# *Assignment 1*
# 3-gram Language models

Mourad Akandouch
INFOY004
Academic year: 2017-2018
ULB - VUB

## TABLE OF CONTENTS

Section 1

## *Tasks we had to do*

The aim of our first assignment was to write language models based on a training set. After that, regarding the generated language models, we had to detect in which varieties of English some sentences are written.

The classification was based on *perplexity.* The language models we had to build are based on trigrams letter and the training set consists of a corpus of three varieties of English: British, Australian and American.

In addition, we had to generate a random output based on the generated language models for each varieties of English. Those generated strings have a fixed length $k$. In this report, I will put a string of length $k = 200$ as mentioned in the assignment.

After that, we had to score each sentence in the test set with each of our generated language models.

One important thing to mention is that we had to “clean” the training set by removing all non-alphabetical symbols and replacing all spaces by two underscores. I also added one underscore at the beginning and ending of the training text. I did so because the underscore’s role on our assignment is the detection of the beginning/ending of a word. A very easy way to clean the text is simply using regular expression.

Section 2

# Results

In this section, I will first discuss about the way I cleaned the text. Then, I will put a random output generated by each language models. One interesting thing is that the output generates words that *could* be in the English vocabulary. After that, I will put an excerpt of the language model for each of the varieties of English.

Finally, I will give the computed perplexity of each test sentence for each language models.

## Cleaning the text

First, I removed all whitespaces by replacing them by two underscores and I also lowered the case of each characters of the string. For instance, if the string is *“John ate all my Leonidas chocolate bars!”*, then it will becomes:

*“john__ate__all__my__leonidas__chocolate__bars__!”*

After that, I have removed all non-alphabetical symbol with the following regular expression

```
[^_a-z]
```

In addition, I have added one underscore at the beginning and the end of the string, which give us the following string after the cleaning:

*“_john__ate__all__my__leonidas__chocolate__bars___”*

The two additional underscores are not part of the assignment but it is a personal additional assumption. I did so because the first letter of the first word of the text has to be a candidate for the first word that I randomly generate. When I generate a random output, I choose the first trigram from those whose starting letters are at least one underscore.

## Generation of a random output for each language model

In order to generate a random output from a language model, I followed the “algorithm” from the slides. The Shannon’s model. I first generate a trigram that starts with at least one underscore, then, iteratively, I generate a letter based on the two previous generated word and according to the apparition’ likelihood that each trigram carries. For instance, the trigram *“the”* is more likely to appear than *“thz”*.

For the **Australian** English, my language model generated the following output:

***“_ull__aborge___call__ren__eilam___dies__foraire__thand__of__sesed__se__japhesider___debrouts___by__hatigthe___neddecitions__ative__of__lemed__eald__packad__wek__orebamen__paaw__st__cor___maket___bildrap”***

Beautiful, isn’t it? The random output generated words that are already in the English corpus while some others *could* by the way they sound when we read them. I wanted to remove all the underscores and replace them by spaces but the character counts will therefore not sum up to $k = 200$. However, here is the same text, without underscores:

*“ull aborge call ren eilam dies foraire thand of sesed se japhesider debrouts by hatigthe neddecitions ative of lemed eald packad wek orebamen paaw st cor maket bildrap”*

For the **British** English, here is what my language model generated:

***“_youbland___onat_____ith___havent___th___exiscurnew___thastorded___ sim___a___the_____min___or___thicut___cliese___and_______me___ressab ould_______s___to___thereachughtriniq_exas___and___worke___attioncif_ ___we___of___creeked___examand”***

Finally, here is what does looks like a random output with the **American** English language model:

***“_zezed___is___is___forsel___the___trults___ableartisto___trougargor___ the___in___buto___pertual_____raeopere___truall___en___man___wits___ acen___bou___examerhatiated___to___ebal___her___bovick___wit___cone xpothe_______that___ca_____on”***

We can see that there are more spaces. Maybe due to the different *values* used in my *Add-k* smoothing.

## Excerpt of the language models

In my project, I generate the language models in the same folder where the training set is. Here is an excerpt for the bigram *“iz”* in **British** English:

| Trigram starting by “*iz*” | Likelihood (British English) |
|---|---:|
| *A* | 0.18125643666323377 |
| *B* | 0.0010298661174047373 |
| *C* | 0.0010298661174047373 |
| *D* | 0.003089598352214212 |
| *E* | 0.5612770339855818 |
| *F* | 0.0010298661174047373 |
| *G* | 0.0020597322348094747 |
| *H* | 0.003089598352214212 |
| *I* | 0.04737384140061792 |
| *J* | 0.0010298661174047373 |
| *K* | 0.0010298661174047373 |
| *L* | 0.0010298661174047373 |
| *M* | 0.0010298661174047373 |
| *N* | 0.0010298661174047373 |
| *O* | 0.03913491246138002 |
| *P* | 0.0010298661174047373 |
| *Q* | 0.0010298661174047373 |
| *R* | 0.0010298661174047373 |
| *S* | 0.0020597322348094747 |
| *T* | 0.0010298661174047373 |
| *U* | 0.015447991761071062 |
| *V* | 0.0010298661174047373 |
| *W* | 0.0010298661174047373 |
| *X* | 0.0010298661174047373 |
| *Y* | 0.0020597322348094747 |
| *Z* | 0.042224510813594233 |
| _ | 0.029866117404737384 |

We can see that the vowels have a higher probability of apparition than the other letters. It would sound weird if it were not the case though. In addition, there is no zeros because I have smoothed the counts with an *add-k* smoothing with $k = 1.2$.

The following table shows the same excerpt but for **American** English:

| Trigram starting by "*iz*" | Likelihood (American English) |
|---|---:|
| *A* | 0.20625 |
| *B* | 0.0014423076923076924 |
| *C* | 0.0009615384615384616 |
| *D* | 0.0009615384615384616 |
| *E* | 0.5649038461538461 |
| *F* | 0.0009615384615384616 |
| *G* | 0.0009615384615384616 |
| *H* | 0.0014423076923076924 |
| *I* | 0.07067307692307692 |
| *J* | 0.0009615384615384616 |
| *K* | 0.0009615384615384616 |
| *L* | 0.0009615384615384616 |
| *M* | 0.002403846153846154 |
| *N* | 0.0014423076923076924 |
| *O* | 0.03125 |
| *P* | 0.0009615384615384616 |
| *Q* | 0.0009615384615384616 |
| *R* | 0.0009615384615384616 |
| *S* | 0.0009615384615384616 |
| *T* | 0.0014423076923076924 |
| *U* | 0.012980769230769231 |
| *V* | 0.0009615384615384616 |
| *W* | 0.0009615384615384616 |
| *X* | 0.0009615384615384616 |
| *Y* | 0.0009615384615384616 |
| *Z* | 0.022596153846153846 |
| _ | 0.016826923076923076 |

The results differ a little from the previous excerpt, but it is worth to mention that my $k$ is different for the British and American. Here, $k = 1.8$ but even with a different value, the vowels are approximatively the same. My $k$'s are different because I wanted to smooth the results in order to have the best prediction against the test set. I will discuss more about this in the corresponding section.

## Score for the test set

Here is a table showing the perplexity score for each test sentence and for each language model. I highlight the best (min) result for each sentence. A green highlight means a correct prediction and a red one means a bad prediction.

| Variety | Sentence # | GB's perplexity | AU's perplexity | US' perplexity |
|---|---|---|---|---|
| AU | 1 | 6.3306 | 6.1526 | 6.3094 |
| AU | 2 | 6.4762 | 6.2701 | 6.4581 |
| AU | 3 | 7.0007 | 6.7841 | 6.8412 |
| GB | 4 | 5.3015 | 5.4509 | 5.8185 |
| GB | 5 | 6.0157 | 6.2074 | 6.0260 |
| GB | 6 | 5.7486 | 5.7876 | 5.7909 |
| US | 7 | 5.3189 | 5.6103 | 5.2716 |
| US | 8 | 5.9902 | 6.0240 | 5.9890 |
| US | 9 | 5.1371 | 5.3915 | 5.1740 |

Surprisingly, the language model can even guess the variety of English but not in all cases. The last sentence were not been well classified. Despite it was an US sentence, the British language model had the least perplexity. When we look at the British and American's perplexity, we remark that there is nearly no difference. The most remarkable is the second last sentence where GB's perplexity is 5.99 while the US' is 5.989! That is, 0.001 of difference! Furthermore, I think that the chosen *k* of my smoothing plays a great role in those values.

Globally, American language model classify well the sentence in *100%* of the cases. For the Australian language model, its precision is also *100%*. Nevertheless, for the British one, there was one error. Therefore, on nine sentences, 8/9 had a "good" perplexity.

Section 3

# Discussion

Here we discuss a little bit more on what I did and why. Furthermore, I answer to the questions asked in the assignment.

**Question 1** – *Why is it useful to replace single whitespaces between words by a double underscore?*

Because by doing it that way, we can detect which letter is at the beginning and at the end of a word. In addition, when we generate a random output, one can simply look at the trigram starting by at least one underscore in order to pick one of those at random.

**Question 2** – *Do you need to run the test set on all three language models or is the score from a single variety model sufficient?*

Yes, we need to do so in order to choose the language model that has the smallest perplexity and therefore predict which variety the sentences come from.

**Question 3** – *Would a unigram or bigram language model work as well? Explain why (not).*

I think it would still work for bigrams but extremely less well than a trigram language model. For unigram, it will simply not work. Because when we decrease the $N$ in n-grams, the probabilities in the language model become less contextual. They depend on a smaller amount of previous grams. Therefore, the lesser $N$ becomes, the less the prediction is accurate.

**Question 4** – *Do the language models show anything about similarity of the varieties? Why (not)?*

Not really because the perplexities of each language models are nearly the same. However, statistically, I remark that some trigrams are more frequent in some varieties than in others. For instance, the trigram *"the"* has a probability of 0.67 in Australian English and 0.64 in American English.

*Do Americans say more the trigram "the" than Australians do?* I think that we need a bigger training set in order to confirm that!

**Question 5** – *Can you think of a better way to make a language variety guesser?*

May be if we include a lot of idioms or figure of speech that only exist in the corresponding variety and by considering words instead of letters, we should be able to have a better guess.

## Section 4

## Appendix

Here is the code of the assignment. However, in order to make it work, one must respect the folder hierarchy of the project. Because the random output are exported in another folder and I take the training set from a specific folder. The running working assignment is available on Github on the following link:

https://github.com/mourad1081/natural-language-processing/tree/master/assignments/assignment 1

There is two files. The main running file and a class representing a language model.

Concerning the implementation, I use a 2D dictionary. The first dimension stores all the possible bigrams the second stores all the letters that follow a given bigram. I store the counts/probability in the second dimension as value for the letter (which is the key). For instance, here is an excerpt of the 2D dictionary for the Australian language model:

```
{
    "qg": {
        "h": 0.012345679012345678,
        "i": 0.012345679012345678,
        "e": 0.012345679012345678,
        "o": 0.012345679012345678,
        "z": 0.012345679012345678,
        "m": 0.012345679012345678,
        "j": 0.012345679012345678,
        "l": 0.012345679012345678,
        "c": 0.012345679012345678,
        "d": 0.012345679012345678,
        "_": 0.012345679012345678,
        "k": 0.012345679012345678,
        "f": 0.012345679012345678,
        "g": 0.012345679012345678,
        "v": 0.012345679012345678,
        "u": 0.012345679012345678,
        "p": 0.012345679012345678,
        "x": 0.012345679012345678,
        "a": 0.012345679012345678,
        "t": 0.012345679012345678,
        "q": 0.012345679012345678,
        "y": 0.012345679012345678,
        "w": 0.012345679012345678,
        "r": 0.012345679012345678,
        "s": 0.012345679012345678,
        "n": 0.012345679012345678,
        "b": 0.012345679012345678
    },
    "ko": {
        "h": 0.02231237322515213,
        "i": 0.008113590263691683,
        "e": 0.008113590263691683,
        "o": 0.05273833671399594,
        "z": 0.006085192697768763,
        "m": 0.014198782961460446,
        "j": 0.016227180527383367,
        "l": 0.028397565922920892,
        "c": 0.008113590263691683,
        "d": 0.010141987829614604,
        "_": 0.034482758620689655,
        "k": 0.008113590263691683,
        "f": 0.016227180527383367,
        "g": 0.04056795131845842,
        "v": 0.014198782961460446,
        "u": 0.04665314401622718,
        "p": 0.016227180527383367,
        "x": 0.004056795131845842,
        "a": 0.02028397565922921,
        "t": 0.016227180527383367,
        "q": 0.004056795131845842,
        "y": 0.004056795131845842,
        "w": 0.014198782961460446,
        "r": 0.23732251521298176,
        "s": 0.028397565922920892,
        "n": 0.09127789046653144,
        "b": 0.010141987829614604
    }
…
}
```

There is another 2D dictionary storing the counts in the same manner.

For the smoothing, I have use an *Add-k* smoothing in order to tune a bit more the language models and not just add 1 to all counts. You can see in the code below the value I gave for each language model.

Now, the code. Here is the main file:

## App.py

```python
from src.LanguageModel import LanguageModel, ANSI


def generate_language_model(path):
    print(ANSI.header, "Training", path, "corpus")
    print(" --------------------------------------------", ANSI.endc)
    lm = LanguageModel(path)

    lm.generate_trigrams_counts()

    k = 2 if path.endswith("AU") else 1.2 if path.endswith("GB") else 1.8
    lm.add_k_smoothing(k=k)
    lm.maximum_likelihood()

    lm.export()
    lm.generate_random_output(length=200, export_to_file=True)
    print("\n")
    return lm


"""
Python-like "main function"
"""
if __name__ == '__main__':
    training_set = [
        "../training_set/training.GB",
        "../training_set/training.AU",
        "../training_set/training.US"
    ]

    test_set = ["../test_set/test"]

    # We create the LM's
    language_models = []
    for corpus in training_set:
        language_models.append(generate_language_model(corpus))

    # We classify the test set
    for test in test_set:
        file = open(test, 'r')
        perplexities = {}
        i = 0
        for line in file.readlines():
            perplexities['line ' + str(i)] = {}
            print(ANSI.ok_blue, "Line to classify (results below):")
            print("➤➤➤", line[:50] + '...', ANSI.endc)
            print(" ➤➤➤ Results:")
            print(" -----------")
            for lmodel in language_models:
                perplexities['line ' + str(i)][lmodel.name] = lmodel.get_perplexity_from(line)
                print(" ➤➤➤ Perplexity of", lmodel.name, "=", perplexities['line ' +
str(i)][lmodel.name])

            x = min(perplexities['line ' + str(i)].keys(), key=(lambda k: perplexities['line ' +
str(i)][k]))
            print(ANSI.header, '★★★★★★★★ Best result:', x, '★★★★★★★★★★', ANSI.endc)
            i += 1

        file.close()
```

And the language model code:

## LanguageModel.py

```python
import os
import random
import re as regexp
import json
from decimal import Decimal
from operator import itemgetter


class LanguageModel:

    def __init__(self, path_file):
        """
        Creates an object that can modelize a language
        given the path to a corpus.
        :param path_file Path to the file to process.
        """
        self.path_file = path_file
        file = open(path_file, "r", encoding="utf-8")
        self.text = "".join(file.readlines())
        self.name = os.path.basename(path_file)
        self.vocabulary = None
        self.trigrams_count = None
        self.trigrams_normalized = None
        self.k_smoothed = False
        self.k = 0
        self.preprocessing_text()
        self.generate_vocabulary()

        file.close()

    def preprocessing_text(self):
        """
        Cleans the text without modifying the input file.
        The function transforms all non-char characters
        to double underscores.
        """
        print(' ➤➤➤ Cleaning text...', end='', flush=True)
        self.text = regexp.sub(" ", "__", self.text.lower(), flags=regexp.MULTILINE)
        self.text = "_" + regexp.sub("[^_a-zA-Z]", "", self.text, flags=regexp.MULTILINE) + "_"
        print(ANSI.ok_green, 'OK ✓', ANSI.endc)

    def export(self):
        """
        Exports the current state of the model in a file.
        """
        if self.trigrams_normalized is not None:
            with open(self.path_file + "__language_model.json", 'w') as file:
                json.dump(self.trigrams_normalized, file, indent=4)
            print(ANSI.bold, "➤➤➤ Language model successfully generated.", ANSI.endc)
            print("     Path to language model: ", ANSI.ok_green, self.path_file +
"__language_model.json", ANSI.endc)

    def generate_vocabulary(self):
        """
        Generates the vocabulary of the text.
        """
        print(' ➤➤➤ Generating vocabulary...', end='', flush=True)
        self.vocabulary = {}
        for letter in self.text:
            if letter in self.vocabulary.keys():
                self.vocabulary[letter] += 1
            else:
                self.vocabulary[letter] = 1
        print(ANSI.ok_green, 'OK ✓', ANSI.endc)

    def generate_trigrams_counts(self):
        """
        Counts all letter 3-grams
        """
        print(' ➤➤➤ Generating trigram counts...', end='', flush=True)
        # We create a 2D matrix : column = P(w_i | w_i-2, w_i-1)
        self.trigrams_count = self.generate_all_trigrams()
        start, end, i = 0, 2, 2

        while i < len(self.text):
            preceding_bigram = self.text[start:end]
            current_letter = self.text[i]

            self.trigrams_count[preceding_bigram][current_letter] += 1

            start += 1
            end += 1
            i += 1
        print(ANSI.ok_green, 'OK ✓', ANSI.endc)

    def maximum_likelihood(self):
        """
        Transform the matrix to a language model.
        """
        print(' ➤➤➤ Generating trigram probabilities...', end='', flush=True)
        self.trigrams_normalized = self.generate_all_trigrams()
        v = len(self.vocabulary)
        for bigram, following_letters in self.trigrams_count.items():
            for letter, count in following_letters.items():
                if self.k_smoothed:
                    denominator = sum([cpt + self.k for key, cpt in following_letters.items()])
                    probability = following_letters[letter] / (denominator + (self.k * v))
                else:
                    denominator = sum([cpt for k, cpt in following_letters.items()])
                    probability = following_letters[letter] / denominator

                self.trigrams_normalized[bigram][letter] = probability
        print(ANSI.ok_green, 'OK ✓', ANSI.endc)

    def add_k_smoothing(self, k=1):
        """
        Smoothes de language model by adding k to each counts.
        :param k: Amount to add to each count.
        """
        for letter, preceding_bigrams in self.trigrams_count.items():
            for preceding_bigram, count in preceding_bigrams.items():
                preceding_bigrams[preceding_bigram] += k
        self.k_smoothed = True
        self.k = k

    def generate_random_output(self, length=300, export_to_file=False):
        """
        Generates a random output according
        to the generated language model.
        :param export_to_file: True if you want to export the result into a file.
        :param length: Length k (3 < k < 300), according to its probabilistic model
        """
        # Step 1. We choose a random trigram starting with one or two underscores.
        w = random.choice([w for w in self.trigrams_normalized.keys() if w.startswith('_')])
        text = w
        for i in range(length):
            # Step 2. for a given length k, now choose
            #         a random bigram (w, x) according to its probability
            current_bigram = self.trigrams_normalized[w]
            possible_letters = [(k, v) for k, v in current_bigram.items()]
            text += self.generate_random_from(possible_letters)
            w = text[-2:]
            # And so on, until the string reaches the desired length

        # text = text.replace("__", " ").replace("_", "")
        print(ANSI.bold, "➤➤➤ Random output for the language model:", ANSI.endc)
        print("    ", text)
        if export_to_file:
            path = "random_output_" + os.path.basename(self.path_file)
            print(ANSI.bold, "➤➤➤ Random output successully exported.", ANSI.endc)
            print("     Path to random output: ", ANSI.ok_green, path, ANSI.endc)
            with open(path, 'w') as file:
                file.write(text)

        return text

    def get_perplexity_from(self, text):
        """
        Gives the perplexity of a text regarding the current language model.
        :param text: the text to evaluate
        :return: The perplexity of a text according to the current language model.
        :rtype: float
        """
        # [3:] because the first chars are metadatas
        txt = self.preprocess_text(text[3:])
        # Computation of the perplexity
        start, end = 0, 2
        index_current_letter = 2
        p = Decimal(1.0)
        while index_current_letter < len(txt):
            p *=
Decimal(Decimal(1.0)/Decimal(self.trigrams_normalized[txt[start:end]][txt[index_current_letter]]))
            start += 1
            end += 1
            index_current_letter += 1

        res = Decimal(p ** Decimal(1/len(txt)))
        return res

    @staticmethod
    def generate_all_trigrams():
        """
        Generates all possible bigram
        """
        z = {}
        alphabet = "abcdefghijklmnopqrstuvwxyz_"
        for first_letter in alphabet:
            for second_letter in alphabet:
                z[first_letter + second_letter] = {}
                for third_letter in alphabet:
                    z[first_letter + second_letter][third_letter] = 0
        return z

    @staticmethod
    def generate_random_from(possible_letters):
        """
        Generates a random
        :parameter possible_letters: The list of (letter, corresponding proba).
        """
        # As the probabilities does not sum to 1... I think, our random will not be
        # in the range 0...1, but 0... sum{probas of the third letters for a given bigram}

        # The sort here is mandatory in order to
        # facilitate the election of the letter
        possible_letters.sort(key=itemgetter(1), reverse=True)
        r = random.uniform(0, sum([item[1] for item in possible_letters]))
        cpt = 0.0
        for item in possible_letters:
            cpt += item[1]
            if r < cpt:
                return item[0]

        # The following instructions should never be reachable
        # but in case of doubt, I put them anyway.
        print(ANSI.fail, "It seems that I could not generate a random letter", ANSI.endc)
        return random.choice(possible_letters)[0]

    @staticmethod
    def preprocess_text(text):
        """
        Cleans a text.
        :parameter text: the text to clean
        """
        text = regexp.sub(" ", "__", text.lower(), flags=regexp.MULTILINE)
        return "_" + regexp.sub("[^_a-zA-Z]", "", text, flags=regexp.MULTILINE) + "_"


# Just for pretty printings
class ANSI:
    header = '\033[95m'
    ok_blue = '\033[94m'
    ok_green = '\033[92m'
    warning = '\033[93m'
    fail = '\033[91m'
    endc = '\033[0m'
    bold = '\033[1m'
    underline = '\033[4m'
```