Sentivi

Release 1.0.4

Duy V. Huynh

Mar 31, 2021

CONTENTS:

A simple tool for sentiment analysis which is a wrapper of scikit-learn and PyTorch Transformers models (for more specific purpose, it is recommend to use native library instead). It is made for easy and faster pipeline to train and evaluate several classification algorithms.

- Install standard version from PyPI:

```
pip install sentivi
```

- Install latest version from source:

```
git clone https://github.com/vndee/sentivi
cd sentivi
pip install .
```

CHAPTER

ONE

EXAMPLE:

```
from sentivi import Pipeline
from sentivi.data import DataLoader, TextEncoder
from sentivi.classifier import SVMClassifier
from sentivi.text_processor import TextProcessor

text_processor = TextProcessor(methods=['word_segmentation', 'remove_punctuation',
↪'lower'])

pipeline = Pipeline(DataLoader(text_processor=text_processor, n_grams=3),
                    TextEncoder(encode_type='one-hot'),
                    SVMClassifier(num_labels=3))

train_results = pipeline(train='./train.txt', test='./test.txt')
print(train_results)

pipeline.save('./weights/pipeline.sentivi')
_pipeline = Pipeline.load('./weights/pipeline.sentivi')

predict_results = _pipeline.predict(['hàng ok đu tuýp có mt s không va c sit. ch đc␣
↪mt s đu thôi .cn '
                                     'nht đu tuýp 14 mà không có. không đt yêu cu ca␣
↪mình s dng',
                                     'Son đpppp, mùi hng vali thm nhng hi nng, cht son␣
↪mn, màu lên chun, '
                                     'đpppppp'])
print(predict_results)
print(f'Decoded results: {_pipeline.decode_polarity(predict_results)}')
```

Console output:

```
One Hot Text Encoder: 100%|| 6/6 [00:00<00:00, 11602.50it/s]
One Hot Text Encoder: 100%|| 2/2 [00:00<00:00, 4966.61it/s]
Input features view be flatten into np.ndarray(6, 35328) for scikit-learn classifier.
Training classifier...
Testing classifier...
Training results:
              precision    recall  f1-score   support

           0       1.00      0.00      0.00         1
           1       0.75      1.00      0.86         3
           2       1.00      1.00      1.00         2

    accuracy                           0.83         6
   macro avg       0.92      0.67      0.62         6
```

(continues on next page)

(continued from previous page)

```
weighted avg       0.88      0.83      0.76         6

Test results:
              precision    recall  f1-score   support

           1       1.00      1.00      1.00         1
           2       1.00      1.00      1.00         1

    accuracy                           1.00         2
   macro avg       1.00      1.00      1.00         2
weighted avg       1.00      1.00      1.00         2


Saved model to ./weights/pipeline.sentivi
Loaded model from ./weights/pipeline.sentivi
Input features view be flatten into np.ndarray(2, 35328) for scikit-learn classifier.
[2 1]
Decoded results: ['#NEG', '#POS']
One Hot Text Encoder: 100%|| 2/2 [00:00<00:00, 10796.15it/s]
```

1.1 Pipeline

Pipeline is a sequence of callable layer (`DataLayer`, `ClassifierLayer`). These layers will be executed with given input (text file) sequentially. Output of the pipeline is the output of last executed layer.

Pipeline can be initialized by default constructor, callable layer can be passed through pipeline in initialization once or use append method.

For example:

```
from sentivi import Pipeline
from sentivi.data import DataLoader, TextEncoder
from sentivi.classifier import SVMClassifier
from sentivi.text_processor import TextProcessor

text_processor = TextProcessor(methods=['word_segmentation', 'remove_punctuation',
↪'lower'])

pipeline = Pipeline(DataLoader(text_processor=text_processor, n_grams=3),
                    TextEncoder(encode_type='one-hot'),
                    SVMClassifier(num_labels=3))
```

or

```
pipeline = Pipeline()
pipeline.append(DataLoader(text_processor=text_processor, n_grams=3))
pipeline.append(TextEncoder(encode_type='one-hot'))
pipeline.append(SVMClassifier(num_labels=3))
```

Executing pipeline with given corpus (text file). By default text file should be in our format, double newline character `(\n\n)` is the separated symbol of training samples:

```
#corpus.txt
polarity_01
```

(continues on next page)

(continued from previous page)

```
sentence_01

polarity_02
sentence_02
```

Pipeline also accept arbitrary keyword arguments when executed function is call, these arguments is passed through executed functions of each layer. Training results will be represented as text in the form of `sklearn.metrics.classification_report`.

```
results = pipeline(train='train.txt', test='test.txt')
```

```
#results

Training classifier...
Testing classifier...
Saved classifier model to ./weights/svm.sentivi

Training results:
              precision    recall  f1-score   support

           0       1.00      0.00      0.00         1
           1       0.75      1.00      0.86         3
           2       1.00      1.00      1.00         2

    accuracy                           0.83         6
   macro avg       0.92      0.67      0.62         6
weighted avg       0.88      0.83      0.76         6

Test results:
              precision    recall  f1-score   support

           1       1.00      1.00      1.00         1
           2       1.00      1.00      1.00         1

    accuracy                           1.00         2
   macro avg       1.00      1.00      1.00         2
weighted avg       1.00      1.00      1.00         2
```

Predict polarity with given texts:

```
predict_results = pipeline.predict(['hàng ok đu tuýp có mt s không va c sit. ch đc mt␣
↪s đu thôi .cn '
                                    'nht đu tuýp 14 mà không có. không đt yêu cu ca␣
↪mình s dng',
                                    'Son đpppp, mùi hng vali thm nhng hi nng, cht son␣
↪mn, màu lên chun, '
                                    'đpppppp'])
print(predict_results)
print(f'Decoded results: {pipeline.decode_polarity(predict_results)}')
```

```
[2 1]
Decoded results: ['#NEG', '#POS']
```

For persistency, pipe can be save and load later:

```
pipeline.save('./weights/pipeline.sentivi')
_pipeline = Pipeline.load('./weights/pipeline.sentivi')

predict_results = _pipeline.predict(['hàng ok đu tuýp có mt s không va c sit. ch đc␣
↪mt s đu thôi .cn '
                                     'nht đu tuýp 14 mà không có. không đt yêu cu ca␣
↪mình s dng',
                                     'Son đpppp, mùi hng vali thm nhng hi nng, cht son␣
↪mn, màu lên chun, '
                                     'đpppppp'])
print(predict_results)
print(f'Decoded results: {_pipeline.decode_polarity(predict_results)}')
```

class sentivi.**Pipeline**(**args*, ***kwargs*)

Pipeline instance

__init__(**args*, ***kwargs*)

Initialize Pipeline instance

Parameters

- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

append(*method*)

Append a callable layer

Parameters **method** – [DataLayer, ClassifierLayer]

Returns None

decode_polarity(*x: Optional[list]*)

Decode numeric polarities into label polarities

Parameters **x** – List of numeric polarities (i.e [0, 1, 2, 1, 0])

Returns List of label polarities (i.e ['neg', 'neu', 'pos', 'neu', 'neg']

Return type List

forward(**args*, ***kwargs*)

Execute all callable layer in self.apply_layers

Parameters

- **args** –
- **kwargs** –

Returns

get_labels_set()

Get labels set

Returns List of labels

Return type List

get_server()

Serving model

Returns

get_vocab()
Get vocabulary

Returns Vocabulary in form of List

Return type List

keyword_arguments()
Return pipeline's protected attribute and its value in form of dictionary.

Returns key-value of protected attributes

Return type Dictionary

static load(*model_path: str*)
Load model from disk

Parameters **model_path** – path to pre-trained model

Returns

predict(*x: Optional[list]*, **args*, ***kwargs*)
Predict target polarity from list of given features

Parameters

- **x** – List of input texts
- **args** – arbitrary positional arguments
- **kwargs** – arbitrary keyword arguments

Returns List of labels corresponding to given input texts

Return type List

save(*save_path: str*)
Save model to disk

Parameters **save_path** – path to saved model

Returns

to(*device*)
To device

Parameters **device** –

Returns

1.2 Text Processor

Sentivi provides a simple text processor layer base on regular expression. `TextProcessor` must be defined as a attribute of `DataLoader` layer, it is a required parameter.

List of pre-built methods can be initialized as follows:

```
text_processor = TextProcessor(methods=['remove_punctuation', 'word_segmentation'])

# or add methods sequentially
text_processor = TextProcessor()
text_processor.remove_punctuation()
text_processor.word_segmentation()
```

(continues on next page)

(continued from previous page)

```
text_processor('Trng đi hc,   Tôn Đc Thng, H; Chí Minh.')
```

Result:

```
Trng đi_hc Tôn_Đc_Thng H_Chí_Minh
```

You can also add more regex pattern:

```
text_processor.add_pattern(r'[0-9]', '')
```

Or you can add your own method, use-defined method should be a lambda function.

```
text_processor.add_method(lambda x: x.strip())
```

Split n-grams example:

```
TextProcessor.n_gram_split('bài tp phân tích cm xúc', n_grams=3)
```

```
['bài tp phân', 'tp phân tích', 'phân tích cm', 'tích cm xúc']
```

class sentivi.text_processor.**TextProcessor**(*methods: Optional[list] = None*)

A simple text processor base on regex

__init__(*methods: Optional[list] = None*)

Initialize TextProcessor instance

Parameters **methods** – list of text preprocessor methods need to be applied, for example: ['remove_punctuation', 'word_segmentation']

add_method(*method*)

Add your method into TextProcessor

Parameters **method** – Lambda function

Returns

add_pattern(*pattern*, *replace_text*)

It is equivalent to re.sub()

Parameters

- **pattern** – regex pattern
- **replace_text** – replace text

Returns

capitalize()

It is equivalent to str.upper()

Returns

capitalize_first()

Capitalize first letter of a given text

Returns

lower()

Lower text

Returns

static n_gram_split(*_x*, *_n_grams*)
Split text into n-grams form

Parameters

- **_x** – Input text
- **_n_grams** – n-grams

Returns List of words

Return type List

remove_punctuation()
Remove punctuation of given text

Returns

word_segmentation()
Using PyVi to tokenize Vietnamese text. Note that this feature only use for Vietnamese text analysis.

Returns

1.3 Data Loader

`DataLoader` is a required layer of any `Pipeline`, it provides several methods for loading data from raw text file and preprocessing data by apply `TextProcessor` layer. As mentioned before, default data format of text corpus should be described as follows:

Polarity first, and text is in following line. Training samples are separated by `\n\n`.

```
# train.txt
polarity_01
sentence_01

polarity_02
sentence_02

...

# test.txt
polarity_01
sentence_01

polarity_02
sentence_02

...
```

```
data_loader = DataLoader(text_processor=text_processor)
data = data_loader(train='train.txt', test='test.txt')
```

You can set your own `delimiter` (separator between polarity and text), `line_separator` (separator between samples). For instance, if `delimiter='\t'` and `line_separator='\n'`, your data should be:

```
# train.txt
polarity_01     sentence_01
polarity_02     sentence_02
```

(continues on next page)

(continued from previous page)

```
...

# test.txt
polarity_01     sentence_01
polarity_02     sentence_02
...
```

```
data_loader = DataLoader(text_processor=text_processor, delimiter='\t', line_
↪separator='\n')
data = data_loader(train='train.txt', test='test.txt')
```

`DataLoader` will return a `sentivi.data.data_loader.Corpus` instance when executed.

class sentivi.data.**DataLoader**(*delimiter: Optional[str] = '\n', line_separator: Optional[str] = '\n\n', n_grams: Optional[int] = 1, text_processor: Optional[sentivi.text_processor.TextProcessor] = None, max_length: Optional[int] = 256, mode: Optional[str] = 'sentivi'*)

DataLoader is an inheritance class of DataLayer.

__init__(*delimiter: Optional[str] = '\n', line_separator: Optional[str] = '\n\n', n_grams: Optional[int] = 1, text_processor: Optional[sentivi.text_processor.TextProcessor] = None, max_length: Optional[int] = 256, mode: Optional[str] = 'sentivi'*)

Parameters

- **delimiter** – separator between polarity and text
- **line_separator** – separator between samples
- **n_grams** – n-gram(s) use to split, for TextEncoder such as word2vec or transformer, n-gram should be 1
- **text_processor** – sentivi.text_processor.TextProcessor instance
- **max_length** – maximum length of input text

forward(**args*, ***kwargs*)

Execute loading data pipeline

Parameters

- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns loaded data

Return type *sentivi.data.data_loader.Corpus*

1.3.1 Corpus

class sentivi.data.data_loader.**Corpus**(*train_file: Optional[str] = None, test_file: Optional[str] = None, delimiter: Optional[str] = '\n', line_separator: Optional[str] = None, n_grams: Optional[int] = None, text_processor: Optional[sentivi.text_processor.TextProcessor] = None, max_length: Optional[int] = None, truncation: Optional[str] = 'head', mode: Optional[str] = 'sentivi'*)

Text corpus for sentiment analysis

__init__(*train_file: Optional[str] = None*, *test_file: Optional[str] = None*, *delimiter: Optional[str] = '\n'*, *line_separator: Optional[str] = None*, *n_grams: Optional[int] = None*, *text_processor: Optional[sentivi.text_processor.TextProcessor] = None*, *max_length: Optional[int] = None*, *truncation: Optional[str] = 'head'*, *mode: Optional[str] = 'sentivi'*)

Initialize Corpus instance

Parameters

- **train_file** – Path to train text file
- **test_file** – Path to test text file
- **delimiter** – Separator between text and labels
- **line_separator** – Separator between samples.
- **n_grams** – N-grams
- **text_processor** – sentivi.text_processor.TextProcessor instance
- **max_length** – maximum length of input text

build()

Build sentivi.data.data_loader.Corpus instance

Returns sentivi.data.data_loader.Corpus instance

Return type sentivi.data.data_lodaer.Corpus

get_test_set()

Get test samples

Returns Input and output of test samples

Return type Tuple[List, List]

get_train_set()

Get training samples

Returns Input and output of training samples

Return type Tuple[List, List]

text_transform(*text*)

Preprocessing raw text

Parameters **text** – raw text

Returns text

Return type str

1.4 Text Encoder

`TextEncoder` is a class that receives pre-processed data from `sentivi.data.DataLoader`, its responsibility is to provide appropriate data to the respective classifications.

```
text_encoder = TextEncoder('one-hot') # ['one-hot', 'word2vec', 'bow', 'tf-idf',
↪'transformer']
```

One-hot Encoding The simplest encoding type of `TextEncoder`, each token will be represented as a one-hot vector. These vectors indicate the look-up index of given token in corpus vocabulary. For example, `vocab = ['I', 'am', 'a', 'student']`:

- one-hot('I') = [1, 0, 0, 0]
- one-hot('am') = [0, 1, 0, 0]
- one-hot('a') = [0, 0, 1, 0]
- one-hot('student') = [0, 0, 0, 1]

Bag-of-Words A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things: A vocabulary of known words. A measure of the presence of known words. More detail: https://machinelearningmastery.com/gentle-introduction-bag-words-model/

Term Frequency - Inverse Document Frequency tf-idf or TFIDF, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. This tf-idf version implemented in TextEncoder is logarithmically scaled version. More detail: https://en.wikipedia.org/wiki/Tf%E2%80%93idf

Word2Vec Word2vec (Mikolov et al, 2013) is a method to efficiently create word embeddings using distributed representation method. This implementation using gensim, it is required model_path argument in initialization stage. For vietnamese, word2vec model should be downloaded from https://github.com/sonvx/word2vecVN

```
text_encoder = TextEncoder(encode_type='word2vec', model_path='./pretrained/wiki.
↪vi.model.bin.gz')
```

Transformer Transformer text encoder is equivalent to transformer.AutoTokenizer from https://huggingface.co/transformers

class sentivi.data.**TextEncoder**(*encode_type: Optional[str] = None*, *model_path: Optional[str] = None*)

__init__(*encode_type: Optional[str] = None*, *model_path: Optional[str] = None*)
Simple text encode layer

Parameters

- **encode_type** – one type in ['one-hot', 'word2vec', 'bow', 'tf-idf', 'transformer']
- **model_path** – model is required for word2vec option

bow(*x*, *vocab*, *n_grams*) → numpy.ndarray
Bag-of-Words encoder

Parameters

- **x** – list of texts
- **vocab** – corpus vocabulary
- **n_grams** – n-grams parameters

Returns Bag-of-Words vectors

Return type numpy.ndarray

forward(*x: Optional[sentivi.data.data_loader.Corpus]*, **args*, ***kwargs*)
Execute text encoder pipeline

Parameters

- **x** – sentivi.data.data_loader.Corpus instance
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns Training and Test batch encoding

Return type Tuple, Tuple

one_hot(*x*, *vocab*, *n_grams*) → numpy.ndarray
Convert corpus into batch of one-hot vectors.

Parameters

- **x** – list of texts
- **vocab** – corpus vocabulary
- **n_grams** – n-grams parameters

Returns one-hot vectors

Return type numpy.ndarray

predict(*x*, *vocab*, *n_grams*, **args*, ***kwargs*)
Encode text for prediction purpose

Parameters

- **x** – list of text
- **vocab** – corpus vocabulary
- **n_grams** – n-grams parameters
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns encoded text

Return type transformers.BatchEncoding

tf_idf(*x*, *vocab*, *n_grams*) → numpy.ndarray
Simple TF-IDF vectors

Parameters

- **x** – list of texts
- **vocab** – corpus vocabulary
- **n_grams** – n-grams parameters

Returns encoded vectors

Return type numpy.ndarray

transformer_tokenizer(*tokenizer*, *x*)
Transformer tokenizer and encoder

Parameters

- **tokenizer** – transformer.AutoTokenizer
- **x** – list of texts

Returns encoded vectors

Return type BatchEncoding

word2vec(*x*, *n_grams*) → numpy.ndarray
word2vec embedding

Parameters

- **x** – list of texts

- **n_grams** – n-grams parameters

Returns encoded vectors

Return type numpy.ndarray

1.5 Scikit-learn Classifier

This module is a wrapper of scikit-learn library. You can initialize classifier instance as same as when initialize `scikit-learn` instance. Initialize arguments of scikit-learn is fully accept.

class sentivi.classifier.sklearn_clf.**ScikitLearnClassifier**(*num_labels: int = 3*, **args*, ***kwargs*)

Scikit-Learn Classifier-based

__init__(*num_labels: int = 3*, **args*, ***kwargs*)

Initialize ScikitLearnClassifier instance

Parameters

- **num_labels** – number of polarities
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

forward(*data*, **args*, ***kwargs*)

Train and evaluate ScikitLearnClassifier instance

Parameters

- **data** – Output of TextEncoder
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns Training and evaluating result

Return type str

load(*model_path*, **args*, ***kwargs*)

Load model from disk

Parameters

- **model_path** – path to pre-trained model path
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns

predict(*x*, **args*, ***kwargs*)

Predict polarities given sentences

Parameters

- **x** – TextEncoder.predict output
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns list of polarities

Return type list

save(*save_path*, **args*, ***kwargs*)
Save model to disk

Parameters

- **save_path** – path to save model
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns

1.5.1 Naive Bayes

class sentivi.classifier.**NaiveBayesClassifier**(*num_labels: int = 3*, **args*, ***kwargs*)

__init__(*num_labels: int = 3*, **args*, ***kwargs*)
Initialize NaiveBayesClassifier

Parameters

- **num_labels** – number of polarities
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

1.5.2 Decision Tree

class sentivi.classifier.**DecisionTreeClassifier**(*num_labels: int = 3*, **args*, ***kwargs*)

__init__(*num_labels: int = 3*, **args*, ***kwargs*)
Initialize DecisionTreeClassifier

Parameters

- **num_labels** – number of polarities
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

1.5.3 Gaussian Process

class sentivi.classifier.**GaussianProcessClassifier**(*num_labels: int = 3*, **args*, ***kwargs*)

__init__(*num_labels: int = 3*, **args*, ***kwargs*)
Initialize GaussianProcessClassifier

Parameters

- **num_labels** – number of polarities
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

1.5.4 Multi-Layer Perceptron

class sentivi.classifier.**MLPClassifier**(*num_labels: int = 3*, **args*, ***kwargs*)

__init__(*num_labels: int = 3*, **args*, ***kwargs*)
Initialize MLPClassifier

Parameters

- **num_labels** – number of polarities
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

1.5.5 Nearest Centroid

class sentivi.classifier.**NearestCentroidClassifier**(*num_labels: int = 3*, **args*, ***kwargs*)

__init__(*num_labels: int = 3*, **args*, ***kwargs*)
Initialize NearestCentroidClassifier.txt

Parameters

- **num_labels** – number of polarities
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

1.5.6 Stochastic Gradient Descent

class sentivi.classifier.**SGDClassifier**(*num_labels: int = 3*, **args*, ***kwargs*)

__init__(*num_labels: int = 3*, **args*, ***kwargs*)
Initialize SGDClassifier

Parameters

- **num_labels** – number of polarities
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

1.5.7 Support Vector Machine

class sentivi.classifier.**SVMClassifier**(*num_labels: int = 3*, **args*, ***kwargs*)

__init__(*num_labels: int = 3*, **args*, ***kwargs*)
Initialize SVMClassifier

Parameters

- **num_labels** – number of polarities
- **args** – arbitrary arguments

- **kwargs** – arbitrary keyword arguments

1.6 Neural Network Classifier

This classifier is based on Neural Network Model

class sentivi.classifier.nn_clf.**NeuralNetworkClassifier**(*num_labels: int = 3*, *embedding_size: Optional[int] = None*, *max_length: Optional[int] = None*, *device: Optional[str] = 'cpu'*, *num_epochs: Optional[int] = 10*, *learning_rate: Optional[float] = 0.001*, *batch_size: Optional[int] = 2*, *shuffle: Optional[bool] = True*, *random_state: Optional[int] = 101*, *hidden_size: Optional[int] = 512*, *num_workers: Optional[int] = 2*, **args*, ***kwargs*)

Neural Network Classifier

__init__(*num_labels: int = 3*, *embedding_size: Optional[int] = None*, *max_length: Optional[int] = None*, *device: Optional[str] = 'cpu'*, *num_epochs: Optional[int] = 10*, *learning_rate: Optional[float] = 0.001*, *batch_size: Optional[int] = 2*, *shuffle: Optional[bool] = True*, *random_state: Optional[int] = 101*, *hidden_size: Optional[int] = 512*, *num_workers: Optional[int] = 2*, **args*, ***kwargs*)

Neural Network Classifier

Parameters

- **num_labels** – number of polarities
- **embedding_size** – input embedding size
- **max_length** – maximum number of input text
- **device** – training device
- **num_epochs** – maximum number of epochs
- **learning_rate** – training learning rate
- **batch_size** – training batch size
- **shuffle** – whether DataLoader is shuffle or not
- **random_state** – random.seed
- **hidden_size** – hidden size
- **num_workers** – number of DataLoader workers
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

static compute_metrics(*preds*, *targets*, *eval=False*)
Compute accuracy and F1

Parameters

- **preds** – prediction output
- **targets** – ground-truth value
- **eval** – whether is eval or not

Returns

fit(**args*, ***kwargs*)
Feed-forward network

Parameters

- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns

forward(*data*, **args*, ***kwargs*)
Training and evaluating NeuralNetworkClassifier

Parameters

- **data** – TextEncoder output
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns training and evaluating results

Return type str

get_overall_result(*loader*)
Get overall result

Parameters **loader** – DataLoader

Returns overall result

Return type str

load(*model_path*, **args*, ***kwargs*)
Load model from disk

Parameters

- **model_path** – path to model path
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns

save(*save_path*, **args*, ***kwargs*)
Save model to disk

Parameters

- **save_path** – path to saved model
- **args** – arbitrary arguments

- **kwargs** – arbitrary keyword arguments

Returns

1.6.1 Text Convolutional Neural Network

class sentivi.classifier.**TextCNNClassifier**(*num_labels: int*, *embedding_size: Optional[int] = None*, *max_length: Optional[int] = None*, *device: Optional[str] = 'cpu'*, *num_epochs: Optional[int] = 10*, *learning_rate: Optional[float] = 0.001*, *batch_size: Optional[int] = 2*, *shuffle: Optional[bool] = True*, *random_state: Optional[int] = 101*, **args*, ***kwargs*)

__init__(*num_labels: int*, *embedding_size: Optional[int] = None*, *max_length: Optional[int] = None*, *device: Optional[str] = 'cpu'*, *num_epochs: Optional[int] = 10*, *learning_rate: Optional[float] = 0.001*, *batch_size: Optional[int] = 2*, *shuffle: Optional[bool] = True*, *random_state: Optional[int] = 101*, **args*, ***kwargs*)

Initialize TextCNNClassifier

Parameters

- **num_labels** – number of polarities
- **embedding_size** – input embedding size
- **max_length** – maximum length of input text
- **device** – training device
- **num_epochs** – maximum number of epochs
- **learning_rate** – training learning rate
- **batch_size** – training batch size
- **shuffle** – whether DataLoader is shuffle or not
- **random_state** – random.seed
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

forward(*data*, **args*, ***kwargs*)

Training and evaluating method

Parameters

- **data** – TextEncoder output
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns training and evaluating results

Return type str

predict(*X*, **args*, ***kwargs*)

Predict polarity with given sentences

Parameters

- **X** – TextEncoder.predict output
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns list of numeric polarities

Return type list

1.6.2 Long Short Term Memory

class sentivi.classifier.**LSTMClassifier**(*num_labels: int*, *embedding_size: Optional[int] = None*, *max_length: Optional[int] = None*, *device: Optional[str] = 'cpu'*, *num_epochs: Optional[int] = 10*, *learning_rate: Optional[float] = 0.001*, *batch_size: Optional[int] = 2*, *shuffle: Optional[bool] = True*, *random_state: Optional[int] = 101*, *hidden_size: Optional[int] = 512*, *hidden_layers: Optional[int] = 2*, *bidirectional: Optional[bool] = False*, *attention: Optional[bool] = True*, **args*, ***kwargs*)

__init__(*num_labels: int*, *embedding_size: Optional[int] = None*, *max_length: Optional[int] = None*, *device: Optional[str] = 'cpu'*, *num_epochs: Optional[int] = 10*, *learning_rate: Optional[float] = 0.001*, *batch_size: Optional[int] = 2*, *shuffle: Optional[bool] = True*, *random_state: Optional[int] = 101*, *hidden_size: Optional[int] = 512*, *hidden_layers: Optional[int] = 2*, *bidirectional: Optional[bool] = False*, *attention: Optional[bool] = True*, **args*, ***kwargs*)

Initialize LSTMClassifier

Parameters

- **num_labels** – number of polarities
- **embedding_size** – input embeddings' size
- **max_length** – maximum length of input text
- **device** – training device
- **num_epochs** – maximum number of epochs
- **learning_rate** – model learning rate
- **batch_size** – training batch size
- **shuffle** – whether DataLoader is shuffle or not
- **random_state** – random.seed number
- **hidden_size** – Long Short Term Memory hidden size
- **bidirectional** – whether to use BiLSTM or not
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

forward(*data*, **args*, ***kwargs*)

Training and evaluating methods

Parameters

- **data** – TextEncoder output
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns training results

predict(*X*, **args*, ***kwargs*)

Predict polarity with given sentences

Parameters

- **X** – TextEncoder.predict output
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns list of numeric polarities

Return type list

1.7 Transformer Classifier

`TransformerClassifier` base on transformers library. This is a wrapper of `transformers.AutoModelForSequenceClassification`, language model should be one of shortcut in transformers pre-trained models or using one in `['vinai/phobert-base', 'vinai/phobert-large']`

```
TransformerClassifier(num_labels=3, language_model_shortcut='vinai/phobert-base',␣
↪device='cuda')
```

class sentivi.classifier.**TransformerClassifier**(*num_labels: Optional[int] = 3*, *language_model_shortcut: Optional[str] = 'vinai/phobert'*, *freeze_language_model: Optional[bool] = True*, *batch_size: Optional[int] = 2*, *warmup_steps: Optional[int] = 100*, *weight_decay: Optional[float] = 0.01*, *accumulation_steps: Optional[int] = 50*, *save_steps: Optional[int] = 100*, *learning_rate: Optional[float] = 3e-05*, *device: Optional[str] = 'cpu'*, *optimizer=None*, *criterion=None*, *num_epochs: Optional[int] = 10*, *num_workers: Optional[int] = 2*, **args*, ***kwargs*)

class TransformerDataset(*batch_encodings*, *labels*)

__init__(*batch_encodings*, *labels*)

Initialize transformer dataset

Parameters

- **batch_encodings** –
- **labels** –

class TransformerPredictedDataset(*batch_encodings*)

__init__(*batch_encodings*)
Initialize transformer dataset

Parameters **batch_encodings** –

__init__(*num_labels: Optional[int] = 3*, *language_model_shortcut: Optional[str] = 'vinai/phobert'*, *freeze_language_model: Optional[bool] = True*, *batch_size: Optional[int] = 2*, *warmup_steps: Optional[int] = 100*, *weight_decay: Optional[float] = 0.01*, *accumulation_steps: Optional[int] = 50*, *save_steps: Optional[int] = 100*, *learning_rate: Optional[float] = 3e-05*, *device: Optional[str] = 'cpu'*, *optimizer=None*, *criterion=None*, *num_epochs: Optional[int] = 10*, *num_workers: Optional[int] = 2*, **args*, ***kwargs*)
Initialize TransformerClassifier instance

Parameters

- **num_labels** – number of polarities
- **language_model_shortcut** – language model shortcut
- **freeze_language_model** – whether language model is freeze or not
- **batch_size** – training batch size
- **warmup_steps** – learning rate warm up step
- **weight_decay** – learning rate weight decay
- **accumulation_steps** – optimizer accumulation step
- **save_steps** – saving step
- **learning_rate** – training learning rate
- **device** – training and evaluating rate
- **optimizer** – training optimizer
- **criterion** – training criterion
- **num_epochs** – maximum number of epochs
- **num_workers** – number of DataLoader workers
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

forward(*data*, **args*, ***kwargs*)
Training and evaluating TransformerClassifier instance

Parameters

- **data** – TransformerTextEncoder output
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns training and evaluating results

Return type str

get_overall_result(*loader*)
Get overall result

Parameters **loader** – DataLoader

Returns overall result

Return type str

load(*model_path*, **args*, ***kwargs*)
Load model from disk

Parameters

- **model_path** – path to model path
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns

predict(*X*, **args*, ***kwargs*)
Predict polarities with given list of sentences

Parameters

- **X** – list of sentences
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns list of polarities

Return type str

save(*save_path*, **args*, ***kwargs*)
Save model to disk

Parameters

- **save_path** – path to saved model
- **args** – arbitrary arguments
- **kwargs** – arbitrary keyword arguments

Returns

1.8 Ensemble Learning

Ensemble and Stacking methods

1.9 Serving

Sentivi use FastAPI to serving pipeline. Simply run a web service as follows:

```
# serving.py
from sentivi import Pipeline, RESTServiceGateway

pipeline = Pipeline.load('./weights/pipeline.sentivi')
server = RESTServiceGateway(pipeline).get_server()
```

```
# pip install uvicorn python-multipart
uvicorn serving:server --host 127.0.0.1 --port 8000
```

Access Swagger at http://127.0.0.1:8000/docs or Redoc http://127.0.0.1:8000/redoc. For example, you can use curl to send post requests:

```
curl --location --request POST 'http://127.0.0.1:8000/get_sentiment/' \
     --form 'text=Son đpppp, mùi hng vali thm nhng hi nng'

# response
{ "polarity": 2, "label": "#POS" }
```

Deploy using Docker

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7

COPY . /app

ENV PYTHONPATH=/app
ENV APP_MODULE=serving:server
ENV WORKERS_PER_CORE=0.75
ENV MAX_WORKERS=6
ENV HOST=0.0.0.0
ENV PORT=80

RUN pip install -r requirements.txt
```

```
docker build -t sentivi .
docker run -d -p 8000:80 sentivi
```

INDEX

Symbols

A

B

C

D

F

G

K

L

M

N

O

P

R

S

T

W