# DeepCASE

***Release 1.0.1***

**Thijs van Ede**

**Aug 01, 2023**

# CONTENTS:

This is the official documentation for the DeepCASE tool by the authors of the IEEE S&P DeepCASE: Semi-Supervised Contextual Analysis of Security Events paper. Please cite this work when using the software for academic research papers, see *Citing* for more information.

DeepCASE introduces a semi-supervised approach for the contextual analysis of security events. This approach automatically finds correlations in sequences of security events and clusters these correlated sequences. The clusters of correlated sequences are then shown to security operators who can set policies for each sequence. Such policies can ignore sequences of unimportant events, pass sequences to a human operator for further inspection, or (in the future) automatically trigger response mechanisms. The main contribution of this work is to reduce the number of manual inspection security operators have to perform on the vast amounts of security events that they receive.

CHAPTER

ONE

# INSTALLATION

The most straightforward way of installing DeepCASE is via pip

```
pip install deepcase
```

## 1.1 From source

If you wish to stay up to date with the latest development version, you can instead download the source code. In this case, make sure that you have all the required *dependencies* installed. You can clone the code from GitHub:

```
git clone git@github.com:Thijsvanede/deepcase.git
```

Next, you can install the latest version using pip:

```
pip install -e <path/to/DeepCASE/directory/containing/setup.py>
```

### 1.1.1 Dependencies

DeepCASE requires the following python packages to be installed:

- Argformat: https://pypi.org/project/argformat/
- Numpy: https://numpy.org
- Pandas: https://pandas.pydata.org/
- PyTorch: https://pytorch.org/
- Scikit-learn: https://scikit-learn.org/stable/index.html
- Scipy: https://www.scipy.org/
- Tqdm: https://tqdm.github.io/

All dependencies should be automatically downloaded if you install DeepCASE via pip. However, should you want to install these libraries manually, you can install the dependencies using the requirements.txt file

```
pip install -r requirements.txt
```

Or you can install these libraries yourself

```
pip install -U argformat numpy pandas torch torchvision torchaudio scikit-learn scipy␣
↪tqdm
```

CHAPTER

# TWO

# USAGE

The DeepCASE package offers both a commond-line tool for easy access and a rich API for full customisation. This section gives a high-level overview of the different steps taken by DeepCASE to perform a contextual analysis of security events and explains how DeepCASE clusters events to reduce the workload of security analysts. We also include several working examples to guide users through the code. For detailed documentation of individual methods, we refer to the *Reference* guide.

## 2.1 Overview

This section gives a high-level overview of the different steps taken by DeepCASE to perform a contextual analysis of security events and explains how DeepCASE clusters events to reduce the workload of security analysts.

1) *Event sequencing*
2) *Context Builder*
3) *Interpreter*
4) *Manual Analysis*
5) *Semi-automatic Analysis*

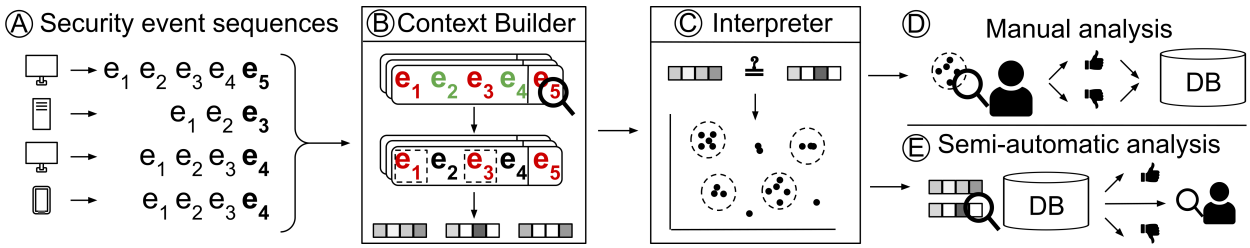


Fig. 1: Figure 1: Overview of DeepCASE.

### 2.1.1 Event sequencing

The first step is to transform events stored in your local format into a format that DeepCASE can handle. For this step, we use the *Preprocessor* class, which is able to take events stored in a `.csv` and `.txt` format and transform them into DeepCASE sequences. For the required formats for both the `.csv` and `.txt` files, we refer to the *Preprocessor* reference.

### 2.1.2 Context Builder

Next, DeepCASE passes the sequences to the *ContextBuilder*. When receiving sequences, the *ContextBuilder* first applies its `fit()` method to train its neural network. Once the network is trained, we use the *ContextBuilder*'s `predict()` method to get the `confidence` in each event with its context and `attention` for all events in the context. These `confidence` and `attention` values can then be passed to the *Interpreter* together with the `events` and their `context` for clustering.

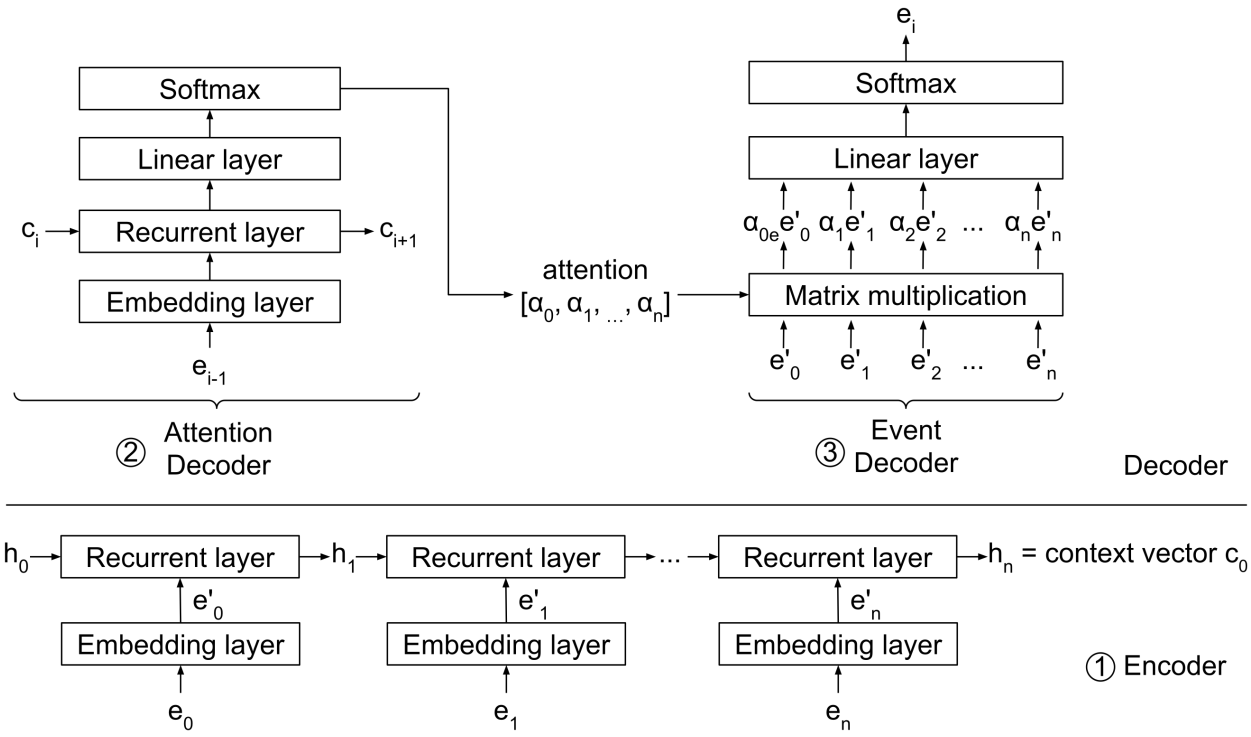


Fig. 2: Figure 2: Architecture of DeepCASE's Context Builder.

### 2.1.3 Interpreter

The main task of the *Interpreter* is to take sequences (consisting of `context` and `events`) and cluster them. To this end, the *Interpreter* invokes the *ContextBuilder*'s `predict()` method and applies the `attention_query()` to obtain a vector representing each sequence. These vectors are then used for clustering. Afterwards, clusters can be manually analysed and assigned a score. After assigning a score to existing clusters, the *Interpreter* can compares new `context` and `events` to existing clusters and assign the scores (semi-)automatically. For sequences that cannot be assigned automatically, the *Interpreter* gives an output indicating why a sequence could not be assigned automatically.

### 2.1.4 Manual Analysis

In manual mode, we use the *interpreter.Interpreter.cluster()* method to cluster sequences consisting of `context` and `events`. This method returns the cluster corresponding to each input, or `-1` if no cluster could be found.

Next, we can manually assign scores using the *interpreter.Interpreter.score()* function. This function takes a score for each clustered sequence and assigns it to the corresponding clusters such that these scores can be used for predicting new sequences.

---

**Note:**

**The *interpreter.Interpreter.score()* function requires:**

1. that all sequences used to create clusters are assigned a score.
2. that all sequences in the **same** cluster are assigned the **same** score.

If you do not have labels for all clusters or different labels within the same cluster, the *interpreter.Interpreter.score_clusters()* method prepares scores such that both conditions are satisfied.

---

### 2.1.5 Semi-automatic Analysis

In semi-automatic mode, we use the *interpreter.Interpreter.predict()* method to assign scores to new sequences (`context` and `events`) based on known clusters. It will either assign the score of the given cluster or a score of:

- `-1`, if the *ContextBuilder* is not confident enough for a prediction.
- `-2`, if the `event` was not in the training dataset.
- `-3`, if the nearest cluster is a larger distance than `epsilon` away from the sequence.

## 2.2 Command line tool

When DeepCASE is installed, it can be used from the command line. The `__main__.py` file in the `deepcase` module implements this command line tool. The command line tool provides a quick and easy interface to predict sequences from `.csv` or `.txt` files. The full command line usage is given in its help page:

```
usage: deepcase.py [-h] [--csv CSV] [--txt TXT] [--events EVENTS] [--length LENGTH] [--
↪timeout TIMEOUT]
                   [--save-sequences SAVE_SEQUENCES] [--load-sequences LOAD_SEQUENCES] [-
↪-hidden HIDDEN]
                   [--delta DELTA] [--save-builder SAVE_BUILDER] [--load-builder LOAD_
↪BUILDER]
                   [--confidence CONFIDENCE] [--epsilon EPSILON] [--min_samples MIN_
↪SAMPLES]
                   [--save-interpreter SAVE_INTERPRETER] [--load-interpreter LOAD_
↪INTERPRETER]
                   [--save-clusters SAVE_CLUSTERS] [--load-clusters LOAD_CLUSTERS]
                   [--save-prediction SAVE_PREDICTION] [--epochs EPOCHS] [--batch BATCH]␣
↪[--device DEVICE]
                   [--silent]
                   {sequence,train,cluster,manual,automatic}
```

(continues on next page)

(continued from previous page)

```
DeepCASE: Semi-Supervised Contextual Analysis of Security Events

positional arguments:
  {sequence,train,cluster,manual,automatic}  mode in which to run DeepCASE

optional arguments:
  -h, --help                                 show this help message and exit

Input/Output:
  --csv CSV                                  CSV events file to process
  --txt TXT                                  TXT events file to process
  --events EVENTS                            number of distinct events to handle
↪(default =  auto)

Sequencing:
  --length LENGTH                            sequence LENGTH
↪(default =    10)
  --timeout TIMEOUT                          sequence TIMEOUT (seconds)
↪(default = 86400)
  --save-sequences SAVE_SEQUENCES            path to save sequences
  --load-sequences LOAD_SEQUENCES            path to load sequences

ContextBuilder:
  --hidden HIDDEN                            HIDDEN layers dimension
↪(default =   128)
  --delta DELTA                              label smoothing DELTA
↪(default =   0.1)
  --save-builder SAVE_BUILDER                path to save ContextBuilder
  --load-builder LOAD_BUILDER                path to load ContextBuilder

Interpreter:
  --confidence CONFIDENCE                    minimum required CONFIDENCE
↪(default =   0.2)
  --epsilon EPSILON                          DBSCAN clustering EPSILON
↪(default =   0.1)
  --min_samples MIN_SAMPLES                  DBSCAN clustering MIN_SAMPLES
↪(default =     5)
  --save-interpreter SAVE_INTERPRETER        path to save Interpreter
  --load-interpreter LOAD_INTERPRETER        path to load Interpreter
  --save-clusters SAVE_CLUSTERS              path to CSV file to save clusters
  --load-clusters LOAD_CLUSTERS              path to CSV file to load clusters
  --save-prediction SAVE_PREDICTION          path to CSV file to save prediction

Train:
  --epochs EPOCHS                            number of epochs to train with
↪(default =    10)
  --batch BATCH                              batch size       to train with
↪(default =   128)

Other:
  --device DEVICE                            DEVICE used for computation (cpu|cuda|auto)
```

(continues on next page)

(continued from previous page)

```
↪(default =  auto)
  --silent                                    silence mode, do not print progress
```

### 2.2.1 Examples

Below, we provide various examples of using the command-line tool for running DeepCASE.

#### Event sequencing

Transform `.csv` or `.txt` files into sequences and store them in the file `sequences.save`.

```
python3 deepcase sequence --csv <path/to/file.csv> --save-sequences sequences.save
python3 deepcase sequence --txt <path/to/file.txt> --save-sequences sequences.save
```

#### ContextBuilder

Train the *ContextBuilder* on the input samples loaded from the file `sequences.save` and store the trained ContextBuilder in the file `builder.save`.

```
python3 deepcase train\
     --load-sequences sequences.save\
     --save-builder builder.save
```

#### Interpreter

Run in manual mode where the *Interpreter* clusters the given sequences. We load the sequences from `sequences.save` and the trained ContextBuilder from `builder.save`. We store the interpreter (containing all clusters) to the file `interpreter.save` and the generated clusters to `clusters.csv`. The `clusters.csv` file contains two columns: `cluster` and `label`. We can manually label the individual samples within the cluster by changing the `label` value, note that the rows of the csv file corresond to the loaded sequences. If the sequences itself contained labels, these labels are used for storing in the `csv` file, otherwise, all clusters are assigned a label of `-1`.

```
python3 deepcase cluster\
     --load-sequences sequences.save\
     --load-builder builder.save\
     --save-interpreter interpreter.save\
     --save-clusters clusters.csv
```

#### Manual Mode

Once we (manually) provided a label to each cluster, we can assign these label in manual mode and save the updated interpreter.

---

**Note:** If `--load-clusters` is not specified, DeepCASE will try to use the labels extracted from the `sequences` it processes (see *Preprocessor*). If no `labels` were provided there either, DeepCASE throws an error.

---

```
python3 deepcase manual\
     --load-sequences sequences.save\
     --load-builder builder.save\
     --load-interpreter interpreter.save\
     --load-clusters clusters.csv\
     --save-interpreter interpreter_fitted.save
```

#### (Semi)-automatic Mode

Once we assigned labels to the clusters in the *Interpreter*, we can use DeepCASE to predict labels for new sequences. We save these predicted labels in a file called `prediction.save`.

---

**Note:** If sequences contain `labels` (see *Preprocessor*), we also output a classification report and confusion matrix to show the performance of DeepCASE.

---

```
python3 deepcase automatic\
     --load-sequences sequences.save\
     --load-builder builder.save\
     --load-interpreter interpreter_fitted.save\
     --save-prediction prediction.csv
```

## 2.3 Code integration

To integrate DeepCASE into your own project, you can use it as a standalone module. DeepCASE offers rich functionality that is easy to integrate into other projects. Here we show some simple examples on how to use the DeepCASE package in your own python code. For a complete documentation we refer to the *Reference* guide.

---

**Note:** The code used in this section is also available in the GitHub repository under `examples/example.py`.

---

### 2.3.1 Import

To import components from DeepCASE simply use the following format

```
from deepcase(.<module>) import <Object>
```

For example, the following code imports the *Preprocessor*, *ContextBuilder*, and *Interpreter*.

```
from deepcase.preprocessing   import Preprocessor
from deepcase.context_builder import ContextBuilder
from deepcase.interpreter     import Interpreter
```

### 2.3.2 Loading data

DeepCASE can load sequences from `.csv` and specifically formatted `.txt` files (see *Preprocessor* class).

```
# Create preprocessor
preprocessor = Preprocessor(
    length  = 10,    # 10 events in context
    timeout = 86400, # Ignore events older than 1 day (60*60*24 = 86400 seconds)
)

# Load data from file
context, events, labels, mapping = preprocessor.csv('data/example.csv')
```

In case no labels were explicitly provided as an argument, and no labels could be extracted from the file, we may set labels for each sequence manually. Note that we assign the labels as a numpy array, which requires importing numpy using `import numpy as np`.

```
# In case no labels are provided, set labels to -1
if label is None:
    labels = np.full(events.shape[0], -1, dtype=int)
```

By default, the Tensors returned by the *Preprocessor* are set to the `cpu` device. If you have a system that supports `cuda` Tensors you can cast the Tensors to cuda using the following code. Note that the check in this code requires you to import PyTorch using `import torch`.

```
# Cast to cuda if available
if torch.cuda.is_available():
    events  = events .to('cuda')
    context = context.to('cuda')
```

#### Splitting data

Once we have loaded the data, we will split it into train and test data. This step is not necessarily required, depending on the setup you use, but we will use the training and test data in the remainder of this example.

```
# Split into train and test sets (20:80) by time - assuming events are ordered␣
↪chronologically
events_train  = events [:events.shape[0]//5 ]
events_test   = events [ events.shape[0]//5:]

context_train = context[:events.shape[0]//5 ]
context_test  = context[ events.shape[0]//5:]

label_train   = label  [:events.shape[0]//5 ]
label_test    = label  [ events.shape[0]//5:]
```

### 2.3.3 ContextBuilder

First we create an instance of DeepCASE's *ContextBuilder* using the following code:

```
# Create ContextBuilder
context_builder = ContextBuilder(
    input_size     = 100,   # Number of input features to expect
    output_size    = 100,   # Same as input size
    hidden_size    = 128,   # Number of nodes in hidden layer, in paper we set this to 128
    max_length     = 10,    # Length of the context, should be same as context in␣
↪Preprocessor
)

# Cast to cuda if available
if torch.cuda.is_available():
    context_builder = context_builder.to('cuda')
```

Once the `context_builder` is created, we train it using the `fit()` method.

```
# Train the ContextBuilder
context_builder.fit(
    X             = context_train,                # Context to train with
    y             = events_train.reshape(-1, 1), # Events to train with, note that these␣
↪should be of shape=(n_events, 1)
    epochs        = 10,                           # Number of epochs to train with
    batch_size    = 128,                          # Number of samples in each training␣
↪batch, in paper this was 128
    learning_rate = 0.01,                         # Learning rate to train with, in paper␣
↪this was 0.01
    verbose       = True,                         # If True, prints progress
)
```

#### I/O methods

We can load and save the ContextBuilder to and from a file using the following code:

```
# Save ContextBuilder to file
context_builder.save('path/to/file.save')
# Load ContextBuilder from file
context_builder = ContextBuilder.load('path/to/file.save')
```

### 2.3.4 Interpreter

Once we fitted the `context_builder`, we create in *Interpreter* instance using the following code:

```
# Create Interpreter
interpreter = Interpreter(
    context_builder = context_builder, # ContextBuilder used to fit data
    features        = 100,             # Number of input features to expect, should be␣
↪same as ContextBuilder
    eps             = 0.1,             # Epsilon value to use for DBSCAN clustering, in␣
```

(continues on next page)

(continued from previous page)

```
↪paper this was 0.1
    min_samples     = 5,               # Minimum number of samples to use for DBSCAN␣
↪clustering, in paper this was 5
    threshold       = 0.2,             # Confidence threshold used for determining if␣
↪attention from the ContextBuilder can be used, in paper this was 0.2
)
```

Once the `interpreter` is created, we can use it to cluster samples using the `cluster()` method.

```
# Cluster samples with the interpreter
clusters = interpreter.cluster(
    X          = context_train,               # Context to train with
    y          = events_train.reshape(-1, 1), # Events to train with, note that these␣
↪should be of shape=(n_events, 1)
    iterations = 100,                         # Number of iterations to use for␣
↪attention query, in paper this was 100
    batch_size = 1024,                        # Batch size to use for attention query,␣
↪used to limit CUDA memory usage
    verbose    = True,                        # If True, prints progress
)
```

#### I/O methods

We can load and save the Interpreter to and from a file using the following code:

```
# Save Interpreter to file
interpreter.save('path/to/file.save')
# Load Interpreter from file
interpreter = Interpreter.load(
    'path/to/file.save',
    context_builder = context_builder, # When loading the Interpreter, make sure it is␣
↪linked to the same ContextBuilder used for training.
)
```

### 2.3.5 Manual Mode

When we have used the Interpreter to cluster samples, we can assign a score to the individual clusters. Assigning a score is done through the `score()` method, however, this method has two requirements for assigning a score:

1. that all sequences used to create clusters are assigned a score.
2. that all sequences in the **same** cluster are assigned the **same** score.

Therefore, to make sure these two conditions hold, we first call the `score_clusters()` method and use the result for the `score()` method.

```
# Compute scores for each cluster based on individual labels per sequence
scores = interpreter.score_clusters(
    scores   = labels_train, # Labels used to compute score (either as loaded by␣
↪Preprocessor, or put your own labels here)
    strategy = "max",        # Strategy to use for scoring (one of "max", "min", "avg")
```

(continues on next page)

(continued from previous page)

```
    NO_SCORE = -1,           # Any sequence with this score will be ignored in the␣
↪strategy.
                             # If assigned a cluster, the sequence will inherit the␣
↪cluster score.
                             # If the sequence is not present in a cluster, it will␣
↪receive a score of NO_SCORE.
)

# Assign scores to clusters in interpreter
# Note that all sequences should be given a score and each sequence in the
# same cluster should have the same score.
interpreter.score(
    scores  = scores, # Scores to assign to sequences
    verbose = True,   # If True, prints progress
)
```

### 2.3.6 Semi-automatic Mode

Once we used the *Interpreter* for clustering and assigned a score to each cluster, we can use the :py:meth`predict()` method to predict labels of new sequences. When no cluster could be matched, the :py:meth`predict()` method gives one of three scores for a cluster:

- `-1`, if the *ContextBuilder* is not confident enough for a prediction.
- `-2`, if the `event` was not in the training dataset.
- `-3`, if the nearest cluster is a larger distance than `epsilon` away from the nearest sequence.

```
# Compute predicted scores
prediction = interpreter.predict(
    X          = context_test,               # Context to predict
    y          = events_test.reshape(-1, 1), # Events to predict, note that these should␣
↪be of shape=(n_events, 1)
    iterations = 100,                        # Number of iterations to use for attention␣
↪query, in paper this was 100
    batch_size = 1024,                       # Batch size to use for attention query,␣
↪used to limit CUDA memory usage
    verbose    = True,                       # If True, prints progress
)
```

CHAPTER

# THREE

# REFERENCE

This is the reference documentation for the classes and methods objects provided by the DeepCASE module. Figure 1 gives an overview of the different components that make up DeepCASE as described in the paper.

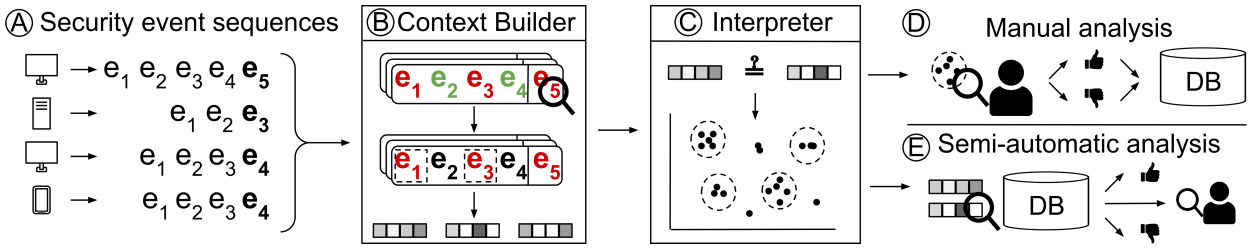


Fig. 1: Figure 1: Overview of DeepCASE.

The ContextBuilder and Interpreter can be used as separate modules within DeepCASE. The DeepCASE class provides a high-level interface that combines all components. The full reference guide can be found here:

## 3.1 Preprocessor

The Preprocessor class provides methods to automatically extract DeepCASE event sequences from various common data formats. To start sequencing, first create the Preprocessor object.

**class** preprocessing.**Preprocessor**(*length*, *timeout*, *NO_EVENT=-1337*)

Preprocessor for loading data from standard data formats.

Preprocessor.**__init__**(*length*, *timeout*, *NO_EVENT=-1337*)

Preprocessor for loading data from standard data formats.

**Parameters**

- **length** (*int*) – Number of events in context.
- **timeout** (*float*) – Maximum time between context event and the actual event in seconds.
- **NO_EVENT** (*int, default=-1337*) – ID of NO_EVENT event, i.e., event returned for context when no event was present. This happens in case of timeout or if an event simply does not have enough preceding context events.

### 3.1.1 Sequencing

All supported formats are wrappers around the sequence method which will produce context and event sequences from given events.

`Preprocessor.sequence(`*data*, *labels=None*, *verbose=False*`)`

Transform pandas DataFrame into DeepCASE sequences.

**Parameters**

- **data** (*pd.DataFrame*) – Dataframe to preprocess.
- **labels** (*int or array-like of shape=(n_samples,), optional*) – If a int is given, label all sequences with given int. If an array-like is given, use the given labels for the data in file. Note: will overwrite any 'label' data in input file.
- **verbose** (*boolean, default=False*) – If True, prints progress in transforming input to sequences.

**Returns**

- **context** (*torch.Tensor of shape=(n_samples, context_length)*) – Context events for each event in events.
- **events** (*torch.Tensor of shape=(n_samples,)*) – Events in data.
- **labels** (*torch.Tensor of shape=(n_samples,)*) – Labels will be None if no labels parameter is given, and if data does not contain any 'labels' column.
- **mapping** (*dict()*) – Mapping from new event_id to original event_id. Sequencing will map all events to a range from 0 to n_events. This is because event IDs may have large values, which is difficult for a one-hot encoding to deal with. Therefore, we map all Event ID values to a new value in that range and provide this mapping to translate back.

### 3.1.2 Formats

**We currently support the following formats:**

- `.csv` files containing a header row that specifies the columns 'timestamp', 'event' and 'machine'.
- `.txt` files containing a line for each machine and a sequence of events (integers) separated by spaces.

Transforming `.csv` files into DeepCASE sequences is the quickest method and is done by the following method call:

`Preprocessor.csv(`*path*, *nrows=None*, *labels=None*, *verbose=False*`)`

Preprocess data from csv file.

---

**Note: Format**: The assumed format of a .csv file is that the first line of the file contains the headers, which should include `timestamp`, `machine`, `event` (and *optionally* `label`). The remaining lines of the .csv file will be interpreted as data.

---

**Parameters**

- **path** (*string*) – Path to input file from which to read data.
- **nrows** (*int, default=None*) – If given, limit the number of rows to read to nrows.
- **labels** (*int or array-like of shape=(n_samples,), optional*) – If a int is given, label all sequences with given int. If an array-like is given, use the given labels for the data in file. Note: will overwrite any 'label' data in input file.

- **verbose** (*boolean, default=False*) – If True, prints progress in transforming input to sequences.

**Returns**

- **events** (*torch.Tensor of shape=(n_samples,)*) – Events in data.
- **context** (*torch.Tensor of shape=(n_samples, context_length)*) – Context events for each event in events.
- **labels** (*torch.Tensor of shape=(n_samples,)*) – Labels will be None if no labels parameter is given, and if data does not contain any 'labels' column.
- **mapping** (*dict()*) – Mapping from new event_id to original event_id. Sequencing will map all events to a range from 0 to n_events. This is because event IDs may have large values, which is difficult for a one-hot encoding to deal with. Therefore, we map all Event ID values to a new value in that range and provide this mapping to translate back.

Transforming `.txt` files into DeepCASE sequences is slower, but still possible using the following method call:

`Preprocessor.text`(*path*, *nrows=None*, *labels=None*, *verbose=False*)

Preprocess data from text file.

---

**Note:** **Format**: The assumed format of a text file is that each line in the text file contains a space-separated sequence of event IDs for a machine. I.e. for *n* machines, there will be *n* lines in the file.

---

**Parameters**

- **path** (*string*) – Path to input file from which to read data.
- **nrows** (*int, default=None*) – If given, limit the number of rows to read to nrows.
- **labels** (*int or array-like of shape=(n_samples,), optional*) – If a int is given, label all sequences with given int. If an array-like is given, use the given labels for the data in file. Note: will overwrite any 'label' data in input file.
- **verbose** (*boolean, default=False*) – If True, prints progress in transforming input to sequences.

**Returns**

- **events** (*torch.Tensor of shape=(n_samples,)*) – Events in data.
- **context** (*torch.Tensor of shape=(n_samples, context_length)*) – Context events for each event in events.
- **labels** (*torch.Tensor of shape=(n_samples,)*) – Labels will be None if no labels parameter is given, and if data does not contain any 'labels' column.
- **mapping** (*dict()*) – Mapping from new event_id to original event_id. Sequencing will map all events to a range from 0 to n_events. This is because event IDs may have large values, which is difficult for a one-hot encoding to deal with. Therefore, we map all Event ID values to a new value in that range and provide this mapping to translate back.

### Future supported formats

**Note:** These formats already have an API entrance, but are currently **NOT** supported.

- `.json` files containing values for 'timestamp', 'event' and 'machine'.
- `.ndjson` where each line contains a json file with keys 'timestamp', 'event' and 'machine'.

`Preprocessor.json`(*path*, *labels=None*, *verbose=False*)

Preprocess data from json file.

**Note:** json preprocessing will become available in a future version.

**Parameters**

- **path** (*string*) – Path to input file from which to read data.
- **labels** (*int or array-like of shape=(n_samples,), optional*) – If a int is given, label all sequences with given int. If an array-like is given, use the given labels for the data in file. Note: will overwrite any 'label' data in input file.
- **verbose** (*boolean, default=False*) – If True, prints progress in transforming input to sequences.

**Returns**

- **events** (*torch.Tensor of shape=(n_samples,)*) – Events in data.
- **context** (*torch.Tensor of shape=(n_samples, context_length)*) – Context events for each event in events.
- **labels** (*torch.Tensor of shape=(n_samples,)*) – Labels will be None if no labels parameter is given, and if data does not contain any 'labels' column.
- **mapping** (*dict()*) – Mapping from new event_id to original event_id. Sequencing will map all events to a range from 0 to n_events. This is because event IDs may have large values, which is difficult for a one-hot encoding to deal with. Therefore, we map all Event ID values to a new value in that range and provide this mapping to translate back.

`Preprocessor.ndjson`(*path*, *labels=None*, *verbose=False*)

Preprocess data from ndjson file.

**Note:** ndjson preprocessing will become available in a future version.

**Parameters**

- **path** (*string*) – Path to input file from which to read data.
- **labels** (*int or array-like of shape=(n_samples,), optional*) – If a int is given, label all sequences with given int. If an array-like is given, use the given labels for the data in file. Note: will overwrite any 'label' data in input file.
- **verbose** (*boolean, default=False*) – If True, prints progress in transforming input to sequences.

**Returns**

- **events** (*torch.Tensor of shape=(n_samples,)*) – Events in data.
- **context** (*torch.Tensor of shape=(n_samples, context_length)*) – Context events for each event in events.
- **labels** (*torch.Tensor of shape=(n_samples,)*) – Labels will be None if no labels parameter is given, and if data does not contain any 'labels' column.
- **mapping** (*dict()*) – Mapping from new event_id to original event_id. Sequencing will map all events to a range from 0 to n_events. This is because event IDs may have large values, which is difficult for a one-hot encoding to deal with. Therefore, we map all Event ID values to a new value in that range and provide this mapping to translate back.

## 3.2 DeepCASE

We provide the DeepCASE class as a wrapper around the *ContextBuilder* and *Interpreter*. The DeepCASE class only implements the `fit()/predict()` methods which requires a priori knowledge of the sequence maliciousness score. If you require a more fine-grained tuning of DeepCASE, e.g., when using the manual labelling mode, we recommend using the individual *ContextBuilder* and *Interpreter* objects as shown in the *Usage*. These individual classes provide a richer API.

**class** module.**DeepCASE**(*features*, *max_length=10*, *hidden_size=128*, *eps=0.1*, *min_samples=5*, *threshold=0.2*)

DeepCASE.**__init__**(*features*, *max_length=10*, *hidden_size=128*, *eps=0.1*, *min_samples=5*, *threshold=0.2*)

Analyse security events with respect to contextual machine behaviour.

---

**Note:** When an Interpreter is trained, it heavily depends on the ContextBuilder used during training. Therefore, we **strongly** suggest **not** to manually change the context_builder attribute, without retraining the interpreter of the DeepCASE object.

---

**Parameters**

- **features** (*int*) – Number of different possible security events.
- **max_length** (*int, default=10*) – Maximum length of context window as number of events.
- **hidden_size** (*int, default=128*) – Size of hidden layer in sequence to sequence prediction. This parameter determines the complexity of the model and its prediction power. However, high values will result in slower training and prediction times.
- **eps** (*float, default=0.1*) – Epsilon used for determining maximum distance between clusters.
- **min_samples** (*int, default=5*) – Minimum number of required samples per cluster.
- **threshold** (*float, default=0.2*) – Minimum required confidence in fingerprint before using it in training clusters.

### 3.2.1 Fit/Predict methods

We provide a `scikit-learn`-like API for DeepCASE to train on sequences with a given maliciousness score and predict the maliciousness score of new sequences.

As DeepCASE is simply a wrapper around the *ContextBuilder* and *Interpreter* objects, the following functionality is equivalent:

***module.DeepCASE.fit()* is equivalent to:**

1. *context_builder.ContextBuilder.fit()*
2. *interpreter.Interpreter.fit()*

***module.DeepCASE.predict()* is equivalent to:**

1. *interpreter.Interpreter.predict()*

***module.DeepCASE.fit_predict()* is equivalent to:**

1. *module.DeepCASE.fit()*
2. *module.DeepCASE.predict()*

#### Fit

The `fit()` method provides an API for directly learning the maliciousness score of sequences. This method combines the `fit()` methods from both the *ContextBuilder* and the *Interpreter*. We note that to use the `fit()` method, scores of sequences should be known a priori. See the *interpreter.Interpreter.fit()* method for an explanation of how these scores are used.

DeepCASE.**fit**(*X*, *y*, *scores*, *epochs=10*, *batch_size=128*, *learning_rate=0.01*, *optimizer=<class 'torch.optim.sgd.SGD'>*, *teach_ratio=0.5*, *iterations=100*, *query_batch_size=1024*, *strategy='max'*, *NO_SCORE=-1*, *verbose=True*)

Fit DeepCASE with given data. This method is provided as a wrapper and is equivalent to calling: - context_builder.fit() and - interpreter.fit() in the given order.

**Parameters**

- **X** (*array-like of type=int and shape=(n_samples, context_size)*) – Input context to train with.
- **y** (*array-like of type=int and shape=(n_samples, n_future_events)*) – Sequences of target events.
- **scores** (*array-like of float, shape=(n_samples,)*) – Scores for each sample in cluster.
- **epochs** (*int, default=10*) – Number of epochs to train with.
- **batch_size** (*int, default=128*) – Batch size to use for training.
- **learning_rate** (*float, default=0.01*) – Learning rate to use for training.
- **optimizer** (*optim.Optimizer, default=torch.optim.SGD*) – Optimizer to use for training.
- **teach_ratio** (*float, default=0.5*) – Ratio of sequences to train including labels.
- **iterations** (*int, default=100*) – Number of iterations for query.
- **query_batch_size** (*int, default=1024*) – Size of batch for query.

- **strategy** (*string (max|min|avg), default=max*) – Strategy to use for computing scores per cluster based on scores of individual events. Currently available options are: - max: Use maximum score of any individual event in a cluster. - min: Use minimum score of any individual event in a cluster. - avg: Use average score of any individual event in a cluster.
- **NO_SCORE** (*float, default=-1*) – Score to indicate that no score was given to a sample and that the value should be ignored for computing the cluster score. The NO_SCORE value will also be given to samples that do not belong to a cluster.
- **verbose** (*boolean, default=True*) – If True, prints progress.

**Returns**
**self** – Returns self.

**Return type**
self

### Predict

When DeepCASE is trained, we can use DeepCASE to predict the score of new sequences. To this end, we provide the `predict()` function which takes `context` and `events` as input and outputs the labels of corresponding predicted clusters. If no sequence could be matched, one of the following scores will be given:

- `-1`: Not confident enough for prediction
- `-2`: Label not in training
- `-3`: Closest cluster > epsilon

---

**Note:** This method is a wrapper around the *interpreter.Interpreter.predict()* method.

---

DeepCASE.**predict**(*X*, *y*, *iterations=100*, *batch_size=1024*, *verbose=False*)

Predict maliciousness of context samples.

**Parameters**

- **X** (*torch.Tensor of shape=(n_samples, seq_length)*) – Input context for which to predict maliciousness.
- **y** (*torch.Tensor of shape=(n_samples, 1)*) – Events for which to predict maliciousness.
- **iterations** (*int, default=100*) – Iterations used for optimization.
- **batch_size** (*int, default=1024*) – Batch size used for optimization.
- **verbose** (*boolean, default=False*) – If True, print progress.

**Returns**

**result** – Predicted maliciousness score. Positive scores are maliciousness scores. A score of 0 means we found a match that was not malicious. Special cases:

- -1: Not confident enough for prediction
- -2: Label not in training
- -3: Closest cluster > epsilon

**Return type**
np.array of shape=(n_samples,)

### Fit_predict

Similar to the `scikit-learn` API, the `fit_predict()` method performs the `fit()` and `predict()` functions in sequence on the same data.

DeepCASE.**fit_predict**(*X*, *y*, *scores*, *epochs=10*, *batch_size=128*, *learning_rate=0.01*, *optimizer=<class 'torch.optim.sgd.SGD'>*, *teach_ratio=0.5*, *iterations=100*, *query_batch_size=1024*, *strategy='max'*, *NO_SCORE=-1*, *verbose=True*)

Fit DeepCASE with given data and predict that same data. This method is provided as a wrapper and is equivalent to calling: - self.fit() and - self.predict() in the given order.

**Parameters**

- **X** (*array-like of type=int and shape=(n_samples, context_size)*) – Input context to train with.
- **y** (*array-like of type=int and shape=(n_samples, n_future_events)*) – Sequences of target events.
- **scores** (*array-like of float, shape=(n_samples,)*) – Scores for each sample in cluster.
- **epochs** (*int, default=10*) – Number of epochs to train with.
- **batch_size** (*int, default=128*) – Batch size to use for training.
- **learning_rate** (*float, default=0.01*) – Learning rate to use for training.
- **optimizer** (*optim.Optimizer, default=torch.optim.SGD*) – Optimizer to use for training.
- **teach_ratio** (*float, default=0.5*) – Ratio of sequences to train including labels.
- **iterations** (*int, default=100*) – Number of iterations for query.
- **query_batch_size** (*int, default=1024*) – Size of batch for query.
- **strategy** (*string (max|min|avg), default=max*) – Strategy to use for computing scores per cluster based on scores of individual events. Currently available options are: - max: Use maximum score of any individual event in a cluster. - min: Use minimum score of any individual event in a cluster. - avg: Use average score of any individual event in a cluster.
- **NO_SCORE** (*float, default=-1*) – Score to indicate that no score was given to a sample and that the value should be ignored for computing the cluster score. The NO_SCORE value will also be given to samples that do not belong to a cluster.
- **verbose** (*boolean, default=True*) – If True, prints progress.

**Returns**

**result** – Predicted maliciousness score. Positive scores are maliciousness scores. A score of 0 means we found a match that was not malicious. Special cases:

- -1: Not confident enough for prediction
- -2: Label not in training
- -3: Closest cluster > epsilon

**Return type**
np.array of shape=(n_samples,)

### 3.2.2 I/O methods

DeepCASE can be saved and loaded from files using the following methods. Please note that the *module.DeepCASE.load()* method is a `classmethod` and must be called statically.

`DeepCASE.`**`save`**(*outfile*)

Save DeepCASE model to output file.

**Parameters**
**outfile** (*string*) – Path to output file in which to store DeepCASE model.

**`classmethod`** `DeepCASE.`**`load`**(*infile*, *device=None*)

Load DeepCASE model from input file.

**Parameters**

- **infile** (*string*) – Path to input file from which to load DeepCASE model.
- **device** (*string, optional*) – If given, cast DeepCASE automatically to device.

**Example:**

```
from deepcase import DeepCASE
deepcase = DeepCASE.load('<path_to_saved_deepcase_object>')
deepcase.save('<path_to_save_deepcase_object>')
```

## 3.3 ContextBuilder

The ContextBuilder is a pytorch neural network architecture that supports scikit-learn like fit and predict methods. The ContextBuilder is used to analyse sequences of security events and can be used to produce confidence levels for predicting future events as well as investigating the attention used to make predictions.

**`class`** `context_builder.`**`ContextBuilder`**(**args: Any*, ***kwargs: Any*)

The *context_builder.ContextBuilder.__init__()* constructs a new instance of the ContextBuilder. For loading a pre-trained ContextBuilder from files, we refer to *context_builder.ContextBuilder.load()*.

`ContextBuilder.`**`__init__`**(*input_size*, *output_size*, *hidden_size=128*, *num_layers=1*, *max_length=10*, *bidirectional=False*, *LSTM=False*)

ContextBuilder that learns to interpret context from security events. Based on an attention-based Encoder-Decoder architecture.

**Parameters**

- **input_size** (*int*) – Size of input vocabulary, i.e. possible distinct input items
- **output_size** (*int*) – Size of output vocabulary, i.e. possible distinct output items
- **hidden_size** (*int, default=128*) – Size of hidden layer in sequence to sequence prediction. This parameter determines the complexity of the model and its prediction power. However, high values will result in slower training and prediction times
- **num_layers** (*int, default=1*) – Number of recurrent layers to use
- **max_length** (*int, default=10*) – Maximum lenght of input sequence to expect

- **bidirectional** (*boolean, default=False*) – If True, use a bidirectional encoder and decoder
- **LSTM** (*boolean, default=False*) – If True, use an LSTM as a recurrent unit instead of GRU

### 3.3.1 Overview

The ContextBuilder is an instance of the pytorch nn.Module class. This means that it implements the functionality of a complete neural network. Figure 1 shows the overview of the neural network architecture of the ContextBuilder.

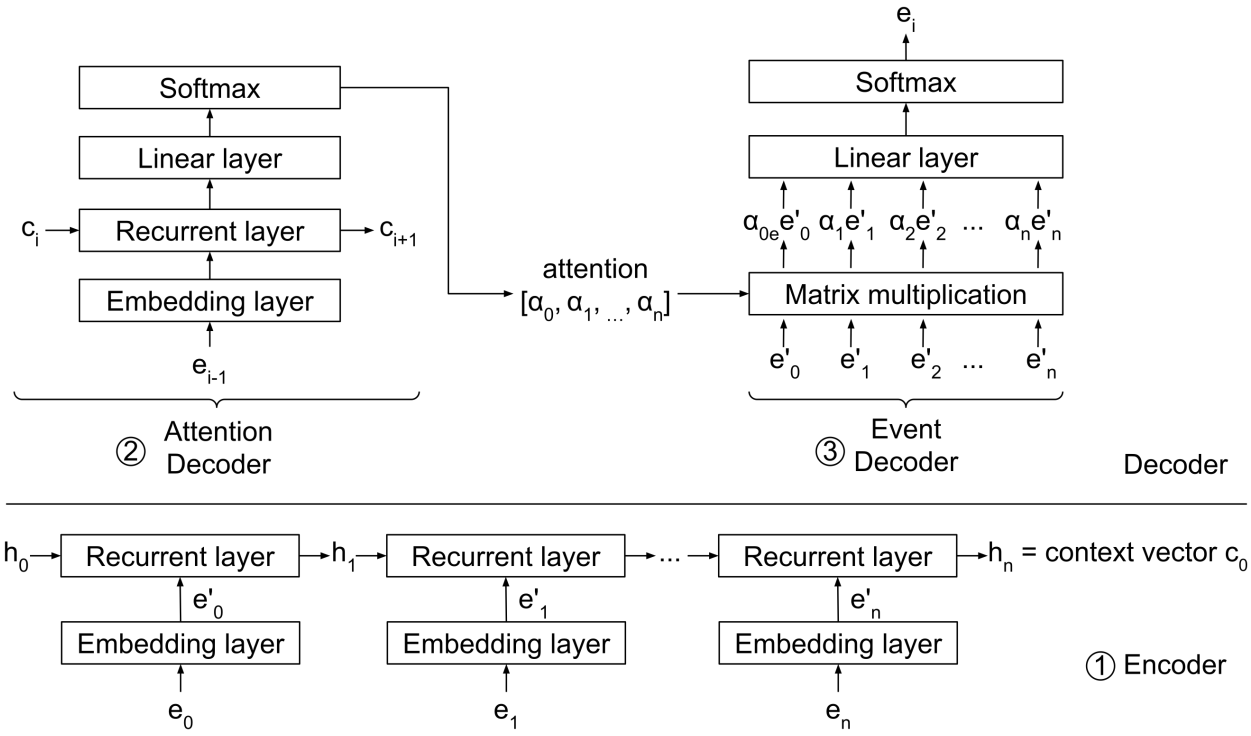


Fig. 2: Figure 1: ContextBuilder architecture.

The components of the neural network are implemented by the following classes:

#### DecoderAttention

The DecoderAttention is an instance of the pytorch nn.Module class. This part of the neural network takes the `context_vector` from the *Encoder* and produces the `attention_vector`.

**class** context_builder.decoders.**DecoderAttention**(**args: Any*, ***kwargs: Any*)

DecoderAttention.**__init__**(*embedding*, *context_size*, *attention_size*, *num_layers=1*, *dropout=0.1*, *bidirectional=False*, *LSTM=False*)

Attention decoder for retrieving attention from context vector.

**Parameters**

- **embedding** (*nn.Embedding*) – Embedding layer to use.
- **context_size** (*int*) – Size of context to expect as input.

- **attention_size** (*int*) – Size of attention vector.
- **num_layers** (*int, default=1*) – Number of recurrent layers to use.
- **dropout** (*float, default=0.1*) – Default dropout rate to use.
- **bidirectional** (*boolean, default=False*) – If True, use bidirectional recurrent layer.
- **LSTM** (*boolean, default=False*) – If True, use LSTM instead of GRU.

### Forward

The `forward()` function takes the `context_vector` and produces the `attention_vector`. This method is also called from the `__call__` method, i.e. when the object is called directly.

`DecoderAttention.`**forward**(*context_vector*, *previous_input=None*)

Compute attention based on input and hidden state.

**Parameters**

- **X** (*torch.Tensor of shape=(n_samples, embedding_dim)*) – Input from which to compute attention
- **hidden** (*torch.Tensor of shape=(n_samples, hidden_size)*) – Context vector from which to compute attention

**Returns**

- **attention** (*torch.Tensor of shape=(n_samples, context_size)*) – Computed attention
- **context_vector** (*torch.Tensor of shape=(n_samples, hidden_size)*) – Updated context vector

## DecoderEvent

The DecoderEvent is an instance of the pytorch nn.Module class. This part of the neural network takes the encoded inputs from the *Encoder* and `attention_vector` from the *DecoderAttention* and predicts the next event in the sequence.

**class** context_builder.decoders.**DecoderEvent**(**args: Any*, ***kwargs: Any*)

`DecoderEvent.`**__init__**(*input_size*, *output_size*, *dropout=0.1*)

### Forward

The `forward()` function takes the `attention_vector` and encoded inputs and predicts the next event in the sequence. This method is also called from the `__call__` method, i.e. when the object is called directly.

`DecoderEvent.`**forward**(*X*, *attention*)

Decode X with given attention.

**Parameters**

- **X** (*torch.Tensor of shape=(n_samples, context_size, hidden_size)*) – Input samples on which to apply attention.
- **attention** (*torch.Tensor of shape=(n_samples, context_size)*) – Attention to use for decoding step

**Returns**
**output** – Decoded output

**Return type**
torch.Tensor of shape=(n_samples, output_size)

### EmbeddingOneHot

The EmbeddingOneHot is an instance of the pytorch nn.Module class. This part of the neural network takes categorical samples and produces a one-hot encoded version of the input. This module is used in the from the *Encoder*.

`class context_builder.embedding.EmbeddingOneHot(*args: Any, **kwargs: Any)`
Embedder using simple one hot encoding.

`EmbeddingOneHot.__init__(input_size)`
Embedder using simple one hot encoding.

**Parameters**
**input_size** (*int*) – Maximum number of inputs to one_hot encode

### Forward

The `forward()` function takes the input values and produces the one-hot encoded equivalent. This method is also called from the `__call__` method, i.e. when the object is called directly.

`EmbeddingOneHot.forward(X)`
Create one-hot encoding of input

**Parameters**
**X** (*torch.Tensor of shape=(n_samples,)*) – Input to encode.

**Returns**
**result** – One-hot encoded version of input

**Return type**
torch.Tensor of shape=(n_samples, input_size)

### Encoder

The Encoder is an instance of the pytorch nn.Module class. This part of the neural network takes the input sequences and produces the embedded outputs as well as the `context_vector` used by the *DecoderAttention* and *DecoderEvent*.

`class context_builder.encoders.Encoder(*args: Any, **kwargs: Any)`

`Encoder.__init__(embedding, hidden_size, num_layers=1, bidirectional=False, LSTM=False)`
Encoder part for encoding sequences.

**Parameters**

- **embedding** (*nn.Embedding*) – Embedding layer to use
- **hidden_size** (*int*) – Size of hidden dimension
- **num_layers** (*int, default=1*) – Number of recurrent layers to use
- **bidirectional** (*boolean, default=False*) – If True, use bidirectional recurrent layer
- **LSTM** (*boolean, default=False*) – If True, use LSTM instead of GRU

### Forward

The `forward()` function takes the input sequences and produces the embedded outputs as well as the `context_vector`. This method is also called from the `__call__` method, i.e. when the object is called directly.

`Encoder.`**`forward`**(*input*, *hidden=None*)

Encode data

**Parameters**

- **input** (*torch.Tensor*) – Tensor to use as input
- **hidden** (*torch.Tensor*) – Tensor to use as hidden input (for storing sequences)

**Returns**

- **output** (*torch.Tensor*) – Output tensor
- **hidden** (*torch.Tensor*) – Hidden state to supply to next input

These additional classes implement methods for training the ContextBuilder:

### LabelSmoothing

The LabelSmoothing is an instance of the pytorch nn.Module class. The LabelSmoothing class implements an adapted version of the label smoothing loss function [1].

**class** `context_builder.loss.`**`LabelSmoothing`**(**args: Any*, ***kwargs: Any*)

`LabelSmoothing.`**`__init__`**(*size*, *smoothing=0.0*)

Implements label smoothing loss function

**Parameters**

- **size** (*int*) – Number of labels
- **smoothing** (*float, default=0.0*) – Smoothing factor to apply

### Forward

The `forward()` function takes actual output `x` and `target` output and computes the loss. This method is also called from the `__call__` method, i.e. when the object is called directly.

`LabelSmoothing.`**`forward`**(*x*, *target*, *weights=None*, *attention=None*)

Forward data

### Reference

[1] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR). [PDF]

### VarAdam

The VarAdam is an instance of the pytorch nn.Module class. The VarAdam class implements an adapted version of the Adam optimizer as introduced in [1].

**class** context_builder.optimizer.**VarAdam**(*model*, *factor=1*, *warmup=4000*, *optimizer=<class 'torch.optim.adam.Adam'>*, *lr=0*, *betas=(0.9, 0.98)*, *eps=1e-09*)

Adam optimizer with variable learning rate.

VarAdam.**__init__**(*model*, *factor=1*, *warmup=4000*, *optimizer=<class 'torch.optim.adam.Adam'>*, *lr=0*, *betas=(0.9, 0.98)*, *eps=1e-09*)

### Update

The following functions update the optimizer with a given number of steps.

VarAdam.**step**()

Update parameters and rate

VarAdam.**rate**(*step=None*)

Compute current learning rate

**Parameters**

**step** (*int, (optional)*) – Number of steps to take

### Reference

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In Advances in neural information processing systems (NIPS). [PDF]

The ContextBuilder itself combines all underlying classes in its `forward()` function. This takes the input of the network and produces the output by passing the data through all internal layers. This method is also called from the `__call__` method, i.e. when the object is called directly.

ContextBuilder.**forward**(*X*, *y=None*, *steps=1*, *teach_ratio=0.5*)

Forwards data through ContextBuilder.

**Parameters**

- **X** (*torch.Tensor of shape=(n_samples, seq_len)*) – Tensor of input events to forward.
- **y** (*torch.Tensor of shape=(n_samples, steps), optional*) – If given, use value of y as next input with probability teach_ratio.
- **steps** (*int, default=1*) – Number of steps to predict in the future.
- **teach_ratio** (*float, default=0.5*) – Ratio of sequences to train that use given labels Y. The remaining part will be trained using the predicted values.

**Returns**

- **confidence** (*torch.Tensor of shape=(n_samples, steps, output_size)*) – The confidence level of each output event.
- **attention** (*torch.Tensor of shape=(n_samples, steps, seq_len)*) – Attention corrsponding to X given as (batch, out_seq, in_seq).

### 3.3.2 Fit/Predict methods

We provide the ContextBuilder as a classifier to learn sequences and predict the output values. To this end, we implement scikit-learn like fit and predict methods for training and predicting with the network.

#### Fit

The `fit()` method automatically trains the network using the given input data `X` and `y`, while allowing the user to set various learning variables such as the number of `epochs` to train with, the `batch_size` and `learning_rate`. Please see the method below for all available options.

ContextBuilder.**fit**(*X*, *y*, *epochs=10*, *batch_size=128*, *learning_rate=0.01*, *optimizer=torch.optim.SGD*, *teach_ratio=0.5*, *verbose=True*)

Fit the sequence predictor with labelled data

**Parameters**

- **X** (*array-like of type=int and shape=(n_samples, context_size)*) – Input context to train with.
- **y** (*array-like of type=int and shape=(n_samples, n_future_events)*) – Sequences of target events.
- **epochs** (*int, default=10*) – Number of epochs to train with.
- **batch_size** (*int, default=128*) – Batch size to use for training.
- **learning_rate** (*float, default=0.01*) – Learning rate to use for training.
- **optimizer** (*optim.Optimizer, default=torch.optim.SGD*) – Optimizer to use for training.
- **teach_ratio** (*float, default=0.5*) – Ratio of sequences to train including labels.
- **verbose** (*boolean, default=True*) – If True, prints progress.

**Returns**
**self** – Returns self

**Return type**
self

#### Predict

The `predict()` method outputs the `confidence` values for predictions of future events in the sequence and `attention` values used for each prediction. The `steps` parameter specifies the number of predictions to make into the future, e.g. `steps=2` will give the next 2 predicted events to occur.

ContextBuilder.**predict**(*X*, *y=None*, *steps=1*)

Predict the next elements in sequence.

**Parameters**

- **X** (*torch.Tensor*) – Tensor of input sequences
- **y** (*ignored*) –
- **steps** (*int, default=1*) – Number of steps to predict into the future

**Returns**

- **confidence** (*torch.Tensor of shape=(n_samples, seq_len, output_size)*) – The confidence level of each output
- **attention** (*torch.Tensor of shape=(n_samples, input_length)*) – Attention corrsponding to X given as (batch, out_seq, seq_len)

#### Fit_predict

The `fit_predict()` method performs the `fit()` and `predict()` functions in sequence on the same data.

`ContextBuilder.`**`fit_predict`**(*X*, *y*, *epochs=10*, *batch_size=128*, *learning_rate=0.01*, *optimizer=torch.optim.SGD*, *teach_ratio=0.5*, *verbose=True*)

Fit the sequence predictor with labelled data

**Parameters**

- **X** (*torch.Tensor*) – Tensor of input sequences
- **y** (*torch.Tensor*) – Tensor of output sequences
- **epochs** (*int, default=10*) – Number of epochs to train with
- **batch_size** (*int, default=128*) – Batch size to use for training
- **learning_rate** (*float, default=0.01*) – Learning rate to use for training
- **optimizer** (*optim.Optimizer, default=torch.optim.SGD*) – Optimizer to use for training
- **teach_ratio** (*float, default=0.5*) – Ratio of sequences to train including labels
- **verbose** (*boolean, default=True*) – If True, prints progress

**Returns**
**result** – Predictions corresponding to X

**Return type**
torch.Tensor

### 3.3.3 Query

The `query()` method implements the *attention query* from the DeepCASE paper. This method tries to find the optimal attention vector for a given input, in order to predict the known output.

`ContextBuilder.`**`query`**(*X*, *y*, *iterations=0*, *batch_size=1024*, *ignore=None*, *return_optimization=None*, *verbose=True*)

Query the network to get optimal attention vector.

**Parameters**

- **X** (*array-like of type=int and shape=(n_samples, context_size)*) – Input context of events, same as input to fit and predict
- **y** (*array-like of type=int and shape=(n_samples,)*) – Observed event
- **iterations** (*int, default=0*) – Number of iterations to perform for optimization of actual event
- **batch_size** (*int, default=1024*) – Batch size of items to optimize
- **ignore** (*int, optional*) – If given ignore this index as attention

- **return_optimization** (*float, optional*) – If given, returns number of items with confidence level larger than given parameter. E.g. return_optimization=0.2 will also return two boolean tensors for elements with a confidence >= 0.2 before optimization and after optimization.
- **verbose** (*boolean, default=True*) – If True, print progress

**Returns**

- **confidence** (*torch.Tensor of shape=(n_samples, output_size)*) – Confidence of each prediction given new attention
- **attention** (*torch.Tensor of shape=(n_samples, context_size)*) – Importance of each input with respect to output
- **inverse** (*torch.Tensor of shape=(n_samples,)*) – Inverse is returned to reconstruct the original array
- **confidence_orig** (*torch.Tensor of shape=(n_samples,)*) – Only returned if return_optimization != None Boolean array of items >= threshold before optimization
- **confidence_optim** (*torch.Tensor of shape=(n_samples,)*) – Only returned if return_optimization != None Boolean array of items >= threshold after optimization

### 3.3.4 I/O methods

The ContextBuilder can be saved and loaded from files using the following methods. Please note that the *context_builder.ContextBuilder.load()* method is a `classmethod` and must be called statically.

`ContextBuilder.`**`save`**(*outfile*)

Save model to output file.

**Parameters**

**outfile** (*string*) – File to output model.

**classmethod** `ContextBuilder.`**`load`**(*infile*, *device=None*)

Load model from input file.

**Parameters**

**infile** (*string*) – File from which to load model.

**Example:**

```
from deepcase.context_builder import ContextBuilder
builder = ContextBuilder.load('<path_to_saved_builder>')
builder.save('<path_to_save_builder>')
```

## 3.4 Interpreter

The Interpreter takes input sequences (`context` and `events`) and clusters them. In order to do this clustering, it uses the `attention` values from the *ContextBuilder* after applying the attention query. Besides clustering, the Interpreter also offers methods to assign scores for Manual analysis, and to predict the scores of unknown sequences for Semi-Automatic analysis.

**class** `interpreter.`**`Interpreter`**(*context_builder*, *features*, *eps=0.1*, *min_samples=5*, *threshold=0.2*)

Interpreter.__init__(*context_builder*, *features*, *eps=0.1*, *min_samples=5*, *threshold=0.2*)

Interpreter for a given ContextBuilder.

**Parameters**

- **context_builder** (ContextBuilder) – ContextBuilder to interpret.
- **features** (*int*) – Number of different possible security events.
- **eps** (*float, default=0.1*) – Epsilon used for determining maximum distance between clusters.
- **min_samples** (*int, default=5*) – Minimum number of required samples per cluster.
- **threshold** (*float, default=0.2*) – Minimum required confidence of ContextBuilder before using a context in training clusters.

### 3.4.1 Fit/Predict methods

We provide a `scikit-learn`-like API for the Interpreter as a classifier to labels for sequences in the form of clusters and predict the labels of new sequences. To this end, we implement scikit-learn like fit and predict methods for training and predicting with the network.

#### Fit

The `fit()` method provides an API for directly learning the maliciousness score of sequences. This method combines Interpreter's Clustering and Manual Mode for sequences where the labels are known a priori. To this end, it calls the `cluster()`, `score_clusters()`, and `score()` methods in sequence. When the labels for sequences are not known in advance, the Interpreter offers the functionality to first cluster sequences, and then manually inspect clusters for labelling as described in the paper. For this functionality, we refer to the methods:

- *interpreter.Interpreter.cluster()*
- *interpreter.Interpreter.score_clusters()*
- *interpreter.Interpreter.score()*

Interpreter.fit(*X*, *y*, *scores*, *iterations=100*, *batch_size=1024*, *strategy='max'*, *NO_SCORE=-1*, *verbose=False*)

Fit the Interpreter by performing clustering and assigning scores.

**Fit function is a wrapper that calls the following methods:**

1. Interpreter.cluster
2. Interpreter.score_clusters
3. Interpreter.score

**Parameters**

- **X** (*torch.Tensor of shape=(n_samples, seq_length)*) – Input context to cluster.
- **y** (*torch.Tensor of shape=(n_samples, 1)*) – Events to cluster.
- **scores** (*array-like of float, shape=(n_samples,)*) – Scores for each sample in cluster.
- **iterations** (*int, default=100*) – Number of iterations for query.
- **batch_size** (*int, default=1024*) – Size of batch for query.

- **strategy** (*string (max|min|avg), default=max*) – Strategy to use for computing scores per cluster based on scores of individual events. Currently available options are: - max: Use maximum score of any individual event in a cluster. - min: Use minimum score of any individual event in a cluster. - avg: Use average score of any individual event in a cluster.
- **NO_SCORE** (*float, default=-1*) – Score to indicate that no score was given to a sample and that the value should be ignored for computing the cluster score. The NO_SCORE value will also be given to samples that do not belong to a cluster.
- **verbose** (*boolean, default=False*) – If True, prints achieved speedup of clustering algorithm.

**Returns**
**self** – Returns self

**Return type**
self

### Predict

When the Interpreter is trained using either the `fit()` method, or by using the individual `cluster()` and `score()` methods, we can use the Interpreter in (semi-)automatic mode. To this end, we provide the `predict()` function which takes `context` and `events` as input and outputs the labels of corresponding predicted clusters. If no sequence could be matched, one of the following scores will be given:

- `-1`: Not confident enough for prediction
- `-2`: Label not in training
- `-3`: Closest cluster > epsilon

---

**Note:** To use the `predict()` method, make sure that **both** the `cluster()` and `score()` methods have been called to cluster samples and assign a score to those samples.

---

`Interpreter.`**predict**(*X*, *y*, *iterations=100*, *batch_size=1024*, *verbose=False*)

Predict maliciousness of context samples.

**Parameters**

- **X** (*torch.Tensor of shape=(n_samples, seq_length)*) – Input context for which to predict maliciousness.
- **y** (*torch.Tensor of shape=(n_samples, 1)*) – Events for which to predict maliciousness.
- **iterations** (*int, default=100*) – Iterations used for optimization.
- **batch_size** (*int, default=1024*) – Batch size used for optimization.
- **verbose** (*boolean, default=False*) – If True, print progress.

**Returns**

**result** – Predicted maliciousness score. Positive scores are maliciousness scores. A score of 0 means we found a match that was not malicious. Special cases:

- -1: Not confident enough for prediction
- -2: Label not in training

- -3: Closest cluster > epsilon

**Return type**
np.array of shape=(n_samples,)

### Fit_predict

Similar to the `scikit-learn` API, the `fit_predict()` method performs the `fit()` and `predict()` functions in sequence on the same data.

`Interpreter.fit_predict`(*X*, *y*, *scores*, *iterations=100*, *batch_size=1024*, *strategy='max'*, *NO_SCORE=-1*, *verbose=False*)

Fit Interpreter with samples and labels and return the predictions of the same samples after running them through the Interpreter.

**Parameters**

- **X** (*torch.Tensor of shape=(n_samples, seq_length)*) – Input context to cluster.
- **y** (*torch.Tensor of shape=(n_samples, 1)*) – Events to cluster.
- **scores** (*array-like of float, shape=(n_samples,)*) – Scores for each sample in cluster.
- **iterations** (*int, default=100*) – Number of iterations for query.
- **batch_size** (*int, default=1024*) – Size of batch for query.
- **strategy** (*string (max|min|avg), default=max*) – Strategy to use for computing scores per cluster based on scores of individual events. Currently available options are: - max: Use maximum score of any individual event in a cluster. - min: Use minimum score of any individual event in a cluster. - avg: Use average score of any individual event in a cluster.
- **NO_SCORE** (*float, default=-1*) – Score to indicate that no score was given to a sample and that the value should be ignored for computing the cluster score. The NO_SCORE value will also be given to samples that do not belong to a cluster.
- **verbose** (*boolean, default=False*) – If True, prints achieved speedup of clustering algorithm.

**Returns**

**result** – Predicted maliciousness score. Positive scores are maliciousness scores. A score of 0 means we found a match that was not malicious. Special cases:

- -1: Not confident enough for prediction
- -2: Label not in training
- -3: Closest cluster > epsilon

**Return type**
np.array of shape=(n_samples,)

### 3.4.2 Clustering

The main task of the Interpreter is to cluster events. To this end, the `cluster()` method automatically clusters sequences from the `context` and `events` that have been given as input.

`Interpreter.cluster`(*X*, *y*, *iterations=100*, *batch_size=1024*, *verbose=False*)

Cluster contexts in X for same output event y.

**Parameters**

- **X** (*torch.Tensor of shape=(n_samples, seq_length)*) – Input context to cluster.
- **y** (*torch.Tensor of shape=(n_samples, 1)*) – Events to cluster.
- **iterations** (*int, default=100*) – Number of iterations for query.
- **batch_size** (*int, default=1024*) – Size of batch for query.
- **verbose** (*boolean, default=False*) – If True, prints achieved speedup of clustering algorithm.

**Returns**

**clusters** – Clusters per input sample.

**Return type**

np.array of shape=(n_samples,)

#### Auxiliary cluster methods

To create clusters, we recall from the DeepCASE paper Section III-C1 we apply `attention_query()` to the result from the ContxtBuilder. Using the obtained attention we create a vector (Section III-B2) representing the context using the method `vectorize()`. Both steps are combined in the method `attended_context()`.

`Interpreter.attended_context`(*X*, *y*, *threshold=0.2*, *iterations=100*, *batch_size=1024*, *verbose=False*)

Get vectors representing context after the attention query.

**Parameters**

- **X** (*torch.Tensor of shape=(n_samples, seq_length)*) – Input context to cluster.
- **y** (*torch.Tensor of shape=(n_samples, 1)*) – Events to cluster.
- **threshold** (*float, default=0.2*) – Minimum confidence required for creating a vector representing the context.
- **iterations** (*int, default=100*) – Number of iterations for query.
- **batch_size** (*int, default=1024*) – Size of batch for query.
- **verbose** (*boolean, default=False*) – If True, prints achieved speedup of clustering algorithm.

**Returns**

- **vectors** (*scipy.sparse.csc_matrix of shape=(n_samples, dim_vector)*) – Sparse vectors representing each context with a confidence >= threshold.
- **mask** (*np.array of shape=(n_samples,)*) – Boolean array of masked vectors. True where input has confidence >= threshold, False otherwise.

Interpreter.**attention_query**(*X*, *y*, *iterations=100*, *batch_size=1024*, *verbose=False*)

Compute optimal attention for given context X.

**Parameters**

- **X** (*array-like of type=int and shape=(n_samples, context_size)*) – Input context of events, same as input to fit and predict.
- **y** (*array-like of type=int and shape=(n_samples,)*) – Observed event.
- **iterations** (*int, default=100*) – Number of iterations to perform for optimization of actual event.
- **batch_size** (*int, default=1024*) – Batch size of items to optimize.
- **verbose** (*boolean, default=False*) – If True, prints progress.

**Returns**

- **confidence** (*torch.Tensor of shape=(n_samples,)*) – Resulting confidence levels in y.
- **attention** (*torch.Tensor of shape=(n_samples,)*) – Optimal attention for predicting event y.

Interpreter.**vectorize**(*X*, *attention*, *size*)

Compute the total attention for each event in the context. The resulting vector can be used to compare sequences.

**Parameters**

- **X** (*torch.Tensor of shape=(n_samples, sequence_length, input_dim)*) – Context events to vectorize.
- **attention** (*torch.Tensor of shape=(n_samples, sequence_length)*) – Attention for each event.
- **size** (*int*) – Total number of possible events, determines the vector size.

**Returns**

**result** – Sparse vector representing each context.

**Return type**

scipy.sparse.csc_matrix of shape=(n_samples, n)

### 3.4.3 Manual mode

Once events have been clusters, we can assign a label or score to each sequence. This way, we manually label the clusters and prepare the Interpreter object for (semi-)automatically predicting labels for new sequences. To assign labels to clusters, we provide the `score()` method.

---

**Note:**

**The `score()` function requires:**

1. that all sequences used to create clusters are assigned a score.
2. that all sequences in the **same** cluster are assigned the **same** score.

If you do not have labels for all clusters or different labels within the same cluster, the *interpreter.Interpreter.score_clusters()* method prepares scores such that both conditions are satisfied.

---

Interpreter.**score**(*scores*, *verbose=False*)

Assigns score to clustered samples.

**Parameters**

- **scores** (*array-like of shape=(n_samples,)*) – Scores of individual samples.
- **verbose** (*boolean, default=False*) – If True, print progress.

**Returns**

**self** – Returns self

**Return type**

self

### Auxiliary manual methods

**As mentioned above, the `score()` function has two requirements:**

1. that all sequences used to create clusters are assigned a score.
2. that all sequences in the **same** cluster are assigned the **same** score.

We provide the `score_clusters()` method for the situations where you only have labels for some sequences, or if the labels for sequences within the same cluster are not necessarily equal. This method will apply a given `strategy` for equalizing the labels per cluster. Additionally, unlabelled clusters will all be labeled using a given `NO_SCORE` score.

Interpreter.**score_clusters**(*scores*, *strategy='max'*, *NO_SCORE=-1*)

Compute score per cluster based on individual scores and given strategy.

**Parameters**

- **scores** (*array-like of float, shape=(n_samples,)*) – Scores for each sample in cluster.
- **strategy** (*string (max|min|avg), default=max*) – Strategy to use for computing scores per cluster based on scores of individual events. Currently available options are: - max: Use maximum score of any individual event in a cluster. - min: Use minimum score of any individual event in a cluster. - avg: Use average score of any individual event in a cluster.
- **NO_SCORE** (*float, default=-1*) – Score to indicate that no score was given to a sample and that the value should be ignored for computing the cluster score. The NO_SCORE value will also be given to samples that do not belong to a cluster.

**Returns**

**scores** – Scores for individual sequences computed using clustering strategy. All datapoints within a cluster are guaranteed to have the same score.

**Return type**

np.array of shape=(n_samples)

### 3.4.4 Semi-automatic mode

See *interpreter.Interpreter.predict()*.

### 3.4.5 I/O methods

The Interpreter can be saved and loaded from files using the following methods. Please note that the *interpreter.Interpreter.load()* method is a `classmethod` and must be called statically.

`Interpreter.`**`save`**(*outfile*)

Save model to output file.

**Parameters**

**`outfile`** (*string*) – File to output model.

**`classmethod`** `Interpreter.`**`load`**(*infile*, *context_builder=None*)

Load model from input file.

**Parameters**

- **`infile`** (*string*) – File from which to load model.
- **`context_builder`** (ContextBuilder, *optional*) – If given, use the given ContextBuilder for loading the Interpreter.

**Returns**

**self** – Return self.

**Return type**

self

**Example:**

```
from deepcase.interpreter import Interpreter
interpreter = Interpreter.load('<path_to_saved_interpreter>')
interpreter.save('<path_to_save_interpreter>')
```

CHAPTER

FOUR

# ROADMAP

This part of the documentation keeps track of desired features in future releases.

- Update usage with an example using the DeepCASE class.

## 4.1 Nice to haves

Features that are listed here would be nice to have for DeepCASE. I probably won't implement them myself, but feel free to send me a pull request.

- None at the moment

## 4.2 Changelog

**Version 1.0.1:**

- Added DeepCASE class

**Version 0.0.2:**

- Added fit/predict functionality to ContextBuilder and Interpreter.

**Version 0.0.1:**

- Initial release

CHAPTER

FIVE

# CONTRIBUTORS

This page lists all the contributors to this project. If you want to be involved in maintaining code or adding new features, please email t(dot)s(dot)vanede(at)utwente(dot)nl.

## 5.1 Code

- Thijs van Ede

## 5.2 Special Thanks

**We want to give our special thanks for people reporting bugs and fixes.**

- Ammar-Amjad

## 5.3 Academic Contributors

- Thijs van Ede
- Hojjat Aghakhani
- Noah Spahn
- Riccardo Bortolameotti
- Marco Cova
- Andrea Continella
- Maarten van Steen
- Andreas Peter
- Christopher Kruegel
- Giovanni Vigna

CHAPTER

SIX

# LICENSE

MIT License

Copyright (c) 2021 Thijs van Ede

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER

SEVEN

# CITING

To cite DeepCASE please use the following publication:

van Ede, T., Aghakhani, H., Spahn, N., Bortolameotti, R., Cova, M., Continella, A., van Steen, M., Peter, A., Kruegel, C. & Vigna, G. (2022, May). DeepCASE: Semi-Supervised Contextual Analysis of Security Events. In 2022 Proceedings of the IEEE Symposium on Security and Privacy (S&P). IEEE.

[PDF]

## 7.1 Bibtex

```
@inproceedings{vanede2020deepcase,
    title={{DeepCASE: Semi-Supervised Contextual Analysis of Security Events}},
    author={van Ede, Thijs and Aghakhani, Hojjat and Spahn, Noah and Bortolameotti,␣
↪Riccardo and Cova, Marco and Continella, Andrea and van Steen, Maarten and Peter,␣
↪Andreas and Kruegel, Christopher and Vigna, Giovanni},
    booktitle={Proceedings of the IEEE Symposium on Security and Privacy (S&P)},
    year={2022},
    organization={IEEE}
}
```

# INDEX

## Symbols



## A



## C



## D



## E



## F



## I



## J



## L



## N



## P

## Q



## R



## S



## T



## V