# RASA

## RASA STACK

Rasa Stack是一套开源人工智能对话框架，用于开发者创建上下文相关的AI助手和聊天机器人。Rasa Stack让开发者可以扩展机器人，而不仅仅是用最少的训练数据回答简单的问题。

Rasa Stack 包括 Rasa NLU 和 Rasa Core，前者负责进行语义理解（意图识别和槽值提取），而后者负责会话管理，控制跟踪会话并决定下一步要做什么，两者都使用了机器学习的方法可以从真实的会话数据进行学习；另外他们之间还相互独立，可以单独使用。

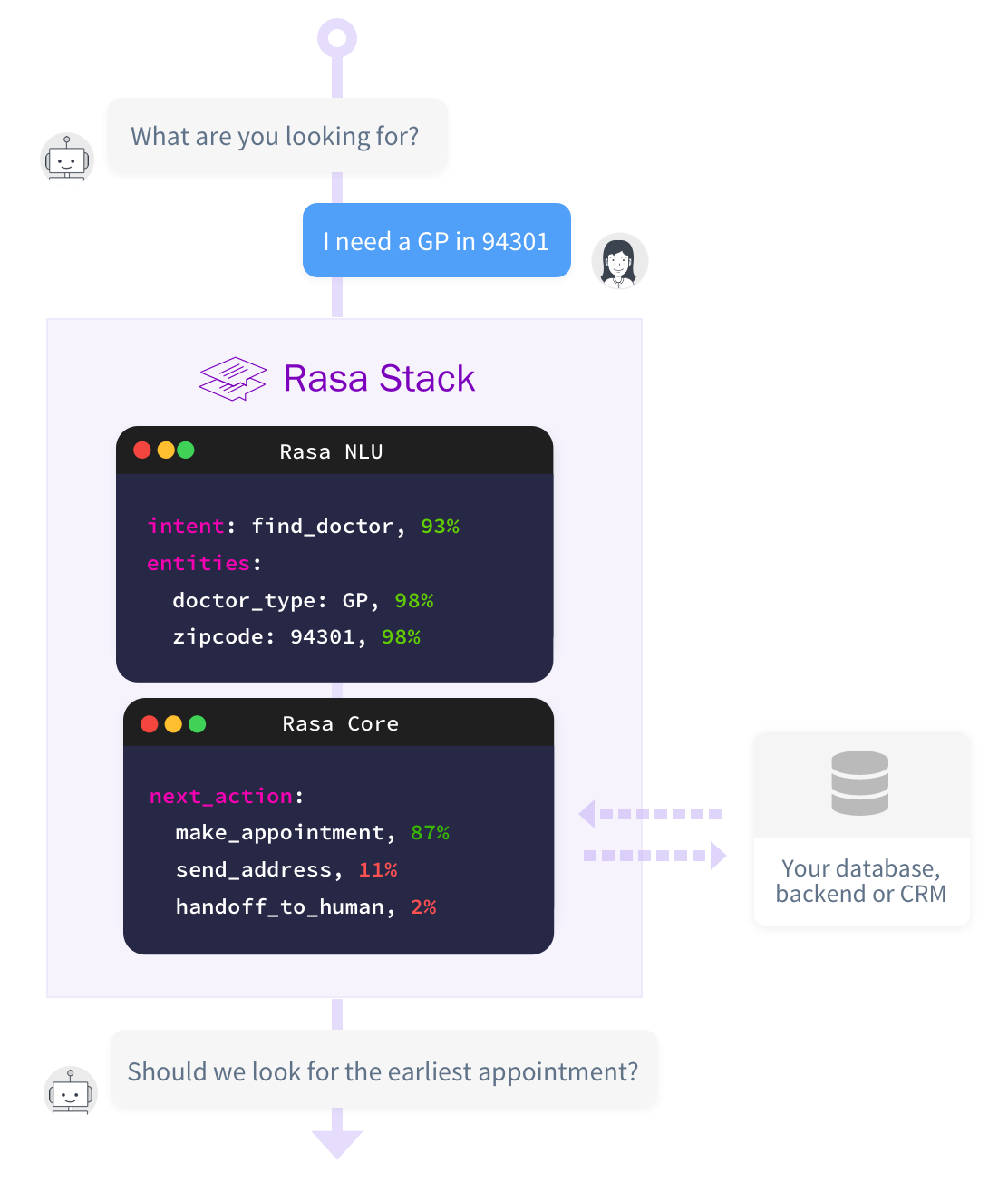
术语解释：

* 意图： 将用户的诉求告知机器。
  + 例如：提出投诉，要求退款等请求。
* 实体： 提取用户诉求细节的属性。
  + 例如：与服务中断、退款等有关的投诉
* 置信度： 一个距离指标，该指标能够体现出NLU分析出的结果与意图列表中诉求相差多少。

下例子能够帮助你更好的理解上述概念。

**输入**：“今天早上我的互联网一直无法使用。”

* **意图**： 服务中断
* **实体**： “服务=互联网”， “持续时间=整个上午”
* **置信度**：0.84（可能根据个人培训方式不同而异）



1. NLU根据训练数据来理解用户的message。

意图分类（Intent classification）:基于预定义的intent解释含义。（例如:我需要94301中的GP是一个find\_doctor意图，有93%的信心）

实体提取（Entity extraction）:识别结构化数据。（例如:GP是doctor\_type和94301 a zipcode）

1. RASA Core决定对话接下来的动作。

基于机器学习的对话管理根据NLU的输入、对话历史和训练数据预测下一个最佳动作。(例如:Core有87%的信心认为make\_appointment是下一个最佳操作,用来与用户确认是否要更改其主要联系信息）。

## 工作流程

使用RASA NLU教Bot理解用户输入消息

### **创建NLU例子**

NLU模型将以一种简单的文本格式接收输入并提取结构化数据，这种结构化数据称为intents，将帮助Bot理解用户消息。

通过定义意图并提供一些用户可能表达意图的方式，来帮助Bot理解用户消息。

将Rasa NLU训练示例:

nlu\_md = """

## intent:greet

- hey

- hello

- hi

- good morning

- good evening

- hey there

## intent:goodbye

- bye

- goodbye

- see you around

- see you later

## intent:mood\_affirm

- yes

- indeed

- of course

- that sounds good

- correct""" %store nlu\_md > nlu.md

### **NLU模型配置**

配置NLU模型是为了定义NLU模型如何被训练以及如何从text输入中提取特征。下面使用预定义的tensorflow\_embedding管道，保存在nlu\_config.yml文件中。

nlu\_config = """

language: en

pipeline: tensorflow\_embedding

"""

%store nlu\_config > nlu\_config.yml

### **训练NLU模型**

训练名字为rasa.nlu的模型：

!python -m rasa\_nlu.train -c nlu\_config.yml --data nlu.md -o models --fixed\_model\_name nlu --project current --verbose

### **测试模型**

下面加载rasa.nlu并返回消息(内容hello)的intent分类结果

from rasa\_nlu.model import Metadata, Interpreter

import json

def pprint(o):

# small helper to make dict dumps a bit prettier

print(json.dumps(o, indent=2))

interpreter = Interpreter.load('./models/current/nlu')

pprint(interpreter.parse(u"Hello"))

使用RASA CORE教Bot响应用户

### **写故事**

RASA CORE训练对话管理模型并预测Bot在对话的特定状态时如何作出响应。

Rasa Core模型以训练“故事”的形式从真实的会话数据中学习。故事是用户和机器人之间的真实对话，其中用户输入表示为Intent，Bot的响应表示为Action。下面是一个简单对话的例子：用户向我们的机器人问好，然后机器人问好。

## story1

\* greet

- utter\_greet

注：

故事以##开头，后跟名称(可选)；

\*后跟用户发送的消息，不是真是消息而是表示用户消息的intent。

-跟Bot需要采取的动作。此情形所有的Action仅仅是发送给用户的消息。

通常Action可以做任何任务，包括API调用以及和外部系统进行交互。

保存示例story在stories.md

stories\_md = """

## happy path

\* greet

- utter\_greet

\* mood\_great

- utter\_happy

## sad path 1

\* greet

- utter\_greet

\* mood\_unhappy

- utter\_cheer\_up

- utter\_did\_that\_help

\* mood\_affirm

- utter\_happy

"""

%store stories\_md > stories.md

### **定义Domain**

Domain定义了Bot所处的世界 - 应该获得的用户输入，应该能够预测的行为，如何响应以及要存储的信息。这是bot的域示例，需要写入domain.yml文件。

domain\_yml = """

intents:

- greet

- goodbye

- mood\_affirm

- mood\_deny

- mood\_great

- mood\_unhappy

actions:

- utter\_greet

- utter\_cheer\_up

- utter\_did\_that\_help

- utter\_happy

- utter\_goodbye

templates:

utter\_greet:

- text: "Hey! How are you?"

utter\_cheer\_up:

- text: "Here is something to cheer you up:"

image: "https://i.imgur.com/nGF1K8f.jpg"

utter\_did\_that\_help:

- text: "Did that help you?"

utter\_happy:

- text: "Great carry on!"

utter\_goodbye:

- text: "Bye"

"""

%store domain\_yml > domain.yml

|  |  |
| --- | --- |
| intents | things you expect users to say. |
| actions | things your bot can do and say |
| templates | template strings for the things your bot can say |

Rasa Core的工作是在对话的每个步骤中选择正确的action来执行。简单的Actions就是在域中的Action,只是向用户发送一条消息，以utter\_开头。它们只会根据templates部分中的模板返回一条消息。

### **训练对话模型**

下一步就是在示例story上训练神经网络。

!python -m rasa\_core.train -d domain.yml -s stories.md -o models/dialogue

该命令将调用RASA CORE的train函数，传入domain和stories参数，训练模型被保存在models/dialogue目录。这个命令的输出是每一个epoch的训练结果。

### **与Bot交谈**

如果有自定义action，请执行此命令，注意此处action\_name为自定义action模块名，如果你的action文件叫做action.py，则写--actions action

python -m rasa\_core\_sdk.endpoint --actions action\_name

启动对话

python -m rasa\_core.run -d models/dialogue -u models/current/nlu --endpoints endpoints.yml

# RASA NLU

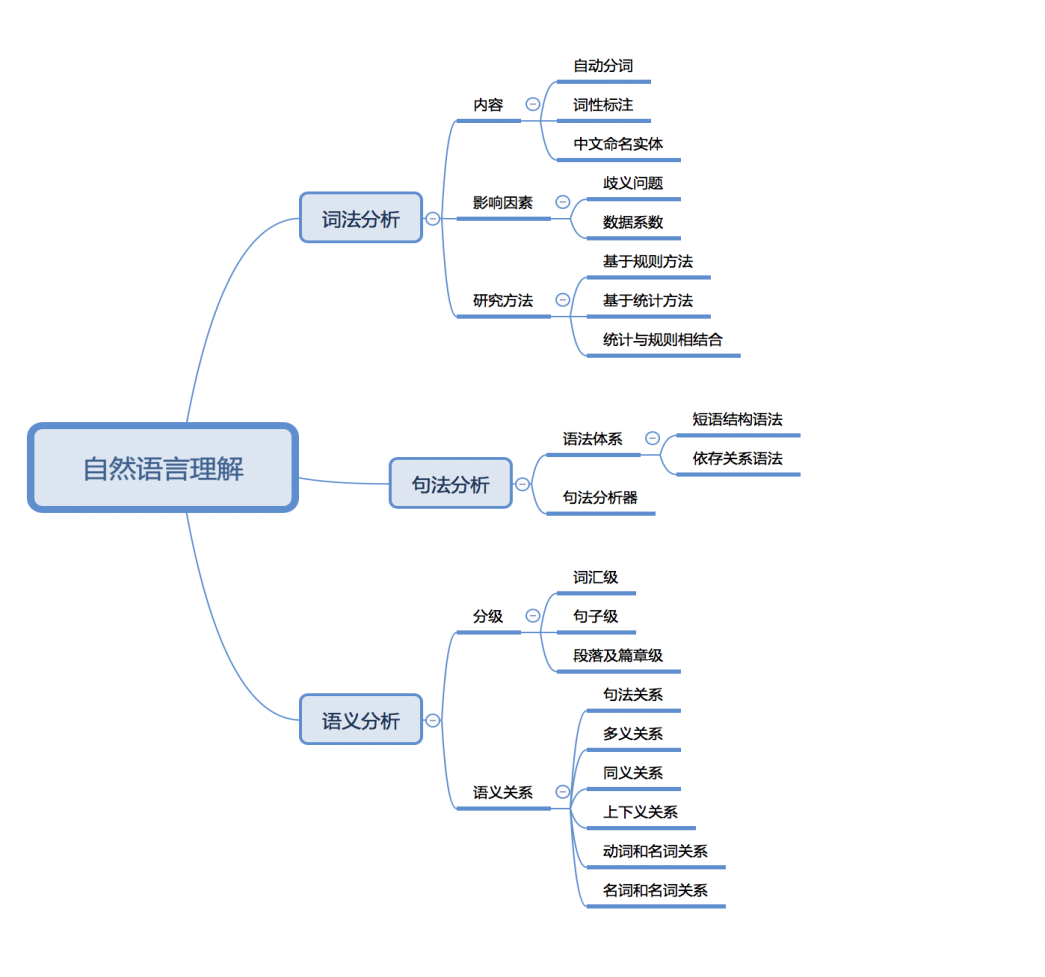
## NLU

自然语言理解（NLU）系统是问答系统、聊天机器人等更高级应用的基石。

基本的NLU工具，包括实体识别和意图识别两个任务。

目前NLU工具，大多是以服务的方式，通过调用远程http的restful API来对目标语句进行解析完成上述两个任务。例如Google的[API.ai](http://api.ai/), Microsoft的[Luis.ai](http://luis.at/), Facebook的[Wit.ai](http://wit.ai/)，被百度收购的[Kitt.ai](http://kitt.ai/)。

NLU的3个层次：词法分析、句法分析、语义分析。



### **词法分析**

词法分析是自然语言处理的技术基础，也是自然语言理解过程的第一层，因此词法分析的性能直接影响到后面句法和语义分析的成果。主要包括自动分词、词性标注、中文命名实体标注三方面内容。

1. 自动分词

现有分词的算法分为三大类：基于词典的分词方法、基于统计的分词方法、基于理解的分词方法。

当前主流的方法还是基于词典进行分词，主要包括正向最大匹配、逆向最大匹配、双向最大匹配。原理是按照既定的规则顺序，将目标字符串依次与词典匹配，匹配成功就取出该词，直到整个字符串全部匹配，如在词典中匹配到，就取出单字。

1. 词性标注

词性标注是对分词结果中的每个单词标注一个正确的词性，例如：每个词是名词、动词还是形容词等。汉语中大多词语只有一个词性，或者出现频次最高的词性远远高于第二位的词性。

因此在词性标注时，一般先针对已存在的词库进行统计学处理，建立词性标注模型，进而通过概率判断每个词的词性。

1. 中文命名实体

命名实体就是将文本中的元素分成预先定义的类，例如：人名、地名、时间、百分比等。它的技术方法主要分为基于规则和词典、基于统计、二者结合的方法。

**基于规则和词典的方法**，大多是由语言学专家构造规则模板然后进行匹配。词典和知识库的创建会直接影响命名实体的准确率。

例如：人名=【姓氏】+【名字】，分别建立“姓氏”、“名字”库，如字串命中，则识别出包含人名实体。

**基于统计的方法**，主要是通过对训练语料所包含的语言信息进行统计和分析，从语料中挖掘出特征。这种方法对语料库的依赖比较大，但是用来建设和评估命名实体识别系统的大规模通用语料库比较少。

### **句法分析**

句法分析的目标是自动推导出句子的句法结构，实现这个目标首先要确定语法体系，不同的语法体系会产生不同的句法结构。常见语法体系有短语结构语法、依存关系语法。

* 依存关系语法

同样分为基于规则和基于统计的两种方法，基本自然语言的技术中，很多都是基于“词典／规则”+“统计”的方法。

（1）基于规则的方法

优点在于：可以最大限度的接近自然语言的句法习惯、表达方式灵活多样，可以最大限度的表达研究人员的思想；

缺点在于：规则刻画的知识粒度难以确定，无法确保规则的一致性，获取规则同样是一个繁琐的过程。

（2）基于统计的方法

目前是句法分析的主流技术，确定语法体系后，需要按照语法体系人工标注句子的语法结构，将其作为训练的语料。因此语料库的建设是非常关键的。

### **语义分析**

语义分析就是指分析话语中所包含的含义，根本目的是理解自然语言。分为词汇级语义分析、句子级语义分析、段落／篇章级语义分析，即分别理解词语、句子、段落的意义。

## Rasa NLU

Rasa NLU本身是只支持英文和德文的。中文因为其特殊性需要加入特定的tokenizer作为整个流水线的一部分，可使用jieba的中文tokenizer。

### **PIPLine**

* **RASA中的pipline**

处理管道是Rasa NLU模型的主要构建块。它定义了从传入用户消息到生成模型输出必须经历的处理阶段。这些阶段可能是标记化，特征化，意图分类，实体提取，模式匹配等。默认情况下，Rasa NLU附带了一堆预构建的组件（甚至是完全设计的管道）供您使用。下面的 pipeline 为 MITIE+Jieba+sklearn， rasa nlu 的配置文件为 config\_jieba\_mitie\_sklearn.yml如下：

language: "zh"

pipeline:

- name: "nlp\_mitie"

model: "data/total\_word\_feature\_extractor\_zh.dat" // 加载 mitie 模型

- name: "tokenizer\_jieba" // 使用 jieba 进行分词

- name: "ner\_mitie" // mitie 的命名实体识别

- name: "ner\_synonyms"

- name: "intent\_entity\_featurizer\_regex"

- name: "intent\_featurizer\_mitie" // 特征提取

- name: "intent\_classifier\_sklearn" // sklearn 的意图分类模型

定义管道后，将逐个调用每个组件并生成输出，该输出可直接添加到Rasa NLU模型输出中，也可用作其他组件的输入。

请务必记住，配置文件中定义组件的方式很重要

rasa nlu 支持不同的 Pipeline，其后端实现可支持 spaCy、MITIE、MITIE + sklearn 以及 tensorflow，其中 spaCy 是官方推荐的，另外值得注意的是从 0.12 版本后，MITIE 就被列入 Deprecated 了。

Rasa基本上提供了一个在各种NLP和ML库之上的高层次的API来负责”意图”的分类和“实体”的提取，这些NLP和ML库则被称为后端，是他们使得Rasa变得智能化。

以下是一些通常与Rasa一起使用的后端：

MITIE： 一个包罗万象的库; 换言之，它有一个内置的用于”实体”提取的NLP库以及一个用于”意图”分类的ML库。

spaCy + sklearn： spaCy是一个只进行”实体”提取的NLP库。而sklearn是与spaCy一起使用的，用于为其添加ML功能来进行”意图”分类操作。

MITIE + sklearn： 该组合使用了两个各自领域里最好的库。该组合既拥有了MITIE中良好的”实体”识别能力又拥有sklearn中的快速和优秀的”意图”分类。

* **选择pipline和配置**

1. short answer

当训练示例少于1000时，使用spacy模型

**language**: "en"

**pipeline**: "pretrained\_embeddings\_spacy"

当有1000个或更多标记的话语时，使用supervised\_embeddings管道

**language**: "en"

**pipeline**: "supervised\_embeddings"

supervised\_embeddings和pretrained\_embeddings\_spacy两个管道之间最大的区别在于，pretrained\_embeddings\_spacy使用来自GloVe或fastText的预先训练的词向量。相反，supervised\_embeddings不使用任何预先训练的单词向量，而是专门为你的数据集拟合这些。 pretrained\_embeddings\_spacy管道的优点是，例如训练示例，“我想买苹果”，在要求Rasa预测“获取梨子”的意图时，模型已经知道“苹果”和“梨“非常相似。如果没有很多训练数据，这将特别有用。

supervised\_embeddings管道的优点是你可以为域自定义单词向量。例如，通常“平衡”一词与“对称”密切相关，但与“现金”一词非常不同。在银行业领域，“平衡”和“现金”密切相关，你希望模型能够捕捉到这一点。此管道不使用特定于语言的模型，因此可以使用任何被标记的语言（在空格上或使用自定义标记生成器）。

注意：意图分类与实体提取无关。因此，有时NLU会获得正确的意图，但实体却是错误的，或者相反。您需要为意图和实体提供足够的数据。

1. 多重意图

如果要将意图分成多个标签，例如对于预测多个意图或建模分层意图结构，只能使用supervised embeddings管道。

为此，请使用以下标志：

intent\_tokenization\_flag如果为true，算法会将intent labels拆分为tokens，并为它们使用bag-of-words 表示;

intent\_split\_symbol设置分隔符字符串以拆分intent labels。默认\_

示例：

**language**: "en"

**pipeline**:

- **name**: "CountVectorsFeaturizer"

- **name**: "EmbeddingIntentClassifier"

**intent\_tokenization\_flag**: true

**intent\_split\_symbol**: "+"

* **加深Pipline的理解**

在Rasa NLU中， incoming messages由一系列组件处理。这些组件在所谓的处理pipline中一个接一个地执行。有实体提取，意图分类，预处理等组件。如果想添加自己的组件，例如运行拼写检查或进行情绪分析，请查看section\_customcomponents。

每个组件处理input并构建output。output可以被pipline中该组件后面的任何组件使用。有些组件只生产被pipline中其他组件使用的信息，还有其他组件生成output属性，这些属性将在处理完成后返回。

例如，对于“我正在寻找中国食物”的句子，输出是：

{

**"text"**: "I am looking for Chinese food",

**"entities"**: [

{**"start"**: 8, **"end"**: 15, **"value"**: "chinese", **"entity"**: "cuisine", **"extractor"**: "CRFEntityExtractor", **"confidence"**: 0.864}

],

**"intent"**: {**"confidence"**: 0.6485910906220309, **"name"**: "restaurant\_search"},

**"intent\_ranking"**: [

{**"confidence"**: 0.6485910906220309, **"name"**: "restaurant\_search"},

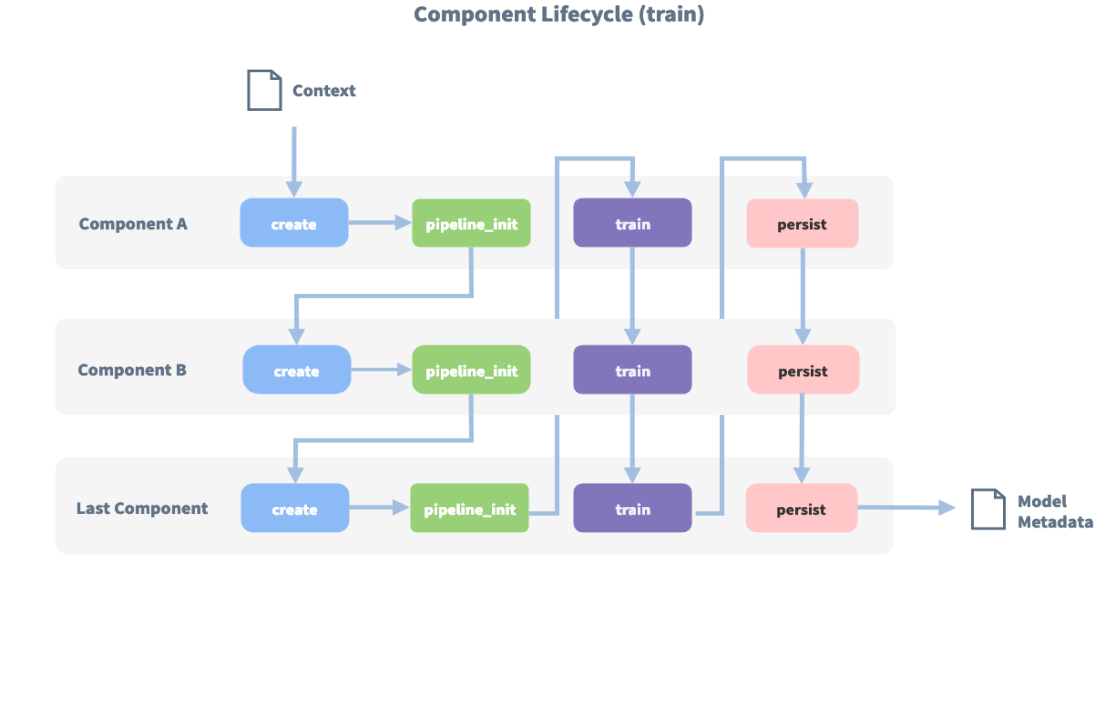
{**"confidence"**: 0.1416153159565678, **"name"**: "affirm"}

]}

这是作为预配置管道pretrained\_embeddings\_spacy中的不同组件的结果的组合而创建的。例如，entities属性由CRFEntityExtractor组件创建。

* **组件生命周期**

每个组件都需要实现Component基类的几个方法;在管道中，将按特定顺序调用这些不同的方法。假设，在配置中添加了以下管道：“管道”："pipeline": ["Component A", "Component B", "Last Component"]。该图显示了此管道培训期间的呼叫顺序：



在创建第一个组件之前调用create函数会创建一个所谓的context（它仅仅是一个python dict）。此context用于在组件之间传递信息。例如，一个组件可以计算训练数据的特征向量，将其存储在上下文中，另一个组件可以从上下文中检索这些特征向量并进行意图分类。

最初，上下文填充了所有配置值，图像中的箭头显示调用顺序并可视化传递的上下文的路径。在训练并持久化所有组件之后，使用最终context dict来持久化模型的元数据。

* **返回实体对象**

在解析返回的对象后，有两个字段显示pipline是如何影响返回的entities。实体的提取器字段extractor告诉您哪个实体提取器找到了该特定实体。处理器字段processors 包含更改此特定实体的组件的名称。

同义词的使用也可能导致value字段与text不精确匹配。相反，它将返回训练有素的同义词。

{

**"text"**: "show me chinese restaurants",

**"intent"**: "restaurant\_search",

**"entities"**: [

{

**"start"**: 8,

**"end"**: 15,

**"value"**: "chinese",

**"entity"**: "cuisine",

**"extractor"**: "CRFEntityExtractor",

**"confidence"**: 0.854,

**"processors"**: []

}

]}

注意：置信度将由CRF实体提取器（CRFEntityExtractor组件）设置。duckling 实体提取器将始终返回1. SpacyEntityExtractor提取器不提供此信息并返回null。

* **预配置的管道**

模板只是完整组件列表的快捷方式。例如，这两种配置是等效的：

**language**: "en"

**pipeline**: "pretrained\_embeddings\_spacy"

和

**language**: "en"

**pipeline**:

- **name**: "SpacyNLP"

- **name**: "SpacyTokenizer"

- **name**: "SpacyFeaturizer"

- **name**: "RegexFeaturizer"

- **name**: "CRFEntityExtractor"

- **name**: "EntitySynonymMapper"

- **name**: "SklearnIntentClassifier"

详细模板见：<https://rasa.com/docs/nlu/choosing_pipeline/>

定制pipline:

不必使用模板，可以通过列出要使用的组件的名称来运行完全自定义管道：

**pipeline**:

- **name**: "SpacyNLP"

- **name**: "CRFEntityExtractor"

- **name**: "EntitySynonymMapper"

* **changelog**

rasa-nul从v0.15.0版本：

tensorflow\_embedding管道现在称为supervised\_embeddings，spacy\_sklearn现在称为pretrained\_embeddings\_spacy

### **置信度**

每个管道将报告置信度分数以及预测的意图，CRFEntityExtractor组件将对提取的实体执行相同的操作。

可以使用置信度分数选择何时忽略Rasa NLU的预测和触发回退行为，例如要求用户重新措辞。如果使用的是Rasa Core，则可以使用后备策略执行此操作。

* 选择信心隔断(Choosing a Confidence Cutoff)

A good way to choose a confidence cutoff is to calculate the model’s confidence on a test set, and compare the confidence values on the correctly and incorrectly predicted examples.

选择置信度隔断的一种好方法是在测试集计算模型的置信度，并比较预测示例的正确和错误的置信度值。

* 置信度分数

置信度得分不是预测正确的真实概率，它只是由模型定义的指标，大致描述了输入与训练数据的相似程度。

例如，pretrained\_embeddings\_spacy管道中的意图分类器通常报告非常低的置信度，而supervised\_embeddings管道通常提供非常高的置信度。一个常见的误解是，如果模型对训练示例有很高的置信度，那么它就是一个“更好”的模型。实际上，这通常意味着模型过度拟合。

### **实体(Entity)**

* 实体提取

CRFEntityExtractor是您使用自己的数据训练的唯一实体提取器，因此是唯一将被评估的实体提取器。如果使用spaCy或duckling预训练实体提取器，Rasa NLU将不会在评估中包含这些。 Rasa NLU将报告CRFEntityExtractor经过训练后识别每种实体类型的召回率，精确度和f1度量。

* 实体得分

为了评估实体提取，我们应用一种简单的基于标签的方法。不考虑BILOU标记，只考虑每个标记的实体类型标记。例如位置实体“near Alexanderplatz” ，我们期望标签LOC LOC而不是基于BILOU的B-LOC L-LOC。我们的方法在评估时更宽松，因为它奖励部分提取并且不惩罚实体的分裂。例如，实体“near Alexanderplatz”和提取“Alexanderplatz”的系统，这奖励提取“Alexanderplatz”并惩罚错过“near”。

| **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) |

然而，基于BILOU的方法将此标记为完全失败，因为它期望Alexanderplatz被标记为实体（L-LOC）中的最后一个标记而不是单个标记实体（U-LOC）。另外请注意，“near”和“Alexanderplatz”的分割提取将在我们的方法上获得满分，但在基于BILOU的方法上获得零分。

注：BILOU是一种对给定句子中的单元做序列标注的方式，用于从给定句子中抽取连续字/词块构成的有意义短语，例如名词短语（noun phrases, NP）、命名实体（named entites, NE）等。对于一个给定句子，将其中每个词标注为B（Beginning，指示某短语起始）、I（Inside）、O（Outside，指示不在短语中）,L(Last 最后一个，相当于End),U(Unit,一个单独单词)。

## 自定义组件增强NLU模型

使用自己的自定义组件（情绪分析器，拼写检查器，字符级标记器，字节对编码器等）来增强现有的Rasa NLU模型。

使用预先构建的Rasa NLU组件可以自由地定制模型。但是，在某些情况下，可能希望添加一个未在Rasa NLU中本地实现的组件。例如，添加一个情绪分析器，使AI助手可以根据用户的心情使用不同的响应，或者添加拼写检查器来纠正用户消息中的拼写错误，然后才能对意图进行分类并提取实体并用于进行API调用或在数据库中查找记录。将自定义组件添加到NLU管道是使用必要方法实现自定义组件类并在Rasa NLU管道配置文件中引用它的过程。通常，您可能希望使用两种类型的自定义组件

## 配置

### **训练数据格式**

Rasa NLU的训练数据分为不同部分：

示例，同义词，正则表达式功能和查找表。

同义词将提取的实体映射到相同的名称，例如将“我的储蓄账户”映射到简单的“储蓄”。但是这仅在提取实体后才会发生，因此您需要提供包含同义词的示例，以便Rasa可以学习如何选择它们。

查找表可以直接指定为列表，也可以指定为包含换行符分隔的单词或短语的txt文件。加载训练数据后，这些文件将用于生成添加到正则表达式功能的不区分大小写的正则表达式模式。例如，在这种情况下，提供货币名称列表，以便更容易选择此实体。

MARKDOWN格式：

**##** 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 [**savings account**](source\_account:savings) <!-- synonyms, method 1-->

**-** Could I pay in [**yen**](currency)? <!-- entity matched by lookup table -->

**##** intent:greet

**-** hey

**-** hello

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

**-** pink pig

**##** regex:zipcode

**-** [0-9]{5}

**##** lookup:currencies <!-- lookup table list -->

**-** Yen**-** USD**-** Euro

**##** lookup:additional\_currencies <!-- no list to specify lookup table file -->

path/to/currencies.txt

JSON格式：

{

**"rasa\_nlu\_data"**: {

**"common\_examples"**: [],

**"regex\_features"** : [],

**"lookup\_tables"** : [],

**"entity\_synonyms"**: []

}}

JSON格式包含一个顶级对象，包括键common\_examples,entity\_synonyms，regex\_features和 common\_examples。

common\_examples用于训练您的模型。您应该将所有训练示例放在common\_examples数组中。正则表达式功能是一种帮助分类器检测实体或意图并提高性能的工具。

Common Examples：

{

**"text"**: "show me chinese restaurants",

**"intent"**: "restaurant\_search",

**"entities"**: [

{

**"start"**: 8,

**"end"**: 15,

**"value"**: "chinese",

**"entity"**: "cuisine"

}

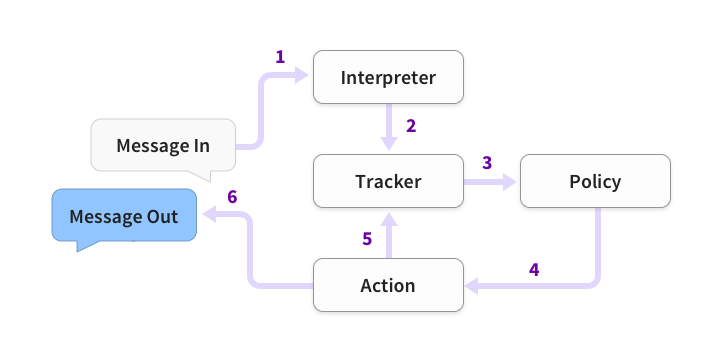
]}

训练数据文件可以存储在一个文件中，也可以在多个文件中，文件位于nlu\_data目录下。

将训练数据拆分为多个文件目前仅适用于markdown和JSON数据。对于其他文件格式，必须使用单文件方法。不能混合markdown和json。

# RASA CORE对话引擎

## 架构



流程介绍：

1. 系统接收Message并传递给解释器，解释器负责将消息转换为一个字典，内容有原始文本，意图(intent)和所有实体(entity)。
2. 跟踪器负责跟踪对话状态的对象。它接收新消息进入时所持的信息。
3. 策略负责接收跟踪器的当前状态。
4. 策略选择下一步的动作。
5. 跟踪器记录下被选中的动作。
6. 系统将相应消息发送给用户。

## 聊天&语音平台

输入通道在模块rasa\_core.channels中定义。目前系统中的输入通道：

facebook, slack, telegram, mattermost and twilio

管理凭证

为了连接通道，需要凭证(或API Token)。

在文件credentials.yml中定义。例如：

twilio:

account\_sid: "ACbc2dxxxxxxxxxxxx19d54bdcd6e41186"

auth\_token: "e231c197493a7122d475b4xxxxxxxxxx"

twilio\_number: "+440123456789"

slack:

slack\_token: "xoxb-286425452756-safjasdf7sl38KLls"

slack\_channel: "@my\_channel"

telegram:

access\_token: "490161424:AAGlRxinBRtKGb21\_rlOEMtDFZMXBl6EC0o"

verify: "your\_bot"

webhook\_url: "your\_url.com/webhook"

mattermost:

url: "https://chat.example.com/api/v4"

team: "community"

user: "user@user.com"

pw: "password"

facebook:

verify: "rasa-bot"

secret: "3e34709d01ea89032asdebfe5a74518"

page-access-token: "EAAbHPa7H9rEBAAuFk4Q3gPKbDedQnx4djJJ1JmQ7CAqO4iJKrQcNT0wtD"

webexteams:

access\_token: "ADD-YOUR-BOT-ACCESS-TOKEN"

room: "YOUR-WEBEXTEAMS-ROOM-ID"

|  |  |  |
| --- | --- | --- |
| **通道名称** | **配置** | **说明** |
| Facebook | 1. 设置Facebook Messenger 和 page 2. 运行方式：脚本运行和python直接连接两种 |  |
| Cisco Webex Teams |  |  |
| Slack | 1、安装Slack app |  |
| mattermost |  |  |
| telegram |  |  |
| Twilio |  |  |
| Rocket.Chat |  |  |
| [Microsoft Bot Framework](https://rasa.com/docs/core/connectors/" \l "id53) |  |  |
| [SocketIO](https://rasa.com/docs/core/connectors/" \l "id56) |  |  |
| [REST Channels](https://rasa.com/docs/core/connectors/" \l "id60) |  | 在网页中防止小部件与机器人交谈 |
| [Ngrok](https://rasa.com/docs/core/connectors/" \l "id59) |  |  |
| custom channel |  | 自定义channel |

## Actions

Actions是Bot对用户输入的响应，在Rasa Core中有3种Action:

|  |  |  |
| --- | --- | --- |
| 类别 | 内容 | 说明 |
| default actions | action\_listen, action\_restart, action\_default\_fallback |  |
| utter actions | action\_endpoint:  url: "http://localhost:5055/webhook" | utterance template(话语模版)以utter\_开头 |
| Custom Actions | action\_endpoint:  url: "http://localhost:5055/webhook" | 使用--endpoints endpoints.yml传递给脚本。  可以使用node.js,.NET,java或其他任何语言构建action server,系统提供python sdk使自定义action更容易。 |

注：不需要为aciton server安装rasa\_core,建议在一个docker容器中部署rasa\_core，为action server另创建一个docker容器,该容器只需要安装rasa\_core\_sdk。

在action.py中定义action,执行如下命令：

python -m rasa\_core\_sdk.endpoint --actions actions

## Using Slots

Slots是bot的存储，用于存储用户提供的信息（例如用户姓名，家乡等信息）以及关于外部世界收集的信息（例如数据库查询的结果）。

插槽会影响对话的进展，不同的行为有不同的插槽类型。

RASA如何使用Slots?

rasa\_core.policies.Policy无权访问您的插槽的值，它接收一个特征表示。例如，一个文本槽值是无关紧要的，策略值是1或0，具体取决于它是否已设置。

在domain.yml可以设置slots初始值。

slots:

name:

type: text

initial\_value: "human"

在对话期间有多种方式设置插槽。

从NLU设置Slots:若NLU模型选择了一个实体，并且域包含一个具有相同名称的插槽，则会自动设置该插槽。

**# story\_01**

\* greet{**"name"**: "Ali"}

- slot{**"name"**: "Ali"}

- utter\_greet

## Slot filling

最常见的对话模式之一是从用户收集一些信息以便做某事（预订餐馆，调用API，搜索数据库等）,称为槽填充。

如果需要连续收集多条信息，建议创建一个FormAction。这是一个单一操作，其中包含循环所需插槽的逻辑，并询问用户此信息。

Domain格式：

forms:

- my\_form

...

表单基础：

以下面Bot为例，单个对话描述所有的happy path:

**## happy path**

\* request\_restaurant

- restaurant\_form

- form{"name": "restaurant\_form"}

- form{"name": null}

用户intent是request\_restaurant，后面是表单动作restaurant\_form。使用form{“name”：“restaurant\_form”}激活表单,并使用form{“name”：null}再次停用表单。

## Bot Response

如果希望机器人响应用户消息，就需要管理机器人响应。

在Bot的训练数据中，可以指定机器人应该执行的操作。这些操作可以使用utterances将消息发送回用户。

管理utterances有两种方法：

1、在域文件（domain.yml）中包含

2、使用外部服务生成响应

在domain.yml中定义utterances。

*# all hashtags are comments :)*intents:

- greet

- default

- goodbye

- affirm

- thank\_you

- change\_bank\_details

- simple

- hello

- why

- next\_intent

entities:

- name

slots:

name:

type: text

templates: templates包含Bot返回给用户的信息

utter\_greet:

- "hey there {name}!" *# {name} will be filled by slot (same name) or by custom action code*

utter\_goodbye:

- "goodbye

- "bye bye *# multiple templates - bot will randomly pick one of them*

utter\_default:

- "default message"

actions:

- utter\_default

- utter\_greet

- utter\_goodbye

如果更改文本或Bot 响应的任何其他部分，需要重新训练Bot才能获取这些更改。

用外部CMS管理Bot utterances

对于某些工作流程，重新训练机器人，仅更改文本可能不是最理想的。所以Core还允许您将响应生成外包并将其与对话学习分开。

Bot根据过去的对话学习并预测action,并对用户输入作出反应，但发送回用户的响应是在Rasa Core之外生成的。

如果Bot想要向用户发送消息，它将使用POST请求调用外部HTTP服务器。要配置此端点，需要创建endpoints.yml并将其传递给运行或服务器脚本。

## 交互学习(interactive learning)

在交互式学习模式中，您可以在与之交谈时向机器人提供反馈。这是探索机器人可以做什么的有效方法，也是解决它所犯错误的最简单方法。基于机器学习的对话的一个优点是，当你的机器人不知道如何做某事时，你可以教它！

## fallback action

有时候你想回到后退行动，比如说“对不起，我不明白”。为此，请将FallbackPolicy添加到策略集合中。

如果意图识别的置信度低于nlu\_threshold，或者没有任何对话策略预测置信度高于core\_threshold，则执行回退操作。

在配置文件中配置：

policies:

- name: "FallbackPolicy"

# min confidence needed to accept an NLU prediction

nlu\_threshold: 0.3

# min confidence needed to accept an action prediction from Rasa Core

core\_threshold: 0.3

# name of the action to be called if the confidence of intent / action

# is below the threshold

fallback\_action\_name: 'action\_default\_fallback'

Form: prefix

表单逻辑由FormAction类描述，而不是由故事描述。机器学习策略不必学习这种行为，如果以后更改表单操作FormAction,机器学习策略则不应该混淆，例如通过添加或删除所需的插槽。当您使用交互式学习生成包含表单的故事时，由表单处理的对话步骤将获得一个表单：prefix，这告诉Rasa Core在训练其他策略时应该忽略这些步骤。你在这里没有什么特别的，所有形式的快乐路径仍然涵盖在Forms Basics中给出的基本故事。

## Training and Policies

Rasa Core的工作原理是从story创建训练数据并在该数据上训练模型。

python -m rasa\_core.train -d domain.yml -s data/stories.md **\**

-o models/current/dialogue -c config.yml

rasa\_core.policies.Policy类决定在对话的每个步骤中采取的操作。 有不同的策略可供选择，您可以在单个rasa\_core.agent.Agent中包含多个策略。在每一个turn，都将使用最高置信度的policy预测下一个action。

在代码中配置policy:

**from** **rasa\_core.policies.memoization** **import** MemoizationPolicy

**from** **rasa\_core.policies.keras\_policy** **import** KerasPolicy

**from** **rasa\_core.agent** **import** Agent

agent = Agent("domain.yml",

policies=[MemoizationPolicy(), KerasPolicy()])

* MemoizationPolicy(备忘录policy)

MemoizationPolicy只记忆训练数据中的对话。如果训练数据中存在这种确切的对话，它会自信地预测下一个动作1.0，否则它会以置信度0.0预测无。

* Keras Policy

KerasPolicy使用Keras中实现的神经网络来选择下一个动作。默认体系结构基于LSTM，但您可以覆盖KerasPolicy.model\_architecture方法以实现您自己的体系结构。

Agent

The agent allows you to train a model, load, and use it. It is a simple API that lets you access most of Rasa Core’s functionality.

代理允许训练模型，加载和使用它。它是一个简单的API，可让您访问Rasa Core的大部分功能。

代理上的方法并没有暴露所有功能。有时需要自己编排不同的组件（域，策略，解释器和跟踪器存储）来自定义它们。

# RASA STACK入门

## 配置和安装

下拉repo仓库

git clone https://github.com/RasaHQ/starter-pack-rasa-stack.git

安装和配置

cd starter-pack-rasa-stack

pip install -r requirements.txt

安装 spaCy 英文语言模型

python -m spacy download en

## 文件解析

### **Rasa NLU model文件**

****data/nlu\_data.md****  该文件包含6个Intent的训练例子:

* + greet
  + goodbye
  + thanks
  + deny
  + joke
  + name (examples of this intent contain an entity called 'name')

****nlu\_config.yml**** ：Rasa NLU pipeline的配置文件

language: "en"

pipeline: spacy\_sklearn

### **Rasa Core模型文件**

* ****data/stories.md****  训练对话文件。
* ****domain.yml**** 域文件，包含intents, entities, slots, templates and actions。
* ****actions.py**** 自定义动作的代码文件，可通过外部API调用。
* ****endpoints.yml****  自定义动作的webhook configuration文件。
* ****policies.yml****  Rasa Core 模型的训练策略配置文件。

### **使用**

1. 训练Rasa NLU模型

make train-nlu

1. 训练 Rasa Core模型

make train-core

1. 在新终端中，通过运行启动服务器以执行自定义操作。

make action-server

1. 测试

make cmdline

# SPACY

## 模型model

# 环境部署

## NLU mitie部署

nlu依赖库安装

#安装 jieba sklearn

$pip install jieba sklearn

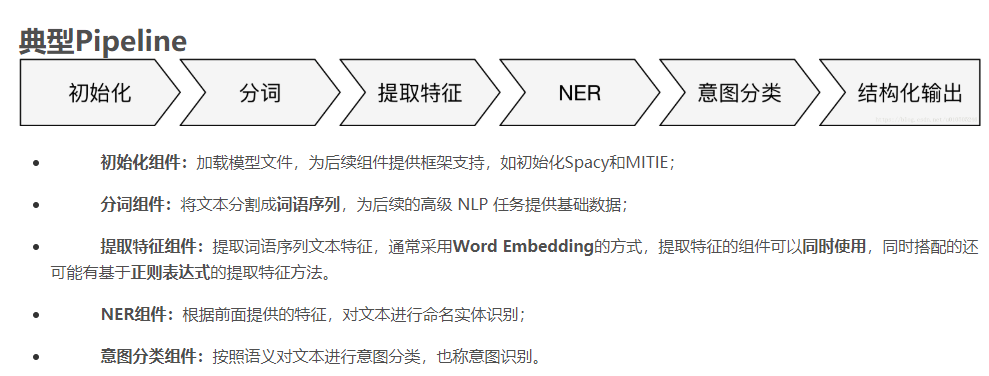
win环境下安装失败：

#安装MITIE

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

#安装Rasa-NLU

$pip install rasa\_nlu[mitie]



<https://blog.csdn.net/u010505246/article/details/82997100>

slot和action:

<https://blog.csdn.net/u012526436/article/details/88061902>

pipeline:

- name: "SpacyNLP"

model: "nlu/data/total\_word\_feature\_extractor\_zh.dat"

- name: "JiebaTokenizer"

# default\_dict: "./default\_dict.big"

# user\_dicts: "./jieba\_userdict"

- name: "RegexFeaturizer"

- name: "SpacyFeaturizer"

- name: "CRFEntityExtractor"

- name: "EntitySynonymMapper"

- name: "SklearnIntentClassifier"

## DOCKER部署

查询docker和docker-compose版本

docker -v && docker-compose -v

训练nara nlu模型

|  |
| --- |
| docker run \  -v /home/sxx/dsmart\_bot:/app/project \  -v /home/sxx/dsmart\_bot/models/rasa\_nlu:/app/model \  rasa/rasa\_nlu:latest-full \  run \  python -m rasa\_nlu.train \  -c /app/project/config.yml \  -d /app/project/data/dsmart.json \  -o /app/model \  --project dsmart\_nlu  上面命令在新版本中已经废弃，不能继续使用。新命令如下：  docker run -v /home/sxx/dsmart\_bot:/app/project -v /home/sxx/dsmart\_bot/models/rasa\_nlu:/app/model rasa/rasa\_nlu:latest-full run python -m rasa train nlu -c /app/project/config.yml -u /app/project/data/dsmart.json --out /app/model --fixed-model-name dsmart\_nlu |

训练rasa core模型

|  |
| --- |
| docker run \  -v $(pwd):/app/project \  -v $(pwd)/models/rasa\_core:/app/models \  rasa/rasa\_core:latest \  train \  --domain project/domain.yml \  --stories project/data/stories.md \  --out models |

命令解析：

-v $(pwd):/app/project 将当前目录挂载到docker容器中的/app/project目录，以便Rasa Core可以在stories.md和domain.yml 上训练模型。

rasa/rasa\_core:latest rasa/rasa\_core为使用的Rasa Core镜像名，tag为latest

train: 在容器中执行 train command 命令

命令执行结果是产生一个rasa core模型，位于目录models/rasa\_core中。

# 情景分析

|  |
| --- |
| **## Generated Story 12345615**  \* 健康分数  - utter\_ask\_for\_uid\_or\_instance\_name  \* inform\_data{"channel":"uid"}  - slot{"channel":"uid"}  - form\_query\_health\_score  - form{"name": "form\_query\_health\_score"}  \* form: inform\_data  - form\_query\_health\_score  - form{"name": null}  - action\_query\_health\_score  - utter\_health\_score |

# 问题

Intent: use\_entities对action选择的影响？

|  |
| --- |
| intents:  - 日检报告  - 其他  - 健康分数（use\_entities:false） |

Use\_entities默认为true，当此处为false时，domain的story找不到对于的action。

|  |
| --- |
| **## Generated Story 12345615**  \* 健康分数  - form\_query\_health\_score  - form{"name": "form\_query\_health\_score"}  - slot{"requested\_slot": "channel"}  \* form: inform\_data  - form\_query\_health\_score  - form{"name": null}  - action\_query\_health\_score  - utter\_health\_score |

新版本

|  |
| --- |
| (bot) sxx@sxx-Aspire-V5-471G:~/projects/dsmart\_bot$ rasa train  Traceback (most recent call last):  File "/home/sxx/.conda/envs/bot/bin/rasa", line 11, in <module>  load\_entry\_point('rasa', 'console\_scripts', 'rasa')()  File "/home/sxx/GIT-PROJECTS/rasa/rasa/\_\_main\_\_.py", line 70, in main  cmdline\_arguments.func(cmdline\_arguments)  File "/home/sxx/GIT-PROJECTS/rasa/rasa/cli/train.py", line 69, in train  kwargs=extract\_additional\_arguments(args),  File "/home/sxx/GIT-PROJECTS/rasa/rasa/train.py", line 48, in train  kwargs=kwargs,  File "uvloop/loop.pyx", line 1451, in uvloop.loop.Loop.run\_until\_complete  File "/home/sxx/GIT-PROJECTS/rasa/rasa/train.py", line 91, in train\_async  training\_files, skill\_imports  File "/home/sxx/GIT-PROJECTS/rasa/rasa/data.py", line 67, in get\_core\_nlu\_directories  story\_files, nlu\_data\_files = get\_core\_nlu\_files(paths, skill\_imports)  File "/home/sxx/GIT-PROJECTS/rasa/rasa/data.py", line 114, in get\_core\_nlu\_files  path, skill\_imports  File "/home/sxx/GIT-PROJECTS/rasa/rasa/data.py", line 139, in \_find\_core\_nlu\_files\_in\_directory  if \_is\_nlu\_file(full\_path):  File "/home/sxx/GIT-PROJECTS/rasa/rasa/data.py", line 157, in \_is\_nlu\_file  content = io\_utils.read\_json\_file(file\_path)  File "/home/sxx/GIT-PROJECTS/rasa/rasa/utils/io.py", line 128, in read\_json\_file  return json.load(f)  File "/home/sxx/.conda/envs/bot/lib/python3.7/json/\_\_init\_\_.py", line 296, in load  parse\_constant=parse\_constant, object\_pairs\_hook=object\_pairs\_hook, \*\*kw)  File "/home/sxx/.conda/envs/bot/lib/python3.7/json/\_\_init\_\_.py", line 338, in loads  s, 0)  json.decoder.JSONDecodeError: Unexpected UTF-8 BOM (decode using utf-8-sig): line 1 column 1 (char 0) |

解决

查找utf-8中含有bom的文件

(bot) sxx@sxx-Aspire-V5-471G:~/projects/dsmart\_bot$ grep -r -i -l $'^\xEF\xBB\xBF' .

./data/dsmart.json

替换：

grep -r -i -l *$'^\xEF\xBB\xBF' . | xargs sed -i 's/^\xEF\xBB\xBF//g'*

# 参考：

[从零了解RASA NLU监督词向量]

<https://medium.com/rasa-blog/supervised-word-vectors-from-scratch-in-rasa-nlu-6daf794efcd8>

[如何处理句中多重intent]

<https://blog.rasa.com/how-to-handle-multiple-intents-per-input-using-rasa-nlu-tensorflow-pipeline/>