

---

# **rasa\_nlu Documentation**

*Release 0.9.0a1*

**Alan Nichol**

**May 21, 2017**



---

## Getting Started

---

<b>1</b>	<b>The quickest quickstart in the west</b>	<b>3</b>
<b>2</b>	<b>About</b>	<b>5</b>



**Attention:** This is the documentation for version 0.9.0a1 of rasa NLU. Make sure you select the appropriate version of the documentation for your local installation!

rasa NLU is an open source tool for intent classification and entity extraction. For example, taking a sentence like

```
"I am looking for a Mexican restaurant in the center of town"
```

and returning structured data like

```
{
  "intent": "search_restaurant",
  "entities": {
    "cuisine": "Mexican",
    "location": "center"
  }
}
```

The intended audience is mainly people developing bots. You can use rasa as a drop-in replacement for [wit](#) , [LUIS](#) , or [api.ai](#), the only change in your code is to send requests to `localhost` instead (see [Migrating an existing app](#) for details).

Why might you use rasa instead of one of those services?

- you don't have to hand over your data to FB/MSFT/GOOG
- you don't have to make a `https` call every time.
- you can tune models to work well on your particular use case.

These points are laid out in more detail in a [blog post](#) .



# CHAPTER 1

---

## The quickest quickstart in the west

---

```
$ python setup.py install
$ python -m rasa_nlu.server -e wit &
$ curl 'http://localhost:5000/parse?q=hello'
[{"_text": "hello", "confidence": 1.0, "entities": {}, "intent": "greet"}]
```

There you go! you just parsed some text. Next step, do the *Tutorial: A simple restaurant search bot*.

---

**Note:** This demo uses a very limited ML model. To apply rasa NLU to your use case, you need to train your own model! Follow the tutorial to get to know how to apply rasa\_nlu to your data.

---





You can think of rasa NLU as a set of high level APIs for building your own language parser using existing NLP and ML libraries. The setup process is designed to be as simple as possible. If you're currently using wit, LUIS, or api.ai, you just:

1. download your app data from wit or LUIS and feed it into rasa NLU
2. run rasa NLU on your machine and switch the URL of your wit/LUIS api calls to `localhost:5000/parse`.

rasa NLU is written in Python, but it you can use it from any language through *Using rasa NLU as a HTTP server*. If your project *is* written in Python you can simply import the relevant classes.

rasa is a set of tools for building more advanced bots, developed by [LASTMILE](#) . This is the natural language understanding module, and the first component to be open sourced.

## Installation

rasa NLU itself doesn't have any external requirements, but to do something useful with it you need to install & configure a backend. Which backend you want to use is up to you.

### Setting up rasa NLU

The recommended way to install rasa NLU is using pip:

```
pip install rasa_nlu
```

If you want to use the bleeding edge version use github + setup.py:

```
git clone git@github.com:golastmile/rasa_nlu.git
cd rasa_nlu
pip install -r requirements.txt
python setup.py install
```

rasa NLU allows you to use components to process your messages. E.g. there is a component for intent classification and there are several different components for entity recognition. The different components have their own requirements. To get you started quickly, this installation guide only installs the basic requirements, you may need to install other dependencies if you want to use certain components. When running rasa NLU it will check if all needed dependencies are installed and tell you which are missing, if any.

---

**Note:** If you want to make sure you got all the dependencies installed any component might ever need, and you don't mind the additional dependencies lying around, you can use

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

to install all requirements.

---

## **Setting up a backend**

Most of the processing pipeline you can use with rasa NLU either require MITIE, spaCy or sklearn to be installed.

### **Option 1 : MITIE**

The MITIE backend is all-inclusive, in the sense that it provides both the NLP and the ML parts.

```
pip install git+https://github.com/mit-nlp/MITIE.git
```

and then download the MITIE models . The file you need is `total_word_feature_extractor.dat`. Save this somewhere and in your `config.json` add `'mitie_file' : '/path/to/total_word_feature_extractor.dat'`.

---

**Note:**

**Training MITIE can be quite slow on datasets with more than a few intents. You can try**

- to use the sklearn + MITIE backend instead (which uses sklearn for the training) or
  - you can install [our mitie fork](#) which should reduce the training time as well.
- 

### **Option 2 : spaCy + sklearn**

You can also run using these two in combination.

installing spacy just requires (for more information visit the [spacy docu](#)):

```
pip install -U spacy
python -m spacy download en
```

If you haven't used numpy/scipy before, it is highly recommended that you use conda. steps are

- install [anaconda](#)
- `conda install scikit-learn`

otherwise if you know what you're doing, you can also just `pip install -U scikit-learn`

### Option 3 : sklearn + MITIE

There is a third backend that combines the advantages of the two previous ones:

1. the fast and good intent classification from sklearn and
2. the good entity recognition and feature vector creation from MITIE

Especially, if you have a larger number of intents (more than 10), training intent classifiers with MITIE can take very long.

To use this backend you need to follow the instructions for installing both, sklearn and MITIE.

## Tutorial: A simple restaurant search bot

---

**Note:** See *Migrating an existing app* for how to clone your existing wit/LUIS/api.ai app.

---

As an example we'll use the domain of searching for restaurants. We'll start with an extremely simple model of those conversations. You can build up from there.

Let's assume that *anything* our bot's users say can be categorized into one of the following intents:

- greet
- restaurant\_search
- thankyou

Of course there are many ways our users might greet our bot:

- *Hi!*
- *Hey there!*
- *Hello again :)*

And even more ways to say that you want to look for restaurants:

- *Do you know any good pizza places?*
- *I'm in the North of town and I want chinese food*
- *I'm hungry*

The first job of rasa NLU is to assign any given sentence to one of the categories: `greet`, `restaurant_search`, or `thankyou`.

The second job is to label words like "Mexican" and "center" as `cuisine` and `location` entities, respectively. In this tutorial we'll build a model which does exactly that.

### Preparing the Training Data

The best way to get training data is from *real users*, and the best way to do that is to *pretend to be the bot yourself*. But to help get you started we have some *data saved*

Download the file and open it, and you'll see a list of training examples like these:

```
{
  "text": "hey",
  "intent": "greet",
  "entities": []
}
```

```
{
  "text": "show me chinese restaurants",
  "intent": "restaurant_search",
  "entities": [
    {
      "start": 8,
      "end": 15,
      "value": "chinese",
      "entity": "cuisine"
    }
  ]
}
```

hopefully the format is intuitive if you've read this far into the tutorial, for details see [Training Data Format](#).

In your working directory, create a data folder, and copy the `demo-rasa.json` file there.

## Visualizing the Training Data

It's always a good idea to *look* at your data before, during, and after training a model. Luckily, there's a [great tool](#) for creating training data in rasa's format. - created by [@azazdeaz](#) - and it's also extremely helpful for inspecting existing data.

For the demo data the output should look like this:

restaurant_search	i'm looking for a place to eat
restaurant_search	i'm looking for a place in the <b>north</b> of town
restaurant_search	show me <b>chinese</b> restaurants

It is **strongly** recommended that you view your training data in the GUI before training.

## Training Your Model

Now we're going to create a configuration file. Make sure first that you've set up a backend, see [Installation](#) . Create a file called `config_spacy.json` or `config_mitie.json`, depending on the pipeline selected, in your working directory which looks like this

```
{
  "pipeline": "spacy_sklearn",
  "path" : "./models",
}
```

```
"data" : "./data/examples/rasa/demo-rasa.json"
}
```

or if you've installed the MITIE backend instead:

```
{
  "pipeline": "mitie",
  "mitie_file": "./data/total_word_feature_extractor.dat",
  "path" : "./models",
  "data" : "./data/examples/rasa/demo-rasa.json"
}
```

Now we can train a spacy model by running:

```
$ python -m rasa_nlu.train -c config_spacy.json
```

If you want to know more about the parameters, there is an overview of the *Configuration*. After a few minutes, rasa NLU will finish training, and you'll see a new dir called something like `models/model_YYYYMMDD-HHMMSS` with the timestamp when training finished.

## Using Your Model

To run your trained model, pass the configuration value `server_model_dirs` when running the server:

```
$ python -m rasa_nlu.server -c config_spacy.json --server_model_dirs=./model_YYYYMMDD-
↪HHMMSS
```

The passed model path is relative to the path configured in the configuration. More information about starting the server can be found in *Using rasa NLU as a HTTP server*.

You can then test our your new model by sending a request. Open a new tab/window on your terminal and run

```
$ curl -XPOST localhost:5000/parse -d '{"q":"I am looking for Chinese food"}' |_
↪python -mjson.tool
```

which should return

```
{
  "text": "I am looking for Chinese food",
  "entities": [
    {
      "start": 8,
      "end": 15,
      "value": "chinese",
      "entity": "cuisine",
      "extractor": "ner_spacy"
    }
  ],
  "intent": {
    "confidence": 0.6485910906220309,
    "name": "restaurant_search"
  },
  "intent_ranking": [
    {
      "confidence": 0.6485910906220309,
      "name": "restaurant_search"
    }
  ],
}
```

```

    {
      "confidence": 0.14161531595656784,
      "name": "affirm"
    }
  ]
}

```

If you are using the `spacy_sklearn` backend and the entities aren't found, don't panic! This tutorial is just a toy example, with far too little training data to expect good performance.

rasa NLU will also print a `confidence` value for the intent classification. For models using `spacy` intent classification this will be a probability. For MITIE models this is just a score, which **might be greater than 1**.

You can use this to do some error handling in your bot (maybe asking the user again if the confidence is low) and it's also helpful for prioritising which intents need more training data.

---

**Note:** The output may contain other or less attributes, depending on the pipeline you are using. For example, the `mitie` pipeline doesn't include the `"intent_ranking"` whereas the `spacy_sklearn` pipeline does.

---

With very little data, rasa NLU can in certain cases already generalise concepts, for example:

```

$ curl -XPOST localhost:5000/parse -d '{"q":"I want some italian"}' | python -mjson.
↪tool
{
  "text": "I want some italian",
  "entities": [
    {
      "end": 19,
      "entity": "cuisine",
      "start": 12,
      "value": "italian",
      "extrator": "ner_mitie"
    }
  ],
  "intent": {
    "confidence": 0.5192305466357352,
    "name": "restaurant_search"
  },
  "intent_ranking": [
    {
      "confidence": 0.5192305466357352,
      "name": "restaurant_search"
    },
    {
      "confidence": 0.2066287604378098,
      "name": "affirm"
    }
  ]
}

```

even though there's nothing quite like this sentence in the examples used to train the model. To build a more robust app you will obviously want to use a lot more data, so go and collect it!

## Configuration

You can provide options to rasa NLU through:

- a json-formatted config file
- environment variables
- command line arguments

Environment variables override options in your config file, and command line args will override any options specified elsewhere. Environment variables are capitalised and prefixed with `RASA_`, so the option `pipeline` is specified with the `RASA_PIPELINE` env var.

## Default

Here is the default configuration including all available parameters:

```
{
  "name": null,
  "pipeline": [],
  "language": "en",
  "num_threads": 1,
  "path": "models",
  "response_log": "logs",
  "config": "config.json",
  "log_level": "INFO",
  "port": 5000,
  "data": null,
  "emulate": null,
  "log_file": null,
  "mitie_file": "data/total_word_feature_extractor.dat",
  "spacy_model_name": null,
  "server_model_dirs": null,
  "token": null,
  "max_number_of_ngrams": 7,
  "duckling_dimensions": null,
  "luis_data_tokenizer": null,
  "entity_crf_BILOU_flag": true,
  "entity_crf_features": [
    ["low", "title", "upper", "pos", "pos2"],
    ["bias", "low", "word3", "word2", "upper", "title", "digit", "pos", "pos2"],
    ["low", "title", "upper", "pos", "pos2"]]
}
```

## Options

A short explanation and examples for each configuration value.

### name

**Type** str

**Examples** "my\_model\_name"

**Description** Defines a models name used to store it and to refere to it when using the http server. The default is `null` which will lead to a generated model name, e.g. `"model_20170426-230305"`.

### pipeline

**Type** `str` or `[str]`

**Examples** `"mitie"` or `["nlp_spacy", "ner_spacy", "ner_synonyms"]`

**Description** The pipeline used for training. Can either be a template (passing a string) or a list of components (array). For all available templates, see *Processing Pipeline*.

### language

**Type** `str`

**Examples** `"en"` or `"de"`

**Description** Language the model is trained in. Underlying word vectors will be loaded by using this language

### num\_threads

**Type** `int`

**Examples** `4`

**Description** Number of threads used during training (not supported by all components, though. Some of them might still be single threaded!).

### path

**Type** `str`

**Examples** `"models/"`

**Description** Directory where trained models will be saved to (training) and loaded from (http server).

### response\_log

**Type** `str` or `null`

**Examples** `"logs/"`

**Description** Directory where logs will be saved (containing queries and responses). If set to `null` logging will be disabled.

### config

**Type** `str`

**Examples** `"config_spacy.json"`

**Description** Location of the configuration file (can only be set as env var or command line option).



### log\_level

**Type** `str`

**Examples** "DEBUG"

**Description** Log level used to output messages from the framework internals.

### port

**Type** `int`

**Examples** 5000

**Description** Port on which to run the http server.

### data

**Type** `str`

**Examples** "data/example.json"

**Description** Location of the training data.

### emulate

**Type** `str`

**Examples** "wit", "luis" or "api"

**Description** Format to be returned by the http server. If `null` (default) the rasa NLU internal format will be used. Otherwise, the output will be formatted according to the API specified.

### mitie\_file

**Type** `str`

**Examples** "data/total\_word\_feature\_extractor.dat"

**Description** File containing `total_word_feature_extractor.dat` (see *Installation*)

### spacy\_model\_name

**Type** `str`

**Examples** "en\_core\_web\_sm"

**Description** If the spacy model to be used has a name that is different from the language tag ("en", "de", etc.), the model name can be specified using this configuration variable. The name will be passed to `spacy.load(name)`.

### server\_model\_dirs

**Type** str

**Examples** "models/"

**Description** Directory containing the model to be used by server or an object describing multiple models. see [HTTP server config](#)

### token

**Type** str or null

**Examples** "asd2aw3r"

**Description** if set, all requests to server must have a ?token=<token> query param. see [Authorization](#)

### max\_number\_of\_ngrams

**Type** int

**Examples** 10

**Description** Maximum number of ngrams to use when augmenting feature vectors with character ngrams (intent\_featurizer\_ngrams component only)

### duckling\_dimensions

**Type** list

**Examples** ["time", "number", "money", "distance"]

**Description** Defines which dimensions, i.e. entity types, the *duckling component* will extract. A full list of available dimensions can be found in the [duckling documentation](#).

### luis\_data\_tokenizer

**Type** str

**Examples** "tokenizer\_mitie"

**Description** Name of the tokenization component used to process luis data (Luis data annotates entities using token offset instead of character offsets, to convert the token offsets to character positions a tokenizer is required.) see [Migrating an existing app](#)

If you want to persist your trained models to S3, there are additional configuration options, see [Model Persistence](#)

### storage

**Type** str

**Examples** "aws" or "gcs"

**Description** Storage type for persistor. See [Model Persistence](#) for more details.

### bucket\_name

**Type** `str`

**Examples** `"my_models"`

**Description** Name of the bucket in the cloud to store the models. If the specified bucket name does not exist, rasa will create it. See *Model Persistence* for more details.

### aws\_region

**Type** `str`

**Examples** `"us-east-1"`

**Description** Name of the aws region to use. This is used only when "storage" is selected as "aws". See *Model Persistence* for more details.

### entity\_crf\_features

**Type** `[[str]]`

**Examples** `[["low", "title"], ["bias", "word3"], ["upper", "pos", "pos2"]]`

**Description** The features are a `[before, word, after]` array with before, word, after holding keys about which features to use for each word, for example, "title" in array before will have the feature "is the preceding word in title case?". Available features are: low, title, word3, word2, pos, pos2, bias, upper and digit

### entitiy\_crf\_BILOU\_flag

**Type** `bool`

**Examples** `true`

**Description** The flag determines whether to use BILOU tagging or not. BILOU tagging is more rigorous however requires more examples per entity. Rule of thumb: use only if more than 100 examples per entity.

## Migrating an existing app

rasa NLU is designed to make migrating from wit/LUIS/api.ai as simple as possible. The TLDR instructions for migrating are:

- download an export of your app data from wit/LUIS/api.ai
- follow the *Tutorial: A simple restaurant search bot*, using your downloaded data instead of `demo-rasa.json`

## Banana Peels

Just some specific things to watch out for for each of the services you might want to migrate from

## wit.ai

Wit used to handle `intents` natively. Now they are somewhat obfuscated. To create an `intent` in wit you have to create an `entity` which spans the entire text. The file you want from your download is called `expressions.json`

## LUIS.ai

When you download your model, the entity locations are specified by the index of the tokens. This is pretty fragile because not every tokenizer will behave the same as LUIS's, so your entities may be incorrectly labelled. Run your training once and you'll get a copy of your training data in the `model_XXXXX` dir. Do any fixes required and use that to train. Use the visualizer (see [Visualizing the Training Data](#)) to spot mistakes easily.

To specify the tokenizer you are using you need to define the `luis_data_tokenizer` configuration variable.

## api.ai

api app exports generate multiple files rather than just one. Put them all in a directory (see `data/examples/api` in the repo) and pass that path to the trainer.

## Emulation

To make rasa NLU easy to try out with existing projects, the server can *emulate* wit, LUIS, or api.ai. In native mode, a request / response looks like this :

```
$ curl -XPOST localhost:5000/parse -d '{"q":"I am looking for Chinese food"}' | python -mjson.tool
{
  "text": "I am looking for Chinese food",
  "intent": "restaurant_search",
  "confidence": 0.4794813722432127,
  "entities": [
    {
      "start": 17,
      "end": 24,
      "value": "chinese",
      "entity": "cuisine"
    }
  ]
}
```

if we run in wit mode (e.g. `python -m rasa_nlu.server -e wit`) then instead have to make a GET request

```
$ curl 'localhost:5000/parse?q=hello' | python -mjson.tool
[
  {
    "_text": "hello",
    "confidence": 0.4794813722432127,
    "entities": {},
    "intent": "greet"
  }
]
```

similarly for LUIS, but with a slightly different response format

```
$ curl 'localhost:5000/parse?q=hello' | python -mjson.tool
{
  "entities": [],
  "query": "hello",
  "topScoringIntent": {
    "intent": "inform",
    "score": 0.4794813722432127
  }
}
```

and finally for api.ai

```
$ curl 'localhost:5000/parse?q=hello' | python -mjson.tool
{
  "id": "ffd7ede3-b62f-11e6-b292-98fe944ee8c2",
  "result": {
    "action": null,
    "actionIncomplete": null,
    "contexts": [],
    "fulfillment": {},
    "metadata": {
      "intentId": "ffdbd6f3-b62f-11e6-8504-98fe944ee8c2",
      "intentName": "greet",
      "webhookUsed": "false"
    },
    "parameters": {},
    "resolvedQuery": "hello",
    "score": null,
    "source": "agent"
  },
  "sessionId": "ffdbd814-b62f-11e6-93b2-98fe944ee8c2",
  "status": {
    "code": 200,
    "errorType": "success"
  },
  "timestamp": "2016-11-29T12:33:15.369411"
}
```

## Training Data Format

The training data for rasa NLU has three arrays inside of a top level object `common_examples`, `intent_examples`, and `entity_examples`. Not all three are required, you can use each of them as needed by the model you are trying to train.

```
{
  "rasa_nlu_data": {
    "common_examples": [],
    "intent_examples": [],
    "entity_examples": []
  }
}
```

The `common_examples` are used to train both the entity and the intent models while the other arrays target intents and entities exclusively.

In many cases it's fine to put all of your training examples in the `common_examples` array. However, if you need lots and lots of examples to train a good entity recogniser, that can mess up your intent model because your classes would become unbalanced. In that case it makes sense to split up these lists.

## Common Examples

Common examples have three components: `text`, `intent`, and `entities`. The first two are strings while the last one is an array.

- The *text* is the search query; An example of what would be submitted for parsing.
- The *intent* is the intent that should be associated with the text.
- The *entities* are specific parts of the text which need to be identified.

Entities are specified with a `start` and `end` value, which together make a python style range to apply to the string, e.g. in the example below, with `text="show me chinese restaurants"`, then `text[8:15] == 'chinese'`. Entities can span multiple words, and in fact the `value` field does not have to correspond exactly to the substring in your example. That way you can map synonyms, or misspellings, to the same value.

```
{
  "text": "show me chinese restaurants",
  "intent": "restaurant_search",
  "entities": [
    {
      "start": 8,
      "end": 15,
      "value": "chinese",
      "entity": "cuisine"
    }
  ]
}
```

## Intent Examples

Omit the entire entities array:

```
{
  "text": "show me chinese restaurants",
  "intent": "restaurant_search"
}
```

## Entity Examples

Simply omit the intent section:

```
{
  "text": "show me chinese restaurants",
  "entities": [
    {
      "start": 8,
      "end": 15,
      "value": "chinese",
      "entity": "cuisine"
    }
  ]
}
```

```
]
}
```

## Entity Synonyms

If you define entities as having the same value they will be treated as synonyms. Here is an example of that:

```
[
  {
    "text": "in the center of NYC",
    "intent": "search",
    "entities": [
      {
        "start": 17,
        "end": 20,
        "value": "New York City",
        "entity": "city"
      }
    ]
  },
  {
    "text": "in the centre of New York City",
    "intent": "search",
    "entities": [
      {
        "start": 17,
        "end": 30,
        "value": "New York City",
        "entity": "city"
      }
    ]
  }
]
```

as you can see, the entity `city` has the value `New York City` in both examples, even though the text in the first example states NYC. By defining the value attribute to be different from the value found in the text between start and end index of the entity, you can define a synonym. Whenever the same text will be found, the value will use the synonym instead of the actual text in the message.

To use the synonyms defined in your training data, you need to make sure the pipeline contains the `ner_synonyms` component (see *Processing Pipeline*).

## Using rasa NLU as a HTTP server

---

**Note:** Before you can use the server, you need to train a model! See *Training Your Model*

---

The HTTP api exists to make it easy for non-python projects to use rasa NLU, and to make it trivial for projects currently using `{wit,LUIS,api}.ai` to try it out.

## Running the server

You can run a simple http server that handles requests using your models with (single threaded)

```
$ python -m rasa_nlu.server -c config_spacy.json --server_model_dirs=./model_YYYYMMDD-  
↪HHMMSS
```

If your server needs to handle more than one request at a time, you can use any WSGI server to run the rasa NLU server. Using gunicorn this looks like this:

```
$ gunicorn -w 4 --threads 12 -k gevent -b 127.0.0.1:5000 rasa_nlu.wsgi
```

This will start a server with four processes and 12 threads. Since there is no standard way to pass command line arguments to the server, all your configuration options need to be placed in your configuration file (including the `server_model_dirs`!). You can set the location of the configuration file using environment variables, otherwise the default configuration from `config.json` will be loaded.

## Emulation

rasa NLU can ‘emulate’ any of these three services by making the `/parse` endpoint compatible with your existing code. To activate this, either add `'emulate' : 'luis'` to your config file or run the server with `-e luis`. For example, if you would normally send your text to be parsed to LUIS, you would make a GET request to

```
https://api.projectoxford.ai/luis/v2.0/apps/<app-id>?q=hello%20there
```

in luis emulation mode you can call rasa by just sending this request to

```
http://localhost:5000/parse?q=hello%20there
```

any extra query params are ignored by rasa, so you can safely send them along.

## Endpoints

### POST `/parse` (no emulation)

You must POST data in this format `'{"q": "<your text to parse>"}`, you can do this with

```
$ curl -XPOST localhost:5000/parse -d '{"q": "hello there"}'
```

### POST `/train`

You can post your training data to this endpoint to train a new model. this starts a separate process which you can monitor with the `/status` endpoint. If you want to name your model to be able to use it during parse requests later on, you should pass the name `/train?name=my_model`. Any parameter passed with the query string will be treated as a configuration parameter of the model, hence you can change all the configuration values listed in the configuration section by passing in their name and the adjusted value.

```
$ curl -XPOST localhost:5000/train -d @data/examples/rasa/demo-rasa.json
```

### GET `/status`

This checks if there is currently a training process running (you can only run one at a time). also returns a list of available models the server can use to fulfill `/parse` requests.



```
$ curl localhost:5000/status | python -mjson.tool
{
  "training" : False
  "models" : []
}
```

### GET /version

This will return the current version of the Rasa NLU instance.

```
$ curl localhost:5000/version | python -mjson.tool
{
  "version" : "0.8.2"
}
```

### GET /config

This will return the currently running configuration of the Rasa NLU instance.

```
$ curl localhost:5000/config | python -mjson.tool
{
  "config": "/app/rasa_shared/config_mitie.json",
  "data": "/app/rasa_nlu/data/examples/rasa/demo-rasa.json",
  "duckling_dimensions": null,
  "emulate": null,
  ...
}
```

## Authorization

To protect your server, you can specify a token in your rasa NLU configuration, e.g. by adding "token" : "12345" to your config file, or by setting the RASA\_TOKEN environment variable. If set, this token must be passed as a query parameter in all requests, e.g. :

```
$ curl localhost:5000/status?token=12345
```

## Serving Multiple Apps

Depending on your choice of backend, rasa NLU can use quite a lot of memory. So if you are serving multiple models in production, you want to serve these from the same process & avoid duplicating the memory load.

---

**Note:** Although this saves the backend from loading the same backend twice, it still needs to load one set of word vectors (which make up most of the memory consumption) per language and backend.

---

You can use the multi-tenancy mode by replacing the `server_model_dirs` config variable with a json object describing the different models.

For example, if you have a restaurant bot and a hotel bot, your configuration might look like this:

```
{
  "server_model_dirs": {
    "hotels" : "./model_XXXXXXX",
    "restaurants" : "./model_YYYYYYY"
  }
}
```

You then pass an extra model parameter in your calls to /parse to specify which one to use:

```
$ curl 'localhost:5000/parse?q=hello&model=hotels'
```

or

```
$ curl -XPOST localhost:5000/parse -d '{"q":"I am looking for Chinese food", "model":
↳ "restaurants"}'
```

If one of the models is named `default`, it will be used to serve requests missing a model parameter. If no model is named `default` requests without a model parameter will be rejected.

## Using rasa NLU from python

Apart from running rasa NLU as a HTTP server you can use it directly in your python program. Rasa NLU supports both Python 2 and 3.

### Training Time

For creating your models, you can follow the same instructions as non-python users. Or, you can train directly in python with a script like the following (using spacy):

```
from rasa_nlu.converters import load_data
from rasa_nlu.config import RasaNLUConfig
from rasa_nlu.model import Trainer

training_data = load_data('data/examples/rasa/demo-rasa.json')
trainer = Trainer(RasaNLUConfig("config_spacy.json"))
trainer.train(training_data)
model_directory = trainer.persist('./models/') # Returns the directory the model is
↳ stored in
```

### Prediction Time

You can call rasa NLU directly from your python script. To do so, you need to load the metadata of your model and instantiate an interpreter. The `metadata.json` in your model dir contains the necessary info to recover your model:

```
from rasa_nlu.model import Metadata, Interpreter

metadata = Metadata.load(model_directory) # where model_directory points to the
↳ folder the model is persisted in
interpreter = Interpreter.load(metadata, RasaNLUConfig("config_spacy.json"))
```

You can then use the loaded interpreter to parse text:

```
interpreter.parse(u"The text I want to understand")
```

which returns the same dict as the HTTP api would (without emulation).

If multiple models are created, it is reasonable to share components between the different models. E.g. the 'nlp\_spacy' component, which is used by every pipeline that wants to have access to the spacy word vectors, can be cached to avoid storing the large word vectors more than once in main memory. To use the caching, a ComponentBuilder should be passed when loading and training models, e.g.:

```
from rasa_nlu.model import Metadata, Interpreter
from rasa_nlu.components import ComponentBuilder
config = RasaNLUConfig("config_spacy.json")

# For simplicity we will load the same model twice, usually you would want to use the
↪ metadata of
# different models
builder = ComponentBuilder(use_cache=True)      # will cache components between
↪ pipelines (where possible)
metadata_model = Metadata.load(model_directory)

interpreter = Interpreter.load(metadata_model, config, builder)
# the clone will share resources with the first model, as long as the same builder is
↪ passed!
interpreter_clone = Interpreter.load(metadata_model, config, builder)
```

## Entity Extraction

There are a number of different entity extraction components, which can seem intimidating for new users. Here we'll go through a few use cases and make recommendations of what to use.

Component	Requires	Model	notes
ner_mitie	MITIE	structured SVM	good for training custom entities
ner_crf	pycrfsuite	conditional random field	good for training custom entities
ner_spacy	spaCy	averaged perceptron	provides pre-trained entities
ner_duckling	duckling	context-free grammar	provides pre-trained entities

**Note:** To use these components, you will probably want to define a custom pipeline, see *Processing Pipeline*. You can add multiple ner components to your pipeline; the results from each will be combined in the final output.

## Use Cases

Here we'll outline some common use cases for entity extraction, and make recommendations on which components to use.

### Places, Dates, People, Organisations

spaCy has excellent pre-trained named-entity recognisers in a number of models. You can test them out in this [awesome interactive demo](#). We don't recommend that you try to train your own NER using spaCy, unless you have a lot of data and know what you are doing. Note that some spaCy models are highly case-sensitive.

## Dates, Amounts of Money, Durations, Distances, Ordinals

The `duckling` package does a great job of turning expressions like “next Thursday at 8pm” into actual datetime objects that you can use. It can also handle durations like “two hours”, amounts of money, distances, etc. Fortunately, there is also a `python wrapper` for duckling! You can use this component by installing the duckling package from PyPI and adding `ner_duckling` to your pipeline.

## Custom, Domain-specific entities

In the introductory tutorial we build a restaurant bot, and create custom entities for location and cuisine. The best components for training these domain-specific entity recognisers are the `ner_mitie` and `ner_crf` components. It is recommended that you experiment with both of these to see what works best for your data set.

## Returned Entities Object

In the object returned after parsing there are two fields that show information about how the pipeline impacted the entities returned. The `extractor` field of an entity tells you which entity extractor found this particular entity. The `processors` field contains the name of components that altered this specific entity.

The use of synonyms can also cause the `value` field not match the `text` exactly. Instead it will return the trained synonym.

```
{
  "text": "show me chinese restaurants",
  "intent": "restaurant_search",
  "entities": [
    {
      "start": 8,
      "end": 15,
      "value": "chinese",
      "entity": "cuisine",
      "extractor": "ner_mitie",
      "processors": []
    }
  ]
}
```

## Improving your models from feedback

When the `rasa_nlu` server is running, it keeps track of all the predictions it’s made and saves these to a log file. By default log files are placed in `logs/`. The files in this directory contain one json object per line. You can fix any incorrect predictions and add them to your training set to improve your parser. After adding these to your training data, but before retraining your model, it is strongly recommended that you use the visualizer to spot any errors, see [Visualizing training data](#).

## Model Persistence

rasa NLU supports using `S3` and `GCS` to save your models.

- **Amazon S3 Storage** S3 is supported using the `boto3` module which you can install with `pip install boto3`.

Start the rasa NLU server with `storage` option set to `aws`. Get your S3 credentials and set the following environment variables:

- `AWS_SECRET_ACCESS_KEY`
- `AWS_ACCESS_KEY_ID`
- `AWS_REGION`
- `BUCKET_NAME`

- **Google Cloud Storage** GCS is supported using the `google-cloud-storage` package which you can install with `pip install google-cloud-storage`

Start the rasa NLU server with `storage` option set to `gcs`.

When running on google app engine and compute engine, the auth credentials are already set up. For running locally or elsewhere, checkout their [client repo](#) for details on setting up authentication. It involves creating a service account key file from google cloud console, and setting the `GOOGLE_APPLICATION_CREDENTIALS` environment variable to the path of that key file.

If there is no bucket with the name `$BUCKET_NAME` rasa will create it. Models are gzipped before saving to cloud.

If you run the rasa NLU server with a `server_model_dirs` which does not exist and `BUCKET_NAME` is set, rasa will attempt to fetch a matching zip from your cloud storage bucket. E.g. if you have `server_model_dirs = ./data/model_20161111-180000` rasa will look for a file named `model_20161111-180000.tar.gz` in your bucket, unzip it and load the model.

## Language Support

Currently rasa NLU is tested and readily available for the following languages:

backend	supported languages
spacy-sklearn	english (en), german (de)
MITIE	english (en)

These languages can be set as part of the *Configuration*.

### Adding a new language

We want to make the process of adding new lanugages as simple as possible to increase the number of supported languages. Nevertheless, to use a language you either need a trained word representation or you need to train that presentation on your own using a large corpus of text data in that language.

These are the steps necessary to add a new language:

#### MITIE

1. Get a ~clean language corpus (a wikipedia dump works) as a set of text files
2. Build and run [MITIE wordrep tool](#) on your corpus. This can take several hours/days depending on your dataset and your workstation. You'll need something like 128GB of RAM for wordrep to run - yes that's alot: try to extend your swap.

3. Set the path of your new `total_word_feature_extractor.dat` as value of the `mitie_file` parameter in `config_mitie.json`

## spacy-sklearn

Spacy already provides a really good documentation page about [Adding languages](#). This will help you train a tokenizer and vocabulary for a new language in spacy.

As described in the documentation, you need to register your language using `set_lang_class()` which will allow rasa NLU to load and use your new language by passing in your language identifier as the `language` [Configuration](#) option.

## Processing Pipeline

The process of incoming messages is split into different components. These components are executed one after another in a so called processing pipeline. There are components for entity extraction, for intent classification, pre-processing and there will be many more in the future.

Each component processes the input and creates an output. The output can be used by any component that comes after this component in the pipeline. There are components which only produce information that is used by other components in the pipeline and there are other components that produce `Output` attributes which will be returned after the processing has finished. For example, for the sentence "I am looking for Chinese food" the output

```
{
  "text": "I am looking for Chinese food",
  "entities": [
    {"start": 8, "end": 15, "value": "chinese", "entity": "cuisine", "extractor":
↪"ner_crf"}
  ],
  "intent": {"confidence": 0.6485910906220309, "name": "restaurant_search"},
  "intent_ranking": [
    {"confidence": 0.6485910906220309, "name": "restaurant_search"},
    {"confidence": 0.1416153159565678, "name": "affirm"}
  ]
}
```

is created as a combination of the results of the different components in the pre-configured pipeline `spacy_sklearn`. For example, the `entities` attribute is created by the `ner_crf` component.

## Pre-configured Pipelines

To ease the burden of coming up with your own processing pipelines, we provide a couple of ready to use templates which can be used by settings the `pipeline` configuration value to the name of the template you want to use. Here is a list of the existing templates:

template name	corresponding pipeline
spacy_sklearn	["nlp_spacy", "ner_crf", "ner_synonyms", "intent_featurizer_spacy", "intent_classifier_sklearn"]
mitie	["nlp_mitie", "tokenizer_mitie", "ner_mitie", "ner_synonyms", "intent_classifier_mitie"]
mitie_sklearn	["nlp_mitie", "tokenizer_mitie", "ner_mitie", "ner_synonyms", "intent_featurizer_mitie", "intent_classifier_sklearn"]
keyword	["intent_classifier_keyword"]

Creating your own pipelines is possible by directly passing the names of the components to rasa NLU in the pipeline configuration variable, e.g. "pipeline": ["nlp\_spacy", "ner\_crf", "ner\_synonyms"]. This creates a pipeline that only does entity recognition, but no intent classification. Hence, the output will not contain any useful intents.

## Built-in Components

Short explanation of every components and it's attributes. If you are looking for more details, you should have a look at the corresponding source code for the component. Output describes, what each component adds to the final output result of processing a message. If no output is present, the component is most likely a preprocessor for another component.

### nlp\_mitie

**Short** MITIE initializer

**Outputs** nothing

**Description** Initializes mitie structures. Every mitie component relies on this, hence this should be put at the beginning of every pipeline that uses any mitie components.

### nlp\_spacy

**Short** spacy language initializer

**Outputs** nothing

**Description** Initializes spacy structures. Every spacy component relies on this, hence this should be put at the beginning of every pipeline that uses any spacy components.

### intent\_featurizer\_mitie

**Short** MITIE intent featurizer

**Outputs** nothing, used used as an input to intent classifiers that need intent features (e.g. intent\_classifier\_sklearn)

**Description** Creates feature for intent classification using the MITIE featurizer.

---

**Note:** NOT used by the intent\_classifier\_mitie component. Currently, only intent\_classifier\_sklearn is able to use precomputed features.

---

### intent\_featurizer\_spacy

**Short** spacy intent featurizer

**Outputs** nothing, used as an input to intent classifiers that need intent features (e.g. `intent_classifier_sklearn`)

**Description** Creates feature for intent classification using the spacy featurizer.

### intent\_featurizer\_ngrams

**Short** Appends char-ngram features to feature vector

**Outputs** nothing, appends its features to an existing feature vector generated by another intent featurizer

**Description** This featurizer appends character ngram features to a feature vector. During training the component looks for the most common character sequences (e.g. `app` or `ing`). The added features represent a boolean flag if the character sequence is present in the word sequence or not.

---

**Note:** There needs to be another intent featurizer previous to this one in the pipeline!

---

### intent\_classifier\_keyword

**Short** Simple keyword matching intent classifier.

**Outputs** `intent`

**Output-Example**

```
{
  "intent": {"name": "greet", "confidence": 0.98343}
}
```

**Description** This classifier is mostly used as a placeholder. It is able to recognize *hello* and *goodbye* intents by searching for these keywords in the passed messages.

### intent\_classifier\_mitie

**Short** MITIE intent classifier (using a `text categorizer`)

**Outputs** `intent`

**Output-Example**

```
{
  "intent": {"name": "greet", "confidence": 0.98343}
}
```

**Description** This classifier uses MITIE to perform intent classification. The underlying classifier is using a multi class linear SVM with a sparse linear kernel (see `mitie trainer code`).



## intent\_classifier\_sklearn

**Short** sklearn intent classifier

**Outputs** intent and intent\_ranking

**Output-Example**

```
{
  "intent": {"name": "greet", "confidence": 0.78343},
  "intent_ranking": [
    {
      "confidence": 0.1485910906220309,
      "name": "goodbye"
    },
    {
      "confidence": 0.08161531595656784,
      "name": "restaurant_search"
    }
  ]
}
```

**Description** The sklearn intent classifier trains a linear SVM which gets optimized using a grid search. In addition to other classifiers it also provides rankings of the labels that did not “win”. The spacy intent classifier needs to be preceded by a featurizer in the pipeline. This featurizer creates the features used for the classification.

## tokenizer\_whitespace

**Short** Tokenizer using whitespaces as a separator

**Outputs** nothing

**Description** Creates a token for every whitespace separated character sequence. Can be used to define tokens for the MITIE entity extractor.

## tokenizer\_mitie

**Short** Tokenizer using MITIE

**Outputs** nothing

**Description** Creates tokens using the MITIE tokenizer. Can be used to define tokens for the MITIE entity extractor.

## tokenizer\_spacy

**Short** Tokenizer using spacy

**Outputs** nothing

**Description** Creates tokens using the spacy tokenizer. Can be used to define tokens for the MITIE entity extractor.

## ner\_mitie

**Short** MITIE entity extraction (using a mitie ner trainer)

**Outputs** appends `entities`

### Output-Example

```
{
  "entities": [{"value": "New York City",
                 "start": 20,
                 "end": 33,
                 "entity": "city",
                 "extractor": "ner_mitie"}]}

```

**Description** This uses the MITIE entity extraction to find entities in a message. The underlying classifier is using a multi class linear SVM with a sparse linear kernel and custom features.

## ner\_spacy

**Short** spacy entity extraction

**Outputs** appends `entities`

### Output-Example

```
{
  "entities": [{"value": "New York City",
                 "start": 20,
                 "end": 33,
                 "entity": "city",
                 "extractor": "ner_spacy"}]}

```

**Description** Using spacy this component predicts the entities of a message. spacy uses a statistical BILUO transition model. As of now, this component can only use the spacy builtin entity extraction models and can not be retrained.

## ner\_synonyms

**Short** Maps synonymous entity values to the same value.

**Outputs** modifies existing entities that previous entity extraction components found

**Description** If the training data contains defined synonyms (by using the `value` attribute on the entity examples). this component will make sure that detected entity values will be mapped to the same value. For example, if your training data contains the following examples:

```
[{
  "text": "I moved to New York City",
  "intent": "inform_relocation",
  "entities": [{"value": "nyc",
                 "start": 11,
                 "end": 24,
                 "entity": "city",
                 "extractor": "ner_mitie",
                 "processor": ["ner_synonyms"]}]}]

```

```

},
{
  "text": "I got a new flat in NYC.",
  "intent": "inform_relocation",
  "entities": [{"value": "nyc",
                 "start": 20,
                 "end": 23,
                 "entity": "city",
                 "extractor": "ner_mitie",
                 "processor": ["ner_synonyms"]}]]
}

```

this component will allow you to map the entities `New York City` and `NYC` to `nyc`. The entity extraction will return `nyc` even though the message contains `NYC`. When this component changes an existing entity, it appends itself to the processor list of this entity.

## ner\_crf

**Short** conditional random field entity extraction

**Outputs** appends entities

**Output-Example**

```

{
  "entities": [{"value": "New York City",
                 "start": 20,
                 "end": 33,
                 "entity": "city",
                 "extractor": "ner_crf"}]
}

```

**Description** This component implements conditional random fields to do named entity recognition. CRFs can be thought of as an undirected Markov chain where the time steps are words and the states are entity classes. Features of the words (capitalisation, POS tagging, etc.) give probabilities to certain entity classes, as are transitions between neighbouring entity tags: the most likely set of tags is then calculated and returned.

## ner\_duckling

**Short** Adds duckling support to the pipeline to unify entity types (e.g. to retrieve common date / number formats)

**Outputs** appends entities

**Output-Example**

```

{
  "entities": [{"end": 53,
                 "entity": "time",
                 "start": 48,
                 "value": "2017-04-10T00:00:00.000+02:00",
                 "extractor": "ner_duckling"}]
}

```

**Description** Duckling allows to recognize dates, numbers, distances and other structured entities and normalizes them (for a reference of all available entities see [the duckling documentation](#)). The

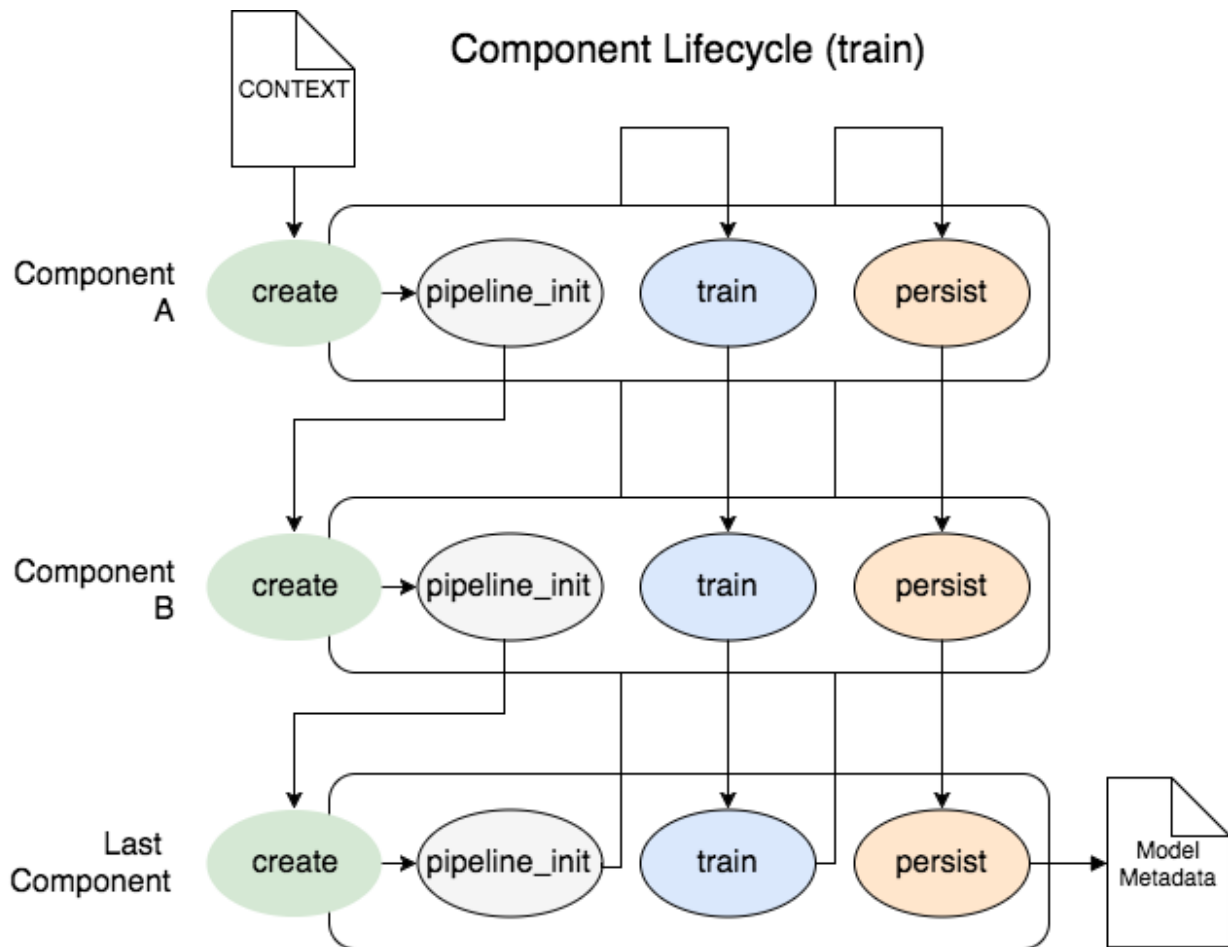
component recognizes the entity types defined by the *duckling dimensions configuration variable*. Please be aware that duckling tries to extract as many entity types as possible without providing a ranking. For example, if you specify both `number` and `time` as dimensions for the duckling component, the component will extract two entities: `10` as a number and `in 10 minutes` as a time from the text `I will be there in 10 minutes`. In such a situation, your application would have to decide which entity type is be the correct one.

## Creating new Components

Currently you need to rely on the components that are shipped with rasa NLU, but we plan to add the possibility to create your own components in your code. Nevertheless, we are looking forward to your contribution of a new component (e.g. a component to do sentiment analysis). A glimpse into the code of `rasa_nlu.components.Component` will reveal which functions need to be implemented to create a new component.

## Component Lifecycle

Every component can implement several methods from the `Component` base class; in a pipeline these different methods will be called in a specific order. Lets assume, we added the following pipeline to our config: `"pipeline": ["Component A", "Component B", "Last Component"]`. The image shows the call order during the training of this pipeline :



Before the first component is created using the `create` function, a so called `context` is created (which is nothing more than a python dict). This context is used to pass information between the components. For example, one

component can calculate feature vectors for the training data, store that within the context and another component can retrieve these feature vectors from the context and do intent classification.

Initially the context is filled with all configuration values, the arrows in the image show the call order and visualize the path of the passed context. After all components are trained and persisted, the final context dictionary is used to persist the models metadata.

## Frequently Asked Questions

### Does it run with python 3?

Yes it does, rasa NLU supports python 2.7 as well as python 3.5 and 3.6. If there are any issues with a specific python version, feel free to create an issue or directly provide a fix.

### Which languages are supported?

There is a list containing all officialy supported languages [here](#). Nevertheless, there are others working on adding more languages, feel free to have a look at the [github issues](#) section or the [gitter chat](#).

### Which version of rasa NLU am I running?

To find out which rasa version you are running, you can execute

```
For Python 2.7:
python -c "import rasa_nlu; print rasa_nlu.__version__"

For Python 3.x:
python -c "import rasa_nlu; print(rasa_nlu.__version__);"
```

If you are using a virtual environment to run your python code, make sure you are using the correct python to execute the above code.

### Why am I getting an UndefinedMetricWarning?

The complete warning is: UndefinedMetricWarning: F-score is ill-defined and being set to 0.0 in labels with no predicted samples. The warning is a result of a lack of training data. During the training the dataset will be splitted multiple times, if there are to few training samples for any of the intents, the splitting might result in splits that do not contain any examples for this intent.

Hence, the solution is to add more training samples. As this is only a warning, training will still succeed, but the resulting models predictions might be weak on the intents where you are lacking training data.

### I have an issue, can you help me?

We'd love to help you. If you are unsure if your issue is related to your setup, you should state your problem in the [gitter chat](#). If you found an issue with the framework, please file a report on [github issues](#) including all the information needed to reproduce the problem.

## Migration Guide

This page contains information about changes between major versions and how you can migrate from one version to another.

### 0.8.x to master

#### 0.7.x to 0.8.x

- The training and loading capability for the spacy entity extraction was dropped in favor of the new CRF extractor. That means models need to be retrained using the crf extractor.
- The parameter and configuration value name of backend changed to pipeline.
- There have been changes to the model metadata format. You can either retrain your models or change the stored metadata.json:
  - rename language\_name to language
  - rename backend to pipeline
  - for mitie models you need to replace feature\_extractor with mitie\_feature\_extractor\_fingerprint. That fingerprint depends on the language you are using, for en it is "mitie\_feature\_extractor\_fingerprint": 10023965992282753551.

#### 0.6.x to 0.7.x

- The parameter and configuration value name of server\_model\_dir changed to server\_model\_dirs.
- The parameter and configuration value name of write changed to response\_log. It now configures the *directory* where the logs should be written to (not a file!)
- The model metadata format has changed. All paths are now relative with respect to the path specified in the configuration during training and loading. If you want to run models that are trained with a version prev to 0.7 you need to adapt the paths manually in metadata.json from

```
{
  "trained_at": "20170304-191111",
  "intent_classifier": "model_XXXX_YYYY_ZZZZ/intent_classifier.pkl",
  "training_data": "model_XXXX_YYYY_ZZZZ/training_data.json",
  "language_name": "en",
  "entity_extractor": "model_XXXX_YYYY_ZZZZ/ner",
  "feature_extractor": null,
  "backend": "spacy_sklearn"
}
```

to something along the lines of this (making all paths relative to the models base dir, which is model\_XXXX\_YYYY\_ZZZZ/):

```
{
  "trained_at": "20170304-191111",
  "intent_classifier": "intent_classifier.pkl",
  "training_data": "training_data.json",
  "language_name": "en",
  "entity_synonyms": null,
}
```

```
"entity_extractor": "ner",
"feature_extractor": null,
"backend": "spacy_sklearn"
}
```

## License

Copyright 2016 LastMile Technologies Ltd

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this project except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

## Roadmap

These are features we are keen on! Check out issues in github to see status of things.

- easier deployment options
- create a platform for rasa NLU users to share models/ data
- support loading training data from a DB instead of a text file
- entity normalisation: as is, the named entity extractor will happily extract *cheap & inexpensive* as entities of the *expense* class, but will not tell you that these are realisations of the same underlying concept. You can easily handle that with a list of aliases in your code, but we want to offer a more elegant & generalisable solution. [Word Forms]([https://github.com/gutfeeling/word\\_forms](https://github.com/gutfeeling/word_forms)) looks promising.
- support for more (human) languages

## Contributing

Contributions are very much encouraged! Please create an issue before doing any work to avoid disappointment.

We created a tag that should get you started quickly if you are searching for interesting topics to get started.

## Python Conventions

Python code should follow the pep-8 spec.

## Python 2 and 3 Cross Compatibility

To ensure cross compatibility between Python 2 and 3 we prioritize Python 3 conventions. Keep in mind that:

- all string literals are unicode strings
- division generates floating point numbers. Use `//` for truncated division

- some built-ins, e.g. `map` and `filter` return iterators in Python 3. If you want to make use of them import the Python 3 version of them from `builtins`. Otherwise use list comprehensions, which work uniformly across versions
- use `io.open` instead of the builtin `open` when working with files
- The following imports from `__future__` are mandatory in every python file: `unicode_literals`, `print_function`, `division`, and `absolute_import`

Please refer to this [cheat sheet](#) to learn how to write different constructs compatible with Python 2 and 3.

## Code of conduct

rasa NLU adheres to the [Contributor Covenant Code of Conduct](#). By participating, you are expected to uphold this code.

## Documentation

Everything should be properly documented. To locally test the documentation you need to install

```
brew install sphinx
pip install sphinx_rtd_theme
```

After that, you can compile and view the documentation using:

```
cd docs
make html
cd _build/html
python -m SimpleHTTPServer 8000 .
```

The documentation will be running on <http://localhost:8000/>.

Code snippets that are part of the documentation can be tested using

```
make doctest
```

## Change Log

All notable changes to this project will be documented in this file. This project adheres to [Semantic Versioning](#) starting with version 0.7.0.

### [Unreleased] - master

---

**Note:** This version is not yet released and is under active development.

---

#### Added

#### Changed



## Removed

### Fixed

- /status endpoint showing all available models instead of only those whose name starts with *model*

## [0.8.5] - 2017-05-10

### Fixed

- Fixed pypi installation dependencies (e.g. flask). fixes #354

## [0.8.4] - 2017-05-10

### Fixed

- Fixed CRF model training without entities. fixes #345

## [0.8.3] - 2017-05-10

### Fixed

- Fixed Luis emulation and added test to catch regression. Fixes #353

## [0.8.2] - 2017-05-08

### Fixed

- deepcopy of context #343

## [0.8.1] - 2017-05-08

### Fixed

- NER training reuses context inbetween requests

## [0.8.0] - 2017-05-08

### Added

- ngram character featurizer (allows better handling of out-of-vocab words)
- replaced pre-wired backends with more flexible pipeline definitions
- return top 10 intents with sklearn classifier #199
- python type annotations for nearly all public functions
- support for arbitrary spacy language model names

- duckling components to provide normalized output for structured entities
- Conditional random field entity extraction (Markov model for entity tagging, better named entity recognition with low and medium data and similarly well at big data level)
- allow naming of trained models instead of generated model names
- dynamic check of requirements for the different components & error messages on missing dependencies
- support for using multiple entity extractors and combining results downstream

### Changed

- unified tokenizers, classifiers and feature extractors to implement common component interface
- `src` directory renamed to `rasa_nlu`
- when loading data in a foreign format (`api.ai`, `luis`, `wit`) the data gets properly split into intent & entity examples
- **Configuration:**
  - added `max_number_of_ngrams`
  - removed `backend` and added `pipeline` as a replacement
  - added `luis_data_tokenizer`
  - added `duckling_dimensions`
- **parser output format changed** from `{"intent": "greeting", "confidence": 0.9, "entities": []}` to `{"intent": {"name": "greeting", "confidence": 0.9}, "entities": []}`
- **entities output format changed** from `{"start": 15, "end": 28, "value": "New York City", "entity": "GPE"}` to `{"extractor": "ner_mitie", "processors": ["ner_synonyms"], "start": 15, "end": 28, "value": "New York City", "entity": "GPE"}` where `extractor` denotes the entity extractor that originally found an entity, and `processor` denotes components that alter entities, such as the synonym component.
- camel cased MITIE classes (e.g. `MITIETokenizer` → `MitieTokenizer`)
- model metadata changed, see migration guide
- updated to `spacy 1.7` and dropped training and loading capabilities for the `spacy` component (breaks existing `spacy` models!)
- introduced compatibility with both Python 2 and 3

### Removed

### Fixed

- properly parse `str` additionally to `unicode` #210
- support entity only training #181
- resolved conflicts between metadata and configuration values #219
- removed tokenization when reading `Luis.ai` data (they changed their format) #241

### [0.7.4] - 2017-03-27

#### Fixed

- fixed failed loading of example data after renaming attributes, i.e. “KeyError: ‘entities’”

### [0.7.3] - 2017-03-15

#### Fixed

- fixed regression in mitie entity extraction on special characters
- fixed spacy fine tuning and entity recognition on passed language instance

### [0.7.2] - 2017-03-13

#### Fixed

- python documentation about calling rasa NLU from python

### [0.7.1] - 2017-03-10

#### Fixed

- mitie tokenization value generation #207, thanks @cristinacaputo
- changed log file extension from `.json` to `.log`, since the contained text is not proper json

### [0.7.0] - 2017-03-10

This is a major version update. Please also have a look at the [Migration Guide](#).

#### Added

- Changelog ;)
- option to use multi-threading during classifier training
- entity synonym support
- proper temporary file creation during tests
- mitie\_sklearn backend using mitie tokenization and sklearn classification
- option to fine-tune spacy NER models
- multithreading support of build in REST server (e.g. using gunicorn)
- multitenancy implementation to allow loading multiple models which share the same backend

**Fixed**

- error propagation on failed vector model loading (spacy)
- escaping of special characters during mitie tokenization

**[0.6-beta] - 2017-01-31**