our method to be automatic to a significant extend. But does our generated content can be classified as rap lyrics? From a technical perspective, we tried to be as elaborate as possible in a field where there was no sure path we could take, there was not a *right way* to do it as it exists on other fields that draw upon years of previous work. We examined previous work from poetry generation, which we found surprisingly similar despite the differences and implemented an array of methods in an attempt to map what was, and still is, uncharted territory. We would like to be honest, and answer that we are not sure if we achieved our goal, and we would be sure that we didn't if we hadn't included the qualitative analysis in our evaluation. But when we saw the participants reciting the generating lyrics in the rythm they made up, we believe we saw something special happening, and while we are still unsure about the answer to that question, we are sure about we why chose to do this particular research regarding generation of creative content.

Now, to some technical comments about our work. An important point we should note is the high variance of the models, sometimes generating really good output and sometimes not so. This, we believe, can be interpreted in both in a positive and in a negative way. On the positive side, it means that all of our models are able to generate content with sufficient rhyming qualities, at least sometimes. Pairing this ability with a generate-and-test approach which is widespread in poetry and other kinds of creative content generation, we already have a pipeline that yields the results we were looking for. The negative way is that our research reads more like an exploratory approach where we tried a lot of things, some of which did work, and less than a straightforward approach setting clear research questions and following up with an equivalent clarity in addressing them.

We did however, see how some simple heuristics can have a huge impact on perceived textual qualities. We also explored some of the latest deep learning models from the literature and add our little bit of knowledge in a field that previous work is really sparse.

5.1.1 Limitations

While we show how language modelling methods can be combined with heuristics to improve the perceived qualities of the generated content, that does not mean there are no limitations to our method. Our pipeline for generating rap lyrics, from content acquisition to text output, is a long chain of interconnected steps, and as every chain it is as weak as the weakest link. Starting off with our dataset, while we did our best to clean up and preprocess the sample we decided to work with, we believe there is still much non-lyrical content in the dataset that our models train on. The only way to be 100% sure that we have a clean dataset is to have a human annotator going through every and each example. As a result, from the first stage of our process, there is a weak link that propagates up to the top.

The next possibly problematic area is the extraction of our templates. While Hirjee and Brown's method truly represents the state of the art, we demonstrated early in our introduction how complex the rhyming and stress patterns of rap lyrics can get, that we need to truly rethink the whole way of representing rhymes as just words that their vowels match. Good rappers are known for complex internal rhymes and rhyming single words with whole phrases. We consider our models monolithic and quite dumb, in that sense.

Recurrent neural networks are a powerful tool and with the availability of high-performance GPUs it is easier than ever to try bigger and deeper architectures. But is this what improvement means? What we are trying to say is that the power should come out of novel ideas and novel architectures and not just bigger networks and better hardware. Considering the orders of magnitude difference between the classical statistical natural language methods and neural networks, and while not arguing that these do not actually yield better results, it makes us reflect upon if so much complexity is needed, as we already assume that it is not desirable.

Finally, the problem of automatically evaluating the quality of the suggested solution has been haunting the field of generation of creative content, and it doesn't look like it is going to be solved soon. Comparing our quantitative methods with the opinion of a handful of humans, it makes us think if all these numbers actually add any meaning or we are just in a rat race to get lower error scores and lower accuracies, actually forgetting the most important judge, that is us, ourselves.

Last but not least, we have full knowledge that we are not going to fool anyone with our generated content. There are rappers out there, there are poets out there creating novel content and expanding the horizons of creativity to domains unseen before. We cannot help but see our struggle as part of a monkey see monkey do game, until and if the vision of Artificial Intelligence comes true.

5.2 Future Work

Despite all of the above reflections, we still maintain the same excitement as the day one we started our endeavor. During the research and writing of the paper, we had many ideas that we would like to try, some being easy to implement and already available, and some being vague directions and gut feelings about which way we should move towards.

Our constraining method with regards to neural networks is a kind of a brute-force method. If we had a week more on our disposal, we would try for sure beam-search instead of just enforcing our output on the network. Beam search is a search heuristic where we do not commit to any particular output but instead explore a number of options simultaneously, each one maintaining their own probability distribution and is a vital part of many state of the art natural language processing approaches, perhaps most notable machine translation (Wu et al., 2016). In our current approach, we force the model to immediately decide

which rhyme to use, and already in the next timestep this is as solid as a rock, even though at a later context the model has the ability to realize that it was probably not the right decision. We believe that implementing our constrains as a beam search, maintaining a number of graphs in parallel and collapsing the tree only when we the model is certain enough that this actually was the right decision is something that will yield instant improvements, and we almost wish this feature was implemented in our current models.

Another possible ground to explore on is multi-task learning (Seltzer and Droppo, 2013; Collobert et al., 2011). It seems that rapping is closely related to the words auditory qualities, so integrating our models to a multi-task setting where we predict also the phonemes from the words or characters could improve also the rapping capability of the model. In the same fashion, predicting POS tags could create models that have a stronger sense of grammar, and predicting the original artist for a given verse could reduce (or increase) plagiarism.

On more theoretical grounds, we strongly believe that all we need is a little more attention. Attentive language models (along with models incorporating subword features) is what our gut feeling drives us towards. While as a concept, especially in neural networks, it something really novel and cutting edge, the first paper already look very promising(Mularczyk, 2017; **?**), and we firmly believe that they can bring a revolution in the field.

Bibliography

Gabriele Barbieri, François Pachet, Pierre Roy, and Mirko Degli Esposti. Markov constraints for generating lyrics with style. In *ECAI*, volume 242, pages 115–120, 2012. 2.2, 3.1.6, 3.2, 3.3.1

Françoise Beaufays. The neural networks behind google voice transcription, 2015. URL https://ai.googleblog.com/2015/08/the-neural-networks-behind-google-voice.html. 2.3.2.2

Margaret A Boden. Creativity and artificial intelligence. *Artificial Intelligence*, 103(1-2): 347–356, 1998. 1.1

Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394, 1999. 2.3.1, 2.3.1.1, 2.3.1.2, 3.3.1

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014. 2.3.2.3

Noam Chomsky. *Syntactic structures*. Walter de Gruyter, 2002. 2.3.1.1

D Manning Christopher, Raghavan Prabhakar, and Schutza Hinrich. Introduction to information retrieval. *An Introduction To Information Retrieval*, 151(177):5, 2008. 3.1.4

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014. 2.3.2.3

Dan CireşAn, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural networks*, 32:333–338, 2012. 2.1.2

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011. 5.2

Simon Colton, Jacob Goodwin, and Tony Veale. Full-face poetry generation. In *ICCC*, pages 95–102, 2012. 2.1.2

Peter Dale. *An introduction to rhyme*. Agenda, 1998. 1.1

Belén Díaz-Agudo, Pablo Gervás, and Pedro A González-Calero. Poetry generation in colibri. In *European Conference on Case-Based Reasoning*, pages 73–87. Springer, 2002. 2.1.2

Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990. 2.3.2.1

Pablo Gervás. An expert system for the composition of formal spanish poetry. In *Applications and Innovations in Intelligent Systems VIII*, pages 19–32. Springer, 2001. 2.1.2

Pablo Gervás. Exploring quantitative evaluations of the creativity of automatic poets. In *Workshop on Creative Systems, Approaches to Creativity in Artificial Intelligence and Cognitive Science, 15th European Conference on Artificial Intelligence*. IOS Press, Lyon, France, 2002. 2.1.2

Pablo Gervás. Computational modelling of poetry generation. In *Artificial Intelligence and Poetry Symposium, AISB Convention*, 2013. 2.1.2

Marjan Ghazvininejad, Xing Shi, Yejin Choi, and Kevin Knight. Generating topical poetry. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1183–1191, 2016. 2.1.2

Yoav Goldberg. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, 10(1):1–309, 2017. 2.1.1

Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013. 2.1.1, 2.2

Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31(5):855–868, 2009. 2.1.2, 2.3.2.2

Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013. 2.3.2.2

Erica Greene, Tugba Bodrumlu, and Kevin Knight. Automatic analysis of rhythmic poetry with applications to generation and translation. In *Proceedings of the 2010 conference on empirical methods in natural language processing*, pages 524–533. Association for Computational Linguistics, 2010. 2.1.2

Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014. 2.1.2

Charles O Hartman. *Virtual muse: experiments in computer poetry*. Wesleyan University Press, 1996. 2.1.2

Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012. 2.1.2

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 3.2

Hussein Hirjee and Daniel Brown. Using automated rhyme detection to characterize rhyming style in rap music. 2010a. 3.1.6, 3.4.1

Hussein Hirjee and Daniel G Brown. Automatic detection of internal and imperfect rhymes in rap lyrics. In *ISMIR*, pages 711–716, 2009. 3.1.6, 3.4.1

Hussein Hirjee and Daniel G Brown. Rhyme analyzer: An analysis tool for rap lyrics. In *Proceedings of the 11th International Society for Music Information Retrieval Conference*. Citeseer, 2010b. 2.2, 3.1.6, 3.4.1, 5.1.1

Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998. 2.3.2.1

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997. 2.3.2.2

Mike Swarbrick Jones. Deeprhyme (d-prime) – generating dope rhymes with deep learning, 2016. URL https://swarbrickjones.wordpress.com/2016/11/07/deeprhyme-d-prime-generating-dope-rhymes-with-deep-learning/. 2.2, 3.1.6, 3.2, 3.3.2

Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning*, pages 2342–2350, 2015. 2.3.2.1

Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015. URL http://karpathy.github.io/2015/05/21/rnn-effectiveness/. 2.1.1, 2.2, 3.3.1

Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015. 2.2

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *AAAI*, pages 2741–2749, 2016. 2.1.1, 2.1.2, 2.2

Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184. IEEE, 1995. 3.3.1, 4.1.1

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 2.1.2

William Sanford LaSor, David Allan Hubbard, Frederic William Bush, and Leslie C Allen. *Old Testament survey: the message, form, and background of the Old Testament*. Wm. B. Eerdmans Publishing, 1996. 1.1

Eric Malmi, Pyry Takala, Hannu Toivonen, Tapani Raiko, and Aristides Gionis. Dopelearning: A computational approach to rap lyrics generation. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 195–204. ACM, 2016. 2.1.3, 2.2, 3.4.1

Hisar Manurung. An evolutionary algorithm approach to poetry generation. 2004. 2.1.2, 3.4.2

Hisar Manurung, Graeme Ritchie, and Henry Thompson. A flexible integrated architecture for generating poetic texts. Technical report, The University of Edinburgh, 2000a. 2.1.2

Hisar Manurung, Graeme Ritchie, and Henry Thompson. Towards a computational model of poetry generation. Technical report, The University of Edinburgh, 2000b. 2.1.2

Heath McCoy. Rap linguistics course 'cool' but challenging, 2014. https://www.ucalgary.ca/utoday/issue/2014-09-22/rap-linguistics-course-cool-challenging, Last accessed: 2018-06-14. 1.1

Wes McKinney. pandas: a foundational python library for data analysis and statistics. 3.1.3

Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017. 2.1.1

Maria Rosa Menocal. *The Arabic role in medieval literary history: a forgotten heritage*. University of Pennsylvania Press, 2004. 1.1

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010. 2.1.2

Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5528–5531. IEEE, 2011. 2.1.2

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013. 2.3.2.4, 3.3.1

Mimino666, 2017. URL https://github.com/Mimino666/langdetect. 3.1.1

Yasumasa Miyamoto and Kyunghyun Cho. Gated word-character recurrent language model. *arXiv preprint arXiv:1606.01700*, 2016. 2.1.1, 2.1.2, 2.2, 3.2, 4.1.2

Benjamin Mularczyk. Guided attention for neural language models. Master's thesis, ETH-Zürich, 2017. 5.2

Hugo Oliveira. Automatic generation of poetry: an overview. *Universidade de Coimbra*, 2009. 2.1.2

Hugo Gonçalo Oliveira. Poetryme: a versatile platform for poetry generation. *Computational Creativity, Concept Invention, and General Intelligence*, 1:21, 2012. 2.1.2

Hugo Gonçalo Oliveira. Tra-la-lyrics 2.0: Automatic generation of song lyrics on a semantic domain. *Journal of Artificial General Intelligence*, 6(1):87–110, 2015. 2.1.2

François Pachet and Pierre Roy. Markov constraints: steerable generation of markov sequences. *Constraints*, 16(2):148–172, 2011. 2.2

François Pachet, Pierre Roy, Gabriele Barbieri, and Sony CSL Paris. Finite-length markov processes with constraints. *transition*, 6(1/3), 2001. 2.2, 2.3.1.3

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013. 2.3.2.1

Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch, 2017. URL https://pytorch.org/. 3.2

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 3.1.3

Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014. 2.3.2.4

Peter Potash, Alexey Romanov, and Anna Rumshisky. Ghostwriter: using an lstm for automatic rap lyric generation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1919–1924, 2015. 2.2, 3.3.2

Peter Potash, Alexey Romanov, and Anna Rumshisky. Evaluating creative language generation: The case of rap lyric ghostwriting. *arXiv preprint arXiv:1612.03205*, 2016. 2.2, 3.3.2

Radim Rehurek and Petr Sojka. Software framework for topic modelling with large corpora. In *In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Citeseer, 2010. 3.2

Eric Sven Ristad. A natural law of succession. *arXiv preprint cmp-lg/9508012*, 1995. 3.3.1

AJ Robinson and Frank Fallside. *The utility driven dynamic error propagation network*. University of Cambridge Department of Engineering, 1987. 2.3.2.1

Tricia Rose. *Black noise: Rap music and black culture in contemporary America*, volume 6. Wesleyan University Press Middletown, CT, 1994. 1.1

Ronald Rosenfeld. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278, 2000. 2.1.1

Alex Rudnicky. The cmu pronouncing dictionary, 2014. URL `http://www.speech.cs.cmu.edu/cgi-bin/cmudict`. 2.2

Haşim Sak, Andrew Senior, Kanishka Rao, Françoise Beaufays, and Johan Schalkwyk. Google voice search: faster and more accurate, 2015. URL `https://ai.googleblog.com/2015/09/google-voice-search-faster-and-more.html`. 2.3.2.2

Michael L Seltzer and Jasha Droppo. Multi-task learning in deep neural networks for improved phoneme recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6965–6969. IEEE, 2013. 5.2

David R Slavitt, Palmer Bovie, Smith Palmer Bovie, et al. *Aristophanes, 2: Wasps, Lysistrata, Frogs, The Sexual Congress*, volume 2. University of Pennsylvania Press, 1999. 1.1

Andreas Stolcke. Srilm-an extensible language modeling toolkit. In *Seventh international conference on spoken language processing*, 2002. 3.2

Jukka Toivanen, Matti Järvisalo, Hannu Toivonen, et al. Harnessing constraint programming for poetry composition. In *The Fourth International Conference on Computational Creativity*, 2013. 2.1.2

A.M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433, 1950. 2

Aäron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. In *SSW*, page 125, 2016. 2.1.2

Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966. 2

Ian H Witten and Timothy C Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *Ieee transactions on information theory*, 37 (4):1085–1094, 1991. 3.3.1

Dekai Wu, Karteek Addanki, Markus Saers, and Meriem Beloucif. Learning to freestyle: Hip hop challenge-response induction via transduction rule segmentation. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 102–112, 2013. 2.1.3

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016. 2.3.2.2, 5.2

Stanley Xie, Ruchir Rastogi, and Max Chang. Deep poetry: Word-level and character-level language models for shakespearean sonnet generation. 2017. 2.1.1, 2.2, 3.3.1, 3.3.2, 3.4.1, 3.4.2, 4.1.2, 4.1.2, 4.3

Tomaž Šolc, 2009. URL https://www.tablix.org/~avian/blog/archives/2009/01/unicode_transliteration_in_python/. 3.1.1

Appendix A

Lyric Samples

A random sample each presented for each heuristic and each model

A.1 Unconstrained

A.1.1 Back-off n-gram

mind my lost i
shit this drastically what lee bruce like poppin' diddy black clingers zee nice of off dumb
thief a pulled then reply listeners the to born we're a you nigga she's
limo class 2 it's whether
fuck the what hear better a boogaloo japan tawana like slicing gees coper little brother a be
wouldn't you that fuck
town the scholars with mate to prepare
face my smell mines 'cause joy the get and front
say why super disciple jav burnt a not i'm
scarface cube
newark game thru men on pavarotti risotto
job nigga's cooler body hard
on cuffs the put
smack slap smack him am 9 from screen the that's and
g 20 and seat driver's the in i'm comedians e the had purse or ignite mansion whack recoup
how step not find you'll then and pocket a got gully get blazin not 50
on go her let

A.1.2 Word-level RNN

ahead of some time
can you get one more time
and it's getting rougher
your time is sedated
you gotta keep quiet

A.1.3 Character-leel RNN

ladies and gentlemen
i got this back that liquor
trying to get all i can say and last and everything
so here's another nigga looking for what you've been tryin
and knowledge are introducing you
cause it's when you say

A.1.4 Gated LSTM

back to why i lived ta
and left my momma with dad
now they got me still strong
so i can go home
but i don't have got nothin' to believe on
trees so what there's gettin nothing's on
find me under a 747 yessir like damn

A.2 Last

A.2.1 Back-off n-gram

watch your weight
before we get the reverend all been things we all gon' bone crushin and sideways
try'na down our knees
put it on down on top of the streets
and since daddy kane and real stage
then bridge cause i'm raheim i wanna i wanna scrape

A.2.2 Constrained Markov model

no up after this here's thrown gimme another microphone
with i them shadows tang comin' in like control
roc i ain't puttin' good no signs
vagabond a think i drop the mic
yea closest squad nigga
you man starts clickin

A.2.3 Word-level RNN

but i ain't gone
when someone comes along
now my momma's on my flight
but she got home tonight
so she needs to be blowed
here we go

A.2.4 Gated LSTM

and i won't fuck around and that's on my crew how we do
ugk keeping it trill slangin snoop
keep bangin movin' like a gangsta rider
blowin cris rubberbands smoke on my tires
bottle on on the ceiling with a fifty off the flo'
leave the house and let the whole crew know

A.3 Follow

A.3.1 Back-off n-gram

a zone it never happened before new playboy being stingy
they try to gamble bondage pursued by haircuts dancing city
certainly
germany
was down to fell off his bail we unholy beastiality thats a goal
i know you gon have to take the attention symbologist on they go

A.3.2 Constrained Markov model

shawty slam thought i had what was that than dope
queensbridge hall life is bear life is a psych yo
ready black mo's
super time yo
oh kicking and enter
girl people don't question

A.3.3 Word-level RNN

ayyo aog out me off whatever this act like you four
outchea tryna get a trump newspaper on top of they thoughts
dishes off somethin' out of power
brighter day hopping in a tower
nigga got love for you nigga or what
open your eyes like a lighting of blood

A.3.4 Gated LSTM

a couple i just could quit and about six pound
but it'll be as soon as this career come 'round
have to tell you to this guy
shut the fuck up for a while
god damn god damn brick
white boy big dope grill

A.4 Template

A.4.1 Back-off n-gram

my heels kid now have visions of overstep
you can't pass the slums less all ships as we rep
knockin please don't up while you only got someone
kicked her and he know me with the top back like run
steal but i stay methodically not jocking
good i ain't hard when they tip so hit the illuminati

A.4.2 Constrained Markov model

although i throw bands bitch these hoes and like a mobile blimp
jake beep as i walk around to ride dicks
floor who put us up y'all
another them niggas don't want no war
and want poor granny offered me a beast
you they mornin of the poor is keeping

A.4.3 Word-level RNN

and i'm worse than adderall
fo' coppin a fucking sawed off
kobe rebadorchin do him so he shoot that proper
farm like wahlberg
clever whole city is too corrupt
i've been on and back my block on the cally crew hustle on cus

A.4.4 Gated LSTM

that's my niggas i rock my gun my nigga
of my niggas' my neighbourhood christmas
motherfuckin' five
death cause we just slept on the mind
i helped to put y'all muthafuckas right on line
and and i shook my fingers lickin ya backside