IEPY Documentation

Release 0.9.6

Rafael Carrascosa, Javier Mansilla, Gonzalo García Berrotarán, Da

Jul 14, 2017

Contents

IEPY is an open source tool for Information Extraction focused on Relation Extraction.

To give an example of Relation Extraction, if we are trying to find a birth date in:

> *"John von Neumann (December 28, 1903 – February 8, 1957) was a Hungarian and American pure and applied mathematician, physicist, inventor and polymath."*

then IEPY's task is to identify "`John von Neumann`" and "`December 28, 1903`" as the subject and object entities of the "`was born in`" relation.

It's aimed at:

- *users* needing to perform Information Extraction on a large dataset.
- *scientists* wanting to experiment with new IE algorithms.

You can follow the development of this project and report issues at http://github.com/machinalis/iepy or join the mailing list here

CHAPTER 1

Features

- *A corpus annotation tool* with a web-based UI
- *An active learning relation extraction tool* pre-configured with convenient defaults.
- *A rule based relation extraction tool* for cases where the documents are semi-structured or high precision is required.
- **A web-based user interface that:**
  - Allows layman users to control some aspects of IEPY.
  - Allows decentralization of human input.
- A shallow entity ontology with coreference resolution via Stanford CoreNLP
- *An easily hack-able active learning core*, ideal for scientist wanting to experiment with new algorithms.

CHAPTER 2

Contents:

IEPY installation

IEPY runs on *python 3*, and it's fully tested with version *3.4*. These installation notes assume that you have a fresh installation of *Ubuntu 14.04*. If you are installing IEPY on a different platform, some details or software versions may be slightly different.

Because of some of its dependencies, IEPY installation is not a single pip install, but it's actually not that hard.

Outline:

- install some system packages
- install iepy itself
- download 3rd party binaries

System software needed

You need to install the following packages:

```
sudo apt-get install build-essential python3-dev liblapack-dev libatlas-dev gfortran
```

They are needed for python Numpy installation. Once this is done, install numpy by doing:

```
pip install numpy
```

And later, for been able to run some java processes:

```
sudo apt-get install openjdk-7-jre
```

Note: Instead of openjdk-7-jre you can use any other java (version 1.6 or higher) you may have.

Java 1.8 will allow you to use the **newest preprocess models**.

Install IEPY package

1. Create a Virtualenv
2. Install IEPY itself

```
pip install iepy
```

3. Configure java & NLTK

   In order to preprocess documents, set the environment variable JAVAHOME=/usr/bin/java (or the path where java was installed) To make this configuration persistent, add it to your shell rc file.

Download the third party data and tools

You should have now a command named "*iepy*". Use it like this to get some required binaries.

```
iepy --download-third-party-data
```

Note: If the java binary pointed by your JAVAHOME is 1.8, newest preprocess models will be acquired and used.

From 0 to IEPY

In this tutorial we will guide you through the steps to create your first Information Extraction application with IEPY. Be sure you have a working *installation*.

IEPY internally uses Django to define the database models, and to provide a web interface. You'll see some components of Django around the project, such as the configuration file (with the database definition) and the `manage.py` utility. If you're familiar with Django, you will move faster in some of the steps.

0 - Creating an instance of IEPY

To work with IEPY, you'll have to create an *instance*. This is going to be where the configuration, database and some binary files are stored. To create a new instance you have to run:

```
iepy --create <project_name>
```

Where <*project_name*> is something that you choose. This command will ask you a few things such as database name, its username and its password. When that's done, you'll have an instance in a folder with the name that you chose.

Read more about the instantiation process *here*.

1 - Loading the database

The way we load the data into the database is importing it from a *csv* file. You can use the script **csv_to_iepy** provided in your application folder to do it.

```
python bin/csv_to_iepy.py data.csv
```

This will load **data.csv** into the database, from which the data will subsequently be accessed.

Learn more about the required CSV file format here.

Note: You might also provide a *gziped csv file.*

2 - Pre-processing the data

Once you have your database with the documents you want to analyze, you have to run them through the pre-processing pipeline to generate all the information needed by IEPY's core.

The pre-processing pipeline runs a series of steps such as text tokenization, sentence splitting, lemmatization, part-of-speech tagging, and named entity recognition

Read more about the pre-processing pipeline here.

Your IEPY application comes with code to run all the pre-processing steps. You can run it by doing:

```
python bin/preprocess.py
```

This *will* take a while, especially if you have a lot of data.

3 - Open the web interface

To help you control IEPY, you have a web user interface. Here you can manage your database objects and label the information that the active learning core will need.

To access the web UI, you must run the web server. Don't worry, you have everything that you need on your instance folder and it's as simple as running:

```
python bin/manage.py runserver
```

Leave that process running, and open up a browser at http://127.0.0.1:8000 to view the user interface home page.

Now it's time for you to *create a relation definition*. Use the web interface to create the relation that you are going to be using.

IEPY

Now, you're ready to run either the *active learning core* or the *rule based core*.

Constructing a reference corpus

To test information extraction performance, IEPY provides a tool for labeling the entire corpus "by hand" and the check the performance experimenting with that data.

If you would like to create a labeled corpus to test the performance or for other purposes, take a look at the *corpus labeling tool*

Instantiation

Here, we'll explain in detail what an instantiation contains and what it does.

Folder structure

The folder structure of an iepy instance is the following:

```
- __init__.py
- settings.py
- database_name_you_picked.sqlite
- bin
|   - csv_to_iepy.py
|   - iepy_rules_runner.py
|   - iepy_runner.py
|   - manage.py
|   - preprocess.py
|   - rules_verifier.py
- extractor_config.json
- rules.py
```

Let's see why each one of those files is there:

Settings file

settings.py is a configuration file where you can change the database settings and all the web interface related settings. This file has a django settings file format.

Database

When you create an instance, a *sqlite* database is created by default. It has no data yet, since you'll have to fill it with your own data.

When working with big datasets, it's recommended to use some database engine other than *sqlite*. To change the database engine, change the *DATABASES* section of the settings file:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'database_name_you_picked.sqlite',
    }
}
```

For example, you can use PostgreSQL like this:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'your_database_name',
    }
}
```

(Remember that you'll need to install `psycopg2` first with a simple `pip install psycopg2`)

Take a look at the django database configuration documentation for more detail.

Note: Each time you change your database (either the engine or the name) you will have to instruct *django* to create all the tables in it, like this:

```
python bin/manage.py migrate
```

Active learning configuration

`extractor_config.json` specifies the configuration of the active learning core in *json* format.

Rules definition

If you decide to use the rule based core, you'll have to define all your rules in the file `rules.py`

You can verify if your rules run correctly using `bin/rules_verifier.py`. Read more about it here.

CSV importer

In the `bin` folder, you'll find a tool to import data from CSV files. This is the script `csv_to_iepy.py`. Your CSV data has to be in the following format:

```
<document_id>, <document_text>
```

Preprocess

To preprocess your data, you will use the `bin/preprocess.py` script. Read more about it *here*

Runners

In the `bin` folder, you have scripts to run the active learning core (`iepy_runner.py`) or the rule based core (`iepy_rules_runner.py`)

Web UI management

For the web server management, you have the `bin/manage.py` script. This is a django manage file and with it you can start up your server.

Instance Upgrade

From time to time, small changes in the iepy internals will require an *upgrade* of existing iepy instances.

The upgrade process will apply the needed changes to the instance-folder structure.

If you made local changes, the tool will preserve a copy of your changes so you can merge the conflicting areas by hand.

To upgrade an iepy instance, simply run the following command

```
iepy --upgrade <instance path>
```

Note: Look at the settings file to find the version of any iepy instance.

Running the active learning core

The active learning core works by trying to predict the relations using information provided by the user. This means you'll have to label some of the examples and based on those, the core will infer the rest. The core will also give you to label the more important examples (those which best helps to figure out the other cases).

To start using it you'll need to define a relation, run the core, label some evidence and re-run the core loop. You can also label evidences and re-run the core as much as you like to have a better performance.

Creating a relation

To create a relation, first open up the web server if you haven't already, and use a web browser to navigate on http://127.0.0.1:8000. There you'll find instructions on how to create a relation.

Running the core

After creating a relation, you can start the core to look for instances of that relation.

You can run this core in two modes: **High precision** or **high recall**. Precision and recall can be traded with one another up to a certain point. I.e. it is possible to trade some recall to get better precision and vice versa.

To visualize better this trade off, lets see an example: A precision of 99% means that 1 of every 100 predicted relations will be wrong and the rest will be correct. A recall of 30% means that only 30 out of 100 existent relations will be detected by the algorithm and the rest will be wrongly discarded as "no relation present".

Run the active learning core by doing:

```
python bin/iepy_runner.py <relation_name> <output>
```

And add `--tune-for=high-prec` or `--tune-for=high-recall` before the relation name to switch between modes. The default is **high precision**.

This will run until it needs you to label some of the evidences. At this point, what you need to do is go to the web interface that you ran on the previous step, and there you can label some evidences.

When you consider that is enough, on the prompt that the iepy runner presented you, continue the execution by typing **run**.

That will cycle again and repeat the process.

Run the active learning core in the command line and ask it to **STOP**. It'll save a csv with the automatic classifications for all evidences in the database.

Also, note that you can only predict a relation for a text that has been inserted into the database. The csv output file has the primary key of an object in the database that represents the evidence that was classified as "relation present" or "relation not present". An evidence object in the database is a rich-in-information object containing the entities and circumstances surrounding the prediction that is too complex to put in a single csv file.

In order to access the entities and other details you'll need to write a script to talk with the database (see iepy/data/models.py).

Fine tuning

If you want to modify the internal behavior, you can change the settings file. On your instance folder you'll fine a file called `extractor_config.json`. There you've all the configuration for the internal classifier, such as:

Classifier

This sets the classifier algorithm to be used, you can choose from:

- sgd: Stochastic Gradient Descent
- knn: Nearest Neighbors
- svc *(default)*: C-Support Vector Classification
- randomforest: Random Forest
- adaboost: AdaBoost

Features

Features to be used in the classifier, you can use a subset of:

- number_of_tokens
- symbols_in_between
- in_same_sentence
- verbs_count
- verbs_count_in_between
- total_number_of_entities
- other_entities_in_between
- entity_distance
- entity_order
- bag_of_wordpos_bigrams_in_between
- bag_of_wordpos_in_between
- bag_of_word_bigrams_in_between
- bag_of_pos_in_between
- bag_of_words_in_between
- bag_of_wordpos_bigrams

- bag_of_wordpos
- bag_of_word_bigrams
- bag_of_pos
- bag_of_words

These can be added as *sparse* adding them into the *sparse_features* section or added as *dense* into the *dense_features*.

The features in the sparse section will go through a stage of linear dimension reduction and the dense features, by default, will be used with a non-linear classifier.

Viewing predictions on the web user interface

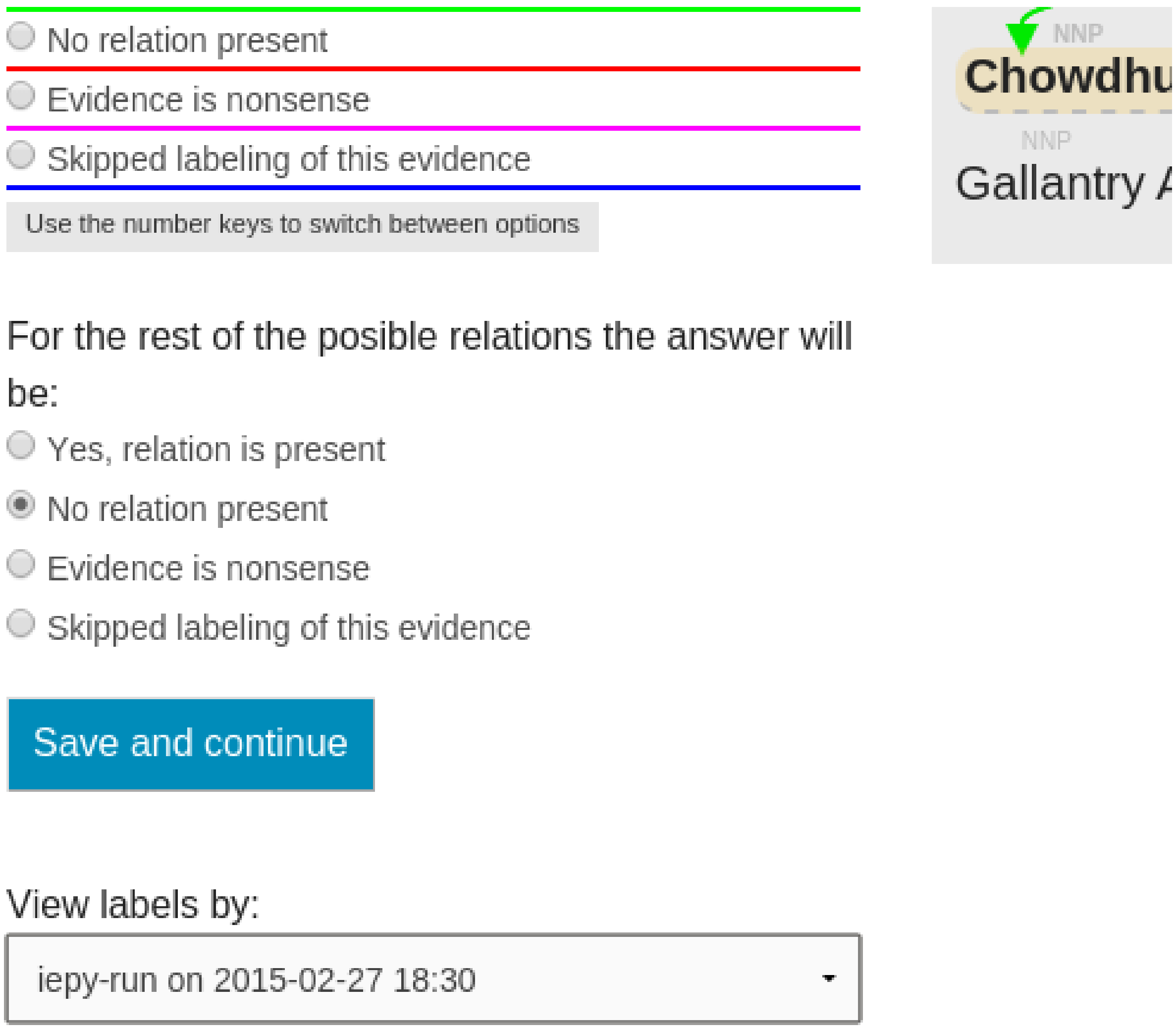
If you prefer to review the predictions using the web interface is possible to run the active learning core in a way that stores the results on the database and they are accesible through the web.

To do so, you'll have to run the core like this:

```
python bin/iepy_runner.py --db-store <relation_name>
```

We do not have an specialized interface to review predictions but you can still view them by using the *interface to create a reference corpus*.

This way, you'll get labels as a new **judge** called iepy-run and a date.



No relation present

Evidence is nonsense

Skipped labeling of this evidence

Use the number keys to switch between options

For the rest of the posible relations the answer will be:

Yes, relation is present

[✓ No relation present]

Evidence is nonsense

Skipped labeling of this evidence

View labels by:

iepy-run on 2015-02-27 18:30

Saving predictor for later use

Since training could be a slow process, you might want to save your trained predictor and re-use it several times without the need to train again.

You can save it this by doing:

```
python bin/iepy_runner.py --store-extractor=myextractor.pickle <relation_name>
↪<output>
```

And re use it like this:

```
python bin/iepy_runner.py --trained-extractor=myextractor.pickle <relation_name>
↪<output>
```

Running the rule based core

Here we will guide you through the steps to use the rule based system to detect relations on the documents.

How they work

In the rule based system, you have to define a set of "regular expression like" rules that will be tested against the segments of the documents. Roughly speaking, if a rule matches it means that the relation is present.

This is used to acquire high precision because you control exactly what is matched.

Anatomy of a rule

Note: If you don't know how to define a python function, check this out

A rule is basically a *decorated python function*. We will see where this needs to be added later, for now lets concentrate on how it is written.

```
@rule(True)
def born_date_and_death_in_parenthesis(Subject, Object):
    """ Example: Carl Bridgewater (January 2, 1965 - September 19, 1978) was shot␣
↪dead """
    anything = Star(Any())
    return Subject + Pos("-LRB-") + Object + Token("-") + anything + Pos("-RRB-") +␣
↪anything
```

First you have to specify that your function is in fact a rule by using the **decorator @rule**.

As you can see in the first line, this is added on top of the function. In this decorator you have to define if the rule is going to be *positive* or *negative*. A positive rule that matches will label the relations as present and a negative one will label it as not present. You can define this by passing the True or False parameter to the rule decorator.

Then it comes the definition of the function. This functions takes two parameters: the **Subject** and the **Object**. This are patterns that will be part of the regex that the function has to return.

After that it comes the body of the function. Here it has to be constructed the regular expression and needs to be returned by the function. This is not an ordinary regular expression, it uses ReFO. In ReFO you have to operate with objects that does some kind of check to the text segment.

For our example, we've chosen to look for the *Was Born* relation. Particularly we look for the date of birth of a person when it is written like this:

```
Carl Bridgewater (January 2, 1965 - September 19, 1978)
```

To match this kind of cases, we have to specify the regex as a sum of predicates. This will check if every part matches.

Rule's building blocks

Aside of every ReFO predicates, iepy comes with a bunch that you will find useful for creating your own rules

- **Subject**: matches the evidence's left part.
- **Object**: matches the evidence's right part.

- **Token**: matches if the token is literally the one specified.
- **Lemma**: matches if the lemma literally the one specified.
- **Pos**: matches the *part of speech* of the token examined.
- **Kind**: matches if the token belongs to an entity occurrence with a given kind.

Setting priority

Using the **rule decorator**, you can set that a rule is more important than another, and because of that it should try to match before.

IEPY will run the rules ordered decreasingly by its priority number, and the default priority is 0.

For example, to set a priority of 1 you do:

```
@rule(True, priority=1)
def rule_name(Subject, Object):
    ...
```

Negative rules

If you spot that your rules are matching things erroneously, you can write a rule that catches that before it is taken by a positive rule.

You do this by setting the rule as a *negative rule* using the decorator. Also is recommended to set higher priority so it is checked before the other ones.

Example:

```
@rule(False, priority=1)
def incorrect_labeling_of_place_as_person(Subject, Object):
    """
    Ex:  Sophie Christiane of Wolfstein (24 October 24, 1667 - 23 August 1737)

    Wolfstein is a *place*, not a *person*
    """
    anything = Star(Any())
    person = Plus(Pos("NNP") + Question(Token(",")))
    return anything + person + Token("of") + Subject + anything
```

Note that the parameters of the rule decorator are **False** and **priority=1**

Where do I place the rules

On your project's instance folder, there should be a *rules.py* file. All rules should be place there along with a **RELATION** variable that sets which relation is going to be used.

This is the file that will be loaded when you run the *iepy_rules_runner*.

Example

This is a portion of the example provided with IEPY, you can view the complete file here.

```python
from refo import Question, Star, Any, Plus
from iepy.extraction.rules import rule, Token, Pos

RELATION = "was born"

@rule(True)
def was_born_explicit_mention(Subject, Object):
    """
    Ex: Shamsher M. Chowdhury was born in 1950.
    """
    anything = Star(Any())
    return anything + Subject + Token("was born") + Pos("IN") + Object + anything


@rule(True)
def is_born_in(Subject, Object):
    """
    Ex: Xu is born in 1902 or 1903 in a family of farmers in Hubei ..
    """
    anything = Star(Any())
    return Subject + Token("is born in") + Object + anything


@rule(True)
def just_born(Subject, Object):
    """
    Ex: Lyle Eugene Hollister, born 6 July 1923 in Sioux Falls, South Dakota,␣
↪enlisted in the Navy....
    """
    anything = Star(Any())
    return Subject + Token(", born") + Object + anything
```

Verifying your rules

During the construction of your rules, you might want to check whether if the rules are matching or if they aren't. Even more, if you have tagged data in your corpus, you can know how good is the performance.

The rules verifier is located on your instance under the `bin` directory, it's called `rules_verifier.py`

You can run the verifier with every rule or with a single rule, on all of the segments or in a sample of those. Take a look at the parameters on the rules verifier to find out how to use them by running:

```
$ python bin/rules_verifier.py --help
```

If you have labeled data on your corpus, the run will calculate how it scored in terms of precision, recall and other metrics. You have to keep in mind that this is not exactly what you'll get when you run the rules core, even if you run the verifier with all the rules and all the data, the numbers are going to be a little different because this will run every evidence with every rule, and the core instead stops at the first match. This is just a warning so you don't get too excited or too depressed with these results.

About the Pre-Process

The preprocessing adds the metadata that iepy needs to detect the relations, which includes:

- Text tokenization and sentence splitting.

- Text lemmatization
- Part-Of-Speech (POS) tagging.
- Named Entity Recognition (NER).
- Gazettes resolution
- Syntactic parsing.
- TextSegments creation (internal IEPY text unit).

We're currently running all this steps (except the last one) using the Stanford CoreNLP tools. This runs in a all-in-one run, but every step can be *modified to use a custom version* that adjust your needs.

About the Tokenization and Sentence splitting

The text of each Document is split on tokens and sentences, and that information is stored on the document itself, preserving (and also storing) for each token the offset (in chars) to the original document text.

The one used by default it's the one that the Stanford CoreNLP provides.

Note: While using the Stanford tokenizer, you can customize some of tokenization options.

First read here: tokenizer options

On your instance *settings.py* file, add options as keys on the CORENLP_TKN_OPTS dict. You can use as key any of the "known options", and as value, use True or False for booleans, or just strings when option requires a text. Example:

```
CORENLP_TKN_OPTS = {
    'latexQuotes': False
}
```

Lemmatization

Note: Lemmatization was added on the version 0.9.2, all instances that were created before that, need to run the preprocess script again. This will run only the lemmatization step.

The text runs through a step of lemmatization where each token gets a lemma. This is a canonical form of the word that can be used in the classifier features or the rules core.

Part of speech tagging (POS)

Each token is augmented with metadata about its part of speech such as noun, verb, adjective and other grammatical tags. Along the token itself, this may used by the NER to detect an entity occurrence. This information is also stored on the Document itself, together with the tokens.

The one used by default it's the one that the Stanford CoreNLP provides.

Named Entity Recognition (NER)

To find a relation between entities one must first recognize these entities in the text.

As an result of NER, each document is added with information about all the found Named Entities (together with which tokens are involved in each occurrence).

An automatic NER is used to find occurrences of an entity in the text.

The default pre-process uses the Stanford NER, check the Stanford CoreNLP's documentation to find out which entity kinds are supported, but includes:

- Location
- Person
- Organization
- Date
- Number
- Time
- Money
- Percent

Others remarkable features of this NER (that are incorporated to the default pre-process) are:

- pronoun resolution
- simple co-reference resolution

This step can be customized to find entities of kinds defined by you, or anything else you may need.

Gazettes resolution

In case you want to add named entity recognition by matching literals, iepy provides a system of gazettes. This is a mapping of literals and entity kinds that will be run on top of the basic stanford NER. With this, you'll be able to recognize entities out of the ones done by the stanford NER, or even correct those that are incorrectly tagged.

Learn more about here.

Syntactic parsing

Note: Syntactic parsing was added on the version 0.9.3, all instances that were created before that, need to run the preprocess script again. This will run only the syntactic parsing step.

The sentences are parsed to works out the syntactic structure. Each sentence gets an structure tree that is stored in Penn Treebank notation. IEPY presents this to the user using a NLTK Tree object.

By default the sentences are processed with the Stanford Parser provided within the Stanford CoreNLP.

For example, the syntactic parsing of the sentence `Join the dark side, we have cookies` would be:

```
(ROOT
  (S
    (S
      (VP (VBN Join)
        (NP (DT the) (JJ dark) (NN side))))
    (, ,)
    (NP (PRP we))
    (VP (VBP have)
      (NP (NNS cookies)))))
```

About the Text Segmentation

IEPY works on a **text segment** (or simply **segment**) level, meaning that will try to find if a relation is present within a segment of text. The pre-process is the responsible for splitting the documents into segments.

The default pre-process uses a segmenter that creates for documents with the following criteria:

- for each sentence on the document, if there are at least 2 Entity Occurrences in there

How to customize

On your own IEPY instances, there's a file called `preprocess.py` located in the `bin` folder. There you'll find that the default is simply run the Stanford preprocess, and later the segmenter. This can be changed to run a sequence of steps defined by you

For example, take this pseudo-code to guide you:

```
pipeline = PreProcessPipeline([
    CustomTokenizer(),
    CustomSentencer(),
    CustomLemmatizer(),
    CustomPOSTagger(),
    CustomNER(),
    CustomSegmenter(),
], docs)
pipeline.process_everything()
```

Note: The steps can be functions or callable objects. We recommend objects because generally you'll want to do some load up of things on the __*init*__ method to avoid loading everything over and over again.

Each one of those steps will be called with each one of the documents, meaning that every step will be called with all the documents, after finishing with that the next step will be called with each one of the documents.

Running in multiple cores

Preprocessing might take a lot of time. To handle this you can run the preprocessing on several cores of the same machine or even run it on differents machines to accelerate the processing.

To run it on the same machine using multiple cores, all you need to do is run:

```
$ python bin/preprocess.py --multiple-cores=all
```

This will use all the available cores. You can also specify a number if you want to use less than that, like this:

```
$ python bin/preprocess.py --multiple-cores=2
```

Running in multiple machines

Running the preprocess on different machines it's a bit tricky, here's what you'll need:

- A iepy instance with a database that allows remote access (such as postgres)
- One iepy instance on each extra machine that has the database setting pointing to the main one.

Then you'll need to decide on how many parts do you want to split the document set and run each part on a different machine. For example, you could split the documents in 4 and run 2 processes on one machine and 2 on another one. To do this you'll run:

On one of the machines, in two different consoles run:

```
$ python bin/preprocess.py --split-in=4 --run-part=1
```

```
$ python bin/preprocess.py --split-in=4 --run-part=2
```

And on the other machine:

```
$ python bin/preprocess.py --split-in=4 --run-part=3
```

```
$ python bin/preprocess.py --split-in=4 --run-part=4
```

Gazettes resolution

We call a gazette a mapping between a list of tokens and an entity kind. If that list of tokens matches exactly on your text, then that would be tagged as an entity.

All the entities occurrences that where detected by a gazette and share the same set of tokens, will share the same entity. This means that if you have a gazette that finds `Dr. House` and tags it as a `PERSON`, all the occurrences in the text that matches those tokens, will belong to the same entity.

Basic usage: Loading from csv

The basic usage would be including a set of gazettes before running the preprocess step. To include the gazettes on your database, you can use the script `gazettes_loader.py` that comes included with your instance. This will take a csv file with the following format:

```
<literal>,<class>
```

Literal can be a single token or multiple tokens separated by space. The only restriction is that every literal is unique.

For example, a gazettes csv file could be:

```
literal,class
Dr. House,PERSON
Lupus,DISEASE
Headache,SYMPTOMS
```

Removing elements

When deleting an entity, all the occurrences are deleted with it along the gazette item that introduced them. Same goes the other way, if you delete a gazette item, the entity, and therefore the occurrences, will be deleted as well.

To delete a gazette item, go to the database admin page and find the Gazette section. You'll be able to find the one that you want to remove.

To remove an entity, find an occurrence by exploring a document on any of its views, and right click it. There you'll find a delete link that enables you to remove the whole entity. Keep in mind that this action will delete the gazette item.

Creating a reference corpus

IEPY provides a web tool for creating a reference corpus in a simple and fast way. This corpus can be used for evaluation or simply to have a labeled corpus of relations between entity occurrences.

Running the web server

First of all, you need to run the web server that will provide the interface. This is done by running a *Django* server.

Assuming you have an iepy instance and it's your current directory, to start the server you need to run

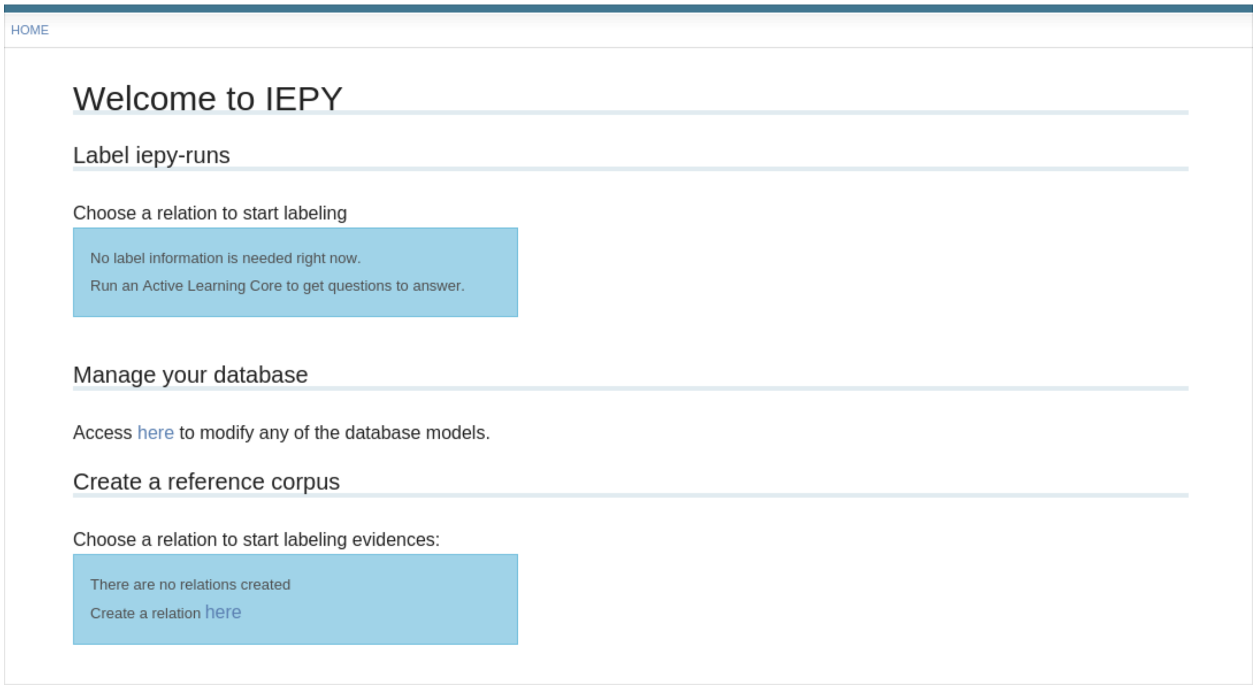
```
$ python bin/manage.py runserver
```

You will see a message like this:

```
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Home page

At this point, you can go on and open a browser and access the URL http://127.0.0.1:8000 and you will get a screen like this:



Welcome to IEPY

Label iepy-runs

Choose a relation to start labeling

No label information is needed right now.
Run an Active Learning Core to get questions to answer.

Manage your database

Access here to modify any of the database models.

Create a reference corpus

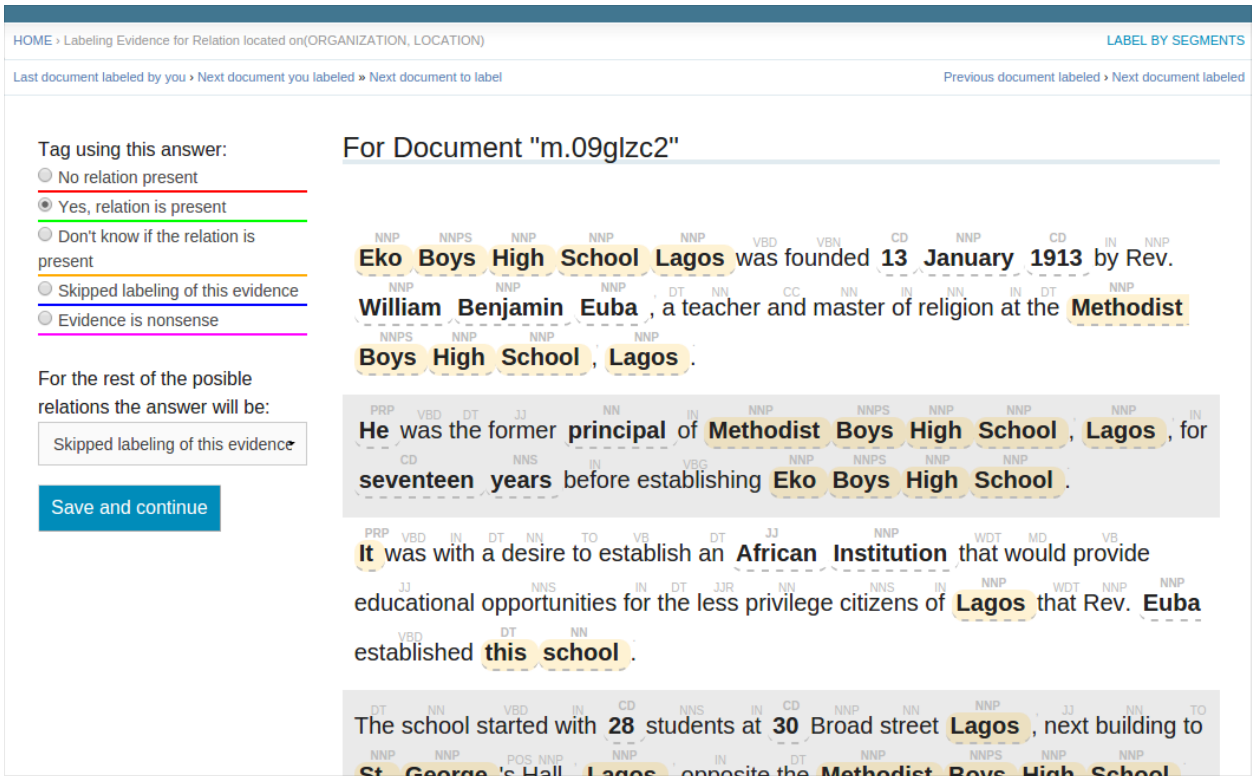
Choose a relation to start labeling evidences:

There are no relations created
Create a relation here

After creating a relation, you can access it on the `Create a reference corpus` section of the home page. Once you get there, you'll find that there are two different ways to label evidences: by segment and by document. The default one is by document but you can switch between both of them.

Document based labeling

This view presents a complete document with all the segments that make sense to show. These are the ones that have present entities with the entity kind that your relation uses.



HOME › Labeling Evidence for Relation located on(ORGANIZATION, LOCATION) LABEL BY SEGMENTS

Last document labeled by you › Next document you labeled » Next document to label Previous document labeled › Next document labeled

Tag using this answer:

- No relation present
- [✓ Yes, relation is present]
- Don't know if the relation is present
- Skipped labeling of this evidence
- Evidence is nonsense

For the rest of the posible relations the answer will be:

Skipped labeling of this evidence

Save and continue

For Document "m.09glzc2"

Eko Boys High School Lagos was founded 13 January 1913 by Rev. William Benjamin Euba , a teacher and master of religion at the Methodist Boys High School , Lagos .

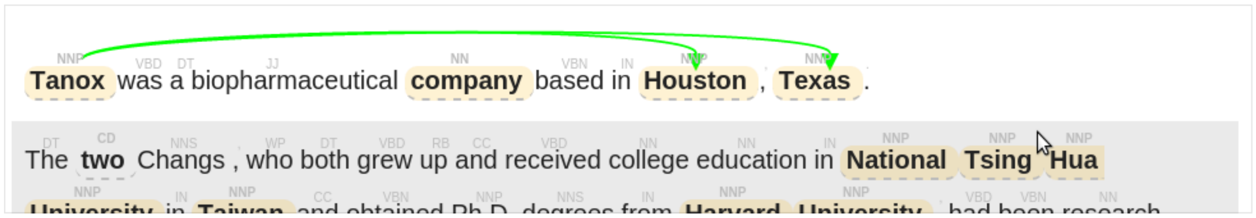
He was the former principal of Methodist Boys High School , Lagos , for seventeen years before establishing Eko Boys High School .

It was with a desire to establish an African Institution that would provide educational opportunities for the less privilege citizens of Lagos that Rev. Euba established this school .

The school started with 28 students at 30 Broad street Lagos , next building to St George 's Hall Lagos opposite the Methodist Boys High School

On the left side of the page, you'll see a list of the options that you have to label the evidences. To start labeling information what you need to do is choose one of this options, then click on two entity occurrences (marked in yellow on the text).

IEPY will only let you click on entity occurrences that has the type that your relation need. Even when you select the first entity occurrence, you will only be able to click on entities of the other entity type.



Tanox was a biopharmaceutical company based in Houston , Texas .

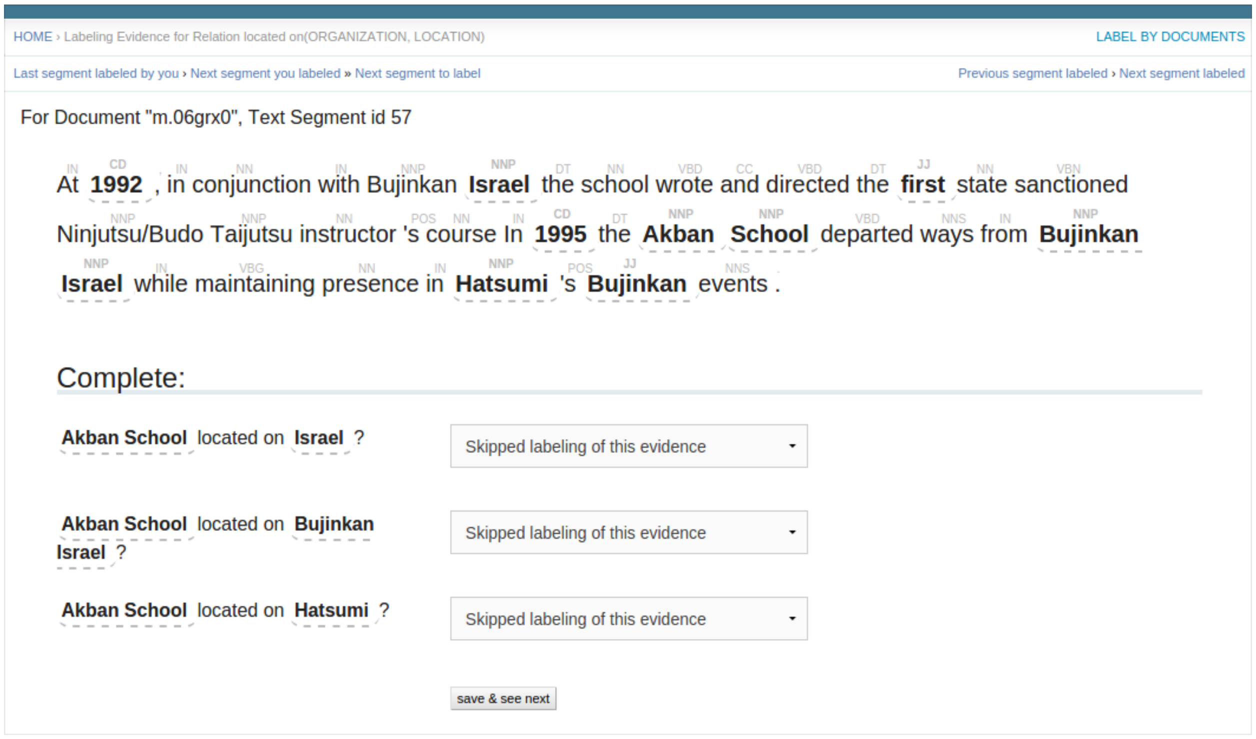
The two Changs , who both grew up and received college education in National Tsing Hua University in Taiwan and obtained Ph.D. degrees from Harvard University had been research

After saving, IEPY will take you to automatically to the next document. Also on top you have some navigation controls.

Note: Be careful with the navigation buttons because it won't save the changes that you've made on this document.

Segment based labeling

When labeling by segment, you are presented with a segment of a document, and you will have to answer if the relation is present on all the possible combinations of entity occurrences.


HOME › Labeling Evidence for Relation located on(ORGANIZATION, LOCATION)

LABEL BY DOCUMENTS

Last segment labeled by you › Next segment you labeled » Next segment to label

Previous segment labeled › Next segment labeled

For Document "m.06grx0", Text Segment id 57

At 1992 , in conjunction with Bujinkan Israel the school wrote and directed the first state sanctioned Ninjutsu/Budo Taijutsu instructor 's course In 1995 the Akban School departed ways from Bujinkan Israel while maintaining presence in Hatsumi 's Bujinkan events .

Complete:

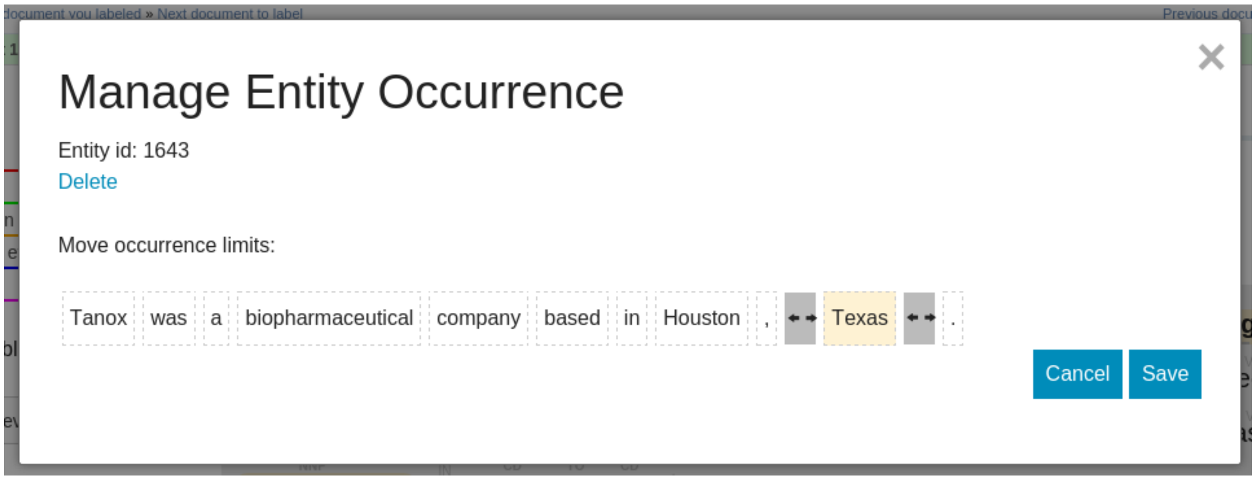
Akban School located on Israel ?	Skipped labeling of this evidence
Akban School located on Bujinkan Israel ?	Skipped labeling of this evidence
Akban School located on Hatsumi ?	Skipped labeling of this evidence

Here what you will need to do is complete every evidence whether the relation is present or not. When saving you will get another segment to label and so on.

On top you have navigation controls and on the far right you have link to switch view for one by document.

Fixing mistagged entity occurrences

It is possible that the automatic process that detects entities have been mistaken. This leads to an entity tagged partially or incorrectly. In this case, we provide a tool to fix this problems. You can access this tool by right clicking in the problematic entity and choosing **Modify entity occurrence**


Manage Entity Occurrence

Entity id: 1643

Delete

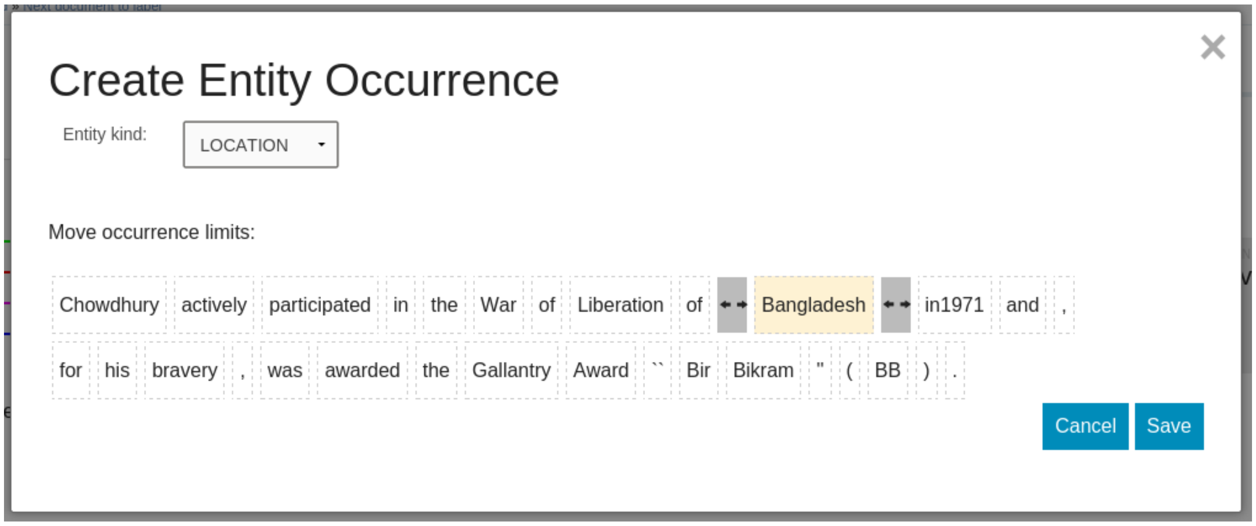
Move occurrence limits:

Tanox was a biopharmaceutical company based in Houston , Texas .

There you can completely remove the entity or change the limits so it holds more (or less) tokens.

Creating new occurrences

If an entity occurrence wasn't detected automatically, you can add it manually. To do so, right click on any token and choose **Create entity occurrence**.


Create Entity Occurrence
Entity kind:
LOCATION
Move occurrence limits:
Chowdhury actively participated in the War of Liberation of ↔ Bangladesh ↔ in1971 and ,
for his bravery , was awarded the Gallantry Award `` Bir Bikram '' (BB) .
Cancel
Save

You can modify the limits of the tokens and the entity kind there. After this operation, new *evidence candidates* will be created if needed.

How to Hack

There are several places where you can incorporate your own ideas and needs into IEPY. Here you'll see how to modify different parts of the iepy core.

Altering how the corpus is created

On the preprocess section was already mentioned that you can customize how the corpus is created.

Using your own classifier

You can change the definition of the *extraction classifier* that is used when running iepy in *active learning* mode.

As the simplest example of doing this, check the following example. First, define your own custom classifier, like this:

```
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import make_pipeline
from sklearn.feature_extraction.text import CountVectorizer


class MyOwnRelationClassifier:
    def __init__(self, **config):
        vectorizer = CountVectorizer(
            preprocessor=lambda evidence: evidence.segment.text)
        classifier = SGDClassifier()
        self.pipeline = make_pipeline(vectorizer, classifier)

    def fit(self, X, y):
        self.pipeline.fit(X, y)
        return self

    def predict(self, X):
        return self.pipeline.predict(X)

    def decision_function(self, X):
        return self.pipeline.decision_function(X)
```

and later, in iepy_runner.py of your IEPY instance, in the **ActiveLearningCore** creation, provide it as a configuration parameter like this

```
iextractor = ActiveLearningCore(
    relation, labeled_evidences,
    tradeoff=tuning_mode,
    extractor_config={},
    extractor=MyOwnRelationClassifier
)
```

Implementing your own features

Your classifier can use features that are already built within iepy or you can create your own. You can even use a rule (as defined in the *rules core*) as feature.

Start by creating a new file in your instance, you can call it whatever you want, but for this example lets call it `custom_features.py`. There you'll define your features:

```
# custom_features.py
from featureforge.feature import output_schema

@output_schema(int, lambda x: x >= 0)
def tokens_count(evidence):
    return len(evidence.segment.tokens)
```

Note: Your features can use some of the Feature Forge's capabilities.

Once you've defined your feature you can use it in the classifier by adding it to the configuration file. You should have one on your instance with all the default values, it's called `extractor_config.json`.

There you'll find 2 sets of features where you can add it: dense or sparse. Depending on the values returned by your feature you'll choose one over the other.

To include it, you have to add a line with a python path to your feature function. If you're not familiarized with the format you should follow this pattern:

```
{project_name}.{features_file}.{feature_function}
```

In our example, our instance is called `born_date`, so in the config this would be:

```
"dense_features": [
    ...
    "born_date.custom_features.tokens_count",
    ...
],
```

Remember that if you want to use that configuration file you have to use the option `--extractor-config`

Using rules as features

In the same way, and without doing any change to the rule, you can add it as feature by declaring it in your config like this:

Suppose your instance is called `born_date` and your rule is called `born_date_in_parenthesis`, then you'll do:

```
"dense_features": [
    ...
    "born_date.rules.born_date_in_parenthesis",
    ...
],
```

This will run your rule as a feature that returns 0 if it didn't match and 1 if it matched.

Using all rules as one feature

Suppose you have a bunch of rules defined in your rules file and instead of using each rule as a different feature you want to use a single feature that runs all the rules to test if the evidence matches. You can write a custom feature that does so. Let's look an example snippet:

```
# custom_features.py
import refo

from iepy.extraction.rules import compile_rule, generate_tokens_to_match, load_rules

rules = load_rules()


def rules_match(evidence):
    tokens_to_match = generate_tokens_to_match(evidence)

    for rule in rules:
        regex = compile_rule(rule, evidence.relation)

        if refo.match(regex, tokens_to_match):
            if rule.answer:  # positive rule
                return 1
            else:  # negative rule
                return -1
    # no rule matched
    return 0
```

This will define a feature called `rules_match` that tries every rule for an evidence until a match occurs, and returns one of three different values, depending on the type of match.

To use this you have to add this single feature to your config like this:

```
"dense_features": [
    ...
    "born_date.custom_features.rules_match",
    ...
],
```

Documents Metadata

While building your application, you might want to store some extra information about your documents. To avoid loading this data every time when predicting, we've separated the place to put this information into another model called **IEDocumentMetadata** that is accessible through the **metadata** attribute.

IEDocumentMetadata has 3 fields:

- title: for storing document's title
- url: to save the source url if the document came from a web page
- itmes: a dictionary that you can use to store anything you want.

By default, the **csv importer** uses the document's metadata to save the filepath of the csv file on the *items* field.

Troubleshooting

32 bit architecture issues

We've experience some memory issues when using a computer with 32 bit architecture. This is because by default we use the Stanford CoreNLP (java based), which has some special needs about the memory. Read about them more in detail here

We quote:

> The system requires Java 1.8+ to be installed. Depending on whether you're running 32 or 64 bit Java and the complexity of the tagger model, you'll need somewhere between 60 and 200 MB of memory to run a trained tagger (i.e., you may need to give java an option like java -mx200m)

What have worked for us is adding the following environment variable before running iepy:

```
export _JAVA_OPTIONS='-Xms1024M -Xmx1024m'
```

You can modify those numbers to your convenience.

Preprocess not running under MacOS

> Problems with the preprocess under MacOS? Apparently a change in the CoreNLP script is needed to be run. You need to change the file `corenlp.sh` that is located on `/Users/<your user>/Library/Application Support/iepy/stanford-corenlp-full-2014-08-27/` and change ``scriptdir=`dirname $0` `` for ``scriptdir=`dirname "$0"` `` (ie, add double quotes around `$0`)

Can't install IEPY with python 2

> Indeed, IEPY works with Python 3.4 or higher.

Language support

By default IEPY will use English models, but it's also able to work with different languages.

The preprocess machinery that's provided by default (Stanford Core NLP) has support for some other languages, so, check their models and documentation in case you need this.

Note: The main goal until now was to architecture IEPY to allow different languages. Right now, the only fully supported languages are English, Spanish and German. If you need something else, do not hesitate in contacting us.

Language Installation and Models

The language models used by IEPY (the information used during preprocessing phase) are stored on your IEPY installation. Several models for different languages can be installed on the same installation.

In order to download Spanish models you should run

```
iepy --download-third-party-data --lang=es
```

In order to download German models you should run

```
iepy --download-third-party-data --lang=de
```

Note: Check Stanford Core NLP documentation and files to download for more language packages.

Language Definition and Instances

Every IEPY instance works for a single language, which is declared on the settings.py file like this:

To change the instance language, change the settings file on the section where it says *IEPY_VERSION*:

```
IEPY_VERSION = 'en'
```

To create an IEPY instance for a different language, you should run

```
iepy --create --lang=es <folder_path>
```

CHAPTER 3

Authors

IEPY is © 2014 Machinalis in collaboration with the NLP Group at UNC-FaMAF. Its primary authors are:

- Rafael Carrascosa <rcarrascosa@machinalis.com> (rafacarrascosa at github)
- Javier Mansilla <jmansilla@machinalis.com> (jmansilla at github)
- Gonzalo García Berrotarán <ggarcia@machinalis.com> (j0hn at github)
- Franco M. Luque <francolq@famaf.unc.edu.ar> (francolq at github)
- Daniel Moisset <dmoisset@machinalis.com> (dmoisset at github)

CHAPTER 4

Changelog

0.9.6

- Fixed some dependencies declarations to provide support for python 3.5
- Bug fix respect to active learning predictions
- Added support for German preprocess (thanks @sweh)

0.9.5

- Bug fix on TokenizerSentencerRunner (thanks ezesalta)
- Fix on installation dependencies
- Tokenization options can be handled from instance settings file
- **Github Bugs fixed:**
  - https://github.com/machinalis/iepy/issues/75
  - https://github.com/machinalis/iepy/issues/76
  - https://github.com/machinalis/iepy/issues/77
  - https://github.com/machinalis/iepy/issues/86
  - https://github.com/machinalis/iepy/issues/89

0.9.4

- Added multicore preprocess
- Added support for Stanford 3.5.2 preprocess models

0.9.3

- Added grammatical parsing to the preprocess flow of documents
- Added support for Spanish preprocess
- Restricted each iepy-instance to a single language
- Gazetter support

- Labeling UI improvements
- Performance and memory usage improvements
- Model simplifications (labels, metadata)
- Storage & view of predictions

0.9.2

- Add ability to use custom features (http://iepy.rtfd.org/en/latest/how_to_hack.html#implementing-your-own-features)
- Add ability to use rules as features (http://iepy.rtfd.org/en/latest/how_to_hack.html#using-rules-as-features)
- Add rules verifier (http://iepy.rtfd.org/en/latest/rules_tutorial.html#verifying-your-rules)
- Fixed bugs of compatibility with firefox [thanks dchaplinsky for the bug report]
- Skip instead of crashing when a document could not be loaded via csv importer [thanks dchaplinsky for the report and suggestion]
- Performance improvement on rules runner
- Change instance files schema, now it's a python package and renamed settings.
- Add lemmatization to the pre-process (http://iepy.rtfd.org/en/latest/preprocess.html#lemmatization)
- Fix critical bug on loading rules
- Fix critical bug on ranking questions on the active learning extraction runner

0.9.1

- Add entity kind on the modal dialog
- Change arrows display to be more understandable
- Join skip and don't know label options
- Change options dropdown for radio buttons
- Show help for shortcuts and change the order of the options
- Documents rich view (without needing to be labeling the document for some relation)
- instance upgrader