Athena AIML NLP Tools

Release main

Athena AIML Contributors

Feb 27, 2024

CONTENTS

sphinx-quickstart on Wed Feb 21 22:09:41 2024. You can adapt this file completely to your liking, but it should at least contain the root *toctree* directive.

CHAPTER

ONE

STRING FILTER

1.1 Overview

Defines the StringFilter class which is used to filter Mattermost messages

class at_nlp.filters.string_filter.**StringFilter**(*verbose=True*, *model_path: Path | None = None*)

This is a conceptual class representation of a simple BLE device (GATT Server). It is essentially an extended combination of the

acronym_mapping: Dict[str, str]

Mapping of acronyms to their meanings from provided CSV

add_labeling_fn(*labeling_fn: LabelingFunction*) → None

Adds a labeling function to the filter

Parameters

labeling_fn (*LabelingFunction*) – A Snorkel labeling function to be used in the ensemble. The labeling function takes in a Panda's Series and returns an integer representing the label. See the provided example for more information.

Returns

None

Raises

ValueError – If the supplied function is not a Snorkel labeling function

Example

```
>>> from at_nlp.filters.string_filter import StringFilter
>>> from snorkel.labeling import LabelingFunction
>>> sf = StringFilter()
>>> @labeling_function()
>>> def lf_example(ds: pd.Series) -> int:
>>>     # This function will test for string lengths greater than 10
>>>     col_name = "Test"
>>>     if len(ds[col_name]) >= 10:
>>>         return 1
>>>     return 0
>>> sf.add_labeling_fn(lf_example)  # noqa
```

add_multiple_labeling_fns(*labeling_fn_list: list[LabelingFunction]*) → None

Convenience function to add multiple labeling functions to the filter

Parameters
labeling_fn_list (*list[LabelingFunction]*) – List of Snorkel labeling functions to be added

Returns
None

Raises:

applier: PandasLFApplier

Applies functions to Pandas DataFrame

class_likelihood = 0.6

Threshold for class probabilities

cv

Coverts messages to a sparse matrix of token counts

drain_config: TemplateMinerConfig

Configuration dictionary from drain3.ini file in cur dir

drain_config_path: Path

Path to drain3.ini file as pathlib.Path object

evaluate(*test_data: pd.DataFrame | np.array*, *test_labels: pd.Series*, *classifier_id: str = 'rf'*) → None

Evaluate trained weak learners.

filter_result: FilterResult

Enumeration of categories for each message

keyword_register: List[str] = []

Iterable of keywords to strain

label_model: LabelModel

Ensemble of labeling models

latency_trace(*test_data: pd.DataFrame | np.array*) → None

Evaluate the inference speed of the classifiers

load_models(*model_dir: Path*) → None

Restore models from a directory

max_str_len: int = None

Maximum length of a message

min_str_len: int = None

Minimum length of a message

mlp = MLPClassifier(alpha=1, max_iter=1000)

Simple MLP classifier for ensemble of weak learners

msg_len_cutoff = 7

Number of characters in a message to define a short message

predict(*in_data: DataFrame*) → ndarray

Predict the labels for a supplied Pandas data frame

print_weak_learner_info(*l_train*)

Prints the weak learners collisions, etc.

register_keywords(*keywords: list[str]*, *make_lowercase: bool = True*) → None

Register new keywords to be used in the labeling functions

remove_labeling_fn(*labeling_fn: LabelingFunction*) → None

Remove a labeling function from the filter.

Parameters
labeling_fn (*LabelingFunction*) – Labeling function to remove from the filter

Returns
None

Raises
ValueError – If the labeling function is not in the filter

Example

```
>>> from at_nlp.filters.string_filter import StringFilter
>>> from snorkel.labeling import LabelingFunction
>>> sf = StringFilter()
>>> # Define a labeling function
>>> @labeling_function()
>>> def lf_example(ds: pd.Series) -> int:
>>>     # This function will test for string lengths greater than 10
>>>     col_name = "Test"
>>>     if len(ds[col_name]) >= 10:
>>>         return 1
>>>     return 0
>>> sf.add_labeling_fn(lf_example)  # noqa
>>> # Remove the previously added labeling function
>>> sf.remove_labeling_fn(lf_example)  # noqa
```

classmethod reset() → None

Reset the class to its default state

Returns
None

Example

```
>>> sf = StringFilter()
>>> sf.reset()
```

rf

Random forest classifier for ensemble of weak learners

salutations = ['hello', 'hola', 'aloha', 'mornin', 'ello govna', 'good morning', 'good evening', 'good night', 'good <:*:>', 'hey <:*:>', 'hi <:*:>', 'haha <:*:>']

List of salutations to filter

save_models(*save_path_stub: Path*) → None

Save trained models to directory with a random uuid to prevent collisions

save_template_miner_cluster_information() → None

Save template miner clusters to a JSON for analysis

set_string_len_bounds(*lower_bound: int*, *upper_bound: int*) → None

Set the lower and upper bounds for the string length labeling function

stage_one_test_data: DataFrame

Data used to test stage one

stage_one_train(*in_data: DataFrame*, *train_config: dict*)

Train the MLP and RF on the reserved stage one training data

stage_one_train_data: DataFrame

Data used to train stage one

stage_two_test_data: DataFrame

Data used to test stage two

stage_two_train(*in_data: DataFrame*, *train_config: Dict*)

Train the ensemble on the reserved stage two training data

stage_two_train_data: DataFrame

Data used to train stage two

template_miner: TemplateMiner

Drain3 template miner to convert log messages to cluster templates

static template_miner_transform(*in_row: Series*, *tm: TemplateMiner*) → str

Helper function to transform messages into their cluster templates

trace_mode: bool = False

Toggle tracing mode

trace_stack: Dict = {}

Retain performance metrics for each classifier

train(*in_data: DataFrame*, *train_conf: Dict*, *serialize=False*)

Trains both the first and second stages

train_template_miner(*in_data: DataFrame*) → None

Train the drain3 template miner first on all available data

transform(*in_data: array*, *pred_fun: MLPClassifier | SVC | RandomForestClassifier*) → array

Generic prediction function that calls the predict method of the supplied callable

update_labeling_fn(*labeling_fn: LabelingFunction*) → None

Update an existing labeling function in the filter

Parameters

labeling_fn (*LabelingFunction*) – Updated labeling function

Returns

None

Raises

ValueError – If the labeling function is not in the filter

Example

```
>>> from at_nlp.filters.string_filter import StringFilter
```

use_random_forest(*random_forest: RandomForestClassifier*) → None

Sets up the StringFilter to use a provided RandomForest Classifier from sklearn

Parameters
random_forest (*RandomForestClassifier*) – Sklearn RandomForest Classifier reference

Returns
None

Example

```
>>> from at_nlp.filters.string_filter import StringFilter
>>> from sklearn.ensemble import RandomForestClassifier  # noqa
>>> random_forest = RandomForestClassifier()  # noqa
>>> string_filter = StringFilter()
>>> @labeling_function()
>>> def new_labeling_function(ds: pd.Series) -> pd.Series:
        x = cv.transform(ds)
        x =
>>>
>>> string_filter.use_random_forest(random_forest)
```

vectorize_text(*ds: Series*) → array

Helper function to vectorize the messages in a pandas Series

verbose: bool

Whether to print diagnostic information to stdout

CHAPTER

TWO

PREPROCESSOR STACK

2.1 Overview

class at_nlp.filters.preprocessor_stack.**PreprocessorStack**

The PreprocessorStack is an iterable of preprocessors designed to operate on a dataframe. Each preprocessor takes a single Pandas Series and a column index and returns a Pandas Series. Multiprocessing is available via the 'multiprocessing' flag.

add(*preprocessor: Callable[[Series, int | str], Series]*, *position: int = -1*) → None

Adds a preprocessor to the filter. Pre-processors must have the following function signature: (pd.Series, int) -> pd.Series. The second argument of the function is column index or name of the item in the series to operate on.

Parameters

- **preprocessor** – A preprocessor function that operates on Pandas Series.
- **position** – Index position of the preprocessor in the stack

Returns

None

Raises

IndexError – If the preprocessor function is not callable or has the wrong signature.

Example

```
>>> from at_nlp.filters.preprocessor_stack import PreprocessorStack
>>> def make_lower_case(ds: pd.Series, col_idx: int):
>>>     s: str = ds.iat[col_idx]
>>>     ds.iat[col_idx] = s.lower()
>>>     return ds
>>> stack = PreprocessorStack()
>>> stack.add(make_lower_case)
>>> # stack.add(make_lower_case, 0) # a position can be given in the range [0,␣
↪len(stack))
```

add_csv_preprocessor(*csv_path: Path*, *search_idx: int = 0*, *replace_idx: int = 1*, *order: int | None = None*) → None

Registers a CSV file to be used for preprocessing. This is different from registering a CSV file for weak learning since we replace the strings before the weak learners are trained and applied. If you wish to use the CSV for weak learning then use the `register_csv_weak_learner()` method instead.

Note: The CSV file will not be serialized when saving the StringFilter object. Internally we will store the search and replacement strings in a dictionary that will get pickled with the object. Thus, when loading the object the CSV file is not necessary.

Parameters

- **csv_path** (*Path*) – Path to the CSV file.
- **search_idx** (*int*) – Index of the column containing the string to be replaced (Defaults to 0).
- **replace_idx** (*int*) – Index of the column containing the replacement string (Defaults to 1).
- **order** (*int | None*) – The position in the call stack to place the preprocessor function. The default is None which places the caller at the end of the stack.

Returns
None

Raises
AssertionError – raised if the CSV file does not exist, or the indices are not integers.

Example

```
>>> from at_nlp.filters.preprocessor_stack import PreprocessorStack
>>> stack = PreprocessorStack()
>>> stack.add_csv_preprocessor(Path("replacement_text1.csv"), 0, 1)
>>> stack.add_csv_preprocessor(Path("replacement_text2.csv"))
```

add_multiple(*preprocessors: Iterable[tuple[int, Callable[[Series, int | str], Series]]]*)

Adds multiple preprocessors to the stack. Takes in a tuple of indices and preprocessors, using the indices for insertion position.

Parameters

- **(Iterable[tuple[int** (*preprocessors*) –
- **Callable[[pd.Series** –
- **str]** (*int |*) –
- **pd.Series]]]** –

Returns
None

Example

```
>>> from at_nlp.filters.preprocessor_stack import PreprocessorStack
>>> stack = PreprocessorStack()
>>> processor0 = ... # func with signature (pd.Series, int) -> pd.Series
>>> processor1 = ... # func with signature (pd.Series, int) -> pd.Series
>>> stack.add_multiple([(0, processor0), (1, processor1)])
```

append(*preprocessor: Callable[[Series, int | str], Series]*) → None

convenience function that calls add(fn, -1)

remove(*preprocessor: Callable[[Series, int | str], Series]*)

Remove a preprocessor from the stack.

Parameters
preprocessor (*Callable[[pd.Series, int | str], pd.Series]*) – Preprocessor reference to be removed

Returns
None

Raises
ValueError – If the preprocessor is not callable or the preprocessor is not in the stack.

Example

```
>>> from at_nlp.filters.preprocessor_stack import PreprocessorStack
>>> stack = PreprocessorStack()
>>> # Define a preprocessor
>>> def example_preprocessor(ds: pd.Series, position: int) -> pd.Series:
>>>     # This function will test for string lengths greater than 10
>>>     if len(ds.iat[position]) >= 10:
>>>         return ds
>>>     ds.iat[position] = ""
>>>     return ds
>>> stack.append(example_preprocessor)
>>> # Remove the previously added preprocessor
>>> stack.remove(example_preprocessor)
```

update(*preprocessor: Callable[[Series, int | str], Series]*)

Update an existing preprocessor in the stack

Parameters
preprocessor (*Preprocessor*) – Updated preprocessor

Returns
None

Raises
ValueError – If the preprocessor is not in the stack.

Example

```
>>> from at_nlp.filters.preprocessor_stack import PreprocessorStack
>>> stack = PreprocessorStack()
>>> # Define a preprocessor
>>> def example_preprocessor(ds: pd.Series, position: int) -> pd.Series:
>>>     s: str = ds.iat[position]
>>>     ds.iat[position] = s.lower()
>>>     return ds
>>> stack.append(example_preprocessor)
>>> # event necessitates changing a preprocessor
>>> def example_preprocessor(ds: pd.Series, position: int) -> pd.Series:
>>>     s: str = ds.iat[position]
>>>     ds.iat[position] = s.upper() # <-- change
>>>     return ds
>>> stack.update(example_preprocessor)
```

CHAPTER

THREE

RETRIEVAL AUGMENTED GENERATION (RAG)

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

S

T

U

V