Athena AIML NLP Tools

Release main

Athena AIML Contributors

Feb 21, 2024

CONTENTS

CHAPTER

ONE

FILTERS.STRING_FILTER

1.1 Overview

Defines the StringFilter class which is used to filter Mattermost messages

class filters.string_filter.**StringFilter**(*verbose=True*, *model_path: Path | None = None*, *acronyms_path: Path | None = None*, *drain_config_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 `bluepy.btle.Peripheral` and `bluepy.btle.ScanEntry` classes

Parameters

- **client** (class:*simpleble.SimpleBleClient*) – A handle to the `simpleble.SimpleBleClient` client object that detected the device
- **addr** (*str, optional*) – Device MAC address, defaults to None
- **addrType** (*str, optional*) – Device address type - one of ADDR_TYPE_PUBLIC or ADDR_TYPE_RANDOM, defaults to ADDR_TYPE_PUBLIC
- **iface** (*int, optional*) – Bluetooth interface number (0 = /dev/hci0) used for the connection, defaults to 0
- **data** (*list, optional*) – A list of tuples (adtype, description, value) containing the AD type code, human-readable description and value for all available advertising data items, defaults to None
- **rssi** (*int, optional*) – Received Signal Strength Indication for the last received broadcast from the device. This is an integer value measured in dB, where 0 dB is the maximum (theoretical) signal strength, and more negative numbers indicate a weaker signal, defaults to 0
- **connectable** (*bool, optional*) – *True* if the device supports connections, and *False* otherwise (typically used for advertising 'beacons')., defaults to *False*
- **updateCount** (*int, optional*) – Integer count of the number of advertising packets received from the device so far, defaults to 0

acronym_mapping: Dict[str, str] = {}

Mapping of acronyms to their meanings from provided CSV

applier: PandasLFApplier

Applies functions to Pandas DataFrame

class_likelihood = 0.6

Threshold for class probabilities

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: DataFrame | array*, *test_labels: Series*, *classifier_id: str = 'rf'*) → None

Return a list of random ingredients as strings.

Parameters
kind (*list[str] or None*) – Optional "kind" of ingredients.

Raises
lumache.InvalidKindError – If the kind is invalid.

Returns
The ingredients list.

Return type
list[str]

filter_result

Enumeration of categories for each message

alias of `FilterResult`

fix_text(*in_text: str*) → str

fractional_max_pool2d(input, kernel_size, output_size=None, output_ratio=None, return_indices=False, _random_samples=None)

Applies 2D fractional max pooling over an input signal composed of several input planes.

Fractional MaxPooling is described in detail in the paper Fractional MaxPooling by Ben Graham

The max-pooling operation is applied in $kH \times kW$ regions by a stochastic step size determined by the target output size. The number of output features is equal to the number of input planes.

Parameters

- **kernel_size** – the size of the window to take a max over. Can be a single number k (for a square kernel of $k \times k$) or a tuple *(kH, kW)*
- **output_size** – the target output size of the image of the form $oH \times oW$. Can be a tuple *(oH, oW)* or a single number oH for a square image $oH \times oH$
- **output_ratio** – If one wants to have an output size as a ratio of the input size, this option can be given. This has to be a number or tuple in the range (0, 1)
- **return_indices** – if `True`, will return the indices along with the outputs. Useful to pass to `max_unpool2d()`.

Examples::

```
>>> input = torch.randn(20, 16, 50, 32)
>>> # pool of square window of size=3, and target output size 13x12
>>> F.fractional_max_pool2d(input, 3, output_size=(13, 12))
>>> # pool of square window and target output size being half of input␣
↪image size
>>> F.fractional_max_pool2d(input, 3, output_ratio=(0.5, 0.5))
```

keyword_register: List[str] = []

Iterable of keywords to strain

label_model: LabelModel

Ensemble of labeling models

latency_trace(*test_data: DataFrame | array*) → None

Evaluate the inferencing 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 mesasge

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

register_new_labeling_fn(*fns: List[LabelingFunction]*)

Register any new labeling functions

replace_acronyms(*in_text: str*) → str

Replace acronyms with their meaningful names

rf = RandomForestClassifier()

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*) → None

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_applier()

Update applier with new lists

vectorize_text(*ds: Series*) → array

Helper function to vectorize the messages in a pandas Series

vectorizer = CountVectorizer()

Coverts messages to a sparse matrix of token counts

verbose: bool

Whether to print diagnostic information to stdout

CHAPTER

TWO

RETRIEVAL AUGMENTED GENERATION (RAG)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

T

U

V