NLP Assignment 1
3-Grams Language Model

Alberto Parravicini

1 Introduction

The goal of the assigment is to build **N-grams** language models from given *corpora* of text, and use them to **generate** new text and to **classify** existing sentences, i.e. say, for a given sentence, which language model most closely resembles the sentence.

Text generation and classification are common challanges in **Natural language processing**, and in this report it is presented one of the simplest and most intuitive ways to accomplish such tasks.

The report is structured as follows: first, it is presented a brief introduction on **N-grams** models; then, it is explained how the input text is *preprocessed*, before being used to build a language model or classified.
After the preprocessing, it is shown how the **N-grams** language models was built. The fourth section is about **Text generation**, followed by a section that explains how to **classify** sentences.
Each of these sections is introduced by a theoretical background, followed by the details of the implementation.
The report is concluded by a summary of the main results obtained, and a discussion of how the language models could be furtherly improved.

2 Introduction to character-level N-grams models

In any language, words and characters never occurs in a completely random fashion. Given a certain word, some words will be more likely to follow it compared to others. It is also reasonable that if we consider a sequence of more words, we will shrink even more the set of words that could come next.
The same line of reasoning applies to single characters too: in the English language, the letter **w** is much more likely to be followed by **a**, then by **k**.

Now, how can we compute how *likely* is **w** to be followed by **a**?
This measure can be interpreted as the probability of encountering **a** after **w**, or in other words, $P(a|w)$.
Using the definition of *conditional probability*, $P(a|w) = \frac{P(a,w)}{P(w)}$. These quantities can be estimated with a good degree of confidence, given a large corpus, by counting the number of times where **aw** and **w** appear, divided by the total number of characters in the corpus.
In short, we can estimate

$$P(a|w) \approx \frac{count(a,w)}{count(w)}$$

These sequences of characters are called **N-grams**, where **N** refers to their length (**aw** is a *2-gram*, or *bigram*).

We could think of considering more characters in the past, and look, for instance, at the probability of encountering **w** after **a** and **l** (as in *lawyer*), or at the probability of encountering **w** after an even longer sequence of characters, possibly even infinite, from a theoretical point of view.
However, this approach presents two main drawbacks:

- Keeping track of the combinations of all possible characters is very expensive from a computational point of view, with a size and computation time that grow exponentially to the size of the past characters we consider.
- By considering very long sequences, our model will not be able to generalize: in short, it will be able to reproduce existing content that occurs commonly in the corpus, but it will fail at generating novel sequences.
  When looking at sequences of words, this means that our model is likely to produce sentences and expressions that are commonly found in the training corpus; if we consider character n-grams, our model is quite likely to reproduce very common words, instead of syllables.

Moreover, it is unlikely that characters in the far past will have any significant influence on the current ones! The latter assumption is formalized by the so-called **Markov's property**:

$$P(c_1, c_2, ..., c_n) \approx \prod_{i=1}^{n} P(c_i|c_{i-1}, c_{i-2}, ..., c_{i-k}))$$

where k is the number of past characters we consider.
In our case, we will work with **3-Grams**.

One might ask why consider characters instead of words when building the **N-Grams**. The computational cost is once again a major reason (as the number of charactes is much smaller than the number of words).
Moreover, looking at sequences of characters seems to work well for **document classification**, as characters N-Grams are more robust to misspelt words and can easily identify patters typical of certain languages [1].

3 Text Preprocessing

To effectively compute the probabilities of encountering each **N-Gram**, it is required to have a large text corpus. However, such corpora are noisy or unpolished, in the sense that the text they contain cannot be immediately used to build a model, but must be preprocessed in a way that the model can successfully exploit the information the corpus contains.

In our case, we are given a set of sentences which contains punctuation, bold characters, digits, and other symbols.
It is reasonable to assume that some of these characters do not drastically influence our language model, and their removal will significantly speed up the model generation and the subsequent text classification.

It should be also noted that the text classification will aim at discerning between 3 different variants of English, namely British, American and Australian English. They follow similar rules with respect to punctuation, digits, and upper-case letters, and this fact was kept into account in the preprocessing of the corpus

The following operations have been done on the input corpus:

- All words are converted to lower-case. As mentioned, all variants of English do not have large differences in the usage of upper-cased letters.
- Accented letters have been replaced to their closest match (e.g. **ù** becomes **u**). Even tough English doesn't use many special characters, it could still be useful to convert them to standard letters, if possible, as to preserve the structure of syllables and words.
- Any non-alphabetic character that isn't a white-space is removed. This includes digits, punctuations, parentheses, and other special characters. As explained, this will reduce the computational cost without hindering the quality of the model in significant ways. Removing apostrophes and similar symbols won't affect the quality of the language classification, as the use of this symbols is very similar in every variant of English.

- All the sentences are joined into one single sentence. This won't affect the results of the model, but it will greatly improve the speed of the training.
- White-spaces are substituted with double underscores "__": this is done because we want to separate different words, and make sure that the probability of encountering a character at the start of a word isn't influenced by the characters at the end of the previous one (we need 2 underscores as in *trigrams* we consider the previous 2 characters).
- Double underscores are added also at the start and at the end of the overall sentence. Adding them is useful so that we don't need to evaluate the probabilities of encountering characters at the start and at the end of a sentence in a special way, but we can reuse the probabilities of encountering __**X** and **X**__, respectively (**X** represents an arbitrary character).

4 3-Grams Model Building

In order to construct a **3-Grams** (or *trigram*) language model, it is required to compute the number of occurrences of all possible 3 combinations of character in the alphabet. In our case, we consider as alphabet the letters from **a** to **z**, plus _. There are multiple ways to store the count of occurrences, for any trigram. One could think of using an hash-table in which each trigram is associated to its number of occurrences. However, it is much more efficient to use a 3-D tensor, in which the $[i, j, k]$ element corresponds to the number of occurrences of the trigram ijk.
Note that this approach would scale to any N-gram, as *numpy* offers support for tensors of arbitrary dimensionality.
To compute trigrams occurrences in an efficient way, it is first generated a set containing every trigram that appears in the corpus.
Then we count the occurrences of each of these trigrams. This approach allows to skip trigrams that do not appear in the corpus, which improves by a significant amount the computational speed.

Then, another 3-D tensor is created, in which the $[i, j, k]$ is the empirical probability of finding letter k after letters i, j. This value is computed by dividing the occurrences count of ijk by the occurrences count of ij, which can be rapidly computed by slicing the tensor across the axis i, j.

Note that **Laplace smoothing** is applied to the occurrences count, i.e. a value of 1 is added to the count of each trigram. This is used to deal with trigrams and bigrams which never appears in the entire corpus, to prevent division by 0 in the computation of the empirical probabilities and in the computation of perplexity.

To get an insight of how trigrams are distributed, it is possible to visualize the sparsity of one of the 3-D tensor used to count the occurrences.
As an example, it is visualized the sparsity of the tensor that represents British English.
Here, for each character k of our vocabulary(*a-z* plus _), it is displayed a matrix where each coordinate i, j represents the trigram i, j, k. Blue dots correspond to non-zero occurrences.
There are few trigrams ending with uncommon letters such as **w** or **x**, while many words end with vowels.

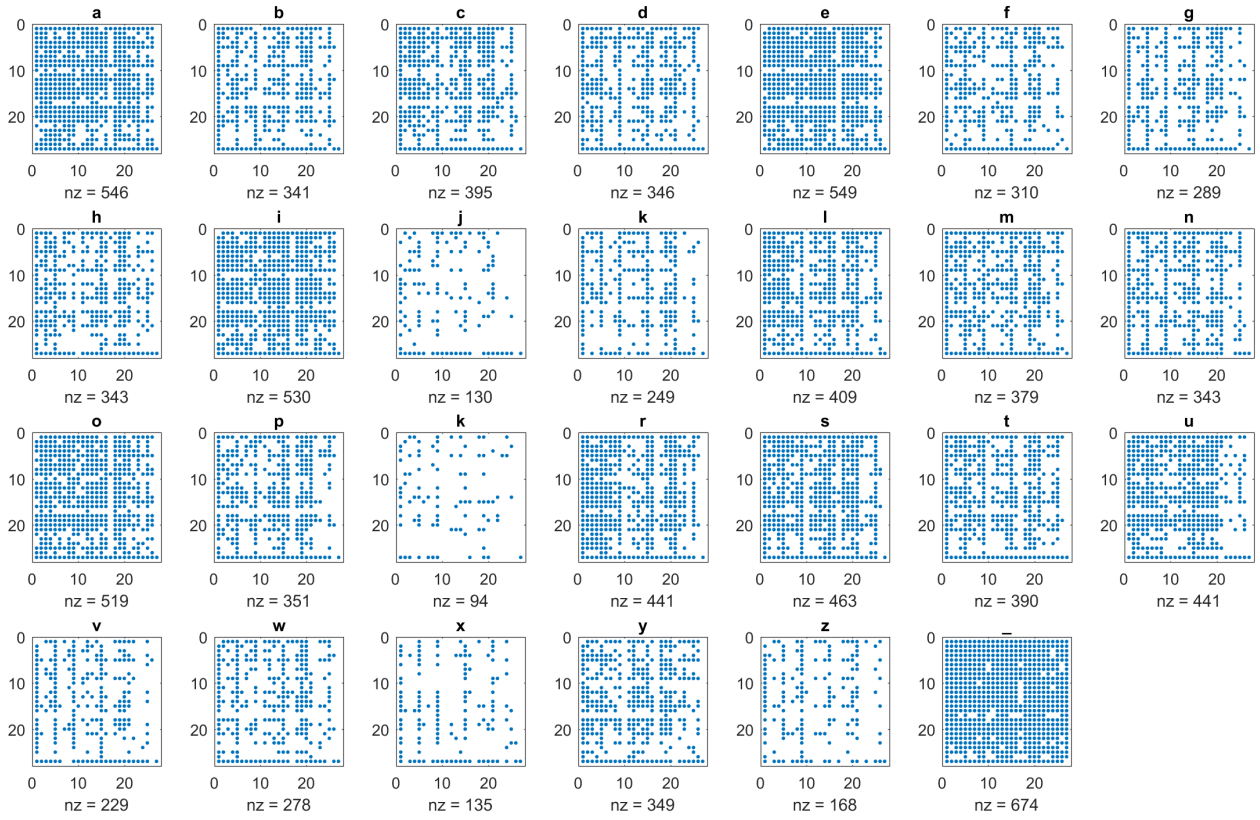

Figure 1: *Sparsity patterns of the occurrences matrices, for the British English corpus. In each matrix are visualized the non-zero occurring trigrams ending with a given letter.*

We can also look at the differences of the 3 language models, once again by looking at the occurrences tensors.
In this case, one approach is to compute the absolute difference of between 2 given language model tensors, and then subtract the mean value of occurrences (multiplied by an arbitrary factor, so that differences becomes more evident), and keep the positive values.
In this way, we will end up with a sparse tensor in which non-zero elements

are trigrams that are more commonly found in one English variety compared to another.

As an example, we can compare the British and the American varieties: it is known that British English tends to use the suffix *-ise*, while American English uses *-ize* (as in *analyse* and *analyze*). The same applies to *-nce* and *-nse* (as in *defense* and *defence*).

Variety	British	American	Australian
-ise	2151	1637	2307
-ize	544	1173	758
-nse	988	1273	1269
-nce	6677	6596	7559

Table 1: Number of occurrences of common suffixes across the 3 English variety.

It seems that *-ize* is indeed more prevalent in the American variety, along with *-nse*. On the other hand, *-nce* is more commonly found in British and Australian English.

5 Text Generation

After computing the probability of encountering each trigram, it is possible to automatically generate text.
To do so, it is possible to keep track of the last 2 characters i, j that have occurred in the sentence, and sample from the discrete probability distribution defined by $P(.|i, j)$.

For each of the 3 language models, it is reported a sample of generated text.

British English:
`__she__yound__as__of__to__buts__being__wil__lif__aboure__to__formse__to__expeame__theit__i__done__the__ancon__is__ho__extudy__oniont__wittils__smar__of__fromfore__hooking__re__ing__the__a__whicanes__res__`

American English:
`__ind__cols__pecan__pubjectic__of__of__thersoo__spe__gold__al__hustare__leve__the__thiclitune__initmannew__wis__of__to__an__the__chis__lat__pars__whe__off__has__musibille__bas__ree__figest__ple__tickdfl__`

Australian English:
`__wouril__ced__thave__rettiessindbase__inge__devisamicenter__in__the__tords__dirs__torthater__wor__facted__pair__mantince__thaparfewrapert__accultut__conly__offecommonathe__tions__peal__every__sy__pa__f__`

Clearly, the sentences generated in this way will be meaningless, and it will also be hard to find any existing word in them. However, it is possible to spot syllables and short words (articles, pronouns).
From this samples, it is also not possible to recognize any significant difference between the 3 types of English. To spot differences between the language models it is required to use metrics such as *perplexity*, as shown in the following section.

6 Text Classification

In this section, it will be shown how to employ the previously generated 3-Grams model in order to distinguish if a sentence belongs to British, American or Australian English.
To do so, it will be employed a metric called **Perplexity**, which is defined as:

$$Perp(c_1, c_2, \ldots, c_n) = P(c_1, c_2, \ldots, c_n)^{-1/n} = \frac{1}{\prod_{i=1}^{n} P(c_i|c_{i-1}, c_{i-2}, \ldots, c_{i-k})^{1/n}}$$

with $k = 2$ in case of **trigrams**.

Moreover, when multiplying small probabilities it is a good idea to consider the log-probabilities instead. As such, we compute the **log-perplexity**:

$$log\text{-}Perp(c_1, c_2, \ldots, c_n) = -\frac{1}{n} \sum_{i=1}^{n} \log_2 P(c_i|c_{i-1}, c_{i-2}, \ldots, c_{i-k})$$

Perplexity can be interpreted as the *degree of surprise* of encountering a sequence in a given language model, and minimizing perplexity is the same as maximizing the probability of encountering the sequence.

We can classify a sentence by computing the perplexity of the sequence for all the 3 language models (i.e. the 3 versions of English); then, the sentence is labelled with the version of English that has the lowest perplexity.
Obviously, it is required to compute the perplexity for all the 3 models, as perplexity scores for a given sentence are mainly useful when compared to each other, instead of being considered by themselves.

To evaluate how good our model is, it is possible to build a **confusion matrix** for each of the 3 classes we are considering (**GB**, **US**, **AU**).
The confusion matrix contains 4 cells, in which are visualized the percentages of *true positives*, *false negatives*, *false positives*, *true negatives*. For a given class, for example British English, true positives are sentences in British English that are correctly labelled as such, while true negatives are sentences in other English variants that are correctly not labelled as British English.

False negatives are sentences in British English that aren't labelled correctly, and false positives are sentences that are labelled as British English despite not being in such variant.

Moreover, to get a global insight of the model performances we can compute the overall accuracy, defined as the percentage of correctly labelled sentences.

Sentence	Real Label	British Perp.	American Perp.	Australian Perp.
When t	AU	2.745	2.737	2.693
Safari	AU	2.844	2.836	2.789
We now	AU	3.274	3.240	3.229
Provid	GB	2.685	2.816	2.720
One of	GB	2.654	2.650	2.692
In 199	GB	2.573	2.582	2.578
" Yes	US	2.501	2.490	2.575
While	US	2.700	2.691	2.701
Watchi	US	2.456	2.459	2.523

Table 2: log-Perplexity of the various sentences (shown here before preprocessing), tested on the 3 models. Numbers in blue are correctly predicted labels. Numbers in red are wrongly predicted labels.

We can see that we achieve a good accuracy (**Overall Accuracy:** $7/9 = 0.77$), and, most importantly, the sentences that are incorrectly labelled have values of perplexity that are very close to each other.

	Positive prediction	Negative prediction
Positive Condition	TP: 2	FN: 1
Negative Condition	FP: 1	TN: 5

Table 3: Confusion Matrix of the British English variety.

	Positive prediction	Negative prediction
Positive Condition	TP: 2	FN: 1
Negative Condition	FP: 1	TN: 5

Table 4: Confusion Matrix of the American English variety.

On a side note, using smaller **N-Grams** is very likely to give lower accuracy scores, as we wouldn't be able to capture syllables and patterns of length 3. Indeed, using **1-Gram** would mean that only the number of occurrences of single

	Positive prediction	Negative prediction
Positive Condition	TP: 3	FN: 0
Negative Condition	FP: 0	TN: 6

Table 5: Confusion Matrix of the Australian English variety.

characters is kept into account, and as such the sentences would be classified based on how common a certain character is, for instance, in British English compared to American English.
Clearly, it's hard to imagine that a certain character is more common in any of these variants, and as such the classification wouldn't be effective.

7 Conclusion

In this report it was shown how a relatively simple **character 3-Grams model** is able to learn and replicate syllables and common patterns of a given corpus.
Most importantly, this model is also able to distinguish, with good accuracy, sentences belonging to 3 variants of English, which is certainly an impressive result if we consider how close these variants are to each other.

Still, it is possible to improve even further the **N-Gram** classifier. There are interesting results [3] that use variable length N-Grams, in the sense that the classifier will keep into account N-Grams of different length.
It could also be interesting to explicitly keep into account the differences in number of occurrences of trigrams, across the 3 varieties. In this way we would focus more on the differences between the varieties, and ignore the common patterns that they have.

Other models, such as **Hidden Markov Models**[2] and **Recurrent Neural Networks**[4] take advantage of *hidden states* to model the interdependencies between characters and symbols. In this way, it is possible to consider long term relations between characters or words, which gives better results that keeping an explicit short term memory as done with N-Grams.

8 Annex: Python Code

8.1 *Model Building*

```
import numpy as np
import re
import timeit
import string

# Used to check which packages are installed.
# Check http://stackoverflow.com/questions/1051254/check-if-python-package-is-installed
import pip
installed_packages = pip.get_installed_distributions()
flat_installed_packages = [package.project_name for package in installed_packages]
if 'Unidecode' in flat_installed_packages:
    import unicodedata


##########################################
# TEXT PREPROCESSING #####################
##########################################

def preprocess_string(input_lines):
    # Put all sentences to lowercase.
    lines = [x.lower() for x in input_lines]
    # If the package "unidecode" is installed,
    # replace unicode non-ascii characters (e.g. accented characters)
    # with their closest ascii alternative.
    if 'Unidecode' in flat_installed_packages:
        lines = [unicodedata.normalize("NFKD", x) for x in lines]
    # Remove any character except a-z and whitespaces.
    lines = [re.sub(r"([^\sa-z])+", "", x) for x in lines]
    # Join all the strings into one
    lines = "".join(lines)
    # Remove whitespaces at the start and end of each sentence.
    lines = lines.strip()
    # Substitute single and multiple whitespaces with a double underscore.
    lines = re.sub(r"[\s]+", "__", lines)
    # Also add a double underscore at the start and at the end of each sentence.
    lines = "__" + lines + "__"

    return lines


##########################################
# COUNT TRIGRAMS  ########################
##########################################

# Build the vocabulary, in our case a list of alphabetical character plus _
# Treating _ as character will allow to model the ending of words too!
vocabulary = list(string.ascii_lowercase[:26] + "_")

def count_occ(sentence, trigram):
```

```
    occurrences = 0

    for n in range(len(sentence) - 2):
        if sentence[n:n+3] == trigram:
            occurrences += 1
    return occurrences

def generate_n_grams(sentence, n):
    """
    Generate the set of n-grams for the given sentence.
    :param sentence: input text string
    :param n: size of the n-grams
    :return: the set of n-grams that appear in the sequence.
    """
    return set(sentence[i:i+n] for i in range(len(sentence)-n+1))


def build_occurrencies_matrix(vocabulary, lines, smoothing="laplace"):
    """
    Build a matrix in which position [i, j, k] corresponds
    to the number of occurrences of trigram "ijk" in the given corpus
    :param vocabulary: the characters for which the occurrences are counted
    :param lines: a text string
    :param smoothing: the type of smoothing to be applied
    :return: a 3-D numpy tensor
    """
    start_time = timeit.default_timer()
    occurrencies_matrix = np.zeros(shape=(len(vocabulary), len(vocabulary), len(vocabulary)))
    v_dict = {char: num for num, char in enumerate(vocabulary)}

    # Generate all the trigrams that appear in the corpus
    trigrams = generate_n_grams(lines, 3)
    # For each trigram, count its occurrences
    for i_t, t in enumerate(trigrams):
        print(i_t / len(trigrams))
        occurrencies_matrix[v_dict[t[0]], v_dict[t[1]], v_dict[t[2]]] = count_occ(lines, t)
    end_time = timeit.default_timer()
    print("! -> EXECUTION TIME OF OCCURRENCIES COUNTING:", (end_time - start_time), "\n")

    if smoothing=="laplace":
        # Apply laplacian smoothing
        occurrencies_matrix += 1

return occurrencies_matrix


##########################################
# ESTIMATE PROBABILITIES #################
##########################################

def build_freq_matrix(vocabulary, occurrencies_matrix):
    start_time = timeit.default_timer()

    # Estimate probabilities of encountering "k" after "ij":
    # prob(k | i, j) = count("ijk") / count("ij")
    frequency_matrix = np.zeros(shape=(len(vocabulary), len(vocabulary), len(vocabulary)))
```

```
    for i in range(len(vocabulary)):
        for j in range(len(vocabulary)):
            for k in range(len(vocabulary)):
                frequency_matrix[i, j, k] = occurrencies_matrix[i, j, k]\
                 / (sum(occurrencies_matrix[i, j, :]))
    end_time = timeit.default_timer()
    print("! -> EXECUTION TIME OF PROBABILITIES:", (end_time - start_time), "\n")

    return frequency_matrix

########################################
# MAIN  TRAINING ########################
########################################

def train(version):
    filename = "./data/training." + version + ".txt"

    start_time = timeit.default_timer()
    with open(filename, encoding="utf8") as f:
    lines = f.readlines()

    #lines = lines[:int(len(lines)/200)]
    lines = preprocess_string(lines)

    end_time = timeit.default_timer()
    print("! -> EXECUTION TIME OF TEXT PREPROCESSING:", (end_time - start_time), "\n")

    print(lines[:20])

    occurrencies_matrix = build_occurrencies_matrix(vocabulary, lines)
    np.save("./language_model_occ_" + version, occurrencies_matrix)
    frequency_matrix = build_freq_matrix(vocabulary, occurrencies_matrix)
    np.save("./language_model_freq_" + version, frequency_matrix)

versions = ["GB", "US", "AU"]
for v in versions:
    train(v)
```

8.2 *Text Generation*

```
import numpy as np
import string

def generate_text(language_model, min_length=100,
        vocabulary=list(string.ascii_lowercase[:26] + "_")):
    """
    Generate text from the given 3-Grams language model.
    :param language_model: a 3-D tensor in which [i, j, k] is the probability
        of encountering "k" after "ij".
    :param min_length: minimum length of the generated text. After that,
        the generation will stop after encountering "__"
    :param vocabulary: the set of characters to be used on the generation.
    :return: a string of generated text.
```

```
    """
    length = 0

    generated_text = "__"
    mem_vec = [26, 26]

    while True:
        prob_vec = language_model[mem_vec[0], mem_vec[1], :]
        #print("MEMORY:", [vocabulary[c] for c in mem_vec], list(zip(vocabulary, prob_vec)))
        new_char_index = np.random.choice(np.arange(len(vocabulary)), p=prob_vec)
        generated_text += vocabulary[new_char_index]
        mem_vec = mem_vec[1:] + [new_char_index]
        length += 1
        if length >= min_length and generated_text[-2:] == "__":
            break
    return generated_text

language_model = np.load("language_model_freq_GB.npy")
generated_text = generate_text(language_model)
print(generated_text)
```

8.3 *Text Classification*

```
import pandas as pd
import numpy as np
import re
import timeit
import string

from language_model import preprocess_string

def perplexity(string, language_model, log=True,
        vocabulary=list(string.ascii_lowercase[:26] + "_")):
    """
    Computes the perplexity of a given string, for the specified language model.

    Given a sentence composed by character [c_1, c_2, ..., c_n],
    perplexity is defined as P(c_1, c_2, ..., c_n)^(-1/n).

    :param string: the input string on which perplexity is computed
    :param language_model: language model used to compute perplexity.
    It is a matrix in which entry [i, j, k] is P(k | j, i).
    :param log: returns perplexity in log-space.
    :param vocabulary: the vocabulary that is used to evaluate the perplexity.
    :return: the perplexity of the sentence.
    """
    v_dict = {char: num for num, char in enumerate(vocabulary)}

    perp = 0
    for i in range(len(string) - 2):
        perp += np.log2(language_model[v_dict[string[i-2]], v_dict[string [i-1]],\
            v_dict[string[i]]])
        perp *= -(1/len(string))
```

```
    return perp if log==True else 2**perp

def analyze_results(results, true_cond, perc=True):
    """
    :param results: a list of tuples of the form (real_label, predicted_label)
    :param true_cond: label that should be considered as true condition
    :param perc: if true, give the results as % instead than absolute values
    :return: a dictionary with keys [TP, FN, FP, TN]
    """
    tp = sum([item[0] == true_cond and item[1] == true_cond for item in results])
    fn = sum([item[0] == true_cond and item[1] != true_cond for item in results])
    fp = sum([item[0] != true_cond and item[1] == true_cond for item in results])
    tn = sum([item[0] != true_cond and item[1] != true_cond for item in results])

    confusion_matrix = {"TP": tp, "FN": fn, "FP": fp, "TN": tn}
    return confusion_matrix if not perc else\
        {k: v / len(results) for k, v in confusion_matrix.items()}

def accuracy(results):
    """
    :param results: a list of tuples of the form (real_label, predicted_label)
    :return: accuracy of the results, expressed as the percentage of labels correctly predicted.
    """
    return sum([item[0] == item[1] for item in results]) / len(results)

model_names = ["GB", "US", "AU"]
language_models = {}
for model_name in model_names:
    language_models[model_name] = np.load("language_model_freq_" + model_name + ".npy")


test_filename = "data/test.txt"

results = []
with open(test_filename, encoding="utf8") as f:
    lines = f.readlines()
for l in lines:
    [label, sentence] = l.split("\t", 1)
    sentence = preprocess_string(sentence)
    perp_res = {k: perplexity(sentence, language_model)
        for k, language_model in language_models.items()}
    results.append((label, min(perp_res.keys(), key=(lambda key: perp_res[key]))))
    print(sentence[:6], "-- REAL LABEL:", label, "-- PERP:", perp_res)
print(results)

print("\n======== GB =========\n\n", analyze_results(results, "GB", False))
print("\n======== US =========\n", analyze_results(results, "US", False))
print("\n======== AU =========\n", analyze_results(results, "AU", False))

print("\n===== ACCURACY ======\n", accuracy(results))
```

9 Annex: Language Model Extract

In this section are displayed the probabilities of some of the trigrams, for British and American English.

Trigram	Probability
iza	0.192
izb	0.001
izc	0.001
izd	0.003
ize	0.594
izf	0.001
izg	0.002
izh	0.003
izi	0.050
izj	0.001
izk	0.001
izl	0.001
izm	0.001
izn	0.001
izo	0.041
izp	0.001
izq	0.001
izr	0.001
izs	0.002
izt	0.001
izu	0.016
izv	0.001
izw	0.001
izx	0.001
izy	0.002
izz	0.045
iz_	0.032

Table 6: Probabilities of trigrams starting in **iz** in British English.

Trigram	Probability
iza	0.220
izb	0.001
izc	0.001
izd	0.001
ize	0.604
izf	0.001
izg	0.001
izh	0.001
izi	0.075
izj	0.001
izk	0.001
izl	0.001
izm	0.002
izn	0.001
izo	0.033
izp	0.001
izq	0.001
izr	0.001
izs	0.001
izt	0.001
izu	0.013
izv	0.001
izw	0.001
izx	0.001
izy	0.001
izz	0.024
iz_	0.017

Table 7: Probabilities of trigrams starting in **iz** in American English.

10 Addendum: Kneser-Ney Smoothing

It is also possible to implement **Kneser-Ney Smoothing** for bigrams, and see how it performs compared to the standard 3-Grams model.
The value of δ used by the algorithm was found by optimizing the value of the accuracy function, using *scipy.optimize*. The occurrences matrices were built by using 80% of the sentences, and the value of δ was optimized using the remaining 20%.
However, the results are mediocre, giving only **0.44** of accuracy (on the given test set), compared to the **0.77** of the 3-Grams model.

```
def build_occurrencies_matrix(vocabulary, lines):
    """
    Build a matrix in which position [i, j] corresponds
    to the number of occurrences of bigram "ij" in the given corpus
    :param vocabulary: the characters for which the occurrences are counted
    :param lines: a text string
    :return: a 2-D numpy tensor
    """
    occurrencies_matrix = np.zeros(shape=(len(vocabulary), len(vocabulary)))
    v_dict = {char: num for num, char in enumerate(vocabulary)}

    # Generate all the trigrams that appear in the corpus
    bigrams = generate_n_grams(lines, 2)
    # For each trigram, count its occurrences
    for i_t, t in enumerate(bigrams):
        occurrencies_matrix[v_dict[t[0]], v_dict[t[1]]] = count_occ(lines, t)

    return occurrencies_matrix

def train_kneser_ney(occ_matrix, delta=0.75,
        vocabulary=list(string.ascii_lowercase[:26] + "_")):
    frequency_matrix = np.zeros(shape=(len(vocabulary), len(vocabulary)))

    for i in range(len(vocabulary)):
        for j in range(len(vocabulary)):
            frequency_matrix[i, j] =\
                (max(occ_matrix[i, j] - delta, 0) / sum(occ_matrix[i, :])) \
                + (delta / sum(occ_matrix[i, :])) * sum([x > 0 for x in occ_matrix[i, :]]) \
                * sum([x > 0 for x in occ_matrix[:, j]]) / (occ_matrix > 0).sum()
    return frequency_matrix

def train_occ(lines, version, training_split=0.8):
    lines = lines[:int(len(lines) * training_split)]
    lines = preprocess_string(lines)

    occurrencies_matrix = build_occurrencies_matrix(vocabulary, lines)
    np.save("./language_model_occ_bigrams" + version, occurrencies_matrix)

def train_wrapper(delta):
    versions = ["GB", "US", "AU"]
    language_models = {}
    for v in versions:
        occ_matrix = np.load("./language_model_occ_bigrams" + v + ".npy")
        language_models[v] = bk.train_kneser_ney(occ_matrix, delta)

    test_filename = "test_lines.txt"
    test_labels_filename = "test_labels.txt"

    results = []
    with open(test_filename, encoding="utf8") as f:
        test_lines = f.readlines()
    with open(test_labels_filename, encoding="utf8") as f:
        test_labels = f.readlines()
```

```
        test_labels = [x.strip() for x in test_labels]
    for sentence, label in zip(test_lines, test_labels):
        sentence = bk.preprocess_string(sentence)
        perp_res = {k: perplexity(sentence, language_model)
            for k, language_model in language_models.items()}
        results.append((label, min(perp_res.keys(), key=(lambda key: perp_res[key]))))
    return -accuracy(results)

for v in versions:
    filename = "./data/training." + v + ".txt"
    with open(filename, encoding="utf8") as f:
    lines = f.readlines()
    split = int(len(lines) * training_split)
    train_lines = lines[:split]
    test_lines += lines[split:]
    test_labels += [v] * len(lines[split:])

    train_occ(train_lines, v)

import scipy.optimize as optim
res = -optim.minimize_scalar(train_wrapper, bounds=(0.01, 10), method='bounded')
print(res)
```

References

[1] William B Cavnar, John M Trenkle, et al. N-gram-based text categorization. *Ann Arbor MI*, 48113(2):161–175, 1994.

[2] Paolo Frasconi, Giovanni Soda, and Alessandro Vullo. Hidden markov models for text categorization in multi-page documents. *Journal of Intelligent Information Systems*, 18(2-3):195–217, 2002.

[3] Ioannis Kanaris, Konstantinos Kanaris, Ioannis Houvardas, and Efstathios Stamatatos. Words versus character n-grams for anti-spam filtering. *International Journal on Artificial Intelligence Tools*, 16(06):1047–1067, 2007.

[4] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In *AAAI*, volume 333, pages 2267–2273, 2015.