rasa_nlu Documentation

Release 0.12.0a2

Alan Nichol

Mar 23, 2018

Getting Started

Note: This is the documentation for version 0.12.0a2 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 Dialogflow, 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.

CHAPTER 2

About

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 Dialogflow, 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/Dialogflow 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 Rasa . This is the natural language understanding module, and the first component to be open sourced.

2.1 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.

2.1.1 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 https://github.com/RasaHQ/rasa_nlu.git
cd rasa_nlu
pip install -r requirements.txt
pip install -e .
```

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 alt_requirements/requirements_full.txt
```

instead of `requirements.txt` to install all requirements.

2.1.2 Setting up a backend

Most of the processing pipeline you can use with rasa NLU require spaCy and sklearn to be installed.

Best for most: spaCy + sklearn

You can also run using these two in combination.

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

```
pip install rasa_nlu[spacy]
python -m spacy download en_core_web_md
python -m spacy link en_core_web_md en
```

This will install Rasa NLU as well as spacy and its language model for the english language. We highly recommend using at least the "medium" sized models (`_md`) instead of the spacy's default small `en_core_web_sm` model. Small models will work as well, the downside is that they have worse performance during intent classification.

Note: Using spaCy as the backend for Rasa is the **preferred option**.

2.2 Tutorial: A simple restaurant search bot

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

As an example we'll start a new project covering 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 **intent** 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.

2.2.1 Preparing the Training Data

The training data is essential to develop chatbots. It should include texts to be interpreted and the structured data (intent/entities) we expect chatbots to convert the texts into. The best way to get training texts is from *real users*, and the best way to get the structured data is to pretend to be the bot yourself. But to help get you started, we have some data saved.

Download the file (json format) and open it, and you'll see a list of training examples, each composed of `"text"`, `"intent"` and `"entities"`, as shown below. In your working directory, create a `data` folder, and copy this `demo-rasa.json` file there.

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

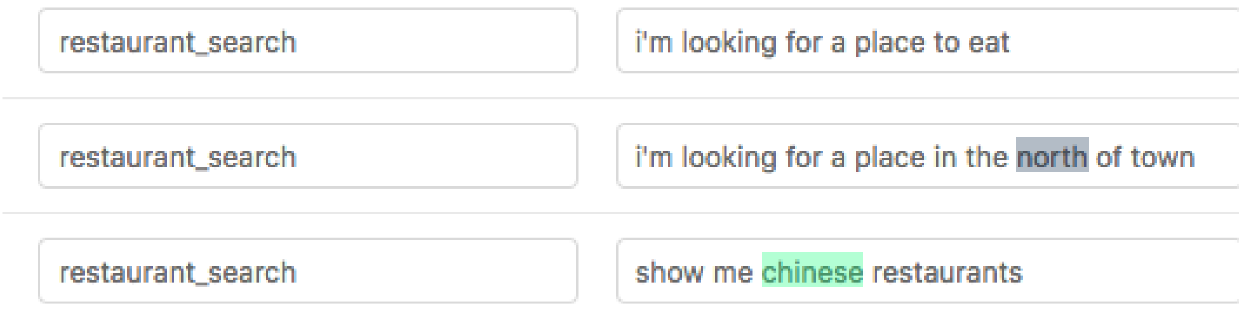
```
{
  "text": "show me chinese restaurants",
  "intent": "restaurant_search",
  "entities": [
    {
      "start": 8,
      "end": 15,
      "value": "chinese",
      "entity": "cuisine",
      "confidence": 0.875
    }
  ]
}
```

Hopefully the format is intuitive if you've read this far into the tutorial, for details see *Training Data Format*. Otherwise, the next section 'visualizing the training data' can help you better read, verify and/or modify the training data.

2.2.2 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 and modifying existing data.

For the demo data the output should look like this:



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

2.2.3 Training a New Model for your Project

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` in your working directory which looks like this

```
{
  "pipeline": "spacy_sklearn",
  "path" : "./projects",
  "data" : "./data/examples/rasa/demo-rasa.json"
}
```

Now we can train a spacy model by running:

```
$ python -m rasa_nlu.train -c sample_configs/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 folder named as `projects/default/model_YYYYMMDD-HHMMSS` with the timestamp when training finished.

2.2.4 Using Your Model

By default, the server will look for all projects folders under the `path` directory specified in the configuration. When no project is specified, as in this example, a "default" one will be used, itself using the latest trained model.

```
$ python -m rasa_nlu.server -c sample_configs/config_spacy.json
```

More information about starting the server can be found in *Using rasa NLU as a HTTP server*.

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

Note: For windows users the windows command line interface doesn't like single quotes. Use doublequotes and escape where necessary. `curl -X POST "localhost:5000/parse" -d "{/"q/":/"I am looking for Mexican food/"}" | python -m json.tool`

```
$ curl -X POST localhost:5000/parse -d '{"q":"I am looking for Mexican food"}' |
python -m json.tool
```

which should return

```
{
    "intent": {
    "name": "restaurant_search",
    "confidence": 0.8231117999072759
    },
    "entities": [
        {
            "start": 17,
            "end": 24,
            "value": "mexican",
            "entity": "cuisine",
            "extractor": "ner_crf",
            "confidence": 0.875
        }
    ],
    "intent_ranking": [
        {
            "name": "restaurant_search",
            "confidence": 0.8231117999072759
        },
        {
            "name": "affirm",
            "confidence": 0.07618757211779097
        },
        {
            "name": "goodbye",
            "confidence": 0.06298664363805719
        },
        {
            "name": "greet",
            "confidence": 0.03771398433687609
        }
    ],
    "text": "I am looking for Mexican food"
}
```

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.

Note: Intent classification is independent of entity extraction, e.g. in "I am looking for Chinese food" the entities are not extracted, though intent classification is correct.

Rasa NLU will also print a `confidence` value for the intent classification. For models using spacy intent classification this will be a probability.

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

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

```
$ curl -X POST localhost:5000/parse -d '{"q":"I want some italian food"}' | python -m␣
↪json.tool
{
    "intent": {
        "name": "restaurant_search",
        "confidence": 0.5792111723774511
```

```
    },
    "entities": [
        {
            "entity": "cuisine",
            "value": "italian",
            "start": 12,
            "end": 19,
            "extractor": "ner_crf",
            "confidence": 0.875
        }
    ],
    "text": "I want some italian food"
}
```

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 training data, so go and collect it!

2.3 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.

2.3.1 Default

Here is the default configuration including all available parameters:

```
{
  "project": null,
  "fixed_model_name": null,
  "pipeline": [],
  "language": "en",
  "num_threads": 1,
  "max_training_processes": 1,
  "path": "projects",
  "response_log": "logs",
  "storage": null,
  "config": "config.json",
  "log_level": "INFO",
  "port": 5000,
  "data": null,
  "emulate": null,
  "log_file": null,
  "spacy_model_name": null,
  "token": null,
  "cors_origins": [],
  "aws_endpoint_url": null,
  "max_number_of_ngrams": 7,
```

```
  "duckling_dimensions": null,
  "duckling_http_url": null,

  "ner_crf": {
    "BILOU_flag": true,
    "features": [
      ["low", "title", "upper", "pos", "pos2"],
      ["bias", "low", "word3", "word2", "upper", "title", "digit", "pos", "pos2",
↪"pattern"],
      ["low", "title", "upper", "pos", "pos2"]],
    "max_iterations": 50,
    "L1_c": 1,
    "L2_c": 1e-3
  },

  "intent_classifier_sklearn": {
    "C": [1, 2, 5, 10, 20, 100],
    "kernel": "linear"
  }
}
```

2.3.2 Options

A short explanation and examples for each configuration value.

project

Type `str`

Examples `"my_project_name"`

Description Defines a project name to train new models for and to refer to when using the http server. The default value is `null` which will lead to using the default project `"default"`. All projects are stored under the `path` directory.

pipeline

Type `str` or `[str]`

Examples `"spacy_sklearn"` 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!).

fixed_model_name

Type `str`

Examples `"my_model_name"`

Description Instead of generating model names (e.g. `model_20170922-234435`) a fixed model name will be used. The model will always be saved in the path `{project_path}/{project_name}/{model_name}`. If the model is assigned a fixed name, it will possibly override previously trained models.

max_training_processes

Type `int`

Examples `1`

Description Number of processes used to handle training requests. Increasing this value will have a great impact on memory usage. It is recommended to keep the default value.

path

Type `str`

Examples `"projects/"`

Description Projects 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 `"sample_configs/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. For JSON and markdown data, this can either be a single file or a directory containing multiple training data files.

cors_origins

Type `list`

Examples `["*"]`, `["*.mydomain.com", "api.domain2.net"]`

Description List of domain patterns from where CORS (cross-origin resource sharing) calls are allowed. The default value is `[]` which forbids all CORS requests.

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.

spacy_model_name

Type `str`

Examples `"en_core_web_md"`

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)`.

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", "amount-of-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.

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.

aws_endpoint_url

Type `str`

Examples `"http://10.0.0.1:9000"`

Description Optional endpoint of the custom S3 compatible storage provider. This is used only when `"storage"` is selected as `"aws"`. See *Model Persistence* for more details.

ner_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`

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.

max_iterations

Type `int`

Examples `50`

Description This is the value given to sklearn_crfcuite.CRF tagger before training.

L1_C

Type `float`

Examples `1.0`

Description This is the value given to sklearn_crfcuite.CRF tagger before training. Specifies the L1 regularization coefficient.

L2_C

Type `float`

Examples `1e-3`

Description This is the value given to sklearn_crfcuite.CRF tagger before training. Specifies the L2 regularization coefficient.

intent_classifier_sklearn

C

Type `[float]`

Examples `[1, 2, 5, 10, 20, 100]`

Description Specifies the list of regularization values to cross-validate over for C-SVM. This is used with the `kernel` hyperparameter in GridSearchCV.

kernel

Type `string`

Examples `"linear"`

Description Specifies the kernel to use with C-SVM. This is used with the `C` hyperparameter in GridSearchCV.

2.4 Migrating an existing app

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

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

2.4.1 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 and `entity` which spans the entire text. The file you want from your download is called `expressions.json`

LUIS.ai

Nothing special here. Downloading the data and importing it into Rasa NLU should work without issues

Dialogflow

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

2.4.2 Emulation

To make Rasa NLU easy to try out with existing projects, the server can *emulate* wit, LUIS, or Dialogflow. 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 Dialogflow

```
$ 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"
}
```

2.5 Training Data Format

The training data for rasa NLU is structured into different parts, `common_examples`, `entity_synonyms` and `regex_features`. The most important one is `common_examples`.

```
{
    "rasa_nlu_data": {
        "common_examples": [],
        "regex_features" : [],
        "entity_synonyms": []
    }
}
```

The `common_examples` are used to train both the entity and the intent models. You should put all of your training examples in the `common_examples` array. The next section describes in detail how an example looks like. Regex features are a tool to help the classifier detect entities or intents and improve the performance.

You can use Chatito , a tool for generating training datasets in rasa's format using a simple DSL or Tracy, a simple GUI to create training datasets for rasa.

2.5.1 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. [required]
- The *intent* is the intent that should be associated with the text. [optional]
- The *entities* are specific parts of the text which need to be identified. [optional]

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"
    }
  ]
}
```

2.5.2 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*).

Alternatively, you can add an "entity_synonyms" array to define several synonyms to one entity value. Here is an example of that:

```
{
  "rasa_nlu_data": {
    "entity_synonyms": [
      {
        "value": "New York City",
        "synonyms": ["NYC", "nyc", "the big apple"]
      }
    ]
  }
}
```

Note: Please note that adding synonyms using the above format does not improve the model's classification of those entities. **Entities must be properly classified before they can be replaced with the synonym value.**

2.5.3 Regular Expression Features

Regular expressions can be used to support the intent classification and entity extraction. E.g. if your entity has a certain structure as in a zipcode, you can use a regular expression to ease detection of that entity. For the zipcode example it might look like this:

```
{
    "rasa_nlu_data": {
        "regex_features": [
            {
                "name": "zipcode",
                "pattern": "[0-9]{5}"
            },
            {
                "name": "greet",
                "pattern": "hey[^\\s]*"
            },
        ]
    }
}
```

The name doesn't define the entity nor the intent, it is just a human readable description for you to remember what this regex is used for. As you can see in the above example, you can also use the regex features to improve the intent classification performance.

Try to create your regular expressions in a way that they match as few words as possible. E.g. using `hey[^\s]*` instead of `hey.*`, as the later one might match the whole message whereas the first one only matches a single word.

Regex features for entity extraction are currently only supported by the `ner_crf` component! Hence, other entity extractors, like `ner_spacy` won't use the generated features and their presence will not improve entity recognition for these extractors. Currently, all intent classifiers make use of available regex features.

Note: Regex features don't define entities nor intents! They simply provide patterns to help the classifier recognize entities and related intents. Hence, you still need to provide intent & entity examples as part of your training data!

2.5.4 Markdown Format

Alternatively training data can be used in the following markdown format. Examples are listed using the unordered list syntax, e.g. minus `-`, asterisk `*`, or plus `+`:

```
## intent:check_balance
- what is my balance <!-- no entity -->
- how much do I have on my [savings](source_account) <!-- entity "source_account" has␣
↪value "savings" -->
- how much do I have on my [my savings account](source_account:savings) <!-- synonyms,
↪ method 1-->

## intent:greet
- hey
- hello

## synonym:savings   <!-- synonyms, method 2 -->
- pink pig


## regex:zipcode
- [0-9]{5}
```

2.5.5 Organization

The training data can either be stored in a single file or split into multiple files. For larger training examples, splitting the training data into multiple files, e.g. one per intent, increases maintainability.

Storing files with different file formats, i.e. mixing markdown and JSON, is currently not supported.

Note: Splitting the training data into multiple files currently only works for markdown and JSON data. For other file formats you have to use the single-file approach.

2.6 Using rasa NLU as a HTTP server

Note: Before you can use the server, you need to train a model! See *Training a New Model for your Project*

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/Dialogflow to try it out.

2.6.1 Running the server

You can run a simple http server that handles requests using your projects with :

```
$ python -m rasa_nlu.server -c sample_configs/config_spacy.json
```

The server will look for existing projects under the folder defined by the `path` parameter in the configuration. By default a project will load the latest trained model.

2.6.2 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.

2.6.3 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"}'
```

By default, when the project is not specified in the query, the `"default"` one will be used. You can (should) specify the project you want to use in your query :

```
$ curl -XPOST localhost:5000/parse -d '{"q":"hello there", "project": "my_restaurant_
↪search_bot"}
```

By default the latest trained model for the project will be loaded. You can also query against a specific model for a project :

```
$ curl -XPOST localhost:5000/parse -d '{"q":"hello there", "project": "my_restaurant_
↪search_bot", "model": <model_XXXXXX>}
```

`POST /train`

You can post your training data to this endpoint to train a new model for a project. This request will wait for the server answer: either the model was trained successfully or the training errored. Using the HTTP server, you must specify the project you want to train a new model for to be able to use it during parse requests later on : `/train?project=my_project`. 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?project=my_project -d @data/examples/rasa/demo-
↪rasa.json
```

You cannot send a training request for a project already training a new model (see below).

`POST /evaluate`

You can use this endpoint to evaluate data on a model. The query string takes the `project` (required) and a `model` (optional). You must specify the project in which the model is located. N.b. if you don't specify a model, the latest one will be selected. This endpoint returns some common sklearn evaluation metrics (accuracy, f1 score, precision, as well as a summary report).

```
$ curl -XPOST localhost:5000/evaluate?project=my_project&model=model_XXXXXX -d @data/
↪examples/rasa/demo-rasa.json | python -mjson.tool

{
    "accuracy": 0.19047619047619047,
    "f1_score": 0.06095238095238095,
    "precision": 0.036281179138321996,
    "predictions": [
        {
            "intent": "greet",
            "predicted": "greet",
            "text": "hey",
            "confidence": 1.0
        },
        ...,
    ]
    "report": ...
}
```

GET /status

This returns all the currently available projects, their status (`training` or `ready`) and their models loaded in memory. also returns a list of available projects the server can use to fulfill `/parse` requests.

```
$ curl localhost:5000/status | python -mjson.tool

{
  "available_projects": {
    "my_restaurant_search_bot" : {
      "status" : "ready",
      "available_models" : [
        <model_XXXXXX>,
        <model_XXXXXX>
      ]
    }
  }
}
```

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_spacy_sklearn.json",
    "data": "/app/rasa_nlu/data/examples/rasa/demo-rasa.json",
    "duckling_dimensions": null,
    "emulate": null,
    ...
}
```

2.6.4 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
```

On default CORS (cross-origin resource sharing) calls are not allowed. If you want to call your rasa NLU server from another domain (for example from a training web UI) then you can whitelist that domain by adding it to the config value `cors_origin`.

2.6.5 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.

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.

As stated previously, Rasa NLU naturally handles serving multiple apps : by default the server will load all projects found under the `path` directory defined in the configuration. The file structure under `path directory` is as follows :

- <path>
  - <project_A>
    - <model_XXXXXX>
    - <model_XXXXXX>

      ...
  - <project_B>
    - <model_XXXXXX>

      ...

    ...

So you can specify which one to use in your `/parse` requests:

```
$ curl 'localhost:5000/parse?q=hello&project=my_restaurant_search_bot'
```

or

```
$ curl -XPOST localhost:5000/parse -d '{"q":"I am looking for Chinese food", "project
↪":"my_restaurant_search_bot"}'
```

You can also specify the model you want to use for a given project, the default used being the latest trained :

```
$ curl -XPOST localhost:5000/parse -d '{"q":"I am looking for Chinese food", "project
↪":"my_restaurant_search_bot", "model":<model_XXXXXX>}'
```

If no project is to be found by the server under the `path` directory, a `"default"` one will be used, using a simple fallback model.

2.7 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.

2.7.1 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.training_data 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("sample_configs/config_spacy.json"))
trainer.train(training_data)
model_directory = trainer.persist('./projects/default/')  # Returns the directory the␣
↪model is stored in
```

2.7.2 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

# where `model_directory points to the folder the model is persisted in
interpreter = Interpreter.load(model_directory, RasaNLUConfig("sample_configs/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.

Here is a short example on how to create a component builder, that can be reused to train and run multiple models, to train a model:

```
from rasa_nlu.training_data import load_data
from rasa_nlu.config import RasaNLUConfig
from rasa_nlu.components import ComponentBuilder
from rasa_nlu.model import Trainer

builder = ComponentBuilder(use_cache=True)      # will cache components between␣
↪pipelines (where possible)

training_data = load_data('data/examples/rasa/demo-rasa.json')
trainer = Trainer(RasaNLUConfig("sample_configs/config_spacy.json"), builder)
trainer.train(training_data)
model_directory = trainer.persist('./projects/default/')  # Returns the directory the␣
↪model is stored in
```

The same builder can be used to load a model (can be a totally different one). The builder only caches components that are safe to be shared between models. Here is a short example on how to use the builder when loading models:

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

# For simplicity we will load the same model twice, usually you would want to use the␣
↪metadata of
# different models

interpreter = Interpreter.load(model_directory, config, builder)     # to use the␣
↪builder, pass it as an arg when loading the model
# the clone will share resources with the first model, as long as the same builder is␣
↪passed!
interpreter_clone = Interpreter.load(model_directory, config, builder)
```

2.8 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_crf`	crfsuite	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

The exact required packages can be found in `dev-requirements.txt` and they should also be shown when they are missing and a component is used that requires them.

To improve entity extraction, you can use regex features if your entities have a distinctive format (e.g. zipcodes). More information can be found in the *Training Data Format*.

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.

2.8.1 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 is the `ner_crf` component.

2.8.2 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_crf",
      "confidence": 0.854,
      "processors": []
    }
  ]
}
```

Note: The *confidence* will be set by the CRF entity extractor (*ner_crf* component). The duckling entity extractor will always return *1*. The *ner_spacy* extractor does not provide this information and returns *null*.

2.9 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*.

2.10 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.

2.11 Language Support

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

backend	supported languages
spacy-sklearn	english (`en`), german (`de`), spanish (`es`), portuguese (`pt`), italian (`it`), dutch (`nl`), french (`fr`)

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

2.11.1 Adding a new language

We want to make the process of adding new languages 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:

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.

2.12 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 ouput 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", "confidence": 0.864}
    ],
    "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.

2.12.1 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 setting 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", "tokenizer_spacy", "intent_entity_featurizer_regex", "intent_featurizer_spacy", "ner_crf", "ner_synonyms", "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.

2.12.2 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_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_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_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.

intent_entity_featurizer_regex

Short regex feature creation to support intent and entity classification

Outputs `text_features` and `tokens.pattern`

Description During training, the regex intent featurizer creates a list of *regular expressions* defined in the training data format. If an expression is found in the input, a feature will be set, that will later be fed into intent classifier / entity extractor to simplify classification (assuming the classifier has learned during the training phase, that this set feature indicates a certain intent). Regex features for entity extraction are currently only supported by the `ner_crf` component!

tokenizer_whitespace

Short Tokenizer using whitespaces as a separator

Outputs nothing

Description Creates a token for every whitespace separated character sequence. This tokenizer can not be used together with spaCy. spaCy has a build-in tokenizer it will use.

tokenizer_spacy

Short Tokenizer using spacy

Outputs nothing

Description Creates tokens using the spacy tokenizer.

ner_spacy

Short spacy entity extraction

Outputs appends `entities`

Output-Example

```
{
    "entities": [{"value": "New York City",
                  "start": 20,
                  "end": 33,
                  "entity": "city",
                  "confidence": null,
                  "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. This extractor does not provide any confidence scores.

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

this component will allow you to map the entities `New York City` and `NYC` to `nyc`. The entitiy extraction will return `nyc` even though the message contains `NYC`. When this component changes an exisiting 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",
                  "confidence": 0.874,
                  "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",
                  "confidence": 1.0,
                  "extractor": "ner_duckling"}]
}
```

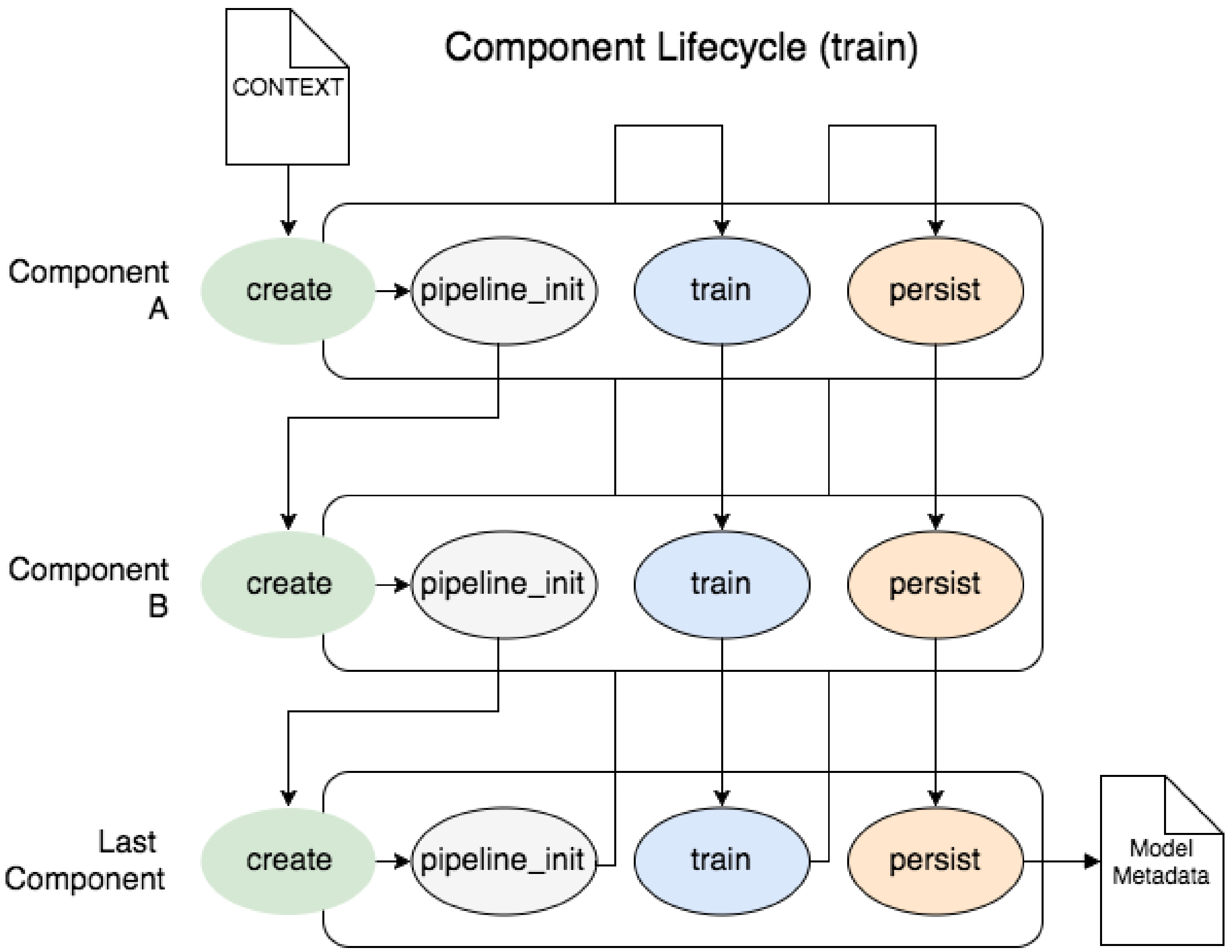
Description Duckling allows to recognize dates, numbers, distances and other structured entities and normalizes them (for a reference of all available entities see the duckling documentation). The component recognizes the entity types defined by the *duckling dimensions configuration variable*. Please 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. The extractor will always return *1.0* as a confidence, as it is a rule based system.

2.12.3 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.

2.12.4 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 model's metadata.

2.13 Evaluation

The evaluation script *evaluate.py* allows you to test your models performance for intent classification and entity recognition. You invoke this script supplying test data, model, and config file arguments:

```
python -m rasa_nlu.evaluate -d data/my_test.json -m models/my_model -c my_nlu_config.
↪json
```

If you would like to evaluate your pipeline using crossvalidation, you can run the evaluation script with the mode crossvalidation flag. This gives you an estimate of how accurately a predictive model will perform in practice. Note that you cannot specify a model in this mode, as a new model will be trained on part of the data for every crossvalidation loop. An example invocation of your script would be:

```
python -m rasa_nlu.evaluate -d data/examples/rasa/demo-rasa.json -c sample_configs/
↪config_spacy.json --mode crossvalidation
```

2.13.1 Intent Classification

The evaluation script will log precision, recall, and f1 measure for each intent and once summarized for all. Furthermore, it creates a confusion matrix for you to see which intents are mistaken for which others.

2.13.2 Entity Extraction

For each entity extractor, the evaluation script logs its performance per entity type in your training data. So if you use `ner_crf` and `ner_duckling` in your pipeline, it will log two evaluation tables containing recall, precision, and f1 measure for each entity type.

In the case `ner_duckling` we actually run the evaluation for each defined duckling dimension. If you use the `time` and `ordinal` dimensions, you would get two evaluation tables: one for `ner_duckling (Time)` and one for `ner_duckling (Ordinal)`.

`ner_synonyms` does not create an evaluation table, because it only changes the value of the found entities and does not find entity boundaries itself.

Finally, keep in mind that entity types in your testing data have to match the output of the extraction components. This is particularly important for `ner_duckling`, because it is not fitted to your training data.

Entity Scoring

To evaluate entity extraction we apply a simple tag-based approach. We don't consider BILOU tags, but only the entity type tags on a per token basis. For location entity like "near Alexanderplatz" we expect the labels "LOC" "LOC" instead of the BILOU-based "B-LOC" "L-LOC". Our approach is more lenient when it comes to evaluation, as it rewards partial extraction and does not punish the splitting of entities. For example, the given the aforementioned entity "near Alexanderplatz" and a system that extracts "Alexanderplatz", this reward the extraction of "Alexanderplatz" and punish the missed out word "near". The BILOU-based approach, however, would label this as a complete failure since it expects Alexanderplatz to be labeled as a last token in an entity (L-LOC) instead of a single token entity (U-LOC). Also note, a splitted extraction of "near" and "Alexanderplatz" would get full scores on our approach and zero on the BILOU-based one.

Here's a comparison between both different scoring mechanisms for the phrase "near Alexanderplatz tonight":

extracted	Simple tags (score)	BILOU tags (score)
[near Alexanderplatz](loc) [tonight](time)	loc loc time (3)	B-loc L-loc U-time (3)
[near](loc) [Alexanderplatz](loc) [tonight](time)	loc loc time (3)	U-loc U-loc U-time (1)
near [Alexanderplatz](loc) [tonight](time)	O loc time (2)	O U-loc U-time (1)
[near](loc) Alexanderplatz [tonight](time)	loc O time (2)	U-loc O U-time (1)
[near Alexanderplatz tonight](loc)	loc loc loc (2)	B-loc I-loc L-loc (1)

2.14 Frequently Asked Questions

2.14.1 How many training examples do I need?

Unfortunately, there is no cookie-cutter answer to this question. It depends on your intents and your entities.

If you have intents that are easily confusable, you will need more training data. Accordingly, as you add more intents, you also want to add more training examples for each intent. If you quickly write 20-30 unique expressions for each intent, you should be good for the beginning.

The same holds true for entities. the number of training examples you will need depends on how closely related your different entity types are and how clearly entities are distinguishable from non-entities in your use case.

To assess your model's performance, *run the server and manually test some messages* , or use the *evaluation script*.

2.14.2 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.

2.14.3 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.

2.14.4 Which version of rasa NLU am I running?

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

```
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.

2.14.5 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.

2.14.6 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.

2.15 Migration Guide

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

2.15.1 0.9.x to 0.10.0

- We introduced a new concept called a `project`. You can have multiple versions of a model trained for a project. E.g. you can train an initial model and add more training data and retrain that project. This will result in a new model version for the same project. This allows you to, allways request the latest model version from the http server and makes the model handling more structured.
- If you want to reuse trained models you need to move them in a directory named after the project. E.g. if you already got a trained model in directory `my_root/model_20170628-002704` you need to move that to `my_root/my_project/model_20170628-002704`. Your new projects name will be `my_project` and you can query the model using the http server using `curl http://localhost:5000/parse?q=hello%20there&project=my_project`
- Docs moved to https://rasahq.github.io/rasa_nlu/
- Renamed `name` parameter to `project`. This means for training requests you now need to pass the `project parameter instead of ``name`, e.g. `POST /train?project=my_project_name` with the body of the request containing the training data
- Adapted remote cloud storages to support projects. This is a backwards incompatible change, and unfortunately you need to retrain uploaded models and reupload them.

2.15.2 0.8.x to 0.9.x

- add `tokenizer_spacy` to trained spacy_sklearn models metadata (right after the `nlp_spacy`). alternative is to retrain the model

2.15.3 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`.

2.15.4 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"
}
```

2.16 License

```
    "Legal Entity" shall mean the union of the acting entity and all
    other entities that control, are controlled by, or are under common
    control with that entity. For the purposes of this definition,
    "control" means (i) the power, direct or indirect, to cause the
    direction or management of such entity, whether by contract or
    otherwise, or (ii) ownership of fifty percent (50%) or more of the
    outstanding shares, or (iii) beneficial ownership of such entity.

    "You" (or "Your") shall mean an individual or Legal Entity
    exercising permissions granted by this License.

    "Source" form shall mean the preferred form for making modifications,
    including but not limited to software source code, documentation
    source, and configuration files.

    "Object" form shall mean any form resulting from mechanical
    transformation or translation of a Source form, including but
    not limited to compiled object code, generated documentation,
    and conversions to other media types.

    "Work" shall mean the work of authorship, whether in Source or
    Object form, made available under the License, as indicated by a
    copyright notice that is included in or attached to the work
    (an example is provided in the Appendix below).

    "Derivative Works" shall mean any work, whether in Source or Object
    form, that is based on (or derived from) the Work and for which the
    editorial revisions, annotations, elaborations, or other modifications
    represent, as a whole, an original work of authorship. For the purposes
    of this License, Derivative Works shall not include works that remain
    separable from, or merely link (or bind by name) to the interfaces of,
    the Work and Derivative Works thereof.

    "Contribution" shall mean any work of authorship, including
    the original version of the Work and any modifications or additions
    to that Work or Derivative Works thereof, that is intentionally
    submitted to Licensor for inclusion in the Work by the copyright owner
    or by an individual or Legal Entity authorized to submit on behalf of
    the copyright owner. For the purposes of this definition, "submitted"
    means any form of electronic, verbal, or written communication sent
    to the Licensor or its representatives, including but not limited to
    communication on electronic mailing lists, source code control systems,
    and issue tracking systems that are managed by, or on behalf of, the
    Licensor for the purpose of discussing and improving the Work, but
    excluding communication that is conspicuously marked or otherwise
    designated in writing by the copyright owner as "Not a Contribution."

    "Contributor" shall mean Licensor and any individual or Legal Entity
    on behalf of whom a Contribution has been received by Licensor and
    subsequently incorporated within the Work.

 2. Grant of Copyright License. Subject to the terms and conditions of
    this License, each Contributor hereby grants to You a perpetual,
    worldwide, non-exclusive, no-charge, royalty-free, irrevocable
    copyright license to reproduce, prepare Derivative Works of,
    publicly display, publicly perform, sublicense, and distribute the
    Work and such Derivative Works in Source or Object form.
```

```
3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.
```

2.17 Community Contributions

Note: This is an (incomplete) list of external resources created by the Rasa community. We list them here because they can help you learn about the Rasa Stack, but they are not officially endorsed by Rasa and we cannot promise that they will be kept up-to-date as the project evolves.

2.17.1 Community Written Documentation

- **A three part tutorial on using Rasa NLU in combination with Node-RED to create a basic chat bot and integrate it with Sl**
  - Part 1 - Installation, Education, and Model Training
  - Part 2 - Back end fulfillment using Node-RED
  - Part 3 - A Complete Chatbot on Slack and Twilio
- **Documentation on using Rasa NLU with Docker**
  - Using Rasa NLU with Docker - The easiest way to get started working with Rasa
- Failing Gracefully with Rasa NLU

2.17.2 Community Open Source Tools/Software

Below is a list of tools and applications written around or for Rasa NLU using a permissive license.

- **Postgres backed UI for interacting with Rasa NLU**
  - Rasa UI
- **A tool for generating training examples from a list of entities**
  - Chatito

2.17.3 Video Tutorials

- Talk about the Rasa Stack at PyData

2.18 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.

2.18.1 Python Conventions

Python code should follow the pep-8 spec.

2.18.2 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.

2.18.3 Code of conduct

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

2.18.4 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 .
# python 3: python -m http.server
```

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

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

```
make doctest
```

2.19 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.

2.19.1 [Unreleased 0.12.0.aX] - master

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

Added

- support for inline entity synonyms in markdown training format
- support for regex features in markdown training format
- support for splitting and training data into multiple and mixing formats
- support for markdown files containing regex-features or synonyms only
- added ability to list projects in cloud storage services for model loading
- server evaluation endpoint at `POST /evaluate`
- CRF entity recognizer now returns a confidence score when extracting entities
- server endpoint at `DELETE /models` to unload models from server memory

Changed

- Regex features are now sorted internally. **retrain your model if you use regex features**
- The keyword intent classifier now returns `null` instead of `"None"` as intent name in the json result if there's no match
- in teh evaluation results, replaced `O` with the string `no_entity` for better understanding
- The `CRFEntityExtractor` now only trains entity examples that have `"extractor": "ner_crf"` or no extractor at all
- Ignore hidden files when listing projects or models
- Docker Images now run on python 3.6 for better non-latin character set support

Removed

- MITIE support - backend is no longer supported.

Fixed

2.19.2 [0.11.1] - 2018-02-02

Fixed

- Changelog doc formatting

- fixed project loading for newly added projects to a running server

2.19.3 [0.11.0] - 2018-01-30

Added

- non ascii character support for anything that gets json dumped (e.g. training data received over HTTP endpoint)
- evaluation of entity extraction performance in `evaluation.py`
- support for spacy 2.0
- evaluation of intent classification with crossvalidation in `evaluation.py`
- support for splitting training data into multiple files (markdown and JSON only)

Changed

- removed `-e .` from requirements files - if you want to install the app use `pip install -e .`
- fixed http duckling parsing for non `en` languages
- fixed parsing of entities from markdown training data files

2.19.4 [0.10.6] - 2018-01-02

Added

- support asterisk style annotation of examples in markdown format

Fixed

- Preventing capitalized entities from becoming synonyms of the form lower-cased -> capitalized

2.19.5 [0.10.5] - 2017-12-01

Fixed

- read token in server from config instead of data router
- fixed reading of models with none date name prefix in server

2.19.6 [0.10.4] - 2017-10-27

Fixed

- docker image build

2.19.7 [0.10.3] - 2017-10-26

Added

- support for new dialogflow data format (previously api.ai)
- improved support for custom components (components are stored by class name in stored metadata to allow for components that are not mentioned in the Rasa NLU registry)
- language option to convert script

Fixed

- Fixed loading of default model from S3. Fixes #633
- fixed permanent training status when training fails #652
- quick fix for None "_formatter_parser" bug

2.19.8 [0.10.1] - 2017-10-06

Fixed

- readme issues
- improved setup py welcome message

2.19.9 [0.10.0] - 2017-09-27

Added

- Support for training data in Markdown format
- Cors support. You can now specify allowed cors origins within your configuration file.
- The HTTP server is now backed by Klein (Twisted) instead of Flask. The server is now asynchronous but is no more WSGI compatible
- Improved Docker automated builds
- Rasa NLU now works with projects instead of models. A project can be the basis for a restaurant search bot in German or a customer service bot in English. A model can be seen as a snapshot of a project.

Changed

- Root project directories have been slightly rearranged to clean up new docker support
- use `Interpreter.create(metadata, ...)` to create interpreter from dict and `Interpreter.load(file_name, ...)` to create interpreter with metadata from a file
- Renamed `name` parameter to `project`
- Docs hosted on GitHub pages now: Documentation
- Adapted remote cloud storages to support projects (backwards incompatible!)

Fixed

- Fixed training data persistence. Fixes #510
- Fixed UTF-8 character handling when training through HTTP interface
- Invalid handling of numbers extracted from duckling during synonym handling. Fixes #517
- Only log a warning (instead of throwing an exception) on misaligned entities during mitie NER

2.19.10 [0.9.2] - 2017-08-16

Fixed

- removed unnecessary *ClassVar* import

2.19.11 [0.9.1] - 2017-07-11

Fixed

- removed obsolete `--output` parameter of `train.py`. use `--path` instead. fixes #473

2.19.12 [0.9.0] - 2017-07-07

Added

- increased test coverage to avoid regressions (ongoing)
- added regex featurization to support intent classification and entity extraction (`intent_entity_featurizer_regex`)

Changed

- replaced existing CRF library (python-crfsuite) with sklearn-crfsuite (due to better windows support)
- updated to spacy 1.8.2
- logging format of logged request now includes model name and timestamp
- use module specific loggers instead of default python root logger
- output format of the duckling extractor changed. the `value` field now includes the complete value from duckling instead of just text (so this is an property is an object now instead of just text). includes granularity information now.
- deprecated `intent_examples` and `entity_examples` sections in training data. all examples should go into the `common_examples` section
- weight training samples based on class distribution during ner_crf cross validation and sklearn intent classification training
- large refactoring of the internal training data structure and pipeline architecture
- numpy is now a required dependency

Removed

- luis data tokenizer configuration value (not used anymore, luis exports char offsets now)

Fixed

- properly update coveralls coverage report from travis
- persistence of duckling dimensions
- changed default response of untrained `intent_classifier_sklearn` from `"intent": None` to `"intent": {"name": None, "confidence": 0.0}`
- `/status` endpoint showing all available models instead of only those whose name starts with *model*
- properly return training process ids #391

2.19.13 [0.8.12] - 2017-06-29

Fixed

- fixed missing argument attribute error

2.19.14 [0.8.11] - 2017-06-07

Fixed

- updated mitie installation documentation

2.19.15 [0.8.10] - 2017-05-31

Fixed

- fixed documentation about training data format

2.19.16 [0.8.9] - 2017-05-26

Fixed

- properly handle response_log configuration variable being set to `null`

2.19.17 [0.8.8] - 2017-05-26

Fixed

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

2.19.18 [0.8.7] - 2017-05-24

Fixed

- Fixed range calculation for crf #355

2.19.19 [0.8.6] - 2017-05-15

Fixed

- Fixed duckling dimension persistence. fixes #358

2.19.20 [0.8.5] - 2017-05-10

Fixed

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

2.19.21 [0.8.4] - 2017-05-10

Fixed

- Fixed CRF model training without entities. fixes #345

2.19.22 [0.8.3] - 2017-05-10

Fixed

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

2.19.23 [0.8.2] - 2017-05-08

Fixed

- deepcopy of context #343

2.19.24 [0.8.1] - 2017-05-08

Fixed

- NER training reuses context inbetween requests

2.19.25 [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
- added alternative method of defining entity synonyms
- 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

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

2.19.26 [0.7.4] - 2017-03-27

Fixed

- fixed failed loading of example data after renaming attributes, i.e. "KeyError: 'entities'"

2.19.27 [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

2.19.28 [0.7.2] - 2017-03-13

Fixed

- python documentation about calling rasa NLU from python

2.19.29 [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

2.19.30 [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

2.19.31 [0.6-beta] - 2017-01-31