NLP Tasks

In this notebook, we will examine some of the currently popular NLP tasks. We use some of the popular open-sourced NLP frameworks/libraries to do that.

There will be no training (backpropagation) in this notebook. We are only interested in making predictions with models, to understand the nature of the tasks.

Tasks to cover in this notebook:

1. (extractive) Question Answering
2. Named Entity Recognition (also called token tagging, or, entity extraction)
3. Sentiment Analysis (a specific case of text classification)
4. Text Generation
5. Conversetional Agents

Important eye-opener: All of the tasks in this notebook could be modelled in another way than we do here. However, we tried to show you the most popular ways as of 2021 to perform these tasks. For example, sentiment analysis task could be modelled as a regression problem, rather than a multi-class classification problem.

1- Using AllenNLP

AllenNLP is an open source NLP framework formed by Allen Institute for AI, a private company & research institute. AI2 is founded by Paul Allen, one of the co-founders of Microsoft.

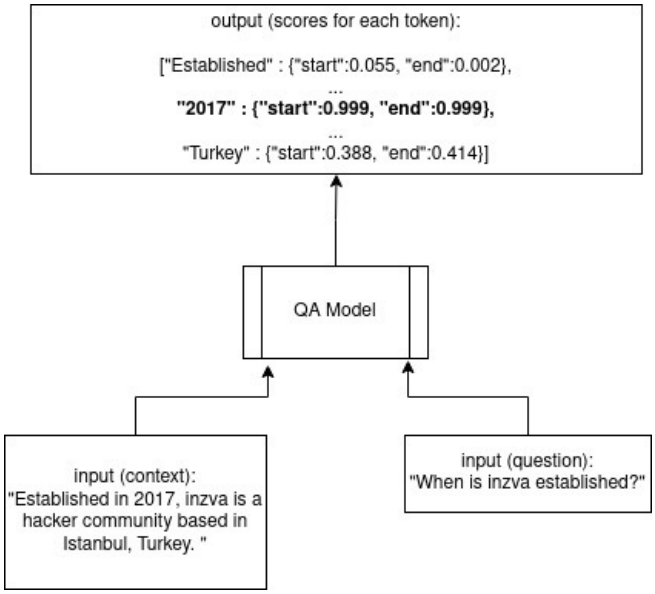
Codes below are obtained from AllenNLP demo site. There are more tasks there. We will examine only three of them.

- https://demo.allennlp.org

a- Reading Comprehension (Question Answering)

Question answering is the task of extracting text spans from context paragraphs. While doing the extraction model also takes a question into account.

- Inputs: Context Paragraph (tokenized string), Question (tokenized string)
- Outputs: A text span (de-tokenized list of tokens), named as the "Answer"
- Popular Methods: BERT based QA models



In the example, we see a context paragraph, giving information on inzva. Then, we have a question, asking inzva's foundation date.

We basically ask the model this question:

- **If we have N tokens in the context, what is the likelihood of n^{th}**
  n^{th}
  token to be the start token of the answer. Likewise, what is the likelihood for each token to be the ending token of the answer?
- By asking that, we find the most likely start and end positions for the answer span.
- Then we can get the answer span between the most likely start token and the most likely end token.
- In the example, token "2017" is the most likely token to be the start token. It is also most likely to be the end token. Between the most likely start token and the most likely end token, the only word is "2017" so we take that as the answer.

We install allennlp_models to use the pretrained models in AllenNLP

In [ ]:
```
!pip install allennlp allennlp_models
```

Then we import the Predictor class, which is a wrapper for ML Model classes.

In [ ]:
```
from allennlp.predictors.predictor import Predictor
import allennlp_models.rc
```

We instantiate a Predictor instance which is a QA Model in our case.

In [ ]:
```
qa_model = Predictor.from_path("https://storage.googleapis.com/allennlp-public-models/bidaf-elmo-model-2020.03.19
```

We make the predictions with the two necessary parameters. Question and the context passage. The answer will be selected from inside the context passage.

In [ ]:
```
results = qa_model.predict(
  passage="The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring Kear
  question="Who stars in The Matrix?")
```

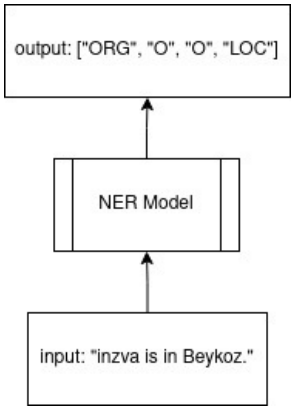
We print three things:

- Our context (just to see it)
- The question (just to see it)
- The answer, which is in the key *best_span_str* .

In [ ]:
```
print(' '.join(results['passage_tokens']), ' '.join(results['question_tokens']), results['best_span_str'], sep =
```

b- Named Entity Recognition (Tagging, Entity Extraction)

Entity recognition is the task of extracting particular spans from text, and sometimes classifying them to classes like *Person* , *Location* , *Orgazination* etc.



In the image, we see an input sentence with 4 input words. NER Model takes the input, and assigns an output class for each token. In this example, outputs are:

- inzva -> Organization
- is -> None
- in -> None
- Beykoz -> Location

Normally, tokens would be subwords, instead of words. In our example we consider words as tokens for simplicity.

To start getting predictions on NER, we import the tagging model from AllenNLP

In [ ]:
```
import allennlp_models.tagging
```

We again create a Predictor instance, which will be our model.

In [ ]:
```
ner_model = Predictor.from_path("https://storage.googleapis.com/allennlp-public-models/ner-model-2020.02.10.tar.g
```

The model makes a prediction.

In [ ]:
```
results = ner_model.predict(
  sentence="Did Uriah honestly think he could beat The Legend of Zelda in under three hours?")
```

We examine the prediction, looking at its keys.

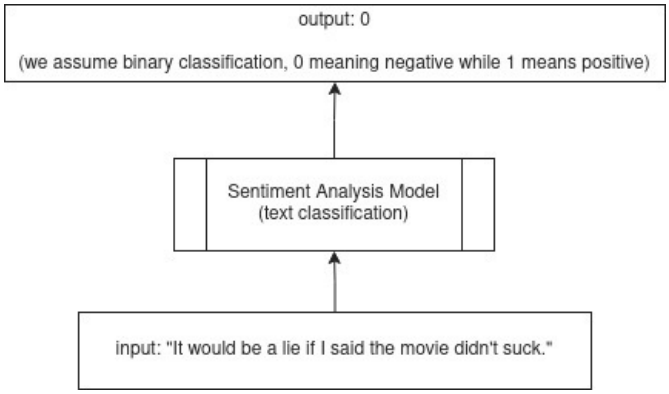
In [ ]:
```
results.keys()
```

We see that "words" and "tags" keys might be relevant to print out each word with their found class. So we print those lists.

In [ ]:
```
for word, tag in zip(results["words"], results["tags"]):
    print(f"{word}\t{tag}")
```

c- Sentiment Analysis

Sentiment Analysis is a text classification problem. One of the easiest ways to do it is to classify the text either as postive or negative.



In the example we see that the person thinks that the movie sucks (is bad).

This is because the person states that if they said the movie *didn't* suck, it would be a *lie* (this is a double negation example).

We import the classification module since the way that we model sentiment analysis is a classification problem.

In [ ]:
```
import allennlp_models.classification
```

We start a predictor class witht the pretrained sentiment analysis model's path.

In [ ]:
```
sentiment_classifier = Predictor.from_path("https://storage.googleapis.com/allennlp-public-models/basic_stanford_
```

We write a negative review.

In [ ]:
```
# Review of the movie "Joker" on Rotten Tomatoes
review = "A movie of a cynicism so vast and pervasive as to \
          render the viewing experience even emptier than its \
          slapdash aesthetic does."
```

We forward our negative review to the model.

In [ ]:
```
result = sentiment_classifier.predict(sentence=review)
```

We print "positive" if the model's confidence is higher on being positive than negative.

Notice that AllenNLP models the sentiment analysis task in a more complex way than binary classification.

They have multiple classes and also confidence values associated with each class.

In [ ]:
```
print('Positive with confidence of', result['probs'][0]) if result['label']=='1' else print('Negative with confic
```

2- Using Huggingface Transformers

Transformers is an open source NLP framework formed by Huggingface, a private company & community.

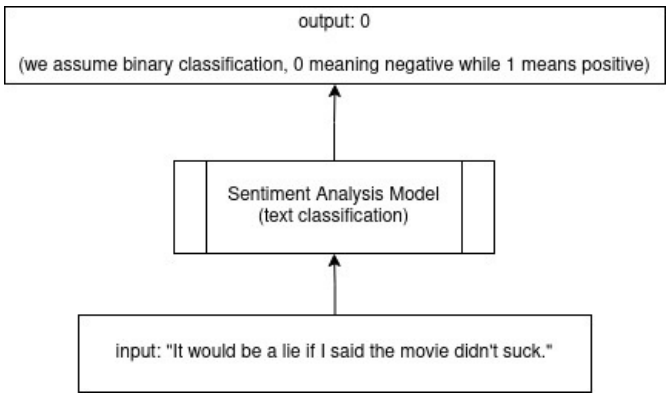
Note: You can also check https://huggingface.co/spaces to get a wider sense on NLP (and also other machine learning) tasks.

Using "Pipeline" for Sentiment Analysis

Pipeline class is a wrapper in Huggingface, just like the Predictor in AllenNLP.

It basically creates a more specialized class instance under the hood, based on the task name that you give.

Sentiment Analysis is a text classification problem. One of the easiest ways to do it is to classify the text either as postive or negative.



In the example we see that the person thinks that the movie sucks (is bad).

This is because the person states that if they said the movie *didn't* suck, it would be a *lie* (this is a double negation example).

In [ ]:
```
# Have either PyTorch >= 1.1 or TensorFlow >= 2.0. Then run the following command to install HuggingFace:
!pip install transformers
```

We start a Pipeline instance, particularly, a sentiment analysis pipeline.

Right after we create it, we forward an input to it. The input is "we love you".

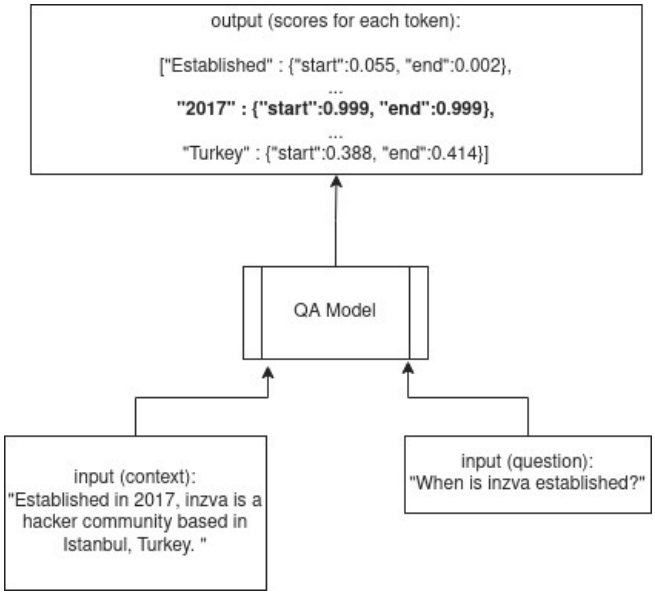
In [ ]:
```
from transformers import pipeline

print(pipeline('sentiment-analysis')('we love you'))
```

Using "Pipeline" for Question Answering

Question answering is the task of extracting text spans from context paragraphs. While doing the extraction model also takes a question into account.

- Inputs: Context Paragraph (tokenized string), Question (tokenized string)
- Outputs: A text span (de-tokenized list of tokens), named as the "Answer"
- Popular Methods: BERT based QA models



In the example, we see a context paragraph, giving information on inzva. Then, we have a question, asking inzva's foundation date.

We basically ask the model this question:

- **If we have N tokens in the context, what is the likelihood of n^{th}**
  n^{th}
  token to be the start token of the answer. Likewise, what is the likelihood for each token to be the ending token of the answer?
- By asking that, we find the most likely start and end positions for the answer span.
- Then we can get the answer span between the most likely start token and the most likely end token.
- In the example, token "2017" is the most likely token to be the start token. It is also most likely to be the end token. Between the most likely start token and the most likely end token, the only word is "2017" so we take that as the answer.

We start a QA pipeline, and name it as "nlp".

We also define one of our inputs (the context). It is an informative paragraph on one of the subtopics of Machine Learning.

In [ ]:
```
nlp = pipeline("question-answering")

context = r"""
Extractive Question Answering is the task of extracting an answer from a text given a question. An example of a
question answering dataset is the SQuAD dataset, which is entirely based on that task. If you would like to fine-
a model on a SQuAD task, you may leverage the examples/question-answering/run_squad.py script.
"""
```

We define the other part of our input (the question). It is a question about one of the subtopics in Machine Learning.

We can ask multiple questions for the same context paragraph, and vice versa in certain cases. For that, we ask another question.

We print:

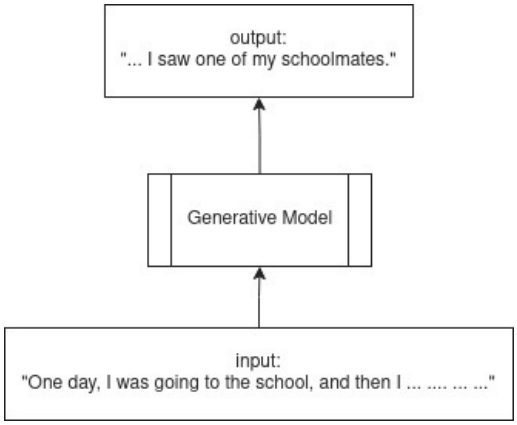
- the answer span
- the score for the answer (similar to confidence, but not a probability value)
- the location (index) of the start of the answer
- the location of the end of the answer

In [ ]:
```
result = nlp(question="What is extractive question answering?", context=context)
print(f"Answer: '{result['answer']}', score: {round(result['score'], 4)}, start: {result['start']}, end: {result[

result = nlp(question="What is a good example of a question answering dataset?", context=context)
print(f"Answer: '{result['answer']}', score: {round(result['score'], 4)}, start: {result['start']}, end: {result[
```

Text Generation with GPT-2

- Text Generation is essentially a language modelling task, however, lately it has started to become more.
- Some text generation models (like GPT-2) can be fine-tuned to perform a huge variety of NLP tasks (QA, classification etc.).
- Some newer text generation models (like GPT-3 and GPT-J) can even zero-shot a wider variety of tasks.



In this example, the task solely Text Generation, similar to writing a story. We will start the story, and expect the model to write new lines. We will use GPT-2 due to practical limitations.

We import specific GPT model and tokenizer classes to form our predictive system, then create tokenizer and model instances.

In [ ]:
```
# For this example, tensorflow >= 2.1 should be installed on the system

import tensorflow as tf
from transformers import TFGPT2LMHeadModel, GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

# add the EOS token as PAD token to avoid warnings
model = TFGPT2LMHeadModel.from_pretrained("gpt2", pad_token_id=tokenizer.eos_token_id)
```

We tokenize the input sentence (the beginning of our story). You can examine the tokenized object if you want.

In [ ]:
```
# encode context the generation is conditioned on
input_ids = tokenizer.encode('I enjoy walking with my cute dog', return_tensors='tf')
```

We generate the continuation of our story. Our arguments specify:

- max_length=50: do not generate more than 50 tokens
- num_beams=5: when you are thinking about the best possible stories, do not think about more than 5 stories at the same time.
- no_repeat_ngram_size=2: ?
- early_stopping=True: ?

We get the output (it is a list of tokens) then we join all of them to get one string.

In [ ]:
```
beam_output = model.generate(
    input_ids,
    max_length=50,
```

```
    num_beams=5,
    no_repeat_ngram_size=2,
    early_stopping=True
)

print("Output:\n" + 100 * '-')
print(tokenizer.decode(beam_output[0], skip_special_tokens=False))
```

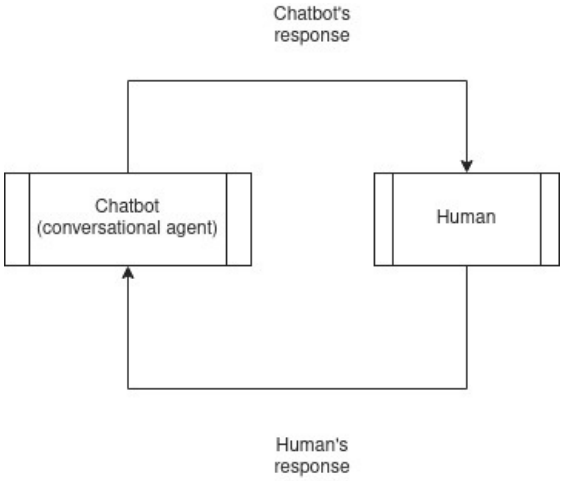
Conversational Agent

Conversational Agents are multi-component systems that are engineered to act as Chatbots. They take action for various queries:

- Order a pizza.
- Call my friend Suzie.
- What's the weather like today?
- ...

In this example, we are trying out an end-to-end conversational agent, that is only designed for chitchat.

It is a finetuned version of one of the GPT models, to do chitchat.



In [ ]:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

tokenizer = AutoTokenizer.from_pretrained("microsoft/DialoGPT-large")
model = AutoModelForCausalLM.from_pretrained("microsoft/DialoGPT-large")

# Let's chat for 5 lines
for step in range(5):
    # encode the new user input, add the eos_token and return a tensor in Pytorch
    new_user_input_ids = tokenizer.encode(input(">> User:") + tokenizer.eos_token, return_tensors='pt')

    # append the new user input tokens to the chat history
    bot_input_ids = torch.cat([chat_history_ids, new_user_input_ids], dim=-1) if step > 0 else new_user_input_ids

    # generated a response while limiting the total chat history to 1000 tokens,
    chat_history_ids = model.generate(bot_input_ids, max_length=1000, pad_token_id=tokenizer.eos_token_id)

    # pretty print last ouput tokens from bot
    print("DialoGPT: {}".format(tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0], skip_special_t
```