Poetic (poetic-py)

Release 1.0.3

Kevin Wang

Jan 23, 2021

CONTENTS

Poetic is a Python package to provide processing and prediction toolchain for poetry and literature in general. This website provides documentation and other package information. For the latest deveopment, please visit the github page: https://github.com/kevin931/poetic.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation Guide

Welcome to Poetic (poetic-py on PyPi). Please see below for installation details.

1.1.1 Pip

System-wide installation:

```
pip install poetic-py
python -c "import nltk; nltk.download('punkt')"
```

The usage of `virtualenv` is also recommended over system-wide installation.

1.1.2 Conda (Recommended)

Or, using `conda` (which I recommend):

```
conda install -c kevin931 poetic-py
python -c "import nltk; nltk.download('punkt')"
```

A few notes on Conda:

If you need documentation on how to get conda working/set up, this is a good guide.

There is a potential bug with tensorflow on Conda for Windows because of maximum path length limitation. If you are unable to install tensorflow from conda, please open an issue or refer to tensorflow's github Issue Tracker for support. For reference, you can also refer to Microsoft's guide . (Caution: Editing registry can have serious consequences. Please proceed with caution and be sure to know the issue.)

1.1.3 Supported Package Versions

Python

- 3.6
- 3.7
- 3.8

Note: Newer versions depend on dependencies.

Dependencies

- tensorflow >= 2
- nltk
- gensim

1.1.4 Troubleshooting

Pip Dependency Issues

If dependencies become an issue, try installing them separately from poetic:

```
pip install tensorflow
pip install gensim
pip install nltk

pip install poetic-py --no-deps
python -c "import nltk; nltk.download('punkt')"
```

If this does not solve the problem or there is any other unforeseeable problems, please head to our Issue Tracker and hopefully I can help you out!

Pip Caching

Pip, by default, caches downloads, which may result in downloading previously installed versions and leading to failure to upgrade. To address this, use the flag:

```
pip install --no-cache-dir poetic-py
```

Python 3.8 for Gensim on Conda

Current gensim builds from Anaconda's default channel do not work with python 3.8. If you are using python 3.8 and are unable to install poetic-py from conda because of this issue, use one of the two solutions from below until gensim has been updated:

```
conda install -c kevin931 poetic-py -c anaconda -c conda-forge
```

Or, this following option will be more specific:

```
conda install -c conda-forge gensim
conda install -c kevin931 poetic-py
```

1.2 Quickstart Reference

Welcome to Poetic (poetic-py on PyPi). Please see below for installation details.

1.2.1 Set Up/Assets Download

First-time Set Up (Default Behavior)

All machine learning models are not shipped with the package because of their size (~1GB). However, poetic provides a utility to download them upon first use.

When the models are needed, the package will call the `poetic.util.Initializer.check_assets()` method to check for assets and if assets are missing, it then subsequently calls the `poetic.util.Initializer.download_assets()` method which will prompt a command-line input:

```
The following important assets are missing:

Downloading from: https://github.com/kevin931/poetic-models/releases/download/v0.1-
↪alpha/sent_model.zip
Download size: 835MB.


Would you like to download? [y/n]
```

This behavior is intended to take bandwidth and user consent into consideration.

Download without Asking

If there is a use case in which command line input is undesirable or inefficient (or if you just don't want to), include the following commands to download them:

```
import poetic
assets =  poetic.util.Initializer.check_assets()
poetic.util.Initializer.download_assets(assets_status=assets, force_download=True)
```

1.2.2 Command-line Mode

Launch GUI

```
python -m poetic
```

Prediction with Sentence Input

```
python -m poetic -s "I am poetic."
```

Prediction with Plain Text File Input

```
python -m poetic -f <PATH_TO_FILE>
```

Save Results to File

```
python -m poetic -f <PATH_TO_FILE> -o <PATH_TO_TXT>
python -m poetic -f <PATH_TO_FILE> -o <PATH_TO_CSV>
python -m poetic -s "I am poetic. Are you?" -o <PATH_TO_TXT>
```

1.2.3 Use in Python

Import Behavior

Directly exposed classes:

- Predictor
- Diagnostics

Utility module:

- util

Make a Simple Prediction

```python
import poetic

new_pred = poetic.Predictor()
sentence_result = new_pred.predict("I am poetic. Are you?") # Directly
file_result = new_pred.predict_file("FILE_PATH.txt") # From a file
```

Prediction Diagnostics

```python
# sentence_result is from the previous section.
sentence_result.run_diagnostics()
sentence_result.to_file("SAVE_PATH.txt")
sentence_result.to_csv("SAVE_PATH.csv")
```

1.3 Tutorials and Examples

This section provides a high-level documentation of the package, including examples, rationale, common usages, tips, and everything without the burden of including all the attributes, methods, class information. If you would like to see the detailed documentation generated by docstring, head to the “Full Documentation” section.

Below is a list of topics covered:

1.3.1 Package Information and Resources

There are many ways to get more information on `poetic`, including official documentation, Github repository, docstrings, and built-in package information. This guide details some of these resources and how they can be best utilized.

GitHub Repository

The GitHub repository is where all the development and latest changes take place, and it is perhaps the best quickstart guide for those who have stumbled upon it.

The default branch is "dev", which is often ahead of the stable release branches, and can be unstable or broken at times. To view the latest stable release codes, use the main branch. All branches with the name "maintenance/" are for patches and bug fixes for previous releases, such as v1.0.x.

Be sure to also check out our Issue Tracker and our new Discussion Page to get involved, report issues, request features, or to contribute.

Documentation

Ta-da!!! You are already reading the official documentation. Congratulations!

The default link poetic.readthedocs.io links to the latest stable release of `poetic`, which corresponds to the `main` branch on GitHub. To view specific versions of the documentation, select the corrresponding versions at the bottom left corner of the page (in green text).

Only the latest patch of each minor release will be viewable on our documentation page: for example, when v1.0.2 is out, its documentation page will replace that of v.1.0.1. All actively supported versions will have its documentation page active until they have reached EOL.

CLI Help

When using `poetic` on through the command-line interface (CLI), there are two options to get help and package versions. For more detailed decumentation on CLI, please see the "**Command Line Interface (CLI)**" section of the documentation.

To get help on the command-line arguments and flags, use the `--help` flag:

```
python -m poetic --help
```

The outputs of the `--help` flag are:

```
Poetry Predictor Command Line Mode

optional arguments:
-h, --help            show this help message and exit
-g, --GUI             Flag to launch GUI. No input needed.
-s SENTENCE, --Sentence SENTENCE
                        A string to be parsed.
-f FILE, --File FILE  Plain text file to be parsed.
-o OUT, --Out OUT     Path to save results (txt or csv).
--version             show program's version number and exit
```

To get the version of the package:

```
python -m poetic --version
```

Info Class

Within the package, the `Info` class provides the basic information of the package, which is useful for obtaining the version and build status. This has to be used within python.

Version

To get the version:

```
import poetic

poetic.util.Info.version()
```

Build status

To get the build status:

```
import poetic

poetic.util.Info.build_status()
```

All official, stable releases will have a build status of "Stable".

Docstrings

All public modules, classes, methods, and specifically defined magic methods have a docstring. It serves as the lowest level documentation for `poetic`. There are two ways of accessing each docstring:

```
import poetic

# Docstring for the Predictor class
help(poetic.Predictor)

# OR:
print(poetic.Predictor.__doc__)
```

All docstrings are also formated in the "**Full Documentation**" section of documentation.

Type Annotation

All functions and methods are type annotated. Although typing, along with dependencies, limit backwards compatibility of `poetic`, this will help with maintainability for developer and ease of use for end users.

Variables and attributes are not yet type annotated. This may be a feature to be considered in the future, when python 3.5 will no longer be supported.

1.3.2 Command Line Interface (CLI)

One of the most common usages of poetic is to make predictions using the command line. A number of arguments are supported to allow some common operations, such as making predictions from string or text file input, saving results to csv, and launching the GUI. This page details all the options and their usecases.

Basic How-To

To use `poetic` in the command-line mode, it is necessary to run the package using python's `-m` argument to run `__main__.py`. The simplest example without any additional option will be the following, which launches the gui by default:

```
python -m poetic
```

To use specific flags and options for poetic (instead of for python), append them to the usual python call:

```
python -m poetic -s "This is poetic"
```

For help, use the following command:

```
python -m poetic --help
```

Platform and Python Interpreter

For documentation including all examples on this page, all shell commands assume the python interpreter can be invoked by `python`. However, for platforms such as **linux**, specifying `python3` is often necessary to ensure that the correct python interpreter is used.

For those who use virtual environment implementations, such as `conda`, or have multiple versions of python interpreters installed, make sure to activate the virtual environment or invoke the correct python interpreter.

Arguments and Flags

Arguments	Type	Functionality
`-h` or `--help`	Flag	Command-line help
`-g` or `--GUI`	Flag	Launch GUI (default option)
`-s` or `--Sentence`	Argument: A string	Sentence input for prediction
`-f` or `--File`	Argument: Input file path	Plain text file input
`-o` or `--Out`	Argument: Output file path	Ouput results to a csv or txt file
`--version`	Flag	Package version

-h

The `-h` and `--help` flags prints out a simple help guide for command-line arguments. It is also invoked when unrecognized flags or arguments are used.

-g

The `-g` and `--GUI` flags launch the package's gui. this flag is intended to be used in combination with `-s` or `-f` to override their default behabvior, which only processes the given inputs and bypasses the GUI.

Launching the GUI is the default behavior of `python -m poetic`, and in the cases when other arguments are not used, `-g` is unnecessary.

-s

The `-s` and `--Sentence` arguments accept a string to be predicted using the `Predictor` class with default parameters. By default, the results will be printed to `stdout`, and no GUI will be launched.

-f

The `-f` and `--File` arguments accept the path to a plain text file as a string. The contents of the file will be predicted using the `Predictor` with default parameters. Like `-s`, outputs will be directed to `stdout` by default and no GUI will be launched.

-o

The `-o` and `--Out` arguments accept the path as a string for ouputing the prediction results. This option has to used in conjunction with `-f` or `-s`, and it does not affect the GUI and its processing, should the GUI is to be launched by `-g`.

Plain text and CSV are supported and parsed according to the file extension, and no strict file extension check will be enforced. However, to save a csv file, use only `.csv` extension.

Argument Combinations

There are a few common configurations and options that are worth listing here separately.

Default

This will lauch the GUI without any additional options (Note: the `-g` flag is unnecessary here.)

```
python -m poetic
```

Prediction with a String Input

The -s argument tells the program to predict using the supplied string. A string can have multiple sentences, which in turn will be tokenized internally.

Without any further arguments, all ouputs will be printed:

```
python -m poetic -s "I love ice cream."
```

To save the results to a txt or csv file, use the -o argument with the appropriate arguments:

```
# Save to txt (File extention not enforced)
python -m poetic -s "I love ice cream." -o "<PATH>.txt"

# Save to csv (File extention enforced)
python -m poetic -s "I love ice cream." -o "<PATH>.csv"
```

Prediction with a File Input

The -f argument accepts a file path to a plain text file, which will be sentence tokenized and treated as strings.

Operationally, -f is similarly to -s for file IO.

```
python -m poetic -f "<LOAD_PATH>"
python -m poetic -f "<LOAD_PATH>" -o "<SAVE_PATH>"
```

File IO

For file input, only plain text files are supported. File extension is not strictly enforced, and all file types will be treated as plain text files.

For file output, both plain text and csv files are supported, and -o option determines the file type using extension. File extensive for csv file has to be .csv for correct parsing, but it does not matter for text files.

.txt File Format

-o will format all plain text output in the following way (Note: This can be subject to change. For consistent results, use csv files instead).

```
Poetic
Version: 1.0.2
For latest updates: www.github.com/kevin931/Poetic

Diagnostics Report

Model: Lexical Model
Number of Sentences: 2

~~~Five Number Summary~~~
Minimum: 0.6363636363636364
Mean: 0.6515151515151515
Median: 0.6515151515151515
Maximum: 0.6666666666666666
```

(continues on next page)

(continued from previous page)

```
Standard Deviation: 0.015151515151515138

~~~All Scores~~~
Sentence #1: 0.6666666666666666
Sentence #2: 0.6363636363636364
```

.csv File Format

When a file path ending in `.csv` is encountered or the `to_csv()` method of the `Diagnostics` class is explicitly called, the results will be formatted with three columns separated with `Sentence_num`, `Sentence`, and `Score` as keywords. Each sentence and its prediction is in a new row, which is essentially the Tidy Data format for optimal compatibility. The `Sentence_num` column can be treated as the index if desired or necessary.

The raw csv file looks like the following:

```
Sentence_num,Sentence,Score
1,Hi.,0.6666666666666666
2,This is poetic.,0.6363636363636364
```

If formated to a table (like if opened in excel or the like), this will be the result:

Sentence_num	Sentence	Score
2	Hi.	0.6666666666666666
	This is poetic.	0.6363636363636364

Launch GUI

The GUI is a useful part of the program for a few purposes such as demo. Given that the main function of the package is to make predictions, the program defaults to launching the GUI when no other arguments are supplied. On the other hand, when argument `-s` or `-f` is supplied, the program assumes that it is used as for command-line purposed only, and the GUI will not be launch.

To launch the GUI anyways, add the `-g` or `--GUI` flag to the existing command, which will process everything else and then launch the GUI:

```
python -m poetic -s "This is for prediction" -g
```

Unsupported Argument Configurations

Given the functionalities of all the options, certain options will conflict and cause exceptions to be raised. All the incompatibilities are listed below.

-s and -f

`-s` and `-f` cannot be used concurrently on the commandline because each call to the `Predictor` supports only one type of input. Further, since results are printed out to `stdout` by default, it does not make sense to print two separate batches of results. Therefore, `UnsupportedConfigError` from the `poetic.excptions` module will be raised.

The `Predictor` supports sentence tokenization with a single string, and if multiple inputs can be formed into a single string with mutiple sentences, this will be the best approach. There is currently no support for command-line processing of mutiple separate input. For this use case, a `poetic` needs to be imported as a package in python.

-o without -s or -f

The `-o` argument by itself, even if a path is provided, will not have any effect on how the package is run. It has to be used with either `-s` or `-f` to save results of predictions made through the command line. `-o` does not change any global parameter otherwise. If no other options are present, it will be ignored and the default GUI will be launched.

-s, -f, or -o without Arguments

`-s`, `-f`, and `-o` all expected one argument. If only the flags are used without supplying the necessary argument, an error will occur and the program will terminate.

1.3.3 Module Overview

This page documents the public interfaces of `poetic` as a high-level overview. While all classes and methods are accessible using their full path, some classes are designed to be called at the package level and a few others are more or less "internal" even if they are not strictly protected with underscores. For more detailed documentation on the parameters and exact behaviors of each class and method, refer to the **Full Documentation** section.

Import Behavior

Modules

There are five modules in total:

- `exceptions`: An internal module for custom exceptions.
- `gui`: An internal module for the GUI invoked by `-g` flag.
- `predictor`: A module including the `Predictor` class to make predictions.
- `results`: A module for prediction results and diagnostics.
- `util`: A utility module for utility class, functions, and package information.

Package-level Classes

Two classes are directly exposed as package-level classes: `Predictor` and `Diagnostics`. Respectively, they can be accessed directly with `poetic.Predictor` and `poetic.Diagnostics`.

These two classes, especially `Predictor`, are the main interfaces of the package. Therefore, they are directly exposed instead of requiring the access through `poetic.predictor.Predictor` and `poetic.results.Diagnostics`, which are both valid as well.

Modules Overview

Each module will be documentation in its own topic at length. This section is a quick overview of their functionalities.

exceptions

The `exceptions` module contains two classes: `InputLengthError` and `UnsupportedConfigError`. They both inherit directly from python's base `Exception`, and they are raised by various methods and classes in `poetic`.

Although the module and its classes are not preceded by _, it is mainly intended for internal use and its accessibility allows for better and more obvious documentation.

gui

The `gui` module contains the `GUI` class which implements a Tkinter GUI for `poetic`. All the methods of `GUI` are private, and the `gui` module is mainly an internal interface.

To launch the GUI, using the `-g` flag on the command line is recommended instead of importing `poetic` and invoking the `GUI` class.

predictor

The `predictor` module and its package-level `Predictor` class serve as the main interface for the package to predict poetic scores using Keras models. The `Predictor` class is a one-step solution for making predictions as it can load models and dictionaries automatically. To make a prediction, simply follow the following example:

```python
import poetic

pred = poetic.Predictor()
prediction = pred.predict("Is this poetic?")
```

The `Predictions` class is the return type of the `predict()` and `predict_file()` methods of the `Predictor`, and it inherits directly from the `Diagnostics` class. It currently contains all the same methods as its base class and is mainly intended to be invoked internally. To use its functionalities without the `Predictor` class, directly use the `Diagnostics` class instead.

results

The `results` module contains the `Diagnostics` class, which is the base class for the `Predictions` class. Its main functionality includes running diagnotics for predictions, outputing to files, and generating diagnostics report. The `Diagnostics` can be used as a standalone class for any predictions, and more utility and functionalities are planned for the future to add more cuseful diagnostics and utilities.

util

The `util` module provides utility functions and classes for `poetic`, including package metadata, loading and downloading assets, and parsing commandline arguments. It contains two public classes: `Info` and `Initializer`. The `_Arguments` class is intended strictly for internal use to parse command-line arguments for `__main__.py`.

The `Info` class provides basic package information: package vesion and build status. To access the information, use methods `poetic.util.Info.version()` and `poetic.util.Info.build()` respectively.

The `Initializer` class initializes assets for the `Predictor` class, and it contains all class methods without no need of a class instance. The most common usage is to load both the model and dictionary using the `poetic.util.Initializer.initialize()` method which is automatically called by the `Predictor` by default.

1.3.4 Keras Models

`poetic` relies on machine learning models trained with the `tensorflow.keras` framework. This page gives an overview on the default models, the package's infrastructure for loading the model and making predictors, and future model support.

Default Models

The default models are the ones that are "shipped" with package, and currently, `poetic` supports only one model: **the lexical model**. See the *The Lexical Model* section for a detailed explanation of the model's performance and its backgrounds.

Downloading

The trained model and weights combined are very large (~838MB), which is impractical to ship with the `pip` or `conda` package. To address this issue, the models are hosted in their own repository, poetic-models, on Github as releases. This can not only address the package size issue but also allow users to decide whether and when to download the models in case there are bandwidth limitations.

A manual download of the models is not necessary. Whenever the `Initializer` class is called to load the default models, it will automatically check for the local presence of the models and in the case the models are not present, the `poetic.util.Initializer.download_assets()` method is called to fetch the models and place it in the correct directory. There is no need to call the method to download the model upon first installation or update.

By default, the `download_assets()` method will ask for user input with the sample output like the following:

```
The following important assets are missing:

Downloading from: https://github.com/kevin931/poetic-models/releases/download/v0.1-
↪alpha/sent_model.zip
```

(continues on next page)

(continued from previous page)

```
Download size: 835MB.


Would you like to download? [y/n]
```

If the user denies the download with letters other than "Y" or "y", the program may halt because of the lack of a model. If there is a need to bypass the command-line input, set `force_download_assets=True` when initializing the `Predictor` class or `force_download=True` for `download_assets()` and `load_model()` methods of the `Initializer` class. The following demonstrates a few valid ways of force downloading without command-line inputs:

```python
import poetic

# Approach #1
poetic.Predictor(force_download_assets=True)
# Approach #2
poetic.util.download_assets(force_download=True)
# Approach #3
poetic.util.load_model(force_download=True)
```

Loading

In the simplest use case of `poetic` through the `Predictor` class, there is no need to manually load the model as the constructor can automatically load the default model if the class is initialzed with the following:

```python
import poetic

pred = poetic.Predictor(force_download_assets=True)
```

However, there are benefits to loading the keras models directly, as it can expose the full keras interface. The `util` module provides a few functions to conveiently load the default model:

- `poetic.util.Initializer.initialize()`: This class method returns the command-line arguments, the default keras model, and the gensim dictionary.
- `poetic.util.Initializer.load_model()`: This class mothod returns the default keras model.

The advantage of using these two models is that only one function is necessary to load the model without having to know the data directory. However, to access the paths of the model and the weights themselves, use the following snippet:

```python
import pkg_resources
import poetic

data_dir = pkg_resources.resource_filename("poetic", "data/")
weights_path = data_dir + "sent_model.h5"
model_path = data_dir + "sent_model.json"
```

Updating

Currently, model updates are planned to be handled with package updates. At of now, there is no plan to update the existing model, except for changing its name from “sentence” to “lexical” model.

On the roadmap, there is plan to support meterical and combined lexical and metrical models. With the release of such models, the package will be updated with the new model urls or a new update mechanism.

If a qualitative update occurs, re-downloading the models will likely prove to be necessary, and similar procedures will be in place as the initial download of the model.

The Lexical Model

The lexical model is currently the only default model available in `poetic`. It is trained using 18th- and 19th-century works with the lexical contexts through embedding (i.e. the contents of the works themselves in the form of words).

Essentially, the model is a classifier that classifies whether a given input is poetic. More precisely, it can be interpreted as whether an input resembles eighteenth- and nineteenth-century poetry. This definition will be the basis of the concept of the **“poetic score”** throughout the package and the package’s main use case.

A quick note on naming: The model is now called the “sentence model” stored with “sent_model.h5” and “sent_model.json” in v.1.0 because all training sets and inputs are sentence tokenized. Since all other future models will also take the same data format in sentence even though they are not necessarily lexical based, the model will be renamed to the lexical model to better reflect how it was trained and what it represents.

Training and Validation Data

All training and validation data come from Project Gutenberg. The datasets consist of solely 18th- and 19th-century works separated into two categories: poetry and prose (non-poetry). The rationale of this time period is that works during these two centuries are vastly avaible in the public domain and digitized. Further, it is also a time when formal poetry was still the norm instead of the rapid rise of free verse. Thus, this dataset will allow the lexical model to train on the most distinguishing features of poetry.

Given the amount of data available on Project Gutenberg, the training and validation sets consist of a random sample of the aforementioned works. Although a different sample or the entire corpus may result in a different model, the amount of data within the sample used can allow reasonable assumption of representativeness of the sample.

Model Architecture

The overall architecture of the lexical model is a *bidirectional long-short-term memmoey neural network* (LSTM) trained using the keras API of tensorflow. LSTM is known to work well with lexical data although its performance has now been surpassed by large language models, such as Google’s BERT.

Below is a high-level overview of the layers used in training the model (in sequential order):

Layer	Output Shape
Input	(None, 456)
Embedding	(None, 456, 128)
LSTM	(None, 456, 128)
LSTM Forward	(None, 128)
LSTM Backward	(None, 128)
Concatenate	(None, 256)
Dropout	(None, 256)
Dense	(None, 64)
Dropout	(None, 64)
Dense/Output	(None, 1)

Model Performance

The confusion matrix:

	Prose	Poetry
Prose	129168	42082
Poetry	38230	125316

Classification Diagnostics:

- Accuracy: 0.7601
- Precision: 0.7662
- Sensitivity: 0.7486

Custom Models

There is infrastucture in place for the `Predictor` class to utilize custom models. However, v1.0.x **does not** support for custom models because the preprocessing pipeline custom models will likely require a different input shape, which is not supported by the preprocessing pipeline.

Future Updates

Custom keras models with the same input dimension and an embedding layer will be fully supported starting v.1.1.0, which is already in development on the `dev` branch of `poetic`. This will also be accompanied by allowing custom `gensim` dictionaries, which are often necessary for different models. No other types of models' support is planned at this stage of development.

1.3.5 Gensim Dictionary

The `Predictor` class uses a gensim dictionary to convert words (also called "tokens") into IDs for the keras model's embedding layer. Each word has a unique numeric ID that allows the model to create deep embedding to capture the lexical context.

This section of the documentation details the use of gensim dictionaries in `poetic`, the default, and custom dictionary options along with examples.

Default Dictionary

The `poetic` package ships with a default dictionary in both `pip` and `conda` distributions. Given its relatively manageable size, it is included as package data with each release, and there is no need to download the dictionary separately, unlike the default keras model.

The dictionary is constructed using the entire corpus of 18th- and 19th-century literary works on Project Gutenberg, as opposed to the randomly sample used for traning. It should cover a wide range of lexicons encountered in both literature of the time and everyday usage although newest words and slangs may be lacking. In the `Predictor` class, all non-existant words are assigned with the value 0 for consistency.

Format

The dictionary is saved with the `save_as_text()` method of the `gensim.corpora.dictionary.Dictionary` class. The file has the following format, which is also documented here:

```
76242402
440 !       4922258
36666       #       12419
17501       $       23781
142 '       2078602
174 ''      5630856
```

The first line is the number of entries, and following lines each has three tokens separated by tabs: ID, word, document frequency.

Only dictionaries of this format are supported at this time because a tab-separated text file will allow the useage without `gensim` dependency for best future support with custom classes. If a custom dictionary is saved with the `save()` method, it needs to be loaded manually as documented below.

Loading

The `Predictor` class loads the default dictionary by default if the `dict` parameter is not specified as in the example below:

```
import poetic

pred = poetic.Predictor()
```

One **important** thing to note is that the `dict` and `model` parameters are independent of each other: loading a custom model will not require a custom dictionary and vice versa. Therefore, if a custom model is used, it is **recommended**, though not required by the package, to use a custom dictionary.

Under the hood, the `Predictor` calls the `Initializer` class to load the dictionary, which is also a valid way of loading the dictionary independently:

```
import poetic

dictionary = poetic.util.Initializer.load_dict()
# If a Predictor is to be used:
pred = poetic.Predictor(dict=dictionary)
```

The above two snippets are functionally equivalent as shown, but the latter approach allows for the use of a dictionary independently, including accessing its own methods attributes, etc.

The dictionary is stored in the data directory of the package. To access the path of the dictionary, use this snippet:

```
import pkg_resources
import poetic

data_dir = pkg_resources.resource_filename("poetic", "data/")
dictionary_path = data_dir + "word_dictionary_complete.txt"
```

Updating

The dictionary will be updated with the package itself. However, on the current roadmap, there is no update planned for the gensim dictionary itself. Should there be a change, the process will be automated without manual downloading, renaming, etc.

Custom Dictionary

The current version, v1.0.x, does not have have full support for custom dictionary although the `Predictor` class does allow a custom dictionary during initialization. Since there is not yet support for custom models, using a custom dictionary will be practically meaningless with one exception: the use of a custom model with the same input shape with a custom model. See the "Keras Models" section for a more detailed explanation of the state of custom models.

To use a custom model with the `Predictor`, the following snippet will work:

```
import poetic
import gensim

dictionary = gensim.corpora.Dictionary.load_from_text(fname="<PATH>")
pred = poetic.Predictor(dict=dictionary)
```

To use a dictionary saved in the format saved with the `save()` method:

```
import poetic
import gensim

dictionary = gensim.corpora.Dictionary.load(fname="<PATH>")
pred = poetic.Predictor(dict=dictionary)
```

1.3.6 Making Predictions

The most important functionality of `poetic` is the interface to make predictions using pretrained keras models. The `Predictor` class provides a simple-to-use, one-stop solution to predict how poetic any given input is and how much it resembles 18th- and 19th-century poetry if the default models are used.

This page documents the usage of the `Predictor` class along with common topics and examples.

Initialization

The `Predictor` class requires instantiation to work properly since there are no utility functions or class methods in this class. To create an instance, simply use `poetic.Predictor()` becasue it is a package-level class:

```
import poetic

pred = poetic.Predictor()
```

In this example, all default parameters are used, and the default lexical model and dictionary will be loaded. There is **not yet** official support for custom model and dictionary. However, `Predictor` does accept previously loaded assets using either keras and gensim's API or poetic's `Initializer` class:

```
import poetic

model = poetic.util.Initializer.load_model()
dictionary = poetic.util.Initializer.load_dict()
pred = poetic.Predictor(model=model, dict=dictionary)
```

If the default models have not been downloaded from its GitHub repo, there is an option to override the user input prompt:

```
import poetic

pred = poetic.Predictor(force_download_assets=True)
```

Once a `Predictor` object is instantiated, it can be reused to make multiple predictions and to preprocess different inputs. No method will have meaningful side effects, although the `tokenize()` method modifies the internal `_sentences`, which temporarily stores the tokenized input and will be overridden with each subsequent operation. Therefore, a `Predictor` instance is fully reuable and safe.

Making Predictions

The `Predictor` allows users to make poetic predictons using either the `predict()` or the `predict_file()` method. All preprocessing steps are automatically handled without any need to manually clean inputs.

Prediction with Strings

To predict a string, use the `predict()` method of the `Predictor` instance. The input string can consist of multiple sentences, which are then tokenized by preprocessor. The **longest** supported sentence (after sentence tokenization) is **456 tokens**, including words and punctuations.

As an example of string prediction:

```
import poetic

pred = poetic.Predictor()
result = pred.predict("Hi. I am poetic. Are you?")
```

The `predict()` method will return a `Predictions` object, which in turn supports post- processing, such as running diagnostics and saving results to file.

Prediction with Text Files

Plain text files are also supported. To load and predict a file, use the `predict_file()` method, and all preprocessing and the object returned will function exactly the same as the `predict()` method.

Under the hood, it loads the file into a single string, and it then calls the `predict()` method. For large files that can potentially exceed system RAM, it will be better to manually load the files and make predictions.

```
import poetic

pred = poetic.Predictor()
result = pred.predict_file("<PATH>")
```

Preprocessing

The preprocessing toolchain consists of the following steps: tokenization, word ID conversion, lower-case conversion, and padding. The latter two steps are primarily for keras models while tokenization can apply to other NLP workflows. This sections documents some of the details and their supported usage.

One-step Preprocessing

To preprocess the input for the default model of `poetic`:

```
import poetic

pred = poetic.Predictor()
model_input = pred.preprocess("This is poetic. Isn't it?")
```

The `preprocess()` method returns a 2-d numpy array of tokenized word IDs that can be directly predicted using the keras model's `predict()` method. However, the predictor's `predict()` method does not support a preprocessed input: only raw input in strings are supported.

Tokenization

Tokenization is the process of separating a string input into tokens, which are units of texts that the algorithms support. The `Predictor` uses NLTK's `sent_tokenize()` and `word_tokenize()` functions respectively to perform two-step tokenization: first, the string, regardless of length, is tokenized into complete sentences; then, each sentence is tokenized into words and punctuations.

The `tokenize()` methods can be used as a stand-alone function although it is not a proper classmethod for compatibility with the `Predictions` class.

As an example:

```
import poetic

pred = poetic.Predictor()
model_input = pred.tokenize("This is poetic. Isn't it?")
```

The output will be a 2-d nested list in the following format:

```
[['This', 'is', 'poetic', '.'], ['Is', "n't", 'it', '?']]
```

Padding

Padding is part of the `preprocess()` method, and it cannot be called seprately. It pads each tokenized input in accordance with the input shape of the default lexical model used, which is 456. There is not yet support to adjust the padding length in this release, and this is the reason why custom model support is very limited.

Under the hood, the `tf.keras.preprocessing.sequence.pad_sequences()` method is called, and the default pre-padding is used. Given that the default lexical model uses an LSTM architechture, the pre-padding strategy makes sense. Currently, there is no support for other types of padding.

Word IDs

All tokens (mostly words, contractions, and punctuations after tokenized) are converted into word IDs, which are all postive `int`. By default, the gensim dictionary shipped by the package is used. However, if a custom dictionary is supplied at initialization of the `Predictor`, it will likely be incomptabile with the default model because models are specifically trained with one set of word IDs. Therefore, it is **not recommended** to use a custom dictionary.

1.3.7 Prediction Results

As the final part of `poetic`'s main workflow, the post-processing of prediction results consists of diagnostics, summary, and file output. The package's `Diagnostics` class provides all these functionalities with a few simple methods, which are documented below.

Predictions Class

Both `predict()` and `predict_file()` methods of the `Predictor` class returns an instance of the `Predictions` class:

```
import poetic

pred = poetic.Predictor()
score = pred.predict("Is this poetic?")
```

In the example above, the `score` object will be a `Predictions` object, which can then call methods to run diagnostics and save results.

Inheritance

The `Predictions` class inherits from the `Diagnostics` class, and all methods are also inherited with the only difference in the constructor. The advantage of using an inherited class instead of using the `Diagnostics` class directly is that the preprocessing of keras predictions can occur separately. Thus, the `Predictions` class serves as an internal interface to distinguish from manually instantiated instances of the `Diagnostics` class.

To use the toolchain and methods separately, use the `Diagnostics` class instead. All methods of the `Predictons` will be documented with the `Diagnostics` class unless they are overridden.

Diagnostics Class

As the base class for `Predictions`, the `Diagnostics` class provides a more genralized framework for working with any prediction results. In the future, more abstractions may be added to allow for more versatility to use independently.

A typical workflow will involve making predictions, running diagnostics, and saving the results to a file:

```
import poetic

pred = poetic.Predictor()
score = pred.predict("Is this poetic?")
score.run_diagnostics()

print(score.generate_report())
score.to_file(path="<PATH>")
```

Instantiation

To use the `Diagnostics` class, only the `predictions` argument is required as a list of floats:

```
import poetic

results = poetic.Diagnostics(predictions = [1, 0, 0.5])

# OR: with sentences
sentences = ["Hi.", "I am poetic", "How about you?"]
results_sentences = poetic.Diagnostics(predictions = [1, 0, 0.5], sentences=sentences)
```

The `sentences` argument is optional. If used, it will store the corresponding sentences of the predictions as a class attribute; otherwise, it will be `None`, and all other methods are largely unaffected, except the contents of the outputs.

Diagnostic Statistics

As of now, the `Diagnostics` class supports five-number summary for predictions. As part of the workflow, it is automatically called by the `run_diagnostics()` method, and the results are stored in the `diagnostics` attribute of the object. As an example:

```python
import poetic

results = poetic.Diagnostics(predictions = [1, 0, 0.5])
results.run_diagnostics()
# Get the diagnostic results
print(results.diagnostics)
```

The `diagnostics` attribute is a dictionary with three keywords: “Sentence_count”, “Five_num”, and “Predictions”. The corresponding values are the following:

- “Sentence_count”: An `int` of the length of entries.
- “Five_num”: Five number summary stored with a dictionary.
- “Predictions”: A `list` of floats from the `predictions` attribute.

To obtain the five number summary separately using the classmethod `five_number()`, which is essentially a utility function that can be use for any array-like objects compatible with `numpy`:

```python
import poetic

results = poetic.Diagnostics(predictions = [1, 0, 0.5])
poetic.Diagnostic.five_number(results.predictions)

# As a stand-alone method:
poetic.Diagnostic.five_number([1, 0, 0.5])
```

Diagnostic Report

A diagnostic report is a string (or plain text) summary of the object with diagnostic statistics. To obtain the diagnostic report, the `run_diagnostics()` method has to be called previously on the object. Otherwise, a type error will be raised because the “diagnostics” attribute will be `None`.

An example usgae of the method is this:

```python
import poetic

results = poetic.Diagnostics(predictions = [1, 0, 0.5])
results.run_diagnostics()
print(results.generate_diagnostics())
```

The contents of the report will be identical to the text file output as documented below.

File Output

The results and diagnostics can be saved to either `.txt` or `.csv` file. The former writes the diagnostics report to a plain text while the latter saves the actual values separated by comma. The usage is essentially identical to using the `-o` option on the command line.

Plain Text File

To save results to a text file:

```python
import poetic

results = poetic.Diagnostics(predictions = [2/3, 7/11])
results.run_diagnostics()
results.to_file("<PATH>")
```

The output format is the following:

```
Poetic
Version: 1.0.2
For latest updates: www.github.com/kevin931/Poetic

Diagnostics Report

Model: Lexical Model
Number of Sentences: 2

~~~Five Number Summary~~~
Minimum: 0.6363636363636364
Mean: 0.6515151515151515
Median: 0.6515151515151515
Maximum: 0.6666666666666666
Standard Deviation: 0.015151515151515138

~~~All Scores~~~
Sentence #1: 0.6666666666666666
Sentence #2: 0.6363636363636364
```

The `to_file()` does not enforce file extension for text files, except for `.csv` ending. In the latter case, it will automatically call the `to_csv()` method. The text file output is more for a quick summary than a way to store data, and the format can potentially change with updates. If an object needs to be restored or data will be further processed, use the csv format instead.

.csv File Format

When a file path ending in `.csv` is encountered or the `to_csv()` method of the `Diagnostics` class is explicitly called, the results will be formatted with three columns separated with `Sentence_num`, `Sentence`, and `Score` as keywords in the first row. Each sentence and its prediction is in a new row, which follows the Tidy Data format for optimal compatibility. The `Sentence_num` column can be treated as the index.

To save to a csv file as an example:

```python
import poetic

```

(continues on next page)

(continued from previous page)

```
results = poetic.Diagnostics(predictions = [2/3, 7/11])
results.run_diagnostics()
results.to_csv("<PATH>")
```

Or, let `to_file()` handle it automatically:

```
import poetic

results = poetic.Diagnostics(predictions = [2/3, 7/11])
results.run_diagnostics()
results.to_file("<PATH>.csv")
```

The raw csv file looks like the following:

```
Sentence_num,Sentence,Score
1,Hi.,0.6666666666666666
2,This is poetic.,0.6363636363636364
```

If formated to a table (like if opened in excel or the like), this will be the result:

Sentence_num	Sentence	Score
1	Hi.	0.6666666666666666
2	This is poetic.	0.6363636363636364

Custom Build-in (Magic) Methods

String Representation

The `str()` method will return a short summary of the object. It will truncate the output to 14 characters after the description:

```
'Diagnostics object for the following predictions: [0.66666666666...'
```

The `repr()` method will return a dictionary cast into a string with all the predictions, sentences, and the `diagnostics` attributes. It does not truncate any results. This will be more appropraite for a full representation of the object. It will have the following format:

```
"{'Predictions': [0.6666666666666666, 0.6363636363636364], 'Sentences': None,
↪'Diagnostics': None}"
```

len()

The `len()` method returns the length of the `predictions` attribute of the object, which is the number of entries in the predictions list. Since the length of `predictions` and `sentences` are intended to match, the returned length logically represents the length of the object.

1.3.8 Further Documentation and Links

There are a few more resources beyond this website, including those included with the package and those of the dependencies. For links and references, see the appropriate sections below.

Package Docstring

All public modules, classes, functions, and methods have docstrings. To view the rendered docstrings as documentation, see the "**Full Documentation**" section. To see how to access such the resources in python, see the "**Package Information and Resources**" section of the documentation.

Installation Documentation

For installation, please see the "**Installation Guide**" page for details. For documentation on pip and conda themselves, links are provided below.

pip

`pip` documentation: https://pip.pypa.io/en/stable/.

conda

General `conda` documentation: https://docs.conda.io/projects/conda/en/latest/index.html

Managing `conda` environments: https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html

`conda` channels: https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/channels.html

---

Dependency Documentation

Keras Models

For furthur documentation on keras models, refer to tensorflow's offical documentation through https://www.tensorflow.org/api_docs/python/tf/keras.

Gensim Dictionaries

For furthur documentation on gensim dictionaries, refer to gensim's offical documentation through https://radimrehurek.com/gensim/corpora/dictionary.html.

Numpy Arrays

Numpy arrays are used in some under-the-hood processing. For documentation on numpy arrays in general, see https://numpy.org/devdocs/reference/generated/numpy.array.html.

For ndarray, which is the return type of the `preprocess()` method of the `Predictor` class, see https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html.

NLTK

The `Predictor` class uses NLTK's tokenization algorithms through its `sent_tokenize()` and `word_tokenize()`. More detailed documentation can be found here: https://www.nltk.org/api/nltk.tokenize.html

1.4 Full Documentation

This section provides a low-level, detailed documentation of all public modules and methods of the package, which are generated from the docstring documentation. This section is ideal for looking at the technicality of the package, such as parameters, methods, etc. There are some brief examples available, but if you would like a high-level explanation of the package, see the "Examples and Package Details" section.

1.4.1 Package Overview

Poetic Package.

This is the Poetic package, which provides functionalities for predicting how poetic language is using a Keras model trained on public-domain works. For further documentation and latest releases, please visit https://github.com/kevin931/poetic.

Modules:

- predictor
- results
- util

Package-level Classes:

- Predictor
- Diagnostics

1.4.2 Docstring Documentation

poetic.predictor module

Making poetic predictions using models.

The predictor module provides interfaces for poetry predictions with the Predictor class. To make prediction, an instance of the Predictor is needed with the necessary model and gensim dictionary is needed.

Examples

The most common use case for the predictor module is with its default settings. To make a prediction with string, below is an example:

```
import poetic

pred = poetic.Predictor()
result = pred.predict("This is an example.")
```

To make a prediction with a text file, use the following codes:

```
import poetic

pred = poetic.Predictor()
result = pred.predict_file("<PATH>")
```

class `poetic.predictor.`**`Predictor`**(*model: Optional[tensorflow.keras.Model] = None, dict: Optional[gensim.corpora.dictionary.Dictionary] = None, force_download_assets: Optional[bool] = False*)

Bases: `object`

The `Predictor()` class processes and predicts inputs for poetic scores. It can be used as the single interface of the package with other modules built as helpers.

Parameters

- **`model`** (*`tensorflow.keras.Model, optional`*) – A pre-trained keras model. The default model will be loaded if no model is supplied. If a custom model is supplied, a custom gensim dictionary is recommended for it to work correctly although not strictly enforced.
- **`dict`** (*`gensim.corpora.dictionary.Dictionary, optional`*) – Gensim dictionary for word IDs. If nothing is supplied, the default dictionary will be loaded. The default dictionary will be required for the the default model to work correctly although it is not strictly enforced.
- **`force_download_assets`** (*`bool, optional`*) – Wheher to download assets (the default models) without asking/user input.

`model`

The pre-trained keras model.

Type tensorflow.keras.Model

`dict`

Gensim dictionary for word IDs.

Type gensim.corpora.dictionary.Dictionary

`force_download_assets`

Wheher to download assets without asking.

Type bool

`predict`(*input: str*) → *poetic.predictor.Predictions*

Predict poetic score from string.

Parameters **`input`** (*`str`*) – Text content to be predicted.

Returns A Predictions object with predicted scores of the given input.

Return type *Predictions*

Raises *`poetic.exceptions.InputLengthError`* – Error for processing input length of zero.

`predict_file`(*path: str*) → *poetic.predictor.Predictions*
Predict poetic score from file.

This method essentially loads the text file into a string of text and then calls the predict method.

Parameters **`path`** (*`str`*) – The path to the text file.

Returns A Predictions object with predicted scores of the given input.

Return type *Predictions*

Raises *`poetic.exceptions.InputLengthError`* – Error for processing empty file, resulting in input length of zero.

`preprocess`(*input: str*) → numpy.ndarray
Preprocess inputs: tokenize, to lower, and padding.

Parameters **`input`** (*`str`*) – Text either in a single string or a list of strings.

Returns A 2-d numpy array of processed inputs.

Return type numpy.ndarray

Raises *`poetic.exceptions.InputLengthError`* – Error for processing input length of zero.

`tokenize`(*input: str*) → List[List[str]]
Tokenize text input into sentences and then words.

Parameters **`input`** (*`str`*) – A string or list of strings of text.

Returns A 2-d list of tokenized words.

Return type list(str)

`word_id`(*input: List[List[str]]*) → List[List[int]]
Convert tokenized words to word IDs using a gensim dictionary.

Parameters **`input`** (*`list`*) – A 2-d list of tokenized words.

Returns A 2-d list of word ids.

Return type list

poetic.results module

Module for processing prediction results.

The results module processes the outputs of prediction results from the Predictor of the predictor module. The functionalities provided include statistical summaries, io, and diagnostic reports.

Examples

The results module's Diagnostics module can be used as part of the prediction workflow. To use its methods with the Predictor class:

```python
import poetic

pred = poetic.Predictor()
result = pred.predict("This is an example.")
result.run_diagnostics() # One-stop function
result.to_file("<PATH>")
```

To use without the Predictor class:

```python
import poetic

pred = [0.1, 0.2, 0.3]
result = Diagnostics(predictions=pred)
five_number_summary = result.five_number()
```

All public methods can be used without the run_diagnostics() method, but they depend on the diagnostic attribute, which the run_diagnostic() method generates.

class `poetic.results.`**`Diagnostics`**(*predictions: List[float]*, *sentences: Optional[List[str]] = None*)

Bases: `object`

Class for storing and processing prediction results.

`Diagnostics` is the default base class of `Predictions`, which is generated by the `Predictor` class. It can also be used as a standalone class for processing any numeric results and strings stored in lists.

Parameters

- **predictions** (*list*) – Predictions of poetic scores.
- **sentences** (*list, optional*) – Sentences associated with the predictions.

predictions

Predictions of poetic scores.

Type list

sentences

Sentences associated with the predictions.

Type list

diagnostics

A dictionary of diagnostics statistics, including sentence count, five number summary, and the predictions themselves.

Type dict

__repr__() → str

String representation for `repr()`.

This string representation returns a dictionary cast into a string with three attributes of this class: predictions, sentences, and diagnostics.

Returns String representation of the object.

Return type str

`__str__``()` → str

String representation for `str()`.

This string representation returns a summary of of the object. It will truncate results if there are more than 15 prediction entries. To get a string representation of all the results in a dictionary cast into string, use repr() method.

Returns String representation of the object.

Return type str

`classmethod five_number``(`*input: Union[numpy.ndarray, List[float]]*`)` → Dict[str, float]

Five number summary.

This methods generates five number summary of a given input. The five number summary includes minimum, mean, median, standard deviation, and maximum. This is a class method.

Parameters **`input`** (*`numpy.ndarrau, list`*) – An array like object.

Returns A dictionary of five number results.

Return type dict(str, float)

`generate_report``()` → str

Generates the diagnostics report in string.

This methods generates a diagnostics report as a string, with Poetic package information, five number summary, and all sentences and their poetic sores.

Returns A string with diagnostic report.

Return type str

`run_diagnostics``()` → None

Run the diagnostics of the predictions.

This methods generate diagnostics of the predictions, which include sentence count, five number summary, and the sentences themselves.

`to_csv``(`*path: str*`)` → None

Saves predictions and sentences to a csv file.

This methods saves the results to a csv file. For a plain text diagnostics, please use the `to_file()` method.

Parameters **`path`** (*`str`*) – An string representing the file path.

`to_file``(`*path: str*`)` → None

Saves diagnostics and predictions to a file.

This methods saves the results to a csv or generates a diagnostics report along with the predictions. The supplied file path's file extension is used to determine which file to save. If a csv is explicitly desired, `to_csv()` method can be used. For all file extensions other than csv, a plain text report will be generated.

Parameters **`path`** (*`str`*) – An string representing the file path.

poetic.util module

Module for package utility.

This module includes the necessary functionalities to load and download assets for other modules in the package and provide basic information for the current build and version as needed.

Examples

To get the build and version information of the package:

```
import poetic

poetic.util.Info.version()
poetic.util.Info.build_status()
```

Under normal circumstances, the methods in the Initializer class is not needed as part of the prediction workflow. One of the most common usage of a first-time user is to download the assets:

```
import poetic

poetic.util.Initializer.download_assets()
```

The tensorflow model and gensim models can also be loaded and returned if they theselves are useful (the Predictor class loads the model automatically):

```
import poetic

poetic.util.Initializer.load_dict()
poetic.util.Initializer.load_model()
```

Both download_assets() and load_model() methods have the force_download parameter which controls whether to download the models without taking commandline inputs when the model is missing. It is default to False so that it does not take up bandwidth unintendedly, but it can also be set to True in cases necessary.

class `poetic.util.`**Info**

Bases: `object`

Info class provides the basic information of the package.

static build_status`()` → str

Get the build status of the current version.

Returns The build status of the current version.

Return type str

static version`()` → str

A single method to return the version of the package.

Returns The current version of the package.

Return type str

class `poetic.util.`**Initializer**

Bases: `object`

Initializes core components of the package.

The Initializer is core part of Poetic that loads and downloads models and other necessary assets. It also facilitates the command line mode by interacting with the _Arguments class.

classmethod check_assets() → Dict[str, bool]

Method to check whether assets requirements are met.

This method checks both the model and its weights in the corresponding directory. It reports back their existence as part of the package requirement.

Returns the status of the assets as a dictionary.

Return type dict

classmethod download_assets(*assets_status: Optional[Dict[str, bool]] = None*, *force_download: Optional[bool] = False*, ***, *_test: Optional[bool] = False*, *_test_input: Optional[str] = None*) → None

Method to download models.

This method downloads models from the poetic-models github repository. Under usual circumstances, other functions will download the models automatically if needed. If all the models already exist, this function will not download them again for package efficiency and bandwidth saving.

If you would like to redownload the assets anyway, a manual download from https://github.com/kevin931/poetic-models/releases is necessary.

Parameters

- **assets_status** (*dict, optional*) – A dictionary generated by `check_assets()` method. It has keys "all_exist", "model", and "weights" with all values being boolean.
- **force_download** (*bool, optional*) – A boolean indicating whether assets should be downloaded regardless of their existence and user inputs.

classmethod initialize(***, *_test: Optional[bool] = False*, *_test_args: Optional[Union[List[str], str]] = None*)

Initializes the package.

This methods checks for any command line arguments, and then loads both the gensim dictionary and the Keras model with its weights.

Returns

Tuple with the following elements

dict: A dictionary of commandline arguments.

tensorflow.keras.Model: A pre-trained Keras model with its weights loaded.

gensim.corpora.dictionary.Dictionary: A gensim dictionary.

Return type tuple

classmethod load_dict() → gensim.corpora.dictionary.Dictionary

Loads gensim dictionary.

Returns A gensim dictionary.

Return type gensim.corpora.dictionary.Dictionary

classmethod load_model(*force_download: Optional[bool] = False*, ***, *_test: Optional[bool] = False*) → tensorflow.python.keras.engine.training.Model

Load Keras models.

This method uses Keras interface to load the previously trained models, which are necessary for the Predictor and the GUI. If the model does not exist, the download_assets() method is automatically called for the option to obtain the necessary assets.

Parameters **`force_download`** (*`bool, optional`*) – A boolean value on whether to download the models without asking if the models do not exist.

Returns Pretrained Keras model

Return type tensorflow.keras.Model

Internal Interfaces

poetic.exceptions Module

Module for custom exceptions.

The exceptions module includes custom classes for specific errors in the poetic package. This is not part of the public interface.

`exception` `poetic.exceptions.`**`InputLengthError`**(*message: Optional[str] = None*)

Bases: `Exception`

Raises Input Length Error.

This exception is used in the Predictor class for the input length of 0.

Parameters **`message`** (*`str`*) – The error message to display.

`exception` `poetic.exceptions.`**`UnsupportedConfigError`**(*message: Optional[str] = None*)

Bases: `Exception`

Raises Unsupported Configuration Error.

This exception is used in the _Arguments class for checking unsupported commandline flags.

Parameters **`message`** (*`str`*) – The error message to display.

poetic.gui Module

Module for Poetry Predictor GUI.

This module includes the class and methods for poetic's desktop Tkinter GUI. As an internal interface, this is not intended or recommended for being used as an import. To run the GUI, follow the given examples or refer to the documentation.

Examples

Simply run the GUI:

```
python -m poetic
```

Make prediction and launch GUI by adding the `-g` or `--GUI` flag:

```
python -m poetic -s "This is poetic." -o "<PATH>" -g
```

`class` `poetic.gui.`**`GUI`**(*predictor: Optional[poetic.Predictor.predictor] = None*, **, _test: Optional[bool] = False*)

Bases: `object`

Launches the GUI, which is equivalent to launching it through the command line.

predictor

A `Predictor` object created through the predictor module. If not supplying a `Predictor` object, the GUI will have no predicting functionality.

Type *Predictor*, optional

poetic.util._Arguments Class

class `poetic.util.`**`_Arguments`**

Bases: `object`

parse(*args: Optional[List[str]] = None*) → Dict[str, Optional[str]]

version() → str

poetic.predictor.Predictions Class

class `poetic.predictor.`**`Predictions`**(*results: List[List[float]], sentences: Optional[List[str]]*)

Bases: *`poetic.results.Diagnostics`*

Class for prediction results from Predictor class.

This class inherets from `Diagnostics` class of the `results` module, and it is intended for being internally called by the Predictor class. To directly access the Diagnostics class's functionality, use it instead.

Parameters

- **results** (*list(list(float))*) – The prediction results predicted with Keras models.
- **sentences** (*list(str), optional*) – A list of strings to represent tokenized sentences predicted by the `Predictor` class.

predictions

Predictions of poetic scores.

Type list

sentences

Sentences associated with the predictions.

Type list

diagnostics

A dictionary of diagnostics statistics, including sentence count, five number summary, and the predictions themselves.

Type dict

1.5 Changelog and Versions

1.5.1 Older Versions

This page contains a running list of all previous release versions, except for the latest supported versions or work in progress changes.

v.1.0.2

- Fixed an issue causing conda build to fail.
- Updated module documentation toctree

v.1.0.1

- Now on **conda** as poetic-py
- Updated documentation for roadmap
- Fixed type-hinting errors
- Updated docstring
- Automated package build process

v.1.0.0

- **FIRST MAJOR RELEASE**
- Now on PyPi
- Support for command-line mode.
- Support for processing text file.
- Added docstring documentation.
- Added official documentation
- Revemped Github repo without LFS.
- Data now hosted on poetic-models
- Added the `exceptions` internal module
- Added support for directly running the package using "python -m poetic"
- Tons of internal optimization

v.0.1.2

- Fixed a bug displaying score without entering anything
- Optimized error handling in Predictor class
- Further optimized the directory tree

v.0.1.1

- Optimized directory structure
- Revamped README with detailed guides
- Added launcher script
- Easier installation

v.0.1.0

- Added GUI for better user experience
- Executable for Windows without need for installation
- Updated code structure
- Updated Project Structure

v.0.0.1

- Project initialization on GitHub.
- Improved interface.
- Updated model.
- LFS support for GitHub.

1.5.2 Deprecations

As documented in **Development and Supoport** section of the documentation, all deprecations will be clearly noted and warned. This page aggregates all current deprecations that are not yet removed from the codebase, and it also provides more details than the release notes. Support for these features will be removed in the next major release.

This list is organized by releases, which represent the time each feature isdeprecated (not the time they are removed).

---

v1.0.0

Launch.py

`Launch.py` is no longer supported starting in this version, and it is not shipped with official releases on `pip` or `conda`. However, it is still tracked by git in the repository, and minimal maintenance updates will occur to ensure that it stays functional. No new features will be supported.

All the functionalities are replaced by `__main__.py`. To launch the program from the command line, use `python -m poetic` intead. All the command-line flags and arguments are still valid.

1.5.3 Versions in Development

This page documents features changed during development for new versions planned. This is highly subject to change, and it is not guaranteed that features appeared here will appear in a stable release. Once a version is release, the changelog will be moved to the **Latest Versions** section under the correct release.

For developers, please see the **Contribution Guideline** section instructions on changes.

Planned Updates

Please see the documentation for the `dev` branch for the latest updates and changelog. This branch is currently up-to-date with the latest release on the `main`.

1.5.4 Latest Versions

v1.0.3

- Added "**Tutorials and Examples**" section to documentation
- Fixed file output spacing issues
- Fixed conda channel priority documentation for python 3.8
- Fixed documentation code highlighting
- Fixed type annotation for the `Predictor` and `Predictions` class
- Fixed docstrings for multiple returns with tuples
- Fixed conda platform conversion commands in setup.py
- Added in-line code highlighting in documentation
- Added import statements to complete examples
- Changed CLI help section wording

1.6 Development and Support

1.6.1 Project Roadmap and Milestones

Major milestones:

- Support for poetic meter: both parsing and predicting
- Support for custom models other than the default model
- Support for more default models

Tentative milestones (subject to change):

- Backwards compatibility with Python 3.5 (No other versions support planned)
- An improved GUI with better front and back end
- Package-level optimization

1.6.2 Future Support and Maintenance

Patches (e.g. x.x.1):

- Each patch consists mainly of bugs fixes, error corrections, and internal improvements.
- No new feature will be introduced.

Minor Updates (e.g. x.1.x):

- Each minor update will be supported and maintained for two minor release cycles after their relase. For example, v.1.0.x will be supported until the release of v.1.3.0.
- The last version before each major release (except versions before v.1.0.0) will potentially have longer life cycles depending on the number of breaking changes.

Major Updates (e.g. 2.0.0):

- There can be breaking changes without prior warnings.
- Currently, there is not yet roadmap for the next major update on the horizon.

Deprecations:

- All deprecations will have clear warnings and be included in release notes.
- Deprecations will be removed in the next major release.
- Backwards compatibility will be maintained before their removed. No breaking changes will occur in minor updates.

1.7 Contribution Guideline

1.7.1 Bug Reporting/Fixes

I get it! I (or the team in the future) make mistakes. If something does not work as intended or if there are any other bugs, please do let us know!

1. Check poetic's Github issue tracker
2. If there is already the same issue, feel free to comment or help out!
3. If not, please open an issue with as much details as possible.
4. Information to include (Do use a template if that works for you):
    - Platform
    - OS
    - Package/Dependency versions (exporting the environment through pip or conda will be best)
    - Codes to reproduce the error

1.7.2 Pull Requests

Any and all help and contrition are welcomed. For consistency and stability, there are a few quick, easy guidelines to follow if you would like to contribute:

- **What to contribute:**
  - Refer to our Issue Tracker and dev branch for the latest updates and developments.
  - If there is something not already covered, open an issue; otherwise, feel free to comment.
  - Take a look at the roadmap.
- **For all changes and source codes:**
  - All functions and methods must be type annotated with `typing`.
  - Unittests must be written to ensure new codes work. Existing tests must pass or modified to pass if relevant. Pytest is used for this package.
  - All public methods, functions, and modules must have a Google-styled docstring. If an interface is modified, docstring must relfect the change accordingly.
  - Comments are fine and sometimes necesary when codes are obscure, but obvious comments should be avoided. Use of expressive variable names are encouraged.
  - Commits must have a simple, meaningful title (e.g. 'Added support for csv files'. Not: 'Quick update').
  - *Breaking changes* and *deprecations* should be implemented with **extreme caution**. Backwards compatibility should be maintained with a deprecation notice unless absolutely impossible.
  - Additional dependencies should be added with caution since it can have remification for end-users.
- **Documentation changes:**
  - **For all changes to the public interface, add such changes to the development changelog at "docs/source/change/d**
    - All new features will go into the next minor release.
    - Most internal changes, bug fixes, and documentation updates will be included in patches.
    - If you're not sure how the change will be integrated into releases, mention this in the message in PR or issue.
  - Updating documentation in general is not mandatory but very much welcomed. Make sure style is consistent.

1.7.3 Feature Request

If there is something that you think will make this better, please let us know!

1. Look through the Issue Tracker and, if appropriate, submit an issue for suggestion (There is a template to follow as well!).
2. If the feature deviates from the roadmap significantly, it will not be implemented. Otherwise, it will be considered based on the following factors:
   - Feature relevance
   - Current project development and timeline
   - Implementation logistics/complexity

- External dependencies (if any)

3. If you yourself would like to implement the feature, say so in the Issue, and I will evaluate it accordingly.
4. Bottomline: it doesn't hurt to ask and I'm pretty openminded.

1.8 License

PYTHON MODULE INDEX

INDEX