Requirements

The theoretical part of this post assumes some familiarity with basic Probability Theory and Machine Learning concepts and terminology. For the coding part, we will be using the R (>= 3.5.0) and the CRAN package `kgrams`:

```
if (!require(kgrams))
        install.packages("kgrams")
```

k-gram language models

Language models, sentence and word probabilities

Many tasks in *Natural Language Processing* (NLP) require, or benefit from, the introduction of a language model, that is a probability distribution over all possible sentences of a natural language. The meaning of these *sentence probabilities* can be understood through the following gedankenexperiment: let $s=w_1w_2\cdots w_l$ be a given sequence of words and denote by $\text{Pr}(s)$ its model probability. Suppose we are given a large collection of natural language text (a *corpus*) and start sampling, uniformly at random, sentences from it. Then $\text{Pr}(s)$ should model the probability for a sampled sentence s' to coincide with our test sentence s.

A slightly more straightforward, but completely equivalent, definition of a language model can be given through the notion of *word continuation probabilities*. Suppose we are given only part of a sentence, a sequence of words $c = w_1 w_2 \cdots w_l$, which we will call the *context*. Then, for any word w, the continuation probability of w in the context of c, denoted $\text{Pr}(w\vert c)$, models the probability that the actual word following c will be w. The equivalence of this formulation with the previous one can be established through the Law of Total Probability, which allows to write the probability of a sentence $s = w_1 w_2 \dots w_i$ as a product of continuation probabilities:
$$P(s) = P(w_i \vert w_1 w_2\cdots w_{i-1})\times P(w_{i-1} \vert w_1 w_2\cdots w_{i-2}) \cdots \times P(w_1\vert)$$
(here $P(w_1|)$ denotes the probability that a sentence starts with word w_1).

Before proceeding to the central topic of this post, k-gram models, I would like to add a few technical details to our current definition of language models. First of all, language models usually come together with a *dictionary*, which is a list of known words (this could be, e.g., the set of words encountered during training). Having a dictionary is actually important for dealing with *unknown* words at prediction time: any out-of-vocabulary (OOV) word, for which no statistics are available, gets mapped to the Unknown-Word token $\langle\text{UNK}\rangle$, which is included as a regular word in the dictionary. Second, an End-Of-Sentence token $\langle\text{EOS}\rangle$, which is supposed to terminate any well-formed sentence, is also included in the dictionary. This is a simple and transparent way to normalize probabilities, i.e. to enforce the requirement that $\sum _s \text{Pr}(s) = 1$, where the sum extends over all well-formed sentences of any length. Finally, k-gram models (to be introduced in a moment) often also include a Begin-Of-Sentence token $\langle\text{BOS}\rangle$ in their dictionary, and sentences are padded on the left with $N-1$ $\langle\text{BOS}\rangle$ tokens (N being the order of the k-gram model), so that sentence probabilities can always be obtained as products of N-gram continuation probabilites.

k-gram models

In k-gram language models, an important simplifying assumption is made on the continuation probabilities $\text {Pr} (w\vert c)$. It is assumed that these depend just on the last $N – 1$ words in the context, where N is called the *order* of the language model. In other words:

$$
\text{Pr}(w\vert c) = \text{Pr}(w\vert w_1 w_2 \cdots w_{N-1})\qquad (c= \cdots w_{-1}w_0 w_1 w_2 \cdots w_{N-1}).
$$

The k-tuples of words $(w_1, w_2, \dots, w_k)$ are called k-grams.

For obvious reasons, k-gram models can usually capture only *short range* word dependencies, unless N is not very large (e.g. larger than the average sentence length). In practice, N is limited by the memory required to store k-gram frequency counts, which tends to grow rapidly with N (for instance, if k-gram counts are stored in a regular hash-table, the space required for storing all counts up to $k \leq N$ is $\mathcal O (V ^ N)$ in the worst case, where V is the size of the model's dictionary). For this reason, more compact and practical models such as *neural* language models are usually preferred over k-gram models for large scale applications. Nonetheless, at smaller scales, k-gram models can often provide a simple, easy to implement and relatively interpretable baseline, and some of the more sophisticated models (such as modified Kneser-Ney) have been shown to achieve surprisingly competitive perplexities on real world corpora.

Estimating continuation probabilities

As the previous discussion should make clear, training a k-gram model boils down to estimating all continuation probabilities $\text {Pr} (w\vert c)$ for contexts of length up to $N – 1$, which are usually obtained from k-gram frequency counts observed in a training corpus. As we shall see in a moment, the sparse nature of k-gram data requires the introduction of *smoothing techniques*, which, loosely speaking, take off some probability mass from frequent k-grams and put this mass on rarer k-grams, thus flattening the whole k-gram probability distribution.

The Maximum Likelihood Estimate for the continuation probability $\text {Pr} (w\vert c)$ is defined by:

$$
\hat{P}_\text{MLE}(w\vert w_1 w_2 \cdots w_\ell) = \frac{C(w_1 w_2\cdots w_\ell w)}{C(w_1 w_2\cdots w_\ell)}.
$$

This definition follows quite naturally from the definition of conditional probability and converges, in the limit of infinite data, to the true continuation probability $\text {Pr} (w\vert c)$. It suffers, however, from two serious issues:

1. k-gram data is sparse: even with a very large training corpus, there will be
   a large number of k-grams which were never observed, and the MLE

incorrectly underestimates their probabilities to zero (and, by consequence, overestimates the probabilities of observed k-grams).
2. If the context $w_1w_2 \dots w_{\ell}$ itself was not observed in the training corpus, the MLE is not defined (both numerator and denominator are zero).

As anticipated above, the solution to both these problems is to *smooth* out the k-gram empirical distribution provided by the training corpus. Chapter 3 of Jurafsky's and Martin's free book on NLP provides a good starting point for learning about these techniques. For the rest of this post, we will use the well known Interpolated Modified Kneser-Ney method, which is considered the state-of-art of k-gram models. Without entering into a technical description of the model, which you can read about in Jurafsky's and Martin's book, this model supplants missing information for high-order k-grams by incorporating lower-order models information, through the concept of generalization probabilities. The algorithm is fully implemented in `kgrams`, so that I won't get into the mathematics of modified Kneser-Ney, but I encourage the interested reader to delve into these details in the suggested reference.

Evaluating language models

To conclude our theoretical introduction, we discuss a popular method for evaluating language models. In general, model evaluation has a two-fold purpose:

- Model *assessment*: estimating the performance of a predictive model.
- Model *selection*: choosing the best out of several different models. This includes also tuning the parameters of a given model.

For both use cases, it is clear that one has in mind a specific *evaluation metric*, which depends in general on the specific end-to-end task the model was built for (e.g. if the language model is used for text autocompletion, a sensible metric could be the binary accuracy of next word predictions).

In spite of this, language model evaluations often employ an intrinsic, task-independent metric called perplexity. On the one hand, perplexity is often found to correlate positively with task-specific metrics; moreover, it is a useful tool for making generic performance comparisons, without any specific language model task in mind.

Perplexity is given by $P = e^H$, where H is the cross-entropy of the language model sentence probability distribution against a test corpus empirical distribution:

$$
H = – \dfrac{1}{W}\sum _s\ \ln (\text {Prob}(s))
$$

Here W is total number of words in the test corpus (we include counts of the EOS token, but not the BOS token, in W), and the sum extends over all sentences in the test corpus. Again, perplexity does not give direct information on the performance of a language model in a specific end-to-end task, but is often found to correlate with the latter, which provides a

practical justification for the use of this metric. Notice that better models are associated with lower perplexities, and that \(H\) is proportional to the negative log-likelihood of the corpus under the language model assumption.

Training a \(k\)-gram model in R

We now move on to the fun part of this post: I will explain you how to train your own \(k\)-gram model in R, using the package `kgrams`. I developed this package with the primary purpose of making small experiments with \(k\)-gram models and to get a grasp about the quantitative and qualitative features of various smoothing methods, either with standard metrics such as perplexity or within an unsupervised task such as random text generation. You can install the released version of `kgrams` from CRAN, using:

```
install.packages("kgrams")
```

So, let us load the package:

```
library(kgrams)
```

Getting the corpus

The first step in training a \(k\)-gram model is, of course, to choose a training corpus. We will take our training corpus from the Folger Shakespeare collection, which offers a convenient query API. We will be using queries of the form https://www.folgerdigitaltexts.org/PLAYCODE/text, which return only the spoken text from a play. The list of available plays, with their corresponding API codes, is the following:

```
playcodes <- c(
        "All's Well That Ends Well" = "AWW",
        "Antony and Cleopatra" = "Ant",
        "As You Like It" = "AYL",
        "The Comedy of Errors" = "Err",
        "Coriolanus" = "Cor",
        "Cymbeline" = "Cym",
        "Hamlet" = "Ham",
        "Henry IV, Part 1" = "1H4",
        "Henry IV, Part 2" = "2H4",
        "Henry V" = "H5",
        "Henry VI, Part 1" = "1H6",
        "Henry VI, Part 2" = "2H6",
        "Henry VI, Part 3" = "3H6",
        "Henry VIII" = "H8",
        "Julius Caesar" = "JC",
        "King John" = "Jn",
        "King Lear" = "Lr",
        "Love's Labor's Lost" = "LLL",
        "Macbeth" = "Mac",
        "Measure for Measure" = "MM",
        "The Merchant of Venice" = "MV",
        "The Merry Wives of Windsor" = "Wiv",
        "A Midsummer Night's Dream" = "MND",
        "Much Ado About Nothing" = "Ado",
```

```
        "Othello" = "Oth",
        "Pericles" = "Per",
        "Richard II" = "R2",
        "Richard III" = "R3",
        "Romeo and Juliet" = "Rom",
        "The Taming of the Shrew" = "Shr",
        "The Tempest" = "Tmp",
        "Timon of Athens" = "Tim",
        "Titus Andronicus" = "Tit",
        "Troilus and Cressida" = "Tro",
        "Twelfth Night" = "TN",
        "Two Gentlemen of Verona" = "TGV",
        "Two Noble Kinsmen" = "TNK",
        "The Winter's Tale" = "WT"
        )
```

We can access, for instance, the text of "Much Ado About Nothing" by opening an R connection as follows (notice that each line is terminated by a line break html tag):

```
get_url_con <- function(playcode) {
        stopifnot(playcode %in% playcodes)
        url <- paste0("https://www.folgerdigitaltexts.org/",
playcode, "/text")
        con <- url(url)
}

con <- get_url_con("Ado")
open(con)
readLines(con, 10)
##  [1] "
"
##  [2] "I learn in this letter that Don
"
##  [3] "Pedro of Aragon comes this night to Messina.
"
##  [4] "He is very near by this. He was not three
"
##  [5] "leagues off when I left him.
"
##  [6] "How many gentlemen have you lost in this
"
##  [7] "action?
"
##  [8] "But few of any sort, and none of name.
"
##  [9] "A victory is twice itself when the achiever
"
## [10] "brings home full numbers. I find here that Don
"

close(con)
```

`kgrams` allows to train language models from out of memory sources of text, such as the url connections we created above. We will use in the followng the function `get_url_con()` to retrieve the text used in our training. We will take all Shakespeare's plays but the “Hamlet” as our training corpus, keeping the latter aside for model evaluations:

```
train_playcodes <- playcodes[names(playcodes) !=
c("Hamlet")]
test_playcodes <- playcodes[names(playcodes) == c("Hamlet")]
```

Text preprocessing and sentence tokenization

It is usually a good idea to apply some transformation to the training corpus, before feeding it to the \(k\)-gram tokenization algorithm. In the following, we will apply the following text preprocessing function, which leverages on the `kgrams::preprocess()` utility and applies some additional transformation required by our specific case:

```
.preprocess <- function(x) {
        # Remove html tags
        x <- gsub("<[^>]+>", "", x)
        # Lower-case and remove characters not alphanumeric
or punctuation
        x <- kgrams::preprocess(x)
        return(x)
}
```

In `kgrams`, you also need to explicitly specify a function used for sentence tokenization (i.e. where should the \(\langle \text{EOS} \rangle\) and \(\langle \text{BOS} \rangle\) tokens be placed in the text). This function should return a character vector, each component of which corresponds to a single sentence. We will use:

```
.tknz_sent <- function(x) {
        # Collapse everything to a single string
        x <- paste(x, collapse = " ")
        # Tokenize sentences
        x <- kgrams::tknz_sent(x, keep_first = TRUE)
        # Remove empty sentences
        x <- x[x != ""]
        return(x)
}
```

This function leverages on the `kgrams::tknz_sent()` utility, and splits sentences in correspondence of any of the punctuation characters “.!?:;”. The argument `keep_first = TRUE` instructs the function to append the first punctuation element terminating a sentence at the end of the tokenized string, separated by a space. In this way, punctuation characters are treated as regular word tokens (this is a simple way to teach the model to distinguish between affirmations, exclamations, questions, etc.).

Extracting \(k\)-gram frequency counts

The second step in the training process is to extract k-gram frequency counts from the training corpus. We will store counts for k-grams of order up to:

```
N <- 5
```

This can be done using the function `kgram_freqs()` and `process_sentences()` from `kgrams`. We use the first function to initialize a new k-gram frequency table of order N:

```
freqs <- kgram_freqs(N, .preprocess = .preprocess,
.tknz_sent = .tknz_sent)
summary(freqs)
## A k-gram frequency table.
##
## Parameters:
## * N: 5
## * V: 0
##
## Number of words in training corpus:
## * W: 0
##
## Number of distinct k-grams with positive counts:
## * 1-grams:0
## * 2-grams:0
## * 3-grams:0
## * 4-grams:0
## * 5-grams:0
```

Here V is the size of the model's dictionary, which was created on the run from all observed words in the training corpus (we could also have used a prefixed dictionary). Notice that the `.preprocess()` and `.tknz_sent()` functions we created above were passed as arguments to `kgram_freqs()`; the k-gram tokenization algorithm will automatically apply this functions before processing k-gram counts.

We can now obtain k-gram counts from Shakespeare as follows:

```
lapply(train_playcodes,
       function(playcode) {
               con <- get_url_con(playcode)
               process_sentences(text = con, freqs = freqs,
verbose = FALSE)
       })
```

The function `process_sentences()` can take as `text` input either a character vector or a connection; here we repeatedly call `process_sentences()` on the connections to the various texts from Shakespeare. Notice that, contrary to many R functions, the function `process_sentences` modifies the object `freqs` *in place*, i.e. without making a copy.

Let us make a few k-gram count queries as a minimal sanity check:

```
query(freqs, c("leonato", "pound of flesh", "smartphones"))

## [1] 23  6  0
```

Looks reasonable, does it?

Building the language model

Finally, to build a language model, we need to choose a smoothing technique. The list of smoothing algorithms available in `kgrams` can be obtained through:

```
smoothers()

## [1] "ml"    "add_k" "abs"   "kn"    "mkn"   "sbo"   "wb"
```

We will use `mkn`, which is the Interpolated Modified Kneser-Ney algorithm described in the previous section. We can get some basic information on this algorithm through:

```
info("mkn")
## Interpolated modified Kneser-Ney
##  * code: 'mkn'
##  * parameters: D1, D2, D3
##  * constraints: 0 <= Di <= 1
```

As it can be seen, the algorithm requires three parameters $D_{1,2,3}$, which correspond to the values of the discount function $D(x)$ for $x = 1, 2$ and $x \geq 3$ respectively used in the Kneser-Ney algorithm.

We can build a `mkn` language model as follows:

```
model <- language_model(freqs, smoother = "mkn", D1 = 0.5,
D2 = 0.5, D3 = 0.5)
summary(model)
## A k-gram language model.
##
## Smoother:
## * 'mkn'.
##
## Parameters:
## * N: 5
## * V: 27133
## * D1: 0.5
## * D2: 0.5
## * D3: 0.5
##
## Number of words in training corpus:
## * W: 955351
##
## Number of distinct k-grams with positive counts:
## * 1-grams:27135
## * 2-grams:296764
```

```
## * 3-grams:631166
## * 4-grams:767564
## * 5-grams:800543
```

The parameters specified here can be modified in any moment with, e.g.:

```
param(model, "D1") <- 0.25
parameters(model)
## $N
## [1] 5
##
## $V
## [1] 27133
##
## $D1
## [1] 0.25
##
## $D2
## [1] 0.5
##
## $D3
## [1] 0.5

param(model, "D1") <- 0.5
```

We can also tell the model to use only information from \(k\)-grams of order \(M < N\), for instance:

```
param(model, "N") <- N - 1
parameters(model)
## $N
## [1] 4
##
## $V
## [1] 27133
##
## $D1
## [1] 0.5
##
## $D2
## [1] 0.5
##
## $D3
## [1] 0.5

param(model, "N") <- N
```

The model can be used to compute sentence probabilities:

```
sentences <- c(
        "I have a letter from monsieur Berowne to one lady
Rosaline.",
        "I have an email from monsieur Valerio to one lady
```

```
Judit."
)
probability(sentences, model)
## [1] 2.407755e-06 3.768191e-40
```

or continuation probability:

```
context <- "pound of"
words <- c("flesh", "bananas")
probability(words %|% context, model)

## [1] 3.930320e-01 5.866405e-08
```

Evaluating and tuning \(k\)-gram models

In this section we use the perplexity metric introduced previously to tune the discount parameters \(D_{1,2,3}\) of our modified Kneser-Ney model. We use as test set the “Hamlet”, which we had left out from the training corpus; recall:

```
test_playcodes
## Hamlet
##  "Ham"
```

To compute the perplexity of our current model on the “Hamlet”, we use:

```
con <- get_url_con(test_playcodes)
perplexity(text = con, model = model)
## [1] 328.5286
```

It is worth to note that the `perplexity()` function uses, by default, the same preprocessing and sentence tokenization transformations applied during model training (this behaviour can be overridden through the `.preprocess` and `.tknz_sent` arguments). In order to get meaningful comparisons, one should always apply the same transformations before perplexity computations for different models.

To select the optimal parameters for our language model, we compute perplexity over a grid of \(D_{1,2,3}\) values and keep the parameters yielding the lower perplexity. We first download the text of the “Hamlet” for efficiency:

```
con <- get_url_con(test_playcodes)
hamlet <- readLines(con)
close(con)
```

Then, we build a grid of \(D_{1,2,3}\) values and compute perplexity for each of these, choosing the parameters yielding the best (lowest) perplexity.

```
tune <- function(D1_grid, D2_grid, D3_grid) {
        res <- list(D1 = 0,  D2 = 0, D3 = 0, perplexity = Inf)
        for (D1 in D1_grid)
                for (D2 in D2_grid)
                        for (D3 in D3_grid) {
                                param(model, "D1") <- D1
                                param(model, "D2") <- D2
                                param(model, "D3") <- D3
```

```
                                perplexity <- perplexity(hamlet, model)
                                if (perplexity < res$perplexity)
                                        res <- list(D1 = D1,
                                                    D2 = D2,
                                                    D3 = D3,
                                                    perplexity =
perplexity)
                        }
        return(res)
}
```

We start with a loose grid:

```
D1_grid <- D2_grid <- D3_grid <- seq(from = 0.5, to = 1.0, by = 0.1)
par <- tune(D1_grid, D2_grid, D3_grid)
par
## $D1
## [1] 0.9
##
## $D2
## [1] 1
##
## $D3
## [1] 1
##
## $perplexity
## [1] 220.6921
```

We can then proceed to fine tune our model using a smaller grid around our current best values:

```
D1_grid <- c(0.88, 0.89, 0.90, 0.91, 0.92)
D2_grid <- D3_grid <- c(0.96, 0.97, 0.98, 0.99, 1.00)
par <- tune(D1_grid, D2_grid, D3_grid)
par
## $D1
## [1] 0.92
##
## $D2
## [1] 1
##
## $D3
## [1] 1
##
## $perplexity
## [1] 219.6056
```

We could repeat (or, better, automate) this process as many times as we want, to search in finer and finer grids, until perplexity stops improving within a certain threshold. Finally, we tune our model with our best values for its parameters:

```
param(model, "D1") <- par$D1
param(model, "D2") <- par$D2
param(model, "D3") <- par$D3
```

In a similar fashion, we can also tune the model k-gram order N:

```
perplexities <- numeric(N)
for (i in 1:N) {
        param(model, "N") <- i
        perplexities[[i]] <- perplexity(hamlet, model)
}
perplexities
## [1] 594.5800 240.6682 219.7559 218.9458 219.6056
```

We see that the 4-gram model is actually performing better (in terms of perplexity) than the 5-gram one, which might be an indication that the higher order models are starting to overfit. For the rest of this post we will use the 4-gram model:

```
param(model, "N") <- 4
```

Generating random text with k-gram models

If you made it up to this point, you know how to train and tune a k-gram language model. Congratulations! As a well deserved reward, we will now use our language model to generate some random Shakespeare-inspired text!

Formally, generating "random" text means to sample from the language model sentence distribution. This sampling can be performed sequentially (i.e. word by word) using the continuation probabilities of the sentence generated so far. In `kgrams`, the relevant function is `sample_sentences()`, which works as follows:

```
set.seed(840)
sample_sentences(model, 10, max_length = 20)
##  [1] "hum ! "
##  [2] "helen to hide them . "
##  [3] "and the rest . "
##  [4] "what is this kindness look palamon signs patiently . "
##  [5] "ill prove it . "
##  [6] "whats here ? "
##  [7] "kneel ? "
##  [8] "kill my court . "
##  [9] "therefore lord for thy tale of these fair maid . "
## [10] "thou wouldst do nothing of a gentleman i did at saint without
the help of devils mytilene return . "
```

Not as good as Shakespeare, but that's a nice start!

An interesting twist in this game presents itself when we introduce a *temperature* parameter. The temperature transformation of a probability distribution is defined by:

$$p_t(i) = \dfrac{\exp(\log{p(i)} / t)} {Z(t)},$$

where $Z(t)$ is the partition function, defined in such a way that $\sum _i p_t(i) \equiv 1$. Notice that, for $t = 1$, we get back the original probability distribution. Intuitively, $t>1$ and $t<1$ temperatures make a probability distribution smoother and rougher, respectively. By making a physical analogy, we can think of less probable words as states with higher energies, and the effect of higher (lower) temperatures is to make more (less) likely to excite these high energy states.

We can introduce temperature in our sampling procedure through the `t` parameter of `sample_sentences()`, which applies a temperature transform to all word continuation probabilities. Here are some experiments:

```
set.seed(841)
sample_sentences(model, 10, max_length = 20) # Normal temperature
##  [1] "thou go along by him yet . "
##  [2] "o that grieve hung their first and whose youth lies in your
child away . "
##  [3] "i am glad i came he could not but that i pity henrys death my
creditors cocksure together . "
##  [4] "verily i swear tis better to greet the besiege like never shut
up a cannon puddle ear . "
##  [5] "the pride of happy but these manner thee for a quarrel . "
##  [6] "i fear thy the watch felt a fellow all was supposed ; "
##  [7] "but who comes here ? "
##  [8] "if thou darst nestorlike bereft ; "
##  [9] "i can produce a course which is already . "
## [10] "tis the petty goaded have on t . "

sample_sentences(model, 10, max_length = 20, t = 10) # High temperature
##  [1] "bellypinchd careers horsemans needy divinely exits calendars
id benevolence plumd sadhearted eaux level league perverse resolve
accouterments luggage amort cherishes [...] (truncated output)"
##  [2] "venom shouldnotwithstanding doomsday swell elseof aloft
furrowweeds dercetus pitythey nutshell poll scorpions presents pericles
scythes placeth potent drooping botcher perversely [...] (truncated
output)"
##  [3] "body ulcerous circumstance whispers sightless reliances
parricides pragging piglike oneandtwenty illfaced apparel biggen
masteri counterfeit uncivil vouchsafed unforced planks sag [...]
(truncated output)"
##  [4] "sweet holdeth cocklight uproar eclipses bastardizing cojoin
antonioo stricken disloyal almain forerun reverted gothe prone branched
spleeny towards upon siri [...] (truncated output)"
##  [5] "ruminat bareheaded mightiness cassius fortress kingas fearhow
dogged counts atwain overtopping thrall learned greediness robbers
loftyplumd hidst commix hereditary ignorance [...] (truncated output)"
##  [6] "wax mildly blench trade gild threwst goal art cloudcapped
onion mun gottst concerns performs picture writer claims close leopard
waxes [...] (truncated output)"
##  [7] "puissance salework sweets brut gravity brazenface becomd
beastliness moist lucrece belief center noces brunt malls welcome
pantingly does fragments popish [...] (truncated output)"
##  [8] "forges mariana french lioness loudhowling commonwealth
commends chapter importunity scared unsettled unreasonably beeves eases
twicetold sworder greeks rump archer gorse [...] (truncated output)"
##  [9] "godly chidst utterd doe didst profferer woodville sins
speediness honeyless altogether panel fittest bretons fount ordure
katherine correcting cushions arcites [...] (truncated output)"
## [10] "disguise pasties rochester raise rain bunchbacked highness
harrow wreakful bursting heartsorrowing softest chosen margareton exegi
```

```
confounding manchild ionian thither distained [...] (truncated output)"

sample_sentences(model, 10, max_length = 20, t = 0.1) # Low temperature
##  [1] "i am not in the world . "
##  [2] "i will not be entreated . "
##  [3] "i am a gentleman . "
##  [4] "i am not . "
##  [5] "i am not in the world . "
##  [6] "i am not in the world . "
##  [7] "i am not in the world . "
##  [8] "i am not . "
##  [9] "i am not . "
## [10] "i am not in the world . "
```

As already explained, sampling at low temperatures gives much more weight to probable sentences, and indeed the output is very repetitive. On the contrary, high temperatures make sentence probabilities more uniform, and our output above looks very random.

Conclusions

This post was a brief introduction to the theory of language models and \(k\)-gram models in particular, and explained how to train, tune and predict with \(k\)-gram models in R, using the package `kgrams`.

For a more comprehensive introduction to \(k\)-gram models, I suggest Chapter 3 of Jurafsky's and Martin's free book on NLP. For a more in-depth analysis of \(k\)-gram models, see the seminal work by Chen and Goodman. If you want to learn about Natural Language Processing from a deep learning point of view, the Sequence Models course from Andrew Ng's Coursera Deep Learning specialization is a good starting point.