Sentiment Analysis With Python

§

Made with Obsidian

Type portfolio-project Category machine-learning Technologies Python

Website Post Link

Sentiment analysis is a natural Language Processing (*NLP*) technique which consists of identifying the emotional tone behind a body of text. This analysis can be applied to multiple contexts such as product review, public opinion, social media polarity, and even support ticket satisfaction measurement.

Sentiment Analysis can be performed using Machine Learning algorithms, computational linguistics, or a combination of both. There are a number of libraries that can be used to achieve this task, the most populars being VADER, TextBlob, SpaCy and Flair, and transformer ML models such as GPT, Google's BERT, RoBERTa and XLNet.

In this Portfolio Project, we'll build an end-to-end Sentiment Analysis application which includes a GUI, two NLP models, a wide variety of options for analysis, and various customizations. It's important to mention that this project is more inclined towards the software engineering aspect of the application, and less towards the sentiment analyis models themselves; we will not be designing our models from scratch.

We will also not include all the project code in this article since it's extense, and will instead limit to examples and explanations for the key sections. Still, the complete project including the resources used can be found in the Portfolio Project Repo.

§

Table of Contents

§

Preface

Sentiment Analysis methods can provide insight regarding the tone, polarity, subjectivity and most prevalent parts of speech of a given text. We can create our own model from scratch, use a pretrained one out of the box, perform transfer learning on a pretrained model with our own datasets, or use a rule-based approach where no ML model is required.

Machine Learning approach

As mentioned before, sentiment analysis can be achieved using **Machine Learning models**. If we think in simple terms, extracting sentiment out of text can be modeled as a classification problem.

Let us illustrate this with an example, where we have a set of movie reviews we would like to classify as positive, neutral or negative:

> A riotous film that finds depth, clarity and refreshment in even the shallowest of pools.

> It would be a disservice to consider this generous film a mere homage.

In general terms, we would preprocess our text data, extract relevant features, train our classification model with labeled data and use our trained model to predict the sentiment of future data sets:

Data preprocessing: The text data is preprocessed by removing stop words, special characters, and converting it to lowercase to reduce noise in the dataset:

- Original text: A riotous film that finds depth, clarity and refreshment in even the shallowest of pools.
- Preprocessed text: riotous film finds depth clarity refreshment even shallowest pools.

Tokenization: Before we pass our sentences to the model, we must also tokenize them, meaning breaking the sentence into tokens consisting of smaller sentences, phrases, symbols or words. Our tokenized sentences would look something like such:

```
rev_1 = ['riotous',
         'film',
         'finds',
         'depth',
         'clarity',
         'refreshment',
         'even',
         'shallowest',
         'pools']

rev_2 = ['would',
         'disservice',
         'consider',
         'generous',
         'film',
         'mere',
         'homage']
```

Feature extraction: The tokenized text is transformed into a numerical representation of features that the model can understand. We can use techniques such as bag-of-words (*BOW*), n-grams, and word embeddings.

The bag-of-words approach considers a vocabulary of all the unique words in the dataset and represent each review as a vector of word counts:

review	rev_1	rev_2
riotous	1	0
film	1	1
finds	1	0
depth	1	0
clarity	1	0
refreshment	1	0
even	1	0
shallowest	1	0
pools	1	0
would	0	1
disservice	0	1
consider	0	1
generous	0	1
mere	0	1
homage	0	1

Table 1: Bag-Of-Words For 2 Movie Reviews

Model training: We now train our sentiment analysis model with previously labeled data, where each text sample (*in this case, a word*) is associated with a sentiment label (*e.g. positive, neutral or negative*). An example of a simple set of labeled words could consist of the following:

Word	Label
riotous	Negative
film	Neutral
finds	Neutral
depth	Neutral
clarity	Positive
refreshment	Positive
even	Neutral
shallowest	Negative
pools	Neutral
would	Neutral
disservice	Negative
consider	Neutral
generous	Positive
mere	Neutral
homage	Positive

Table 2. Sentiment Labels For Words

Since we want to calculate a score for the entire sentence, labels are usually expressed numerically, instead of textually Positive, Neutral or Negative. We could, for example, define a vector $[-1, 0, 1]$, representing each label.

Model testing and validation: The trained model is tested and validated on a separate dataset to evaluate its performance. The model's performance is measured using metrics such as accuracy, precision, recall, and F1 score.

Prediction: Once the model is trained and validated, it can be used to make predictions on new text data. The model will analyze the text and classify it as positive, negative, or neutral sentiment based on the learned patterns and features.

This was a simplified example, but in reality, ML models such as transformer models make associations between words in order to understand the context of a sentence or paragraph by grouping words using Part of Speech (*POS*) tags or other attributes; this technique is called **lemmatization** and is extremely relevant in NLP; even though both of the reviews we used were very positive, there were some words tagged as Negative (*e.g. disservice, riotous*), so the final score would not be 100% Positive.

There are a wide variety of models we can use:

- **Supervised Learning**
  - Naive Bayes Classifier (*NBC*)
  - Support Vector Machines (*SVM*)
  - Logistic Regression (*LR*)
  - Random Forest Classifier (*RFC*)
- **Deep Learning**
  - Convolutional Neural Networks (*CNN*)
  - Recurrent Neural Networks (*RNN*)
  - Deep Belief Networks (*DBN*)
  - Long-Short Term Memory (*LSTM*)

Most available pretrained large models already offer great performance in terms of social media and product review analysis out of the box. On top of that, there are thousands of variations for each existing large model; there are multiple forks containing tuned pretrained models specific for a given application, such as Twitter polarity analysis or IMDB movie rating analysis.

Rule-based approach

As its name suggests, the rule-based approach follows a set of predefined, hardcoded rules in order to classify the text's sentiment. The result is a set of rules based on which the text is labeled as positive/neutral/negative. These rules are also known as lexicons, hence the Rule-based approach is also called **Lexicon-based approach**.

Upon performing the sentiment analysis on a sentence or paragraph, each of the words are scored, and a final score is calculated based on the frequency of each word.

In general, a rule-based approach follows similar initial steps to a ML approach, the biggest difference being there's no model to train, test and validate: We preprocess the text, tokenize it, enrich it with part of speech (*POS*) tagging, and classify it according to a set of predefined rules; it's essentially a simpler process.

The major disadvantage with this approach, is that most libraries are not capable of contextualizing sentences or paragraphs; the final score is given by the cummulative score of each word, without taking context into account.

Still, rule-based algorithms have proven extremely useful and fairly accurate, with a low ammount of effort required in terms of their implementation.

There are two main libraries for performing sentiment analysis using rule-based approaches:

- VADER (*Valence Aware Dictionary for Sentiment Reasoning*)
- TextBlob

§

Concept

We want to design an easy-to-use Guided User Interface which provides the user a way to perform sentiment analysis on one or more datasets. We also want to include deeper analysis capabilities and results exporting, so that the user can visualize textually and graphically the analysis results.

GUI

The Guided User Interface should include the following:

- Option to bulk-download datasets from a list of URLs provided by the user, or bulk-read existing datasets.
- Option to perform sentiment analysis for a user-selected column of a given dataset, using two different models.
- Option to include up to 4 additional columns in order to perform deeper analyses.
- Option to include a Rating column in order to compare sentiment analysis results with actual rating.
- Option to export results using 4 different formats:
  - **Technical:** In-depth analysis including plots and Excel files with results
  - **Business:** Business-like presentation including plots and Excel files.
  - **Visual:** All plots from Technical and Business, without the Excel files.
  - Complete: All plots and Excel files.
- Option to perform in-depth POS analysis correlating POS tags with sentiment scores.
- Option to customize color scheme and transparency for generated plots.

Apart from the main components, we should also include the following:

- A Help popup window containing operation instructions.
- An About popup window containing information related to the project.
- An Appearance Mode menu for selecting System, LIght or Dark.
- A text prompt informing the user about the current progress.
- Progress bars for each step of the process.

In the end, we are looking for an interface like the sketch below:

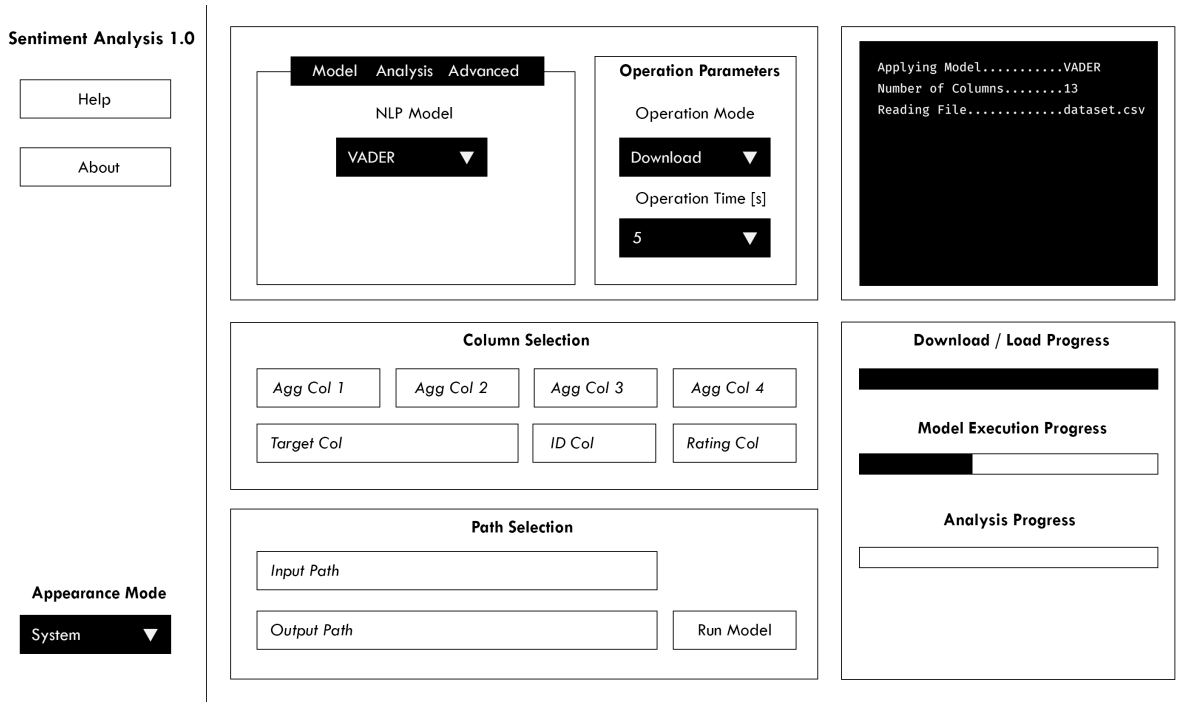

Figure 1: GUI Concept Sketch

Models

For this project we will implement both VADER and TextBlob as options to the user.

VADER is a lexicon and rule-based sentiment analysis tool that is specifically attuned to sentiments expressed in social media.

VADER accepts a string (*word, sentence, paragraph or document*) as input and returns four scores:

- Positiveness probability $[0, 1]$
- Neutrality probability $[0, 1]$
- Negativity probability $[0, 1]$
- Compound score $[-1, 1]$

TextBlob is a library for processing textual data. It provides a simple API for diving into common natural language processing (NLP) tasks such as part-of-speech tagging, noun phrase extraction, sentiment analysis, classification, translation, and more.

The TextBlob sentiment method accepts a `textblob.blob.TextBlob` object containing a string (*word, sentence, paragraph or document*) as input, and returns a tuple of two scores:

- Polarity $[-1, 1]$
- Subjectivity $[-1, 1]$

The advantage of these 2 models, is that both output a polarity score in the same scale $[-1, 1]$, meaning we can use all analysis for both cases without having to rescale or normalize the results. Also, the range is continous and can be used to perform correlational analysis with other continous variables selected by the user.

§

Project structure

When starting a project, the first step is to design a structure which makes sense for what we're building. We can look at the structure as how our folders, files, classes and functions will be ordered. This is extremely important since we'll be writing a lot of modular code, and things can get lost easily.

There are multiple ways of approaching a project structure design; it really depends on each personal taste. We'll be implementing a frontend, backend, configuration files and utility functions, so a main folder with subfolders as packages makes sense for our case:

We will create a master folder (*project folder*) where the inputs, outputs and source code will reside. This folder will have the following structure:

- `sentiment-analysis-with-python`: The project folder.
  - `datasets`: Where our datasets will be downloaded and read from.
  - `outputs`: Where our analyses will be written in.
  - `se_env`: Our virtual environment.
  - `src`: Where the source code will be located.
  - `requirements.txt`: Where all the dependencies will be specified.

The application will be divided into multiple **packages** denoted by folders. Each package will contain **modules** belonging to a similar functionality, denoted by files. Each file will serve a specific purpose and will contain one main **class**. Each class will contain one or more **methods** denoted by functions.

The package structure inside `src` will be as follows:

- `src`: Where the source code will be located.
  - `main.py`: Our main function which the user will execute (*this will be the only point of contact for a typical user*).
  - `application`: Frontend packages
  - `config`: Where we will store configuration files and default parameters for our application.
  - `sentiment_analysis`: Where all the packages related to the analysis will be located.
    - `models:` Where the model definition for VADER will be located.
  - utils: Where helper scripts will be located.

We will define classes using two approaches: the mixin approach and the classical single/multiple inheritance approach.

In Python, a mixin is a class that provides methods to other classes but is not considered a base class itself. In short, a mixin is a class that extends the functionality of other classes without requiring initialization using an `__init__` function, calls to `super()` to initialize parent classes, and other aspects that a conventional class would require.

Below are some other advantages of a mixin approach over conventional classes:

- The main class inherits all mixin class methods directly from `n` mixin classes.
- Parameters & data are defined on the main function, so there's no need to redefine attributes inside mixin classes.
- `self` from the main class is automatically accessible inside mixin class methods.

§

Preparing the environment

We will start by creating our main folder along with the required subfolders:

Code

```
mkdir sentiment-analysis-with-python/src
```

```
cd sentiment-analysis-with-python
```

```
mkdir datasets, outputs
```

```
cd src
```

```
mkdir application, config, sentiment_analysis, utils
```

Creating a virtual environment

We will create a virtual environment tailored for this project. We'll be using Python 3.9.0, which we will need to download if we haven't already.

We will then create our environment using the installed Python version. This environment will be located inside our project folder, `sentiment-analysis-with-python`:

Code

```
C:\Users\username\AppData\Local\Programs\Python\Python39\python.exe -m venv 'se_env'
```

Installing required libraries

Since we'll be using a fair ammount of libraries, its easier for us and for the final user to define a `requirements.txt` file, the reason being we can quickly install all dependencies with a single `pip` command.

The `requirements.txt` file will be located inside our project folder, `sentiment-analysis-with-python`, and will contain the following packages:

```
customtkinter
xlsxwriter
matplotlib
nltk
numpy
pandas
polars
pyarrow
pyinstaller
scikit-learn
seaborn
spyder-kernels
textblob
tk
tomli
wordcloud
```

We can then activate our virtual environment from within the main project folder on the current terminal session, and install all the dependencies. Keep in mind that `Activate.ps1` is meant for `PowerShell`; other shells have their own `activate` script.

Code

```
se_env/Scripts/Activate.ps1
```

```
pip install -r requirements.txt
```

§

Modules

Each package will have multiple files inside, each one representing a module. For each module, we will follow the snake case format with single leading underscore practice `_module.py`, where each file representing a module will be signalled as an internal module; a single leading underscore in front of a variable, a function, or a method name means that these objects are used internally. This also means that, when importing modules using a wildcard `*`, these will not be imported.

We will start by creating our modules inside each `src` package and defining boilerplate code inside them. Our definitions will not make much sense now, but will serve as our project's skeleton and will be explained later on:

```
application
|   _app.py
```

Boilerplate code for `_app.py`:

Code

```
# Third-party packages
import customtkinter
import matplotlib
import matplotlib.pyplot as plt
import tkinter

# Built-in packages
import os
import shutil
import time
import warnings
warnings.filterwarnings("ignore")

# Internal packages
import utils
import sentiment_analysis

# Define classes and functions
class SetGlobalParams(utils.GetParameters):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

class HelpPrompt(SetGlobalParams,
                 customtkinter.CTkToplevel):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

class AboutPrompt(SetGlobalParams,
                  customtkinter.CTkToplevel):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

class MainApplication(SetGlobalParams,
                      customtkinter.CTk,
                      sentiment_analysis.SentimentAnalysis):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def open_help_prompt(self):
        pass

    def return_threshold_val(self, value):
        pass

    def open_about_prompt(self):
        pass

    def change_appearance_mode_event(self, new_appearance_mode: str):
        pass

    def runModel(self):
        pass

if __name__ == '__main__':
    main()
```

It's always best practice to import modules using the following rules:

- Third-party packages first
- Built-in packages second
- Internal packages third
- All imports should ideally be listed alphabetically in ascending order.

```
sentiment_analysis
|   _results_analysis.py
|   _results_writer.py
|   _sentiment_analysis.py
|
├──models
|   |   vader.py
```

Boilerplate code for `_results_analysis`:

Code

```

```

Boilerplate code for `_results_writer.py`:

Code

```

```

Boilerplate code for `_sentiment_analysis.py`:

Code

```

```

```
utils
|   _get_parameters.py
|   _preprocess_data.py
|   _string_formatting.py
```

Boilerplate code for `_get_parameters.py`:

Code

```

```

Boilerplate code for `_preprocess_data.py`:

Code

```
```

Boilerplate code for `_string_formatting.py`:

Code

```
```

Now that we have our modules defined, we can package them.

§

Packages

As mentioned before, we will express our packages as folders inside `src`. For a folder to be used as a package, we need to include a special file, `__init__.py`, inside each package folder. This way, we will be able to import entire packages across files without needing to import each module explicitly; when we import a folder as package from another file, the Python interpreter calls `__init__.py` and looks for module imports. If it finds any, it imports said module as part of the package we're importing.

The basic structure of an `__init__.py` file is very simple. We import the modules we wish to include in the package, along with the classes we wish to use in other files:

```
from ._app import SetGlobalParams
```

In this case, `_app` would be our module (*file*), and `SetGlobalParams` would be a class inside the module we wish to use across our project files.

We prepend our module import with a dot `.` since, even though we're on the same directory as our modules, the Python interpreter doesn't know that. This is why we explicitly have to specify we would like to import the `_app.py` module which is located inside the current directory (`.`).

We can create an `__init__.py` file per package, and populate with our module imports.

Imports for `aplication/__init__.py`:

Code

```
from ._app import SetGlobalParams, HelpPrompt, AboutPrompt, MainApplication
```

Imports for `sentiment_analysis/__init__.py`:

Code

```
from ._results_writer import ResultsWriter
from ._results_analysis import ResultsAnalysis
from ._sentiment_analysis import SentimentAnalysis
```

Imports for `utils/__init__.py`:

Code

```
from ._get_parameters import GetParameters
from ._preprocess_data import PreprocessData
from ._string_formatting import StringFormatting
```

Note that we're not importing any external package here; we're simply importing our own modules (*files*) along with the classes we defined earlier.

§

Configuration Files

Configuration files are extremely usefull when we're writing code and would like to provide a way to configure our application without messing up with the code itself. This technique provides a way for an external user, or even ourselves, to fine-tune any modifiable parameter which changes the behavior of our application's interface or even backend.

The idea is to leave the configuration for the main user-defined parameters in the GUI, and specific parameters such as the GUI's font family, font size, text color, and other parameters inside a configuration file. This way, we purpose the GUI exclusively for model operation, and the configuration files for the application appearance which the general user would not necessarily want to modify. In short, it keeps distractions away while keeping a backdoor for more fine-grained customization.

Also, a parameters file can be used as a collection of default values for the user-defined variables we will include later on. If the GUI were to malfunction, the user could still access the parameters file and set default values for all variables, effectively bypassing the GUI while making none to minor direct modifications to the code itself.

There are multiple file formats such as `.hcl`, `.json` and `.yaml` tailored for configuration files. In this project, we will stick with the `.toml` file format for both configuration and parameters files because of its multiple advantages:

- A
- A
- A
- A
- A

Application configuration

The application configuration file will include parameters related to the appearance of the UI.

User parameters

The user parameters file will include default values for all user-defined parameters inside the GUI.

The `resources` folder will contain all libraries required from `sentimentAnalysisApp`. This is convenient because as we will see in a moment, we can simply import the entire folder and all the libraries inside will be included in our `main.py` file.

For this to work, we will need to create a `__init__.py` file, which will be in charge of including all modules from within our `resources` folder.

The `resources` folder will contain the following elements:

- `__init__.py`
-

Defining TOML files

The `.toml` file format is ...

Defining a TOML configuration file

Aaaa

Defining a TOML parameters file

Input method can be `"Download Mode"` or `"Read Mode"`. This is selected by the user using the Radio Button object. Only one option can be selected simultaneously.

Front-End

UI Components

Text log

We will define a `textlog` object which will display useful messages to the user. Here, the user will be able to monitor the end-to-end process.

A typical insertion has the following generalized structure:

Code

```
# Enable textlog entries
self.textlog.configure(state="normal")

# Insert the required text using the print_position attribute
self.textlog.insert(self.print_position, f"ENTERING DOWNLOAD MODE\n\n")

# Disable textlog entries
self.textlog.configure(state="disabled")

# Update idle tasks
# i.e. Make text appear during run, regardless of current process status
self.update_idletasks()
```

We will also use an alternative structure which will include a string pre-formatting function. This will be useful when we're printing a variable name along with its value to screen; it will make the output clearer.

For these two actions, we will create two separate functions by creating a `string_formatting.py` library inside our `resources` folder:

Code

Code

Back-End

Preprocessing the complete data set

We will define a `preprocessData()` function in order to either download or load data sets, depending on the user's choice.

This function will accept keyword arguments `**kwargs` which we will provide upon the function call later on.

It is bad practice to specify required parameters as part of the `**kwargs` since, if left unspecified, our program will not find them and return an error. To ensure this does not happen, we mentioned that we created a `.toml` file containing default arguments for all keyword arguments passed.

Inside `preprocessData()`, we will define 4 child functions:

- `downloadData()`

downloadData()

This function will download the user-specified datasets if the mode is set to `Download Mode`. A complete data set can be in the form of a .gz file, or a .tsv file.

If a dataset already exists, it will not be downloaded. Otherwise, it will be downloaded

Default arguments

Management

There are some measures we can implement in order to avoid bugs derived from unspecified arguments. These will ensure that our program runs smoothly:

- Specifying a dictionary of default arguments: In this example, a `.toml` file with default arguments is passed. These parameters are initialized at the beginning of `main.py`, so all required arguments will have a default value without exception. These default parameter files are not meant for the user to modify. Instead, the user can change parameters by using the GUI.
- Specifying exception handling carefully: Each input must have an exception handler. If a value. We already have implemented a failsafe for default arguments, but still, the user might input arguments in incorrect

forms. To counteract this, exception handles are implemented on each critical step of the downloading, reading and writing process.

Source file for target URL specification

A `.txt` file must be created inside the input directory, using the name `source.txt`. This file contains all URLs for downloading target datasets which we wish to process and analyze. If this file is not provided, the program will raise an exception.

Preprocessing data

VADER Model

https://www.youtube.com/watch?v=QpzMWQvxXWk
https://www.youtube.com/watch?v=Ew72EAgM7FM
https://towardsdatascience.com/the-most-favorable-pre-trained-sentiment-classifiers-in-python-9107c06442c6
https://www.google.com/search?q=beautiful+python+gui&tbm=isch&client=firefox-b-d&hl=en&sa=X&ved=2ahUKEwja2KT4s779AhX-2skDHWv5B2sQrNwCKAB6BQgBEPQB&biw=1920&bih=919#imgrc=IY1ipTmyw464FM&imgdii=Pu6U445c1OHxsM

Library:

- `from nltk.sentiment import SentimentIntensityAnalyzer`

Assumptions:

- Stop words are removed.
- Each word is scored and combined to a total score.\
- Context is not taken into account.

§

Conclusions

We've reviewed multiple yet simple mechanisms we can employ to make our code cleaner, more elegant, modular, usable, scalable and safer. These measures can not only help us become better programmers but better collaborators. It will make reading code a pleasure instead of an agonizing process and instantly boost our credibility.

§

References

- https://cseweb.ucsd.edu/~jmcauley/datasets/amazon_v2/
- https://towardsdatascience.com/whats-the-meaning-of-single-and-double-underscores-in-python-3d27d57d6bd1
- http://www.qtrac.eu/pyclassmulti.html
- https://www.pythontutorial.net/python-oop/python-mixin/
- http://python-history.blogspot.com/2010/06/method-resolution-order.html

- https://huggingface.co/blog/sentiment-analysis-python

§

Copyright