gencou Documentation

J. Unverfehrt, J. Stanicki, O. Zobel, M. Zadikowitsch

Feb 29, 2020

OVERVIEW

The gencou project revolves around generating counter arguments. Given an argument, locate relevant data in a preconstructed argument knowledge graph and generate the "best" counterargument using a Seq2Seq model to transform the vertices and edges into natural language.

The project was created for the course 'Softwareprojekt' during our bachelor in the Institute of Computational Linguistics Heidelberg and was supervised by Prof. Dr. Annette Frank. The course spanned from October 22nd 2019 to February 29th 2020.

The task is linked here.[1]

[1] https://www.cl.uni-heidelberg.de/courses/ws19/softwareprojekt/material/projects_frank.pdf>

CHAPTER

ONE

GETTING STARTED

1.1 Requirements

At least Python version 3.6.x is needed to run our system. Further, the following packages need to be existent:

python-igraph provides the graph structure

neuralcoref provides the co-references-resolving

allennlp provides a supervised-openIE implementation, used for the information extraction from text

spacy Version 2.1.0, needed by supervised-openIE

pytorch_pretrained_bert contains a pytorch re-implementation of Google's TensorFlow repository for the BERT model

- sklearn
- progressbar2
- torch
- matplotlib
- numpy
- tools
- tqdm
- pandas
- nltk.corpus

To install the required packages, run:

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

Make sure to have nltk.stopwords installed:

```
$ python3
>>> import nltk
>>> nltk.download("stopwords")
```

Additionally make sure that the english language model from spacy is installed:

```
$ python -m spacy download en
```

1.2 Usage of our pretrained model

For using our system, start by running:

```
$ python3 interface.py
```

This module includes both the graph and the frame classification.

If *[1] batch* is chosen, the file *../data/arguments.txt* is loaded, which contains example arguments with their respective stance. Feel free to add or change arguments, but make sure that you keep the correct syntax:

```
argument (all sentences in one line)
stance
argument
...
```

To generate natural language from the found triples, execute `source/seqtoseq_load_model`, which prints the sentences to your terminal. Bring a few minutes at the start, the seq2seq model takes a while to initialize.

This demo is trained with 15000 arguments from the args.me corpus. The system is meant to contain the whole corpus, but due to timing issues (creating only the graph on a cluster took over a week already) we weren't able to implement that yet. We hope that this sample suffices as proof of concept.

1.3 Train yourself

Since our project is scattered in several parts by nature, we haven't yet found a convenient way to integrate an "all-in-one" training processing script. We try our best though to outline here the necessary steps to train our model.

1. *Create the Argument Knowledge Graph*
2. *Train FrameBert model*
3. Train the seq2seq model (with data provided by the graph)

1.4 Create the Argument Knowledge Graph

We recommend adjusting the *akgcreate module* script, which is commented and calls all necessary methods needed to create an argument knowledge graph. For further information on the AKG class, see *akglib module*

Make sure to get your copy of the args.me corpus[1] (894.9 MB in JSON format). Put it in the folder *data/akg/*

If using only the class, make sure to keep the following work order:

[1] https://webis.de/data/args-me.html

1. Create the class (specify the path to your args.me corpus as parameter, if not *default*)
2. Load the corpus *akglib.AKG.load_corpus()*
3. Process the corpus *akglib.AKG.process_arguments()*
4. Construct the graph *akglib.AKG.construct_graph()*
5. *optional* write the training data for the seq2seq model *akglib.AKG.write_seq2seq_train()*

At any point, save your data with the *akglib.AKG.save()* method.

1.5 Train FrameBert model

First, `frame_cluster_frames` needs to be executed. Navigate to the directory and use the command:

```
python3 frame_cluster_frames.py
```

Use this pattern for every mentioned file. This classifies the frames into 10 clusters. These classifications need to be updated in the training corpus with *frame_change_csv*. To split the corpus in a train and a test file, execute: *frame_split_corpus*.

To further preprocess our data and generate tsv-files from the csv-files use *frame_bert_prepare_data*. It also contains the function to convert the argument texts into feature vectors.

To start the training of our BERT fine-tuning model, execute `frame_bert_train`. By executing the file, the pretrained model (bert-base-cased) gets downloaded which is about 400mb. When the model is trained the files created are in *../data/frames/outputs/frames40ep/* . We have to zip the model with the command:

```
~/data/frames/outputs/frames40ep/ tar cfvz frames40ep.tar.gz config.json␣
↪pytorch_model.bin
```

The zipped package needs to be moved to the directory *cache/*. Do that by executing:

```
cp data/frames/outputs/frames40ep/frames40ep.tar.gz data/frames/cache/
```

For evaluating on the test set, use `frame_bert_eval`.

The file *frame_bert_interface* contains the function to predict the frame of a single argument, which is used by the argument knowledge graph.

CHAPTER

TWO

EXEMPLARY RESULTS

These are the results of the first arguments that you get with running `/source/interface` script in batch mode. Obviously not all found triples are really opposing the queryied argument and the seq2seq model is "on point".

Argument Batman would beat Superman in a duel. He has more equipment at his disposal. He is also richer than Superman.

Stance PRO

Triples:

```
[('Batman', 'beat', 'Superman'),
('It', "was n't", 'Batman'),
('Batman',
'has been',
'able to defeat opponents physically more powerful than Batman'),
('Batman', 'is', 'much bigger than Batman'),
('Batman', 'to survive', 'much more than you or I'),
('Batman', 'is', 'most popular than ever'),
('Batman', 'is', 'best'),
('Batman', 'would defeat', 'Iron Man'),
('Batman', 'wants to punch', 'people at night'),
('Batman', 'is', 'superior in raw intellect')]
```

seq2seq:

```
the us that not . . . . . that . . . . . . . . . . . . . . <EOS>
the us . . . . . . . . . . . . . the . . . . . . . . . . . . . the . . . <EOS>
the us that not the us . . . . the . . . . the . . <EOS>
the us that the the . the . . the . . . the . . . the . . . the . . <EOS>
a it not a . . . . . . . . . . . . . . . . <EOS>
the us that the by the . by . the . . . by . the . . . . . <EOS>
the the the the the . the . the . the . . the . . the . . . <EOS>
the us that the the of . the . of . . . the . . . <EOS>
the the that the the the . for the . the . . the . . the . . for the . . .
↪<EOS>
the us that not the . us . . to . . . to . . . . to . . . . <EOS>
```

Argument Immigrants are good for the economy. They do the jobs noone else wants to do.

Stance PRO

Triples:

```
[('migrants', 'came', 'in 2015'),
('migrants', 'coming', 'to Europe'),
('migrants', 'commit', 'a serious crime'),
('migrants', 'came', 'to Europe in 2015'),
('migrants', 'came', 'in to Germany'),
('migrants', 'came', 'into all of Europe')]
```

seq2seq:

```
in . . . . . . . . . . . . . . . . . . . . . . . . . . . . . <EOS>
. . . . . to . to . . . . to . . . . . . to . . . . . . . <EOS>
. . . . . . . . . . . . . . . . . . . . . . . . <EOS>
in . . . . . . . . to . . to . . . . . . . . . . . . . . <EOS>
in . . . . . . . . to . . . . . to . . . . . . . . . . . <EOS>
all of . . . . . . . . . . . . . . . . . . . . . <EOS>
```

Argument Being an adult is worse than being a child. You have to work almost every day, adults don't have free time.

Stance CON

Triples:

```
[('adults', 'have become', 'hypocritical'),
('adults', "do n't grow", 'new brain cells *'),
('adults', 'have faced', 'this era in their lives'),
('adults', "do n't understand", 'Women and the black body political system
↪'),
('You', ''re certified', 'adult'),
('I', 'can trust', 'adults'),
('adults', 'from harming', 'responsible adults'),
('adults', 'are held', 'responsible for adults actions'),
('adults', 'know', 'what The adults're doing'),
('adults', 'make', 'plans for adults future always')]
```

seq2seq:

```
that are that . that . are . . . . that . <EOS>
```

Argument The minimum wage should be raised. Everyone should be able to afford a good lifestyle and not worry about money all the time.

Stance PRO

Triples:

```
[('everyone', 'should be', 'able to'),
('You', 'are not', 'everyone'),
('everyone', 'to know', 'about'),
('everyone', 'wants', 'happy music all the time'),
('everyone', 'knows', 'It'),
('It', 'would benefit', 'everyone'),
('everyone', 'to be', 'A right'),
('everyone', 'can', 'be'),
('everyone',
'should be allowed',
```

(continues on next page)

(continued from previous page)

```
'the right to Life , Liberty , and the Pursuit of Happiness'),
('everyone', 'includes', 'all humans and all sentient animals')]
```

seq2seq:

```
. . . . . . . . . . . . . . <EOS>
that don that that . . . . . <EOS>
have have this . . . have <EOS>
don that that the . and . <EOS>
you that you . . . . . . . . . . . . <EOS>
i can t . . . . . . . . . <EOS>
. . . . . . . . . . . . . . . . . . . . . <EOS>
that are are . . . . that . . . . . . . . that . <EOS>
. . . the the . . <EOS>
. . . . that . . . . . . that . . <EOS>
```

Argument Donald Trump is a good President. He says what he thinks and doesn't care about feelings.

Stance PRO

Triples:

```
[('Donald Trump core', 'will believe', 'in Donald Trump to the end'),
('Donald Trump wife', 'hardly speaks', 'English even though')]
```

seq2seq:

```
. . . . . . . . . . . . . . . . . . . . . <EOS>
it doesn t . . . . . <EOS>
has has has . more . . more than . . . . . more than . . . <EOS>
. is . . . . . . . . . . . . <EOS>
. you that you . more than . . . . . . . <EOS>
. is . . . . . . . . . . . . . . . <EOS>
. . . . . . . . . . . . . . . . <EOS>
. . . . . . . . . . . <EOS>
. . . to to . to . . . to . <EOS>
. is . . . . . . . . . . . . . . . <EOS>
```

Argument The USA should pull their troops from the Middle East. It's none of their business, they should let them figure out themselves.

Stance CON

Triples:

```
[('The US', "wo n't pull", 'out of Afghanistan in 2011'),
('The US', 'trained', 'Chinese troops'),
('The US', 'sends', 'a few troops'),
('The economy', 'from directly challenging', 'The US'),
('The US', 'did', 'it as a pretext'),
('The US', 'took', 'it by force'),
('The US', 'should not change', 'the metric system'),
('The US', 'should ban', 'certain people of having guns'),
('The US', 'should pay', 'for a wall'),
('The US', 'refers', 'To')]
```

seq2seq:

```
. . . . to . to . . to . . . <EOS>
you are not . . . . . . . . . . . . . . . . . . . . . . . . <EOS>
. . . . to . to . . <EOS>
all . . . . . . . . . . . . <EOS>
. . . . . . . . . . <EOS>
it would . . . . . . . . . . <EOS>
. . . . to . . to . . . . . <EOS>
. . . . . . . . . . . . . . . <EOS>
. of the . of . . . . . to . . . <EOS>
all . . . . . . . . . . . . . . . <EOS>
```

Argument We must have some stricter controls on the content of entertainment that is viewed by teenagers. They might get violent and only think about murder.

Stance CON

Triples:

```
[('News programs that use pundits', 'are looking', 'for entertainment')]
```

seq2seq:

```
. . . . to to . the . . . . to . . . <EOS>
. . . . . . . . . . . . . . <EOS>
```

CHAPTER

THREE

EVALUATION

Assignee Oliver Zobel

3.1 Survey

Google Forms Survey to evaluate our generated counterarguments

- Ask about 20 people
- Rate certain characteristics of counterarguments

3.2 Content

Rating one c_arg given arg, c_arg

- grammar: scale 1-5
- logic: scale 1-5
- relevance: scale 1-5
- ethics: scale 1-5
- tone: scale 1-5
- opinion vs fact: scale 1-5
- convincingness: scale 1-5
- other criteria: open question

Ranking three c_args given one arg

- are there other, possibly better c_args: open question

Comparing two c_args

- **logic: c_arg1 or c_arg2** reason: open question
- **relevance: c_arg1 or c_arg2** reason: open question
- **convincingness: c_arg1 or c_arg2** reason: open question

CHAPTER

FOUR

OVERALL DESIGN

The project is separated in four tasks.

The first task is to create the *Argument Knowledge Graph*, necessary preprocessing included. Several measures are applied to the graph data to weight the nodes importance, respective to a query. We use the args.me corpus (see *Data*).

The next task is a frame classifier (see *FrameBert*). This is a BERT model for sequence classification applied to frame classification. It is trained on a separate corpus with annotated frames. The main function is the application and prediction of a passed argument to find better and more fitting arguments.

The *Seq2Seq Model* trains on the whole graph and generates natural language from the passed triples.

For the *Evaluation* a survey needs to be created with different aspects to focus on. It includes looking at one argument to rate, ranking different counter arguments and comparing different possible counter arguments. Since our natural language output doesn't feel very natural, (see *Exemplary results*) we couldn't evaluate the final counter arguments.

CHAPTER

FIVE

DATA

Assignee Jan Stanicki

For the creation of the argument knowledge graph, we use the Args.me Corpus[1]

5.1 Overview

The args.me corpus is a project by Yamen Ajjour, Henning Wachsmuth, Johannes Kiesel, Martin Potthast, Matthias Hagen and Benno Stein. It comprises 387 606 arguments crawled from four debate portals in the middle of 2019. The debate portals are Debatewise, IDebate.org, Debatepedia, and Debate.org. The arguments are extracted using heuristics that are designed for each debate portal. The corpus consists of one JSON-file. The average count of arguments per debate in our datasetamounts to 6.5. Most debates include 6 to 10 arguments. Only a few debates reach more than 20 arguments.

5.2 Statistics

- 387606 arguments from 59637 debates
- count of arguments total: 387606
- count of pro stance: 200099
- count of con stance: 187507
- count of debates: 59637

5.3 Structure

Each argument in the corpus contains the following information:

- premises
  - text: number of sentences
  - stance: “PRO” / “CON”
- context
  - sourceId: e.g. “c67482ba-2019-04-18T13:32:05Z”

[1] Ajjour, Y., Wachsmuth, H., Kiesel, J., Potthast, M., Hagen, M., and Stein, B. (2019b). Data Acquisition for Argument Search: The args.me corpus. In KI 2019. Springer.

  - previousArgumentInSourceId: same as sourceId
  - acquisitionTime: “2019-04-18T13:32:05Z” year-month-dayThours:minutes:secondsLetter
  - discussionTitle: string of topic, long
  - sourceTitle: Type of Debate | discussionTitle | url of source page
  - sourceUrl: concrete url to argument
  - nextArgumentInSourceId: sourceId of next argument
  - id: e.g. “c67482ba-2019-04-18T13:32:05Z-00000-000”
- conclusion: “string”, short sentence or topic

5.4 Example

```
{"premises": [{"text": "My opponent forfeited every round. None of my␣
↪arguments were answered. I don\u2019t like the idea of winning by␣
↪default, but here we are.Tule: it\u2019s good for students to get␣
↪involved and address big issues like teen pregnancy. You need to be able␣
↪to answer arguments like mine and not simply prepare for an abstinence-
↪only type of response. You should also be aware that, in the U.S.,␣
↪condoms may be sold to minors in ANY state. A retailer who says it is␣
↪illegal to sell you them is, frankly, wrong.",

"stance": "CON"}],

"context": {"sourceId": "c67482ba-2019-04-18T13:32:05Z",
↪"previousArgumentInSourceId": "", "acquisitionTime": "2019-04-
↪18T13:32:05Z", "discussionTitle": "Contraceptive Forms for High School␣
↪Students", "sourceTitle": "Debate Argument: Contraceptive Forms for High␣
↪School Students | Debate.org", "sourceUrl": "https://www.debate.org/
↪debates/Contraceptive-Forms-for-High-School-Students/1/",
↪"nextArgumentInSourceId": "c67482ba-2019-04-18T13:32:05Z-00001-000"}, "id
↪": "c67482ba-2019-04-18T13:32:05Z-00000-000",

"conclusion": "Contraceptive Forms for High School Students"},
```

CHAPTER

SIX

ARGUMENT KNOWLEDGE GRAPH

Assignee Julius Unverfehrt

For the actual module documentation, see *akglib module*

6.1 Outline

Input:

The graph is created from args.me arguments which consist of texts of variable length, which are annoted with additional information, see *Data*.

Tasks:

- Reduce input to (several) triples (SUBJ:PRED:OBJ)
  1. Resolve co-references, see *Neuralcoref*
  2. Drop sequences without punctuation or verb, since supervised-openIE needs those
  3. Extract triples, see *Supervised-openIE*
- Add triples to the graph with SUBJ, OBJ as vertices and PRED as edge, see *Construction*
- Merge similar vertices to reduce complexity and increase graph density
- Annotate additional information (conclusion, stance, argument membership)

Output:

The best matching triples that counter a queried argument, see *Query*.

Tasks:

- Identify relevant graph sections
- Rank fitting triples using page rank, similarity measures and frames

6.2 Neuralcoref

To resolve co-references, we use the neuralcoref[1] implementation from huggingface, which is fast and yields satisfying results. The greediness is kept at the 0.5 default value. Identified co-references are resolved, with the idea of making more node merges in the graph possible.

[1] https://github.com/huggingface/neuralcoref

6.3 Supervised-openIE

We use an implementation[2] of openIE from Gabriel Stanovsky et al.[3], which splits sentences at all verb positions and creates triples with SUBJ:PRED:OBJ. For all possible solutions a confidence measure is computed. We take the most confident solution, which tends to be only a part of the whole sentence. The parse is very flat on the dependency level (SUBJ is ARG0, ARG1, etc., same for OBJ). Generally we found out while working on the project, that supervised-openIE might not have been the best parser. For further work we would recommend to try out deeper dependency parser.

6.4 Construction

For adding the triples to the graph, following steps are made:

- Merge vertices if cosine TF-IDF similarity to an existing one is greater than 0.9
- Remember performed merges to improve run time
- Keep shorter vertex (debatable, information loss definitely happened)
- Allow only one merge per existing vertex to prevent further information loss

The cosine check proved to be pretty slow after a certain node threshold was met, which is why we used Sequence matching in later approaches. In this version, the cosine similarity is still in place though.

6.5 Query

A query to the graph consists of an argument with the respective stance. Following steps are taken:

- The argument is processed into triples, with the same method used to construct the graph.
- Relevant graph sections are identified via a simple sequence matcher (similarity ratio of around 0.9 proved to be best). This method proved to be way faster than cosine TF-IDF and yields comparable results.
- Collect all links in the proximity that oppose the query arguments' stance
- Order by
    1. Similarity to the query, presuming that good counterarguments have a high similarity in the vocabulary[4]
    2. Same frame
    3. PageRank
    4. The values are normalized and weighted (refer to *akglib.AKG.get_counter_triples()* for further details)
- Return the (ten) best triples

[2] https://github.com/gabrielStanovsky/supervised-oie

[3] Stanovsky, G., Michael, J., Zettlemoyer, L. & Dagan, I. (2018). Supervised Open Information Extraction. In NAACL 2018.

[4] Wachsmuth, H. [Henning], Syedand, S. & Stein, B. (2018). Retrieval of the Best Counterargument without Prior Topic Knowledge. Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics 2018.

CHAPTER

SEVEN

FRAMEBERT

Assignee Jan Stanicki

7.1 Outline

For the training of the frame classifier, the webis-argument-framing-19[1] corpus is used. First, we need less frames, so we cluster the similar specific frames into a more general frame. Then we write the new frames in the corpus and split it in a train and test file. Next the model is trained and then evaluated on the test corpus. Ultimately a function for integration is written which predicts the frame for a single argument.

7.2 Preprocessing

The following modules & functions are used:

`frame_cluster_frames`

read_webis() reads train corpus and saves the arguments and the respective frame in a list ([[frame1, arg1], [frame2, arg2], [frame2, arg3],...]).

preprocessing() removes stopwords and punctuation from arguments.

vectorize_arguments() vectorizes arguments with TF-IDF, then predicts a cluster for every argument with KMeans.

frameDict() creates a dict with frame as key and for every argument belonging to the frame the predicted cluster is appended to a list (frame: [clus1, clus2, clus1]).

most_common_cluster() takes the frameDict and determines the most frequent cluster by using counter.most_common(). The key is the frame and the value is a tuple consisting of the most frequent cluster and the frequency ({frame: (clus1, 4)}).

create_cluster_sets() A set for every cluster is created. Every frame gets added to the set which was predicted the most frequent cluster. These sets are saved in a pickle-file.

`frame_change_csv`

load_cluster() loads the clusters from pickle-file.

[1] https://webis.de/data/webis-argument-framing-19.html

write_in_csv() replaces frame annotations with cluster numbers. Some clusters are combined because they contain very similar frames. In the end there are 7 different general frames

structure: cluster = general frame name ==> how it's annotated in the corpus:

```
cluster0 = rights ==> 0
cluster1, cluster2 = international politics ==> 1
cluster3 = individual rights ==> 2
cluster4 = education ==> 3
cluster5, cluster6, cluster7 = climate change ==> 4
cluster8 = economics ==> 5
cluster9 = health care ==> 6
```

frame_split_corpus

train_test() splits the corpus in train (80%) and test (20%) set.

7.3 BERT model

frame_bert_prepare_data

Takes the train and test csv-files and transforms it to tsv-files which is standard working with BERT[2] models.

InputExample class that constructs an InputExample for further processing.

DataProcessor base class that collects the InputExamples from the train and test set.

MultiClassificationProcessor class that processes multi classification dataset.

InputFeatures class that defines the structure of a feature vector.

convert_example_to_feature() converts an argument text to a feature vector.

`frame_bert_train`

FrameBert class that extends the BertForSequenceClassification model by a softmax layer.

Creates needed directories if they don't exist. Then calls functions from frame_bert_prepare_data to process training data and feeds the model. Next the model is trained. By running this code, the model trains with 40 epochs.

`frame_bert_eval`

Equivalent to frame_bert_train but processes and uses the test set. Also it creates an evaluation file eval_results.txt.

frame_bert_interface

classifiy_single_argument() function that takes an argument text and processes it to a feature vector, then takes fine-tuned model to predict a frame.

[2] https://github.com/maknotavailable/pytorch-pretrained-BERT

7.4 Lessons learned

The first big problem we encountered was the choice of a training corpus for training the frame classification system. The only fitting corpus we found was the Webis-argument-framing-19 dataset which had more than 1500 different annotated frames. So to make this corpus usable we needed under 20 frames which we tried achieving by clustering similar frames. The first attempt was embedding the frame annotations (e.g. 'economics', 'taxes',...) with BERT but these clusters where completely random, so we discarded this approach. The second approach was vectorizing the arguments with TF-IDF, so we got a predicted cluster for every single argument. To assign a frame to a cluster, we needed to find every argument annotated with the same frame and look for the most common predicted cluster (frame: [arg1->clus1, arg2->clus2, arg3->clus1, arg4->clus3] ==> frame: clus1). There were several problems quite difficult to solve. Firstly, if a frame had not one cluster which was the most frequent one but there were several most frequent ones. We used the most_common() function from collections which then selects a random one. This adds a bit of noise to the data, but no other option came to mind. Second, the final distribution of clusters was even more uneven than thought. That explains the poor performance of the BERT model. Cluster2 is a pretty big one which contains very frequent frames, so very much of the annotation are cluster 2. Maybe clustering similar frames manually would have been a better idea but the way it is now isn't suitable for training. This project being the first one using the cluster of the Institute of Computational Linguistics in Heidelberg was also a big challenge to organize and move the necessary data. For this, more time would have avoided some stress in organization. It was also the first time working with a neural network, which was very interesting getting to know it. Also it was very rewarding when it finally starts working. Preparing the training corpus took a long time unfortunately, so the actual frame classifier with BERT had to work pretty fast. More time for fine-tuning and improving this model or time to experiment with different setups and parameters would have been beneficial. When looking in the eval_results.txt it stands out that the measures all yield the same value. This indicates that it's not working properly which probably is the consequence of the uneven distribution of frames. Altogether it was a project with a lot new elements to work with, which was challenging but also we learned a lot by dealing with the problems.

CHAPTER

EIGHT

SEQ2SEQ MODEL

8.1 Outline

Task:

The goal of the Seq2Seq Model is to learn how to recreate the argument sentences from the triples and to fill in the missing sentence parts.

Input:

The Seq2Seq Neural Network Model is trained with the graph triples extracted from the AKG and the corresponding argument as gold label.

Output:

Sentences generated from the Seq2Seq Model.

Our Model:

- The test corpus had around 1.5 million triple/argument pairs
- The hidden size of the NN was 300
- The learning rate was 0.01
- 20000 iterations over the data

The code for the Seq2Seq model was used from the PyThorch Tutorials Web-page and was adapted and extended for our task[1]

8.2 Usage

The `seqtoseq_train_model` module loads the corpus from the data folder and starts training the Neural Network. After the training is finished the module saves the model in *data/seqtoseq/*. There are two different models for the Encoder NN and the Decoder NN.

The `seqtoseq_load_model:mod:` module loads the pretrained models for the Encoder and the Decoder from the seqtoseq folder and generates sentences from the triples who were saved in *data/triples/*.

[1] https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html#training-the-model

8.3 Results:

Even with a much bigger corpus and a lot more computing power the results were frustratingly bad. The reason behind it is most likely the big difference in word length between the triples and the arguments.

CHAPTER

NINE

INTERFACE MODULE

Small interface for the pretrained args-me_15000_graph.

If you want to use your own sentences for the demo, feel free to adjust /data/arguments.txt, don't change the formating though:

- first line : argument
- second line: stance
- repeat

For own trainings refer to the documentation.

author Julius Unverfehrt

version 0.9

CHAPTER

TEN

AKGLIB MODULE

Provides the class AKG, representing an argument knowledge graph with the methods needed to create and access it.

The graph works with the args-me corpus but could be used with any corpus, small adjustments assumed. Further detail on how to use this class is provided in the Getting started section of the documentation.

author Julius Unverfehrt

version 0.9

class akglib.**AKG**(*corpus_path='../data/args-me.json'*)

Bases: object

This class provides all methods needed to process the args-me corpus into an argument knowledge graph. Coreferences in the documents are resolved with neuralCoref, the information is extracted with supervised-openIE into triples: SUBJ-VERB-OBJ, where SUBJ and OBJ will be nodes in the graph, with VERB as edge. When put into the graph, similar nodes (cosine TF_IDF) are merged (only once per existing node, to prevent information loss). After creation, querys of argument + stance can be made, to recive matching triples that oppose the argument.

Parameters **corpus_path** (*string*) – Path to the corpus (or sample), defaults to "../data/args-me.json"

construct_graph()

Iterates over the processed corpus and adds the triples found by supervised-openIE to the igraph.

If similarity (cosine tf-idf) to a existing node is detected, the nodes will be merged into the shorter one (only once per existing node, to restrict information loss). The merges are therefore memorized, which also improves runtime.

Raises **AttributeError** – corpus not loaded previously and/or not processed

get_cos_sims(*arg*)

Aquires the tf-idf cosine similarites for a new node, respective to the nodes of all arguments in the graph. Returns a list with the similarities, the indices refering to the nodes indices of the graph. The new node is temporarily added to the graph, which prevents creating new lists every time a similarity check is made.

Parameters **arg** (*string*) – one single node

Returns similiarities

Return type list(float)

get_counter_triples(*query*, *stance*, *filename=None*)
Iterates over the graph to identify similar nodes to the query.

For the sake of runtime, we use a simple sequence match for that task, which is way more efficient than cosine similarity and yet yields comparable results. Triples in the neighborhood of identified nodes are estimated as qualified if their stance oposes the queries one. They are then sorted by the normalized scores of:

- respective similarity (cosine TF-IDF), assuming that good counter arguments are similar
- pagerank, assuming that a high pagerank hints to a meaningful node
- frames, assuming that good counterarguments are in the same frame

IMPORTANT: specify a filename if you want to process the triples later, since seq2seq isn't integrated.

Parameters

- **query** (*string*) – argument to which we want the counter argument
- **stance** (*string*) – stance of the argument
- **filename** (*string*) – to saved results on hd, provide a path here, defaults to None

Returns best counter triples (max 10)

Return type list(tuple(string))

Raises

- **TypeError** – query or stance are no strings
- **AttributeError** – corpus and/or graph not loaded
- **ValueError** – no triples extracted from query (might be too short)
- **OSError** – only if filepath is specified

load(*data_proc=False*, *graph=False*)
Load processed data and/or the processed graph, using the location provided from corpus_path.

Parameters

- **data_proc** (*boolean*) – should processed data be loaded, defaults to False
- **graph** (*boolean*) – should the graph be loaded, defaults to False

Raises **OSError** – if files not found / unable to open them

load_corpus(*corpus_path=None*)
Loads the (args.me-)corpus, provided in .json format. Uses the path given from class initialization, alternativly provide one here.

Parameters **corpus_path** (*string*) – Path to the corpus (or sample), defaults to None

Raises

- **TypeError** – invalid corpus_path format

- **OSError** – filesystem errors

my_plot()
Plot the graph with predifined configuration, might take a while for big graphes.

process_arguments()
Main routine for processing the corpus.

Iterates over each argument and executes neuralCoref and supervised-OpenIE. Sentences with triple indicator marks are saved for later, when a Language model which transforms triples into sentences will be trained (other module). Every 1000 arguments, the processed data will be saved, using the location provided from corpus_path.

Raises

- **AttributeError** – corpus not loaded previously
- **TypeError** – args-me structure not detected
- **KeyError** – args-me arguments key not found

save(*data_proc=False*, *graph=False*)
Save processed data and/or the processed graph, using the location provided from corpus_path.

Parameters

- **data_proc** (*boolean*) – should processed data be saved, defaults to False
- **graph** (*boolean*) – should the graph be saved, defaults to False

Raises **OSError** – if not able to write

write_seq2seq_train(*marked=True*)
Writes the training data for the seq2seq model, optionally with markers for the argument positions. Uses corpus_path for the save path.

Parameters **marked** (*boolean*) – if the positions of the arguments should be marked, defaults to True

Raises **OSError** – No access to writefile, or self.data_proc not provided / corrupt

CHAPTER

ELEVEN

AKGCREATE MODULE

Short snippet to batch create the argument knowledge graph from akglib.py. Adjust to your liking and make sure to refer to the documentation.

The following order should be kept:

1. Create the class (specify the path to your args-me corpus as parameter)
2. Load the corpus AKG.load_corpus
3. Process the corpus AKG.process_arguments
4. Construct the graph AKG.construct_graph
5. write the training data for the seq2seq model AKG.write_seq2seq_train

author Julius Unverfehrt

version 0.9

CHAPTER

TWELVE

FRAME_BERT_INTERFACE MODULE

This interface classifies a single argument and assigns a frame. It's the interface to be integrated in other modules.

author Jan Stanicki

class frame_bert_interface.**FrameBert**(*config*, *num_labels=10*)

Bases: pytorch_pretrained_bert.modeling.BertForSequenceClassification

BertForSequenceClassification extended with a softmax layer

frame_bert_interface.**classify_single_argument**(*input_argument*)

takes an input_argument, processes that to a feature vector and classifies this. returns prediction.

Parameters **input_argument** (*string*) – is the argument to be classified.

CHAPTER

THIRTEEN

FRAME_BERT_PREPARE_DATA MODULE

Transforms csv-files to tsv-files and includes some helper classes and the function to convert argument texts to feature vectors.

author Jan Stanicki

class frame_bert_prepare_data.**DataProcessor**
Bases: object

Base class for data converters for sequence classification data sets.

get_dev_examples(*data_dir*)
Gets a collection of ‘InputExample‘s for the dev set.

get_labels()
Gets the list of labels for this data set.

get_train_examples(*data_dir*)
Gets a collection of ‘InputExample‘s for the train set.

class frame_bert_prepare_data.**InputExample**(*guid*, *text_a*, *text_b=None*, *label=None*)
Bases: object

A single training/test example for simple sequence classification.

class frame_bert_prepare_data.**InputFeatures**(*input_ids*, *input_mask*, *segment_ids*, *label_id*)
Bases: object

A single set of features of data.

class frame_bert_prepare_data.**MultiClassificationProcessor**
Bases: *frame_bert_prepare_data.DataProcessor*

Processor for multi classification dataset.

get_dev_examples(*data_dir*)
See base class.

get_labels()
See base class.

get_train_examples(*data_dir*)
See base class.

frame_bert_prepare_data.**convert_example_to_feature**(*example_row*)
Converts an argument example to a feature vector.

Parameters **example_row** (*tuple*) – tuple of example object, label_map, max_seq_length, tokenizer and output_mode needed for preprocessing

CHAPTER

FOURTEEN

FRAME_CHANGE_CSV MODULE

Loads clusters from pickle file and writes a csv

author Jan Stanicki

frame_change_csv.**load_cluster**(*clusterfile_path*)
Loads the clusters saved in .pickle file and returns it.

Parameters **clusterfile_path** (*string*) – path to clusterfile

frame_change_csv.**main**()

frame_change_csv.**write_in_csv**(*original_corpus_path*, *modified_corpus_path*)
Assigns every frame name in corpus file to one of the seven classes (frames) and writes it in a corpus file (Webis-argument-framing-simplified.csv).

Parameters

- **original_corpus_path** (*string*) – path to the original corpus
- **modified_corpus_path** (*string*) – path where the modified corpus is saved

CHAPTER

FIFTEEN

FRAME_SPLIT_CORPUS MODULE

Splits Corpus in train and test file (4:1)

author Jan Stanicki

frame_split_corpus.**train_test**(*in_path*, *train_out_path*, *test_out_path*)
reads corpus file and splits data in 80% train and 20% test and saves it in respective files

CHAPTER

SIXTEEN

SEQTOSEQ_TRAIN_MODEL MODULE

class seqtoseq_train_model.**AttnDecoderRNN**(*hidden_size*, *output_size*, *dropout_p=0.1*, *max_length=100*)

Bases: torch.nn.modules.module.Module

forward(*input*, *hidden*, *encoder_outputs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

initHidden()

class seqtoseq_train_model.**DecoderRNN**(*hidden_size*, *output_size*)

Bases: torch.nn.modules.module.Module

forward(*input*, *hidden*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

initHidden()

class seqtoseq_train_model.**EncoderRNN**(*input_size*, *hidden_size*)

Bases: torch.nn.modules.module.Module

forward(*input*, *hidden*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of

running the registered hooks while the latter silently ignores them.

initHidden()

class seqtoseq_train_model.**Lang**(*name*)

Bases: object

addSentence(*sentence*)

addWord(*word*)

seqtoseq_train_model.**asMinutes**(*s*)

seqtoseq_train_model.**filterPair**(*p*)

seqtoseq_train_model.**filterPairs**(*pairs*)

seqtoseq_train_model.**indexesFromSentence**(*lang*, *sentence*)

seqtoseq_train_model.**normalizeString**(*s*)

seqtoseq_train_model.**prepareData**(*lang1*, *lang2*, *reverse=False*)

seqtoseq_train_model.**readLangs**(*lang1*, *lang2*, *reverse=False*)

seqtoseq_train_model.**showPlot**(*points*)

seqtoseq_train_model.**tensorFromSentence**(*lang*, *sentence*)

seqtoseq_train_model.**tensorsFromPair**(*pair*)

seqtoseq_train_model.**timeSince**(*since*, *percent*)

seqtoseq_train_model.**train**(*input_tensor*, *target_tensor*, *encoder*, *decoder*, *encoder_optimizer*, *decoder_optimizer*, *criterion*, *max_length=100*)

seqtoseq_train_model.**trainIters**(*encoder*, *decoder*, *n_iters*, *print_every=1000*, *plot_every=100*, *learning_rate=0.01*)

seqtoseq_train_model.**unicodeToAscii**(*s*)

CHAPTER

SEVENTEEN

SEQTOSEQ_LOAD_MODEL MODULE

class seqtoseq_load_model.**AttnDecoderRNN**(*hidden_size*, *output_size*, *dropout_p=0.1*, *max_length=100*)

Bases: torch.nn.modules.module.Module

forward(*input*, *hidden*, *encoder_outputs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

initHidden()

class seqtoseq_load_model.**DecoderRNN**(*hidden_size*, *output_size*)

Bases: torch.nn.modules.module.Module

forward(*input*, *hidden*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

initHidden()

class seqtoseq_load_model.**EncoderRNN**(*input_size*, *hidden_size*)

Bases: torch.nn.modules.module.Module

forward(*input*, *hidden*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

initHidden()

class seqtoseq_load_model.**Lang**(*name*)

Bases: object

addSentence(*sentence*)

addWord(*word*)

seqtoseq_load_model.**evaluate**(*encoder*, *decoder*, *sentence*, *max_length=100*)

seqtoseq_load_model.**evaluateRandomly**(*encoder*, *decoder*, *n=5*)

seqtoseq_load_model.**gen_sentence**(*triple*)

seqtoseq_load_model.**indexesFromSentence**(*lang*, *sentence*)

seqtoseq_load_model.**normalizeString**(*s*)

seqtoseq_load_model.**open_graphtriples**(*data*)

seqtoseq_load_model.**prepareData**(*lang1*, *lang2*, *reverse=False*)

seqtoseq_load_model.**readLangs**(*lang1*, *lang2*, *reverse=False*)

seqtoseq_load_model.**tensorFromSentence**(*lang*, *sentence*)

seqtoseq_load_model.**tensorsFromPair**(*pair*)

seqtoseq_load_model.**unicodeToAscii**(*s*)

CHAPTER

EIGHTEEN

APPENDIX

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

I

L

M

N

O

P

R

S

T

U

W