Natural Language Processing
Italian Language Tokenizer with Emoji and Emoticons Support

Daniele Perrella

2022-2023

Contents

Chapter 1

Introduction

Tokenization is a crucial task in Natural Language Processing (NLP) that involves dividing text into meaningful units, or tokens, such as words or punctuation marks. Tokenization is a fundamental step for many NLP applications, including machine translation, sentiment analysis, and named entity recognition.
Italian is a widely spoken language, with over 85 million speakers worldwide, and has its unique linguistic characteristics. Italian has a rich vocabulary and a complex grammar, which makes the task of tokenization particularly challenging. However, tokenization is essential for effective NLP applications in Italian, and the development of a reliable Italian tokenizer can greatly benefit the NLP community.
In this report, I will be presenting an Italian tokenizer, which is capable of tokenizing a wide range of text elements, including literals, punctuation, email and domains, numericals and operators, emoticons, and emoji. Notably, all the tokenization is made using only regular expressions (regex), to accurately segment Italian text into meaningful tokens.
This Italian tokenizer builds upon existing tokenization techniques and incorporates several novel features. I leveraged the power of regex, which is a highly efficient and effective tool for pattern matching in text processing, to develop a comprehensive set of rules for tokenizing Italian text. In particular, this tokenizer is designed to handle a variety of text elements that are often challenging to tokenize, such as emoticons and emoji, which are becoming increasingly prevalent in modern communication.

Chapter 2

Tokenizers

Tokenization is the process of breaking down a sequence of text into smaller units, known as tokens. These tokens are typically words, phrases, or other meaningful units of text that can be used to analyze and understand the content of the text. Tokenization is an important step in many natural language processing tasks, such as text classification, sentiment analysis, and machine translation.

The process of tokenization can vary depending on the specific task and the language being processed. In some cases, tokens are simply individual words, separated by spaces or punctuation marks. In other cases, tokens may be more complex, such as numerical values, internet domains and even emoticons.

There are several different methods for tokenizing text. One common approach is to use regular expressions to match patterns in the text, such as whitespace or specific character sequences. Another approach is to use machine learning algorithms to identify patterns in the text and automatically generate tokens.

Regardless of the specific method used, the goal of tokenization is to break down the text into smaller units that can be more easily analyzed and understood. This allows researchers and developers to extract meaning from large volumes of text data, and to build more accurate and effective natural language processing models.

Chapter 3

Language Tokenization

3.1 Literals

The literals rule is one of the fundamental rules in natural language processing (NLP) and is used to identify words and letters in a given text. The purpose of this rule is to extract meaningful text that carries the main message of the text. A token can be defined as a sequence of characters that represents a unit of meaning in a given context. The tokenizer can split the text into words or letters, depending on the specific task at hand. This rule works by applying a regular expression to the tokens, which matches sequences of characters that represent valid words or letters. The regular expression can be customized to match specific patterns that are relevant to the task at hand.
Overall, the literals rule is an essential component of any NLP pipeline, as it forms the basis for many higher-level language processing tasks.
To support the Italian language, the default configuration of the tokenizer supports all the letters available for Italian, including accented ones:
à è é ì í ò ó ù ú
Both lowercase and uppercase The following expression, is the Regular Expression Rule used to recognize the literals:

```
[A-Za-zÀÀÈÉÌÍÒÓÙÚàèéìíòóùú]+
```

Since there could be concatenations of letters, to form a word, there is the + sign just after the group of characters to consider. This allows the recognition of concatenated letters, and thus words.

3.2 Punctuation

The punctuation rule is used to match any kind of punctuation mark that may appear in text. Punctuation marks include characters such as periods, commas, colons, semicolons, dashes, parentheses, and quotation marks, among others. Punctuation marks are important because they provide structure and clarity to written language, helping to organize ideas and convey meaning. Punctuation marks also play a role in syntax, which is the arrangement of words and phrases to create well-formed sentences in a language.
In regular expressions, the punctuation rule can be expressed using a character class, which is a set of characters enclosed in square brackets. A character class can match any one character from the set of characters it contains.

For this tokenizer, there are two rules for Italian punctuation, one including all the punctuation charaters, and a second one to support exclusively the "..." punctuation.

```
[.][.][.]
```

```
[[]{}:\$¿?§«»¤°€¥£¢¡@&_()=+!’%/.,""’*<>^;÷\|¦¬×¶¯~±›‹‰...•]
```

This character class matches any of the punctuation marks within the brackets. By using the punctuation rule in regular expressions, it is possible to match and manipulate text based on the presence of punctuation marks.

Chapter 4

Email and Domain Support

At the time, since almost every person has a email address, there is an high probability to encounter such content in text, that's why I decided to include a dedicated tokenization rule to handle this type of content. Email domain is not the only type of domain supported, indeed there is also the support for classic URLs.

4.1 Domain Support

This regular expression rule matches URLs, including both http and https protocols, as well as the option of including "www." at the beginning of the domain. The rule also allows for any combination of letters, numbers, hyphens, and underscores before the domain name, and any combination of letters, numbers, hyphens, and dots after the domain name, including paths and query strings.

```
(https?://)?(www.)?[w-]+\\.[w.-]+(/[/?%&=]*)?
```

Here is a breakdown of the different parts of the regular expression rule:

- $(https? : //)?$: This group matches the optional *'http://'* or *'https://'* protocol prefix. The *'?'* character after the *'s'* makes it optional.
- $(www.)?$: This group matches the optional *'www.'* prefix.
- $[w-]+$: This group matches one or more word characters or hyphens.
- .: This matches the literal dot character.

- [w.−]+: This group matches one or more word characters, hyphens, or dots. This matches the domain name.
- (/[/?%& =]∗)?: This group matches an optional path or query string. The path starts with a forward slash, followed by more characters that may include forward slashes, question marks, percent signs, and ampersands.

4.2 Email Support

The regular expression pattern for matching email addresses is a complex one, designed to match a wide variety of valid email addresses while minimizing the number of false positives.

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\.[a-z0-9!#$%&'*+/=?
^_`{|}~-]+)*|(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b
\x5d-\x7f]|[\x01-\x09\x0b\x0c\x0e-\x7f])*)@(?:(?:[a-z0-9]
(?:[a-z0-9-]*[a-z0-9])?.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|
[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}
(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*
[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|
[\x01-\x09\x0b\x0c\x0e-\x7f])+)])
```

The pattern is made up of two main parts: the **local part** and the **domain part**.
The **local part** of the pattern allows for a wide range of characters, including letters, numbers, and special characters It can also include periods (.) and hyphens (-), as long as they are not the first or last character in the local part, and as long as they are not adjacent to each other.
The **domain part** of the pattern is made up of one or more domain name components, separated by periods (.). Each component can include letters, numbers, and hyphens, but cannot begin or end with a hyphen. The final component must be a two-letter country code or a generic top-level domain (such as .com, .org, or .net).
The regular expression pattern also allows for the use of square brackets to enclose an IP address, which can be used instead of a domain name. This allows the pattern to match email addresses that use IP addresses instead of domain names.
Overall, the regular expression pattern for matching email addresses is a highly complex one that is designed to match a wide range of valid email addresses while minimizing the number of false positives.

Chapter 5

Tokenization for Numerical and Operations

In many cases, we need to match and extract numeric values from strings. This is where the following regular expression comes in handy.

```
[-+]?[0-9]*(.|,)?[0-9]+([eE][-+]?[0-9]+)?(%|\u2030)?
```

This regular expression can be used to match numeric values that may contain a sign, a decimal point or comma, scientific notation, and a percentage or permille symbol. In this section, we will explore the details of this regular expression, how it works, and some examples of how to use it in practice.

Here's a breakdown of the different components:

- $[-+]?$: an optional sign (+ or -) at the beginning of the number.
- $[0-9]*$: zero or more digits before the decimal point.
- $(.|,)?$: an optional decimal point, which can be either a period (.) or a comma (,).
- $[0-9]+$: one or more digits after the decimal point.
- $([eE][-+]?[0-9]+)?$: an optional exponent, represented by the letter "e" or "E" followed by an optional sign and one or more digits.
- $(\%|\backslash u2030)?$: an optional percentage or permille symbol, represented by either % or ‰ (Unicode code point for the permille symbol).

Here are some examples of strings that would match this regular expression:

- 123.45
- -0.00123e4
- 6,789.012%
- 42

This flexibility makes the numerical regular expression a useful tool for identifying and extracting numerical data from text.

Chapter 6

Emoticons Support

Emoticons, also known as smileys, are pictorial representations of facial expressions used to convey emotions and attitudes in written communication. They are commonly used in text messaging, social media, and online forums to add tone and convey feelings that might be difficult to express through text alone. Emoticons consist of a combination of characters that resemble a face, and they can convey a wide range of emotions, such as happiness, sadness, anger, surprise, and many others. Emoticons can be simple or complex, and they may include various symbols, letters, numbers, and punctuation marks. In this section, we will discuss a regular expression that can match various types of emoticons, including those with different facial expressions, eyes, mouths, noses, and other features. The regular expression can identify emoticons that use different characters and symbols, such as colons, semicolons, dashes, slashes, brackets, and many others, and it can be useful for various applications, such as sentiment analysis, chatbots, and natural language processing.

The provided regex is a pattern that matches a variety of emoticons commonly used in text-based communication. It consists of two main components separated by the '|' (OR) operator.

```
([<|>}3O~0]?[:;8=BxX%#]['\"]?[-^]?[)(03sSxXDcCpoOPEL\u00de
\u00feb/*\#&$><}{\[\]@|])|([cCD><][-^]?['\"]?[:;8=BxX%#])
```

The first component:

```
"([<|>}3O~0]?[:;8=BxX%#]['\"]?[-^]?[)
(03sSxXDcCpoOPEL\u00de\u00feb/*\\#&$><}{\[\]@|])"
```

matches a wide range of emoticons that start with optional characters such as "<", "|", ">", "3", "O", "~", or "0", followed by a common set of emoti-

con features such as eyes (represented by colons, semicolons, or the characters "8", "B", "x", "X", "%", or "#"), optional eyebrows or glasses (represented by apostrophes or quotation marks), a nose (represented by a hyphen or caret), a mouth (represented by a variety of characters such as parentheses, numbers, letters, or symbols), and optional additional features such as sweat, blush, or tears (represented by characters such as "s", "S", "x", "X", "D", "c", "C", "p", "o", "O", "P", "E", "L", or special characters such as the Icelandic Thorn letter "Þ" or the German Eszett "ß").

The second component

```
"([cCD><][-^]?['\"]?[:;8=BxX%#])"
```

matches a set of emoticons that start with characters such as "c", "C", "D", or "><", followed by optional features such as a nose, eyebrows, or glasses, and ending with eyes.

Examples of supported emoticons that can be matched by this regex include

```
:)  :(  :D  :P  ;)  :/  :O  :S  <3  ^_^
T_T  :3 :|  :@ :S  :')  :*  8)  B)  B|
:X  :>  :<  :{  :}  3:)  >:O :#  o_O
```

and many others.

Chapter 7

Emoji Support

Emojis have become a ubiquitous part of digital communication, conveying emotions, reactions, and ideas with just a single symbol. With their widespread use in social media, messaging apps, and other online platforms, it's no surprise that they have also become a popular feature in regular expressions. In this section, we will explore the regex patterns used to match emojis and some examples of supported emojis.

Matching emojis can be a challenging task due to the various encoding formats used to represent them. To handle this task, this tokenizer makes advantage of a python library **emoji**[2] which converts emojis into text. This text is then handled by the tokenizer, which can match it within the following regular expression

```
:([a-z]*[_][a-z]*)*[-]?([a-z]*[_]?[a-z]*)*:
```

Here is an example of the workflow:

1. The input contains the following emoji

2. The emoji library converts it to the following text:

```
:face_with_raised_eyebrow:
```

3. The text is matched against the regular expression.
4. The regular expression matches the text and recognizes it as an emoji.

The regular expression matches the text and recognizes it as an emoji. The regex pattern includes a set of characters or character ranges that represent the different components of the emoji, such as the base character, modifiers, and skin tone variations Matching these symbols can be useful in a variety of applications, such as sentiment analysis, social media monitoring, and chatbot development. However, it is important to note that the use of emojis in communication is highly contextual and can be subject to interpretation.

Figure 7.1: Example of emoji with skin tone modifier

Chapter 8

The Output

After tokenization, the output feature comes into play. This feature allows the tokenized text to be exported into an external file, which can then be used for further processing or analysis. The ***export_tokens*** function takes two arguments: an output file and the tokenized text. The output file is opened in write mode and the function begins by writing the XML version and root element to the file.

The tokenized text is then iterated over, and for each item in the list, the function writes the item's type and the token values to the output file as XML elements. The type element specifies the type of token (e.g. word, punctuation, number) and the token element contains the actual tokenized text. Once all items in the tokenized text list have been processed, the function writes the closing root element to the output file and closes it. This XML format allows for easy parsing and manipulating of the tokenized text in other programs or scripts. For example, the tokenized text could be loaded into a database for further analysis or visualization
Here's an example of how the output file would look like:

```
<?xml version="1.0" encoding="UTF-8" ?>
<tokenized>
        <item>
                <type>punctuation</type>
                <token>...</token>
        </item>
        <item>
                <type>literal</type>
                <token>test</token>
        </item>
        <item>
                <type>emoji</type>
                <token>:raised_back_of_hand_dark_skin_tone:</token>
        </item>
 </tokenized>
```

In this example, the input text has been tokenized into three items, consisting of one punctuation, one word and one emoji. The output file contains an XML element for each token, with its type and token values specified. Overall, the output feature provides a convenient way to save the results of tokenization. There is also the possibility to save emojis as emojis, and not as text, just in case there is the necessity to keep the original form.

Chapter 9

Custom Configurations

In some cases, you may want to customize the behavior of the tokenizer to better fit your specific use case.
One of the great features of this tokenizer is that you can create your own custom configuration to extend the recognition patterns. The default configuration is good for general text processing, but it may not be the best for specific types of text.

To create your own configuration, you need to modify the JSON file following the same structure as the ***default_config.json*** file. The file should contain a "config" array that contains objects with a "*name*" and a "*regex*" field.
The "*name*" field will be the identifier for the token type you want to create. It is not necessary to be unique. The "*regex*" field should contain the regular expression pattern that will match the text you want to tokenize.
For example, let's say you want to create a token type for hashtags. You can edit the "***custom_config.json***" as follows.

```
{
    "config": [
        {
            "name": "hashtag",
            "regex": "#\\w+"
        }
    ]
}
```

This configuration file contains a single object. The object has the "*name*" field set to "*HASHTAG*" and the "*regex*" field set to `#\w+`

This regular expression will match any text that starts with a "#" an is followed by one or more word characters.

Now, when the tokenizer is run, the custom configuration token types will be included in the output

```
# Input: Check out this #awesome hashtag!

# Output:
    <?xml version="1.0" encoding="UTF-8" ?>
    <tokenized>
        <item>
            <type>literal</type>
            <token>Check</token>
        </item>
        <item>
            <type>literal</type>
            <token>out</token>
        </item>
        <item>
            <type>literal</type>
            <token>this</token>
        </item>
        <item>
            <type>hashtag</type>
            <token>#awesome</token>
        <item>
        <item>
            <type>literal</type>
            <token>hashtag</token>
        <item>
            <type>punctuation</type>
            <token>!</token>
        </item>
     </tokenized>
```

The custom configuration is loaded into the code with the following lines.

```
custom_config_file = open("data/config/custom_config.json")
custom_config_data = json.load(custom_config_file)
custom_config_file.close()
```

It is possible to add more lines inside of the same configuration file, but it is necessary to load the correct file inside of the code, in case you want to create a different configuration file.

In summary, editing the custom configuration is easy and allows you to customize the tokenizer to better fit the needs.

Chapter 10

Conclusion

In this report, we have explored the development and implementation of a tokenization tool that utilizes regular expressions for pattern matching. Through the use of regular expressions, we have successfully achieved the goal of breaking down textual input into meaningful tokens based on predefined patterns. Throughout this report, we have discussed the key components and functionalities of the tokenization tool. We started by introducing the concept of regular expressions and their significance in pattern matching. We explored various regex patterns for different types of tokens, including emojis, website domains, email domains, numbers, operators, punctuation, and literals. Examples were provided to demonstrate how these patterns can be used to effectively identify and extract tokens from text. We also discussed the output feature, which allows the generated tokens to be exported in XML format. The export_tokens() function was presented, showcasing the process of writing tokenized data to an output file. Furthermore, we covered the concept of custom configuration, which provides flexibility in defining and adding new tokenization rules. By modifying the configuration file, users can tailor the tokenization tool to specific requirements and enhance its functionality. Through this project, we have gained valuable insights into the importance of tokenization, the usage of regular expressions, and the development of practical tools for text processing.

Bibliography

[1] Wikipedia Emoticons
https://en.wikipedia.org/wiki/List_of_emoticons

[2] Emoji Library
https://pypi.org/project/emoji/