# Sentiment Analysis With Python, Pt. 2

§

Made with Obsidian

Type portfolio-project | Category data-science | Technologies Python, PowerShell

Website Post Link

In the last segment of this 5-piece Portfolio Project, we discussed what sentiment analysis is and the types of approaches for this technique. We also designed our application's architecture, created our environment and included our project's dependencies, defined our project's directory structure and the interaction between packages & modules, and implemented a fully-fledged GUI using `customtkinter` and `tkinter`.

In this second part, we will design the preprocessing module along with the heart of our application: the sentiment analysis package.

For the preprocessing module, we will implement a main class and several functions to help us download/load datasets, cast entries to their appropriate data types, and extract the columns of interest.

Finally, we will define our models for the sentiment analysis package and include an analysis execution module.

The complete project, including all the resources used, can be found in the Portfolio Project Repo.

§

# Table of Contents

# Preface

Recalling from the last segment, we are looking to build a sentiment analysis GUI where the user can download or read one or more datasets, define different columns such as a target column, an index column, and a rating column, select between two different rule-based sentiment analysis modules (*VADER, TextBlob*), and choose which set of analyses to apply, based on a maximum of 4 possible aggregation columns.

In the last segment, we designed our concept GUI, defined our general project structure using structure charts, and explained how our project would be organized regarding the folder and file structure. We also created all our necessary packages and modules and included a main structure for each file.

In this segment, we'll work with the following packages and modules:

- `utils`
    - `_preprocess_data.py`
- `sentiment_analysis`
    - `models`
        - `vader.py`
    - `_sentiment_analysis.py`

Where the `_preprocess_data.py` module will be responsible for downloading/reading the datasets and making the necessary transformations, the `vader.py` module will contain the VADER model implementation, and the `_sentiment_analysis.py` module will be responsible for running the sentiment analysis upon our main application call.

# Preprocessing

Before we implement our model, we'll need to preprocess our data. We will specify a module inside our `utils` package called `_preprocess_data.py`.

This module will have the following responsibilities:

- Download a dataset/set of datasets if the user specifies Download mode.
- Read a dataset/set of datasets regardless of chosen mode option.
- Validate if selected columns are in selected datasets.
- Cast required data types.
- Select user-defined columns.
- Return a preprocessed DataFrame object.

We will declare six methods inside our `PreprocessData` module:

- `downloadMode`
    - `downloadData`
- `readMode`
    - `selectCols`
    - `castTypes`
    - `readData`

If we recall the general structure, the `_preprocess_data.py` module is directly connected with the `SentimentAnalysis` package. Thus, it will only be called by the latter:

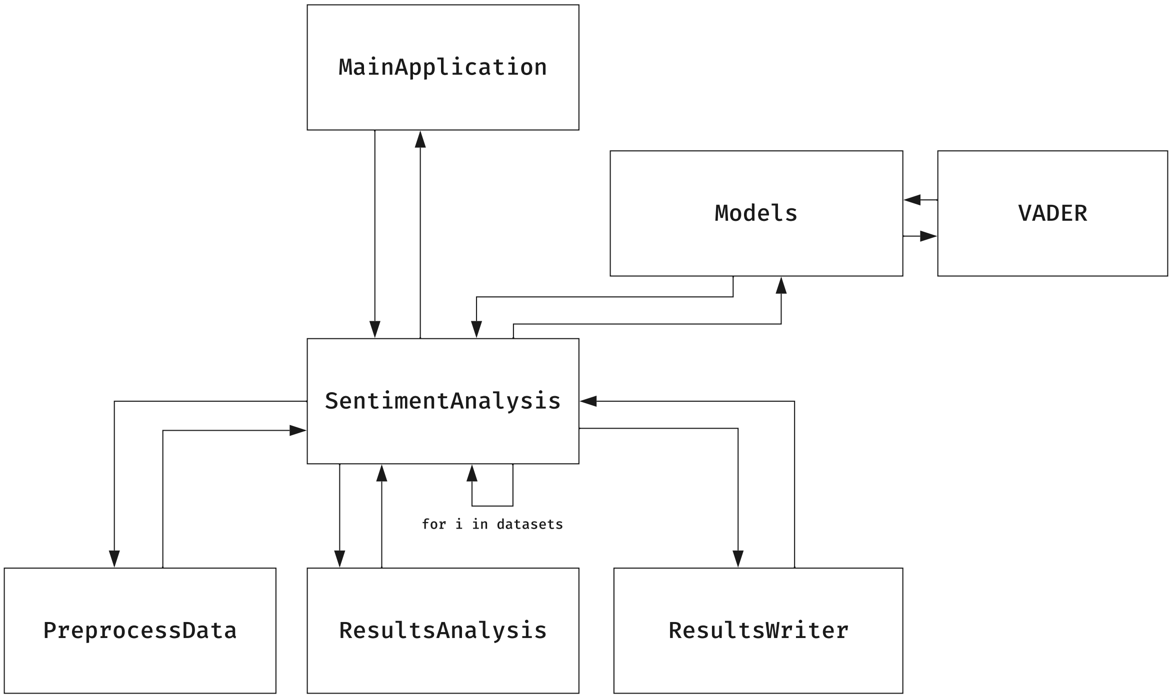


*FIGURE 1: DATA PREPROCESSING GENERAL STRUCTURE*

If we zoom in to our `_preprocess_data.py` module, we have the following structure:

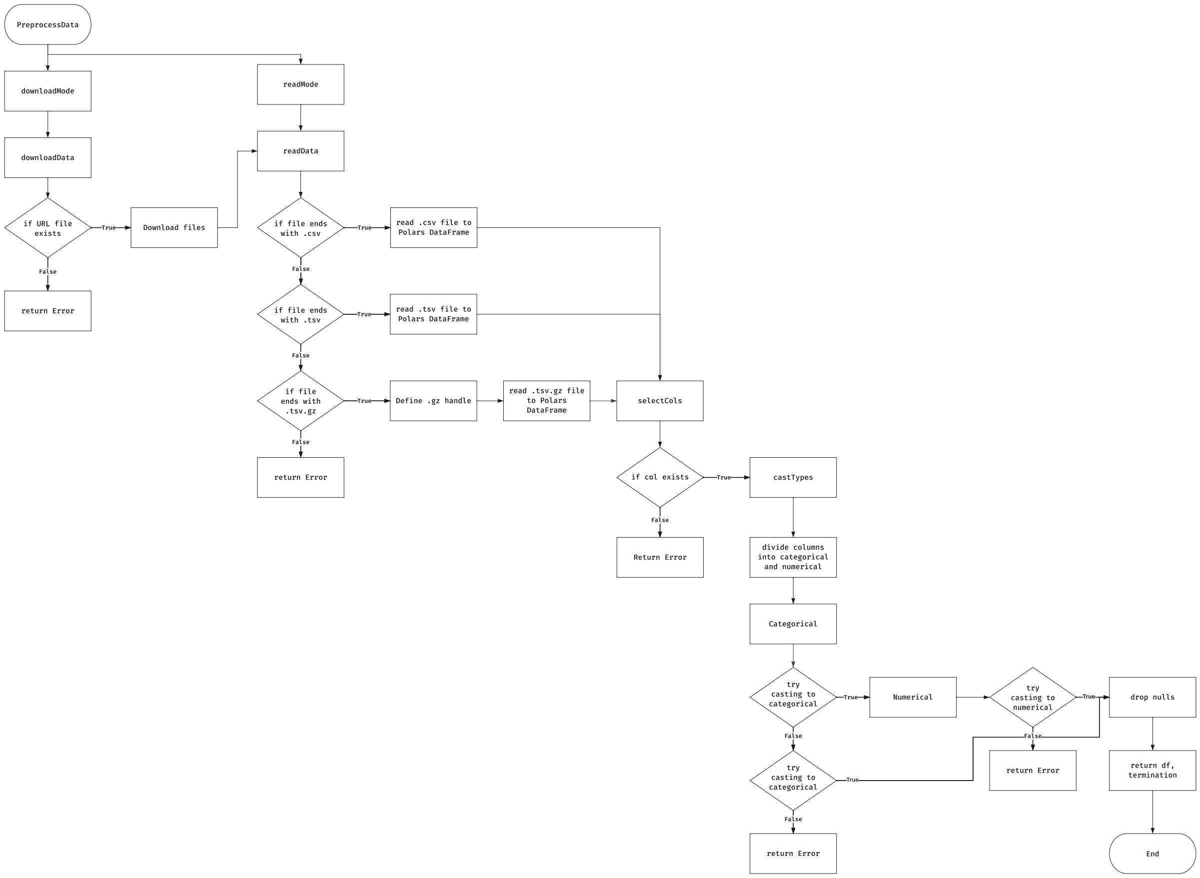


FIGURE 2: DATA PREPROCESSING STRUCTURE CHART

There are some key points worth mentioning:

- **Why choose these specific file formats for dataset reading?**

- Normally, when working with big data files, we encounter `.csv`, `.tsv`, `.parquet`, `.avro`, or compressed file formats.

- Even if we don't have our datasets in the required file formats, the first two are extremely easy to convert to and well-known in the industry.

- The last two are serialized file formats, which we did not include for a specific reason: They can contain unknown data schemas; schema handling in the preprocessing steps could easily result in errors.

- Finally, the most common compression utility for interchanging single files is the gzip standard; contrary to `.zip`, `.gz` can only compress single files, making it ideal for file exchange.

- **What about the `.gz` file format?**

- As mentioned above, a `.gz` file is a compressed file using the GNU Zip (*gzip*) utility. As convenient as it is for exchanging files, the `.gz` file format requires special handling.

- A file handler in Python refers to an object containing a collection of methods for interacting with the file, such as opening, reading, and writing.

- In order to interact with a `.gz` file, we need to use a file handler by including the `gzip` library.
- Once we create a file handler, we can open, read, and write from and to the file.
- **Data type casting exception handling.**
- There are times when we might have an ID column denoting each entry's index (*e.g., Review ID: A1290*)
- By default, this column is treated as Categorical (`string`) in our program, which makes sense since we might have an identifier consisting of alphanumerical characters. Still, there are instances where we might get a numeric column. If this happens, `Polars` will sometimes present problems casting to a Categorical data type. Thus, we define an exception-handling method that takes care of that.
- **Why read to a `Polars` DataFrame?**
- `Polars` reading methods are faster than `Pandas`; `Polars` represents data in memory with `Arrow` arrays, while `Pandas` represents data in memory in `NumPy` arrays. Apache Arrow is an emerging standard for in-memory columnar analytics that can accelerate data load times, reduce memory usage and accelerate calculations by employing parallel processing capabilities.
- If we are to read multiple files and don't support serialized file formats in our program, it'll be faster by using `Polars` vs. `Pandas`.

Once we know the structure, we'll head to our `_preprocess_data.py` file and import the following:

### Code

```
# Third-party packages
import gzip
import polars as pl

# Built-in packages
import os
import time
import urllib.request
import warnings
warnings.filterwarnings('ignore')

# Internal packages
from ._string_formatting import StringFormatting
```

We will also need to turn on the `Polars` global string cache:

### Code

```
pl.toggle_string_cache(True)
```

This will ensure that casts to Categorical types have the categories when string values are equal. Else, we'll get an error when trying to cast.

Finally, we will define our `PreprocessData` class:

### Code

```
class PreprocessData(StringFormatting):
    '''
    - Download (if user specifies) and read datasets into a
    Polars DataFrame object.
    - Cast required data types.
    - Select user-required columns.
    - Return a processed Polars DataFrame.
    '''
```

# 1. downloadData

If the user selects the Download mode and a URL file exists in the `datasets` folder, the datasets in the specified URL will be downloaded to the said folder.

A typical URL file is assumed to be in the `.txt` file format and should look like such:

```
https://example.com/dataset_01.csv
https://example.com/dataset_02.csv
https://example.com/dataset_03.csv
https://example.com/dataset_04.csv
```

Each newline entry is considered a different URL and thus treated as such.

An error will be returned if there is no such URL file and the Download mode is selected.

For this, we'll need to implement a downloader function inside our `PreprocessData` class in our `_preprocess_data.py` module:

### Code

```python
def downloadMode(self):
    '''
    Enter download mode, where all URLs specified on source.txt.
    will be downloaded in the datasets folder.
    '''
    def downloadData():
        '''
        Download the dataset if it does not yet exist.
        '''
        # Define the input file for downloading files using URLs
        input_file = os.path.join(self.project_path, self.var_rdir.get(),
self.var_sourceurl.get()) # type: ignore
        counter = 1

        # Try to get the source file.
        try:
            file = open(input_file)
            file.close()

        # If it does not exist, return error and notify the user
        except Exception as ex:
            self.insertLog(f'ERROR: "{input_file}" DOES NOT EXIST\n\n')

            # We must manage this return value in the function calls.
            return ex

        with open(input_file, 'r') as url_file:
            len_urls = len(url_file.readlines())

            textvar_downloading = self.padStr('DOWNLOADING FROM:', self.var_sourceurl.get()) #
type: ignore
            textvar_linenum = self.padStr('TOTAL URLS:', len_urls)

            # Wait for user to see log
            time.sleep(float(self.var_wait_time.get())) # type: ignore

            self.insertLog(f"{textvar_downloading}\n",
                            f"{textvar_linenum}\n")

        with open(input_file, 'r') as url_file:
            progress_1_step = 1/len_urls
            progress_1_perc = 0
            self.progressbar_1.start() # type: ignore
            for url in url_file:
                # Split on the rightmost / and take everything on the right side of that
                name = url.split('/')[-1].strip('\n')
                filename = os.path.join(self.project_path, self.var_rdir.get(), name) # type:
ignore


                if os.path.isfile(filename):
                    self.insertLog(f"ALREADY DOWNLOADED:\n{filename}\n\n")

                if not os.path.isfile(filename):
                    self.insertLog(f"DOWNLOADING {counter}/{len_urls}:\n{filename}\n\n")
                    urllib.request.urlretrieve(url, filename)
                    self.insertLog(f"DOWNLOADED {counter}/{len_urls}:\n{filename}\n\n")
```

```
            progress_1_perc += progress_1_step
            self.progressbar_1.set(progress_1_perc) # type: ignore
            self.update_idletasks() # type: ignore

            counter += 1

    self.progressbar_1.stop() # type: ignore

    return None

downloadData()

return None
```

Once we have our `downloadMode` method, we can proceed with the reading.

# 2. readMode

As stated earlier, `readMode` will include everything we need to read our dataset to a `Polars` DataFrame object. This includes reading the files, selecting the columns, casting the datatypes, and dropping null values.

We will wrap all of these methods inside a `readMode` function below our `downloadData` method:

### Code

```
def readMode(self, dataset):
    '''
    Enter read mode where a dataset will be read
    if it exists on datasets directory.
    '''
```

## 2.1 selectCols

We want to select the columns defined by our user and only use those as our new DataFrame. The `selectCols` method will try to choose the columns; an error will be returned if they do not exist. It will also specify which columns are meant to be Categorical and which are meant to be Numerical. This will help when we implement our `castTypes` method:

### Code

```python
def selectCols():
    '''
    Get columns required by the user.
    '''
    # Build the column aggregation list.
    cols_agg_raw = [self.col_entry_1.get(), # type: ignore
                    self.col_entry_2.get(), # type: ignore
                    self.col_entry_3.get(), # type: ignore
                    self.col_entry_4.get()] # type: ignore

    cols_agg = []

    # If the user did not specify a given column, do not append it.
    for col in cols_agg_raw:
        if col != '':
            cols_agg.append(col)

    # Extend the list with all other columns
    # We will use this to test if any of the columns do not exist in the dataframe
    cols_all = cols_agg.copy()
    cols_all.extend([self.col_rating.get(), # type: ignore
                        self.col_target.get(), # type: ignore
                        self.col_id.get()]) # type: ignore

    cols_text = cols_agg.copy()
    cols_text.extend([self.col_target.get(), self.col_id.get()]) # type: ignore

    return cols_all, cols_text, self.col_rating.get(), self.col_target.get() # type: ignore
```

## 2.2 castTypes

Once our columns are selected, we must cast them into appropriate data types.

For this to work, we should have the following schema:

- `Agg` columns (max 4, min 1). Can be `str`, `int` or `float` type.
- `ID` column. Can be `str` or `int` type.
- `Target` column. Requires `str` type.
- `Rating` column. Can be `int` or `float` type.

### Code

```
def castTypes(df, cols_text, col_rating):
    '''
    Cast columns to appropriate data types for model execution.
    '''
    # Cast string types
    for text_col in cols_text:
        try:
            df = df.with_column(pl.col(text_col).cast(pl.Categorical))
        except:
            try:
                df = df.with_column(pl.col(text_col).cast(pl.Float64))
            except:
                self.insertLog(f'ERROR: COULD NOT CAST {text_col}.\nPLEASE REVIEW DATA
ENTRIES\n\n')

    # Cast numerical types
    try:
        df = df.with_column(pl.col(col_rating).cast(pl.Float64))

    except:
        self.insertLog(f'ERROR: COULD NOT CAST {col_rating}.\nPLEASE REVIEW DATA ENTRIES\n\n')

    return df
```

## 2.3 readData

We can now implement a reading method that will read a dataset depending on the file format, select the appropriate columns, cast them to the suitable data types, drop any null value present, and return the file format extension. This last step will be helpful when we're setting the dataset name as our identifier:

### Code

```
    def readData(dataset):
        '''
        This function will read one file per iteration
        and return a dataframe.
        It will perform the following tasks:
            - Read the file if it exists, and is of correct file format.
            - Select the user-defined columns if they exist.
            - Cast the user-defined columns to correct data type.
            - Return a processed Polars DataFrame object.
        A data set can be in the form of:
            - A .csv file.
            - A .tsv file.
            - A compressed .gz file containing:
                - A .csv file.
                - A .tsv file.
        For column selection:
            - Agg columns (max 4, min 1). Can be str, int or float type.
            - ID column. Can be str or int type.
            - Target column. Requires str type.
            - Rating column. Can be int or float type.
        '''

        # Define target path for a given iteration
        read_target = os.path.join(self.project_path, self.var_rdir.get(), dataset) # type:
ignore

        # If a .csv file exists, read the .csv file
        if read_target.endswith('.csv'):
            self.insertLog(f"READING:\n{read_target}\n\n")
            termination = '.csv'

            # Wait for user to see params
            time.sleep(float(self.var_wait_time.get())) # type: ignore

            # Read file into df
            df = pl.read_csv(read_target, sep = ',', ignore_errors=True)

            self.insertLog(f"CONCLUDED READING:\n{read_target}\n\n")

            # Wait for user to see params
            time.sleep(float(self.var_wait_time.get())) # type: ignore

        # If a .tsv file exists, read the .tsv file
        elif read_target.endswith('.tsv'):
            self.insertLog(f"READING:\n{read_target}\n\n")
            termination = '.tsv'

            # Wait for user to see params
            time.sleep(float(self.var_wait_time.get())) # type: ignore

            # Read file into df
            df = pl.read_csv(read_target, sep = '\t', ignore_errors=True)

            self.insertLog(f"CONCLUDED READING:\n{read_target}\n\n")

            # Wait for user to see params
```

```python
            time.sleep(float(self.var_wait_time.get())) # type: ignore

        # If a .gz file exists, read the .gz file without explicitly decompressing
        elif read_target.endswith('.tsv.gz'):
            self.insertLog(f"READING:\n{read_target}\n\n")
            termination = '.tsv.gz'

            # Wait for user to see params
            time.sleep(float(self.var_wait_time.get())) # type: ignore

            # Read file into df
            with gzip.open(read_target) as compressed_file:
                df = pl.read_csv(compressed_file.read(), sep = '\t', ignore_errors=True)

            self.insertLog(f"CONCLUDED READING:\n{read_target}\n\n")

            # Wait for user to see params
            time.sleep(float(self.var_wait_time.get())) # type: ignore

        else:
            self.insertLog(f"ERROR:\n{read_target} IS NOT A VALID FILE\n\n")

            # Return None
            return None

        textvar_colnum = self.padStr('COLUMN NUMBER:', len(df.columns))
        self.insertLog(f"{textvar_colnum}\n\n",
                       f"CHECKING COLUMNS\n\n")

        # Wait for user to see params
        time.sleep(float(self.var_wait_time.get())) # type: ignore

        # Extract column list
        cols_all, cols_text, col_rating, col_target = selectCols()

        # Check if user-defined columns exist
        for col in cols_all:
            try:
                df.select(col)
            # If it does not exist, return error and notify user
            except Exception as ex:
                self.insertLog(f'ERROR: "{col}" DOES NOT EXIST\n\n')

                # We will need to manage this return value in the function calls.
                return ex

        df = df.select(cols_all)
        df = castTypes(df, cols_text, col_rating)
        df = df.drop_nulls()

        return df, termination

    df, termination = readData(dataset) # type: ignore

    return df, termination
```

Finally, we will include the following statement at the end of our module:

### Code

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

This will ensure that our module is not executed on import and is instead executed upon explicitly calling it.

§

# Sentiment analysis package

Once we have our preprocessing module, we'll implement our models and a sentiment analysis execution module. We'll be working inside our `sentiment_analysis` package.

The structure will be defined as follows:

- `sentiment_analysis`
  - `models`
    - `vader.py`
  - `_sentiment_analysis.py`

We are not defining a specific module for the TextBlob model since its implementation is straightforward and can be done inside the `_sentiment_analysis.py` module.

We will start by defining the VADER module.

## 1. VADER

We will be working on the `vader.py` module. The VADER model can be accessed through the `nltk` library. We can first import the required libraries:

### Code

```
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
```

We'll next define a function that will:

- Download the required VADER lexicon (*required for sentiment analysis*)
- Define a VADER model object.
- Return a VADER model instance.

### Code

```python
def vaderModel():
    '''
    Download the VADER lexicon first.
    Define Sentiment Analyzer object.
    Return model object.
    '''

    nltk.download('vader_lexicon')
    model = SentimentIntensityAnalyzer()

    return model

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

If we recall the previous segment, we did not specify any model import in our `__init__.py` script. This is because we'll call the VADER model from within the `sentiment_analysis` package.

A typical VADER model import from within the said folder, will consist of the following:

### Code

```python
from sentiment_analysis.models import vader
```

This will automatically import our function if, indeed, we are importing from within the same folder.

# 2. Sentiment analysis

We must implement a module that performs the sentiment analysis on our preprocessed data using the predefined model above. We must also include a TextBlob implementation directly into the `_sentiment_analysis.py` module.

If we recall the general structure, we had the following:

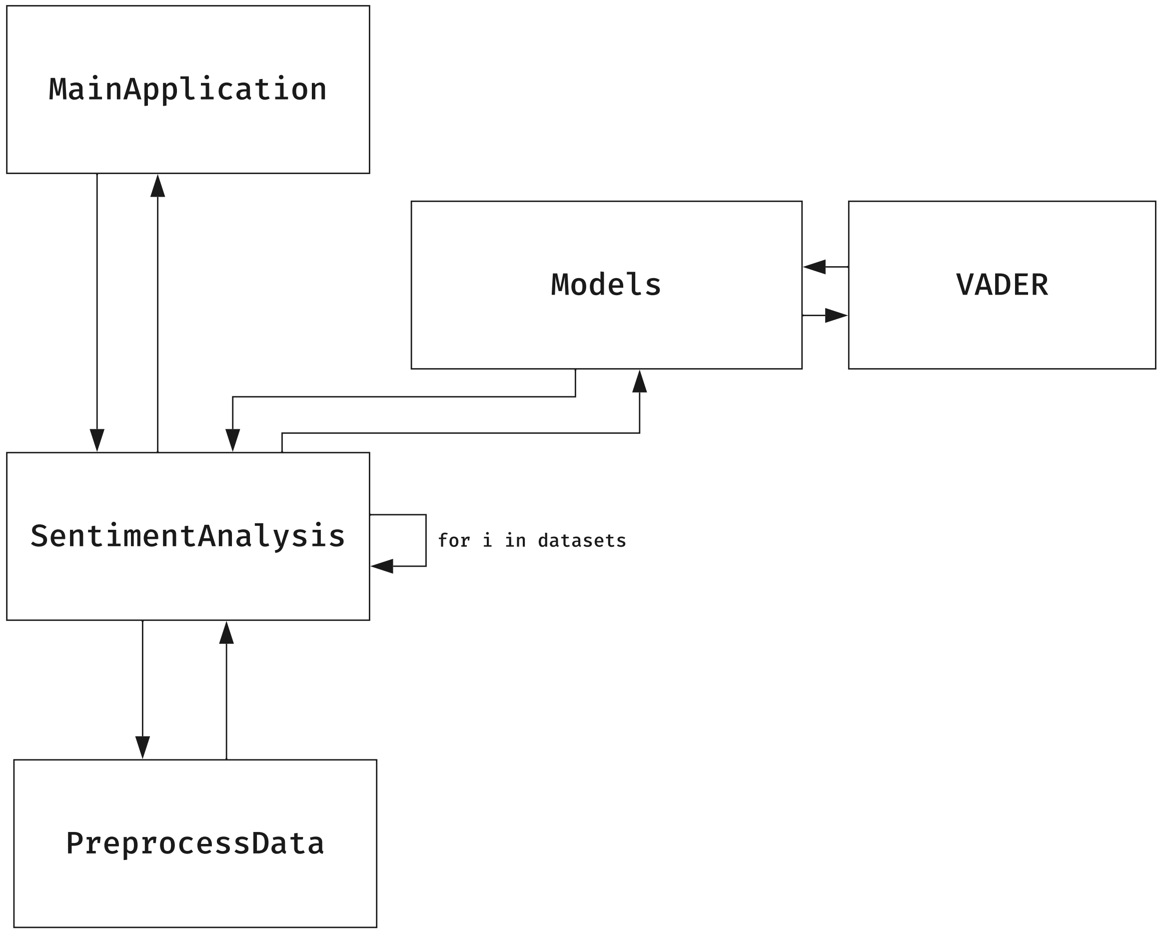


*Figure 3: Sentiment Analysis Execution Process Flow*

If we zoom in on our Sentiment Analysis package, we have the following structure:

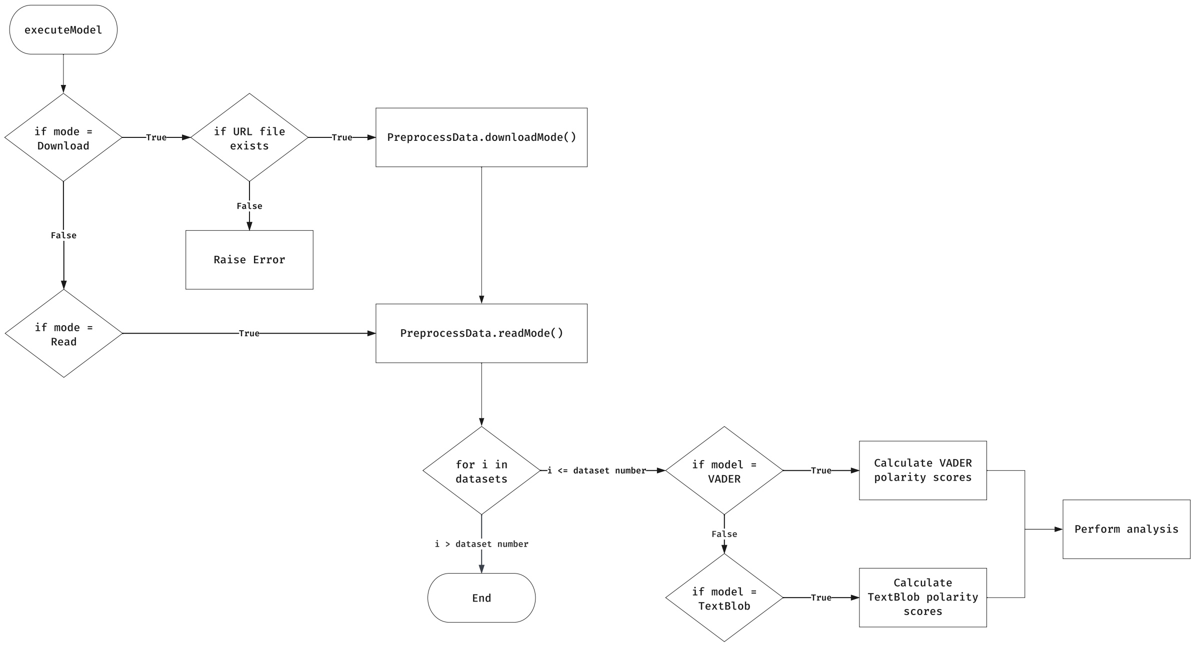


*FIGURE 4: DETAILED SENTIMENT ANALYSIS EXECUTION STRUCTURE CHART*

We will also need to include progress bar updates for the model execution function since we want to reflect the progress percentage using each analyzed target entry (*row*) as the step size.

As we saw in our previous segment, we will need to import some modules beforehand:

## CODE

```python
# Third-party packages
import numpy as np
import pandas as pd
import polars as pl
import pyarrow
from textblob import TextBlob

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

# Internal packages
from utils import PreprocessData
from sentiment_analysis.models import vader
from ._results_analysis import ResultsAnalysis
from ._results_writer import ResultsWriter
```

- `NumPy`, `Pandas`, `Polars`, and `PyArrow` will be used to manipulate our data structures.
- `TextBlob` will be used to implement the `TextBlob` model.
- `os` will be required for system interaction.
- `time` will be required for time pauses between steps.

- `PreprocessData` will be used to include our preprocess datasets.
- `vader` will be used to introduce the VADER model to our system.
- `ResultsAnalysis` and `ResultsWriter` are not yet implemented but will be used to analyze our results and, eventually, write them to the `outputs` folder.

The two target functions will be implemented as methods of our previously defined `SentimentAnalysis` class:

#### Code

```
# Define SentimentAnalysis class
class SentimentAnalysis(PreprocessData,
                        ResultsAnalysis,
                        ResultsWriter):
```

## 2.1 applyModel

The `applyModel` function will have the following responsibilities:

- Select the target column.
- Define a `results` dictionary.
- Apply model based on user selection, for all datasets found in `datasets` folder for all target entries (*rows*).
- Append each result to the results `dictionary`.
- Update progress bar upon each entry analysis completion.
- Consolidate results from dictionary into a `Polars` DataFrame object
- Drop unused columns.
- Rename used columns accordingly.
- Join results with the original preprocessed DataFrame object.
- Return a Polars DataFrame containing preprocessed object and sentiment results.

We can define our `applyModel` function as follows:

#### Code

```python
def applyModel(self, df, dataset):
    '''
    Apply sentiment analysis model depending on the user's choice.
    '''
    self.insertLog(f'APPLYING MODEL TO {dataset.name}\n\n')

    time.sleep(float(self.var_wait_time.get())) # type: ignore

    # Run Sentiment Analysis
    target_data = df.select([self.col_id.get(), # type: ignore
                            self.col_target.get()]) # type: ignore

    # Select model
    if self.var_model.get() == 'VADER': # type: ignore
        model = vader.vaderModel()

        # Define a dictionary to store the resutls
        res_dict = {}

        counter = 0

        # Start progress bar
        total_items = len(df)
        progress_2_step = 1/total_items
        progress_2_perc = 0
        self.progressbar_2.start() # type: ignore

        for col_id, target in target_data.iterrows():
            results = model.polarity_scores(target)

            # Add results to the dictionary
            res_dict[col_id] = results

            # Increment counter
            counter += 1
            self.insertLog(f'ENTRY {counter} OF {total_items}\n')

            # Update progress bar
            progress_2_perc += progress_2_step
            self.progressbar_2.set(progress_2_perc) # type: ignore
            self.update_idletasks() # type: ignore

        # Stop progress bar
        self.progressbar_2.stop() # type: ignore

        df_res = (pl.from_pandas((pd.DataFrame(res_dict).T).
                                reset_index().
                                rename(columns={'index':self.col_id.get(), # type: ignore
                                                'neg':'NEG',
                                                'neu':'NEU',
                                                'pos':'POS',
                                                'compound':'CMP'}
                                                )
                            )
                )
```

```
        # Cast new dataframe types
        df_res = df_res.with_column(pl.col(self.col_id.get()).cast(pl.Categorical)) # type:
ignore

        # We won't be using Positive, Neutral, and Negative Scores, only Compound
        df_res = df_res.drop(columns = ['POS', 'NEU', 'NEG'])

        # Join with original DataFrame
        df_main = df.join(df_res, on = self.col_id.get(), how="inner") # type: ignore

    elif self.var_model.get() == 'TextBlob': # type: ignore
        # Define a dictionary to store results
        res_dict = {}

        counter = 0

        # Start progress bar
        total_items = len(df)
        progress_2_step = 1/total_items
        progress_2_perc = 0
        self.progressbar_2.start() # type: ignore

        for col_id, target in target_data.iterrows():
            results = TextBlob(str(target))
            polarity_score = results.sentiment.polarity # type: ignore

            # Add results to the dictionary
            res_dict[col_id] = {'compound':round(polarity_score, 4)}

            # Increment counter
            counter += 1
            self.insertLog(f'ENTRY {counter} OF {total_items}\n')

            # Update progress bar
            progress_2_perc += progress_2_step
            self.progressbar_2.set(progress_2_perc) # type: ignore
            self.update_idletasks() # type: ignore

        # Stop progress bar
        self.progressbar_2.stop() # type: ignore

        df_res = (pl.from_pandas((pd.DataFrame(res_dict).T).
                            reset_index().
                            rename(columns={'index':self.col_id.get(), # type: ignore
                                            'compound':'CMP'}
                                            )
                            )
                )

        # Cast new dataframe types
        df_res = df_res.with_column(pl.col(self.col_id.get()).cast(pl.Categorical)) # type:
ignore

        # Join with original DataFrame
        df_main = df.join(df_res, on = self.col_id.get(), how="inner") # type: ignore
```

```
    return df_main # type: ignore
```

- We use the `self.var.get()` method in order to get our variables defined in our main application.
- We use a dictionary, `res_dict`, to store our results per dataset.
- We initialize our progress bar and set the step size to be `1/total_items`, meaning once the script goes over all target entries, the progress bar will reach 100%.
- As mentioned earlier, both models output different results, so we'll have to rename the compound score and the polarity score to a common name for our analyses to work:
  - VADER:
    - Negative probability
    - Neutral probability
    - Positive probability
    - Compound: Renamed to `CMP`
  - TextBlob:
    - Polarity: Renamed to `CMP`
- We join the results DataFrame with the preprocessed DataFrame and return it as output.

## 2.2 executeModel

The `executeModel` function will be the second method inside our `SentimentAnalysis` class. It'll be in charge of selecting the operation mode (*Download / Read*), depending on the user's input; the Download mode will only be selected if the user explicitly states so, while the Reading mode will be selected regardless of the user's option.

It will then call the `applyModel` method defined above, the `performAnalysis`, and the `writeResults`, both of which we have yet to define.

### Code

```
def executeModel(self):
    '''
    Downloads the datasets if the user has requested Download operations.
    Loads the datasets one by one and performs the analysis per dataset.
    '''
    # Enter download mode
    if self.var_operation.get() == 'Download Mode': # type: ignore
        self.insertLog("ENTERING DOWNLOAD MODE\n\n")

        # Wait
        time.sleep(float(self.var_wait_time.get())) # type: ignore

        # Download files
        self.downloadMode()

    # Enter reading mode
    elif self.var_operation.get() == 'Read Mode': # type: ignore
        self.insertLog("ENTERING READ MODE\n\n")

        # Wait 2 seconds (for user to see params)
        time.sleep(float(self.var_wait_time.get())) # type: ignore

    # Count number of files
    file_count = 0
    with os.scandir(os.path.join(self.project_path, self.var_rdir.get())) as datasets: # type:
ignore
        for dataset in datasets:
            if dataset.name != self.var_sourceurl.get(): # type: ignore
                file_count =+ 1

    # Read all files in directory, regardless of mode
    with os.scandir(os.path.join(self.project_path, self.var_rdir.get())) as datasets: # type:
ignore

        if file_count == 0:
            self.insertLog("WARNING: FILE NOT FOUND\n\n")

            return None

        else:
            result_dict = {}
            for dataset in datasets:
                # Excluding URL source file
                if dataset.name != self.var_sourceurl.get(): # type: ignore
                    # Read mode
                    self.progressbar_1.set(0) # type: ignore
                    df, termination = self.readMode(dataset)
                    self.progressbar_1.set(1) # type: ignore
                    self.update_idletasks() # type: ignore
                    self.progressbar_1.stop() # type: ignore
                    # Apply model
                    df_main = self.applyModel(df, dataset)

                    # Remove termination from file
                    dataset_name = dataset.name.replace(termination, '')
```

```
                # Apply analysis on each iteration and save results\
                result_dict[dataset_name] = self.performAnalysis(df_main, dataset)

    # Perform writing
    self.writeResults(result_dict) # type: ignore

    return None
```

- Select an operation mode:
  - Download mode:
    - Download.
    - Read.
  - Read mode
    - If the datasets exist, read them.
    - Else, return error.
- Get the preprocessed data set.
- Call `applyModel` using the preprocessed DataFrame and `dataset` name as inputs.
- Remove the `dataset` name's file extension to use as the `results` dictionary key.
- Append a new dictionary entry:
  - Key: Stripped `dataset` name.
  - Value: Analysis results (*has not been implemented yet*)

If we look closely, the `executeModel` method will return `None`. This is because the results from this function are automatically written in the `outputs` directory.

As with previous examples, we will include the following statement at the end of our module:

### Code

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

Now, we have everything ready to start designing our analysis collection.

§

# Conclusions

In this section, we implemented a preprocessing module in charge of selecting the appropriate operation mode, downloading or reading from different file formats depending on the user's choice, selecting the required columns, and casting them to proper data types.

Next, we implemented our `sentiment_analysis` package, consisting of a VADER module and a sentiment analysis execution script responsible for selecting the correct model based on the user's input, applying the model to the target data, and consolidating the results for exporting to an analysis module not yet implemented.

In the next segment, we will implement an analysis module responsible for performing various analyses depending on the user's input and preparing these results for writing. We will also define a results writer module to consolidate all the analysis results and write them to either Excel workbooks or plots, depending on the user's choice.

# References


- Towards Data Science, The Most Favorable Pre-trained Sentiment Classifiers in Python
- cjhutto, vaderSentiment Documentation
- TextBlob, Documentation
- pola-rs, polars
- Polars, User Book
- Python Documentation, gzip


# Copyright