Introduction to Japanese Natural Language Processing Sample Chapters

This PDF includes two sample chapters, and is designed as an early preview of the book. While the chapters included are intended to be complete, they may be revised somewhat as work on the book continues.

To learn more about the book or reserve your copy, check the homepage at japanesenlp.com[1].

[1] https://www.japanesenlp.com/

2.1 An Introduction to fugashi

In this section you'll learn how to do Japanese tokenization using fugashi, a MeCab wrapper, and the unidic-lite dictionary.

surface	pos1	pos2	pos3	lemma	pron	kana	goshu
喫茶	名詞	普通名詞	一般	喫茶	キッサ	キッサ	漢
店	接尾辞	名詞的	一般	店	テン	テン	漢
と	助詞	格助詞	*	と	ト	ト	和
カフェ	名詞	普通名詞	一般	カフェ-cafe	カフェ	カフェ	外
の	助詞	格助詞	*	の	ノ	ノ	和
違い	名詞	普通名詞	一般	違い	チガイ	チガイ	和
は	助詞	係助詞	*	は	ワ	ハ	和
意外	形状詞	一般	*	意外	イガイ	イガイ	漢
と	助詞	格助詞	*	と	ト	ト	和
明確	形状詞	一般	*	明確	メーカク	メイカク	漢

This table is an example of the output available from fugashi and UniDic. Note how besides tokenization it includes a variety of information about each token. This is only some of the fields available in UniDic.

Setup

First you'll need to install fugashi and the dictionary.

fugashi is a wrapper for **MeCab**, a classic Japanese morphological analyzer. fugashi uses Cython to access MeCab's C interface, and also includes some convenient tweaks to make it easier to use in Python.

unidic-lite is a slightly modified version of UniDic 2.1.2. That version of UniDic is somewhat old, but it's small enough that it's easy to install, and high quality enough that it's sufficient for most applications. The **unidic** package on PyPI wraps the latest edition of UniDic, but due to a large increase in dictionary entries, it's harder to set up, so we won't use it for this tutorial.

At time of writing the latest version of fugashi is 1.1.0 and the latest version of unidic-lite is 1.0.8. unidic-lite will work on any system, and fugashi distributes ready-to-use "wheels" for OSX, Linux, and 64 bit Windows. (If you have another operating system you may have to build from source. If you have trouble please feel free to open an issue[1].)

```
%%capture
!pip install fugashi unidic-lite
```

Now that fugashi is installed, you can confirm it works by running it in the terminal. Try running `fugashi -O wakati` and then typing some Japanese. If you push Enter, your input text will be printed with spaces separating tokens. You can use `CTRL+D` to terminate the process. Here's some example output:

```
!echo "毎年東麻布ではかかし祭りが開催されます" | fugashi -O
    wakati
```

```
毎年 東 麻布 で は かかし 祭り が 開催 さ れ ます
```

Note: `wakati` comes from 分かち書き wakachigaki, which refers to the practice of writing Japanese with spaces included, as used in children's books and low resolution displays. In MeCab this refers to the special output mode that just separates tokens with spaces. Note that real wakachigaki uses spaces to separate bunsetsu, not tokens or words.

Next let's use fugashi in code. The main interface to the library is the `Tagger` object, which holds a variety of dictionary related state. The primary way to use the `Tagger` is to simply apply it to input text, which will return a list of `Node` objects. Each `Node` contains the raw text of the token in a `surface`

[1] https://github.com/polm/fugashi

property, and extended dictionary fields are available in the `feature` property.

```
import fugashi

tagger = fugashi.Tagger()

text = "形態素解析をやってみた"
words = tagger(text)
print(words)
print("=====")

for word in words:
    print(word.surface, word.feature.lemma, word.feature.kana
        , sep="\t")
```

```
[形態, 素, 解析, を, やっ, て, み, た]
=====
形態    形態    ケイタイ
素      素      ソ
解析    解析    カイセキ
を      を      ヲ
やっ    遣る    ヤッ
て      て      テ
み      見る    ミ
た      た      タ
```

Note: In Japanese NLP, it's standard to refer to the raw input text form as the "surface" (表層 hyousou), and MeCab uses this in its API. This usage comes from linguistics, where the **surface form** of a word in a particular context (which may be inflected or have unusual orthography) is contrasted with the **lexical form**, which would be a normalized or dictionary form.

For basic tokenization, this is all you need to know. In the next section, we'll look at a slightly more involved application of morphological analysis, and later in this chapter we'll cover advanced tokenization-related topics.

Morphological Analysis Mini Project: Automatic Fuseji

Fuseji (伏せ字) is the practice of replacing some characters with placeholders, usually a circle, to conceal the content of words. A similar thing is some-

times done in English, particularly to avoid using obscene words (a**hole, "you little @#%(*!"). In Japanese fuseji can be used for obscene words, but they can also be used to avoid spoilers, be vague about the names of brands or specific people, or for other reasons.

Let's pretend that we want to automatically apply fuseji for the purpose of hiding spoilers about new movies or other media. While the simplest thing is to replace characters at random from the whole string, it's better to replace certain kinds of words, such as proper nouns. We can use the detailed part of speech information in UniDic, along with word boundaries, to replace proper nouns with fuseji versions.

```
from fugashi import Tagger
from random import sample

tagger = Tagger()


def fuseji_node(text, ratio=1.0):
    """This function will take a node from tokenization and
       actually replace parts of it with filler characters.
       """
    ll = len(text)
    idxs = sample(range(ll), max(1, int(ratio * ll)))
    out = []
    for ii, cc in enumerate(text):
        out.append("○" if ii in idxs else cc)
    return "".join(out)


def fuseji_text(text, ratio=1.0):
    """Given an input string, apply fuseji. """
    out = []
    for node in tagger(text):
        # Normal Japanese text doesn't use white space, but
          this is necessary
        # if you include latin text, for example.
        out.append(node.white_space)
        if node.feature.pos2 != "固有名詞":
            out.append(node.surface)
        else:
            out.append(fuseji_node(node.surface))
    return "".join(out)


```

```
print(fuseji_text("犯人はヤス"))
print(fuseji_text("東京タワーの高さは333m"))
```

```
犯人は○○
○○タワーの高さは333m
```

This code is already reasonably effective, but there are several ways it could be tweaked or improved. For example, sometimes the words that should be concealed aren't just proper nouns; they could also be ordinary nouns or verbs.

How can we find what parts of speech we want to filter? The best way is to use **example sentences** to find what parts of speech we want, as well as to get a better understanding of where our program works well and where it doesn't.

- 新キャラの「カズヤ」は年内に配信予定
- マジルテの水晶の畑エリアにはクリスタルが沢山ある
- 「吾輩は猫である」の作家は夏目漱石
- 『さかしま』（仏: À rebours）は、フランスの作家ジョリス＝カルル・ユイスマンスによる小説

We can check the parts of speech of words in fugashi by using the `node.pos` attribute. This part of speech information comes from UniDic and uses four levels. You can access the individual levels as `node.feature.pos1`, `node.feature.pos2`, and so on. The `node.pos` attribute is a convenience feature that joins the four separate values together and replaces empty values with an asterisk (`*`).

You can check part of speech tags of words by giving a sentence as input with fugashi on the command line, without giving the `-O wakati` command line argument.

```
!echo "毎年東麻布ではかかし祭りが開催されます" | fugashi
```

```
毎年	マイトシ	マイトシ	毎年	名詞-普通名詞-副詞可能
0
東	ヒガシ	ヒガシ	東	名詞-普通名詞-一般	0,3
麻布	アザブ	アザブ	アザブ	名詞-固有名詞-地名-一般
0
```

```
で　デ　デ　で　助詞-格助詞
は　ワ　ハ　は　助詞-係助詞
かかし カカシ カカス 欠かす 動詞-一般　五段-サ行　連用形-一般 0,2
祭り　マツリ マツリ 祭り　名詞-普通名詞-一般　0
が　ガ　ガ　が　助詞-格助詞
開催　カイサイ　カイサイ　開催　名詞-普通名詞-サ変可能　0
さ　サ　スル　為る　動詞-非自立可能　サ行変格　未然形-サ　0
れ　レ　レル　れる　助動詞 助動詞-レル　連用形-一般
ます　マス　マス　ます　助動詞 助動詞-マス　終止形-一般
EOS
```

Censoring Unknown Words

Another thing that'll come up as we're testing is that sometimes words not in the dictionary will be used, like the names of characters in movies and books. From the example sentences above, マジルテ Majirute, the name of a fictional place, is an example of such a word. We basically always want to censor those words to avoid spoilers, so rather than checking part of speech information, we can also check specifically for words that aren't in our dictionary. These are called "unks", from "unknown words", or 未知語 michigo in Japanese. In fugashi you can determine if a given node is in the dictionary just by checking the `node.is_unk` attribute.

Looking at our example sentences, some patterns emerge. We probably don't want to filter verbs, since it's hard to tell when a verb is important. Proper nouns should definitely be filtered. Common nouns may or may not be important, so it's hard to say if we should filter them - for now, let's leave them alone.

Since our conditions for censoring words are getting kind of complicated, let's factor them into a function.

```
def should_hide(node):
    """Check if this node should be hidden or not. """
    if node.is_unk:
        return True
    ff = node.feature
```

```
    if ff.pos1 == "名詞" and ff.pos2 == "固有名詞":
        return True
    return False


def fuseji_text(text, ratio=1.0):
    """Given an input string, apply fuseji. """
    out = []
    for node in tagger(text):
        out.append(node.white_space)
        word = fuseji_node(node.surface) if should_hide(node)
              else node.surface
        out.append(word)
    return "".join(out)


texts = [
    "犯人はヤス",
    "魔法の言葉はヒラケゴマ",
    "『さかしま』（仏：À rebours）は、フランスの作家ジョリス＝カルル・ユイスマンスによる小説",
    "鈴木爆発で最初に解体する爆弾はみかんの形をしている",
]

for text in texts:
    print(fuseji_text(text))
```

```
犯人は○○
魔法の言葉は○○○○○
『さかしま』（仏：○○○○○○○○）は、○○○○の作家○○○○=○○○・○○○○○○による小説
○○爆発で最初に解体する爆弾はみかんの形をしている
```

Use Readings to Censor only Part of Words

At this point our program is pretty effective at applying fuseji to any text we throw at it. That said, censoring the entire text is a little boring. It would be more interesting if we could reveal some letters so that readers can guess the rest of the word, but not quite be certain about it.

There is one potential issue though - if we use kanji, even one character might give the word away in away that's not interesting. What if we could convert words to phonetic versions and then censor part of them?

That would allow us to show part of the word while giving away less information.

Thankfully, UniDic includes a field we can use for this conversion. Every word in the UniDic dictionary has a `kana` field we can use to get the conventional reading for the word in katakana form. (UniDic also has a `pron` field, which uses non-standard orthography to differentiate long vowels.)

One thing to keep in mind is that the kana reading will only be available for words in UniDic, and it won't always be perfect. There are two cases where the reading will be wrong:

1. The word is not in the dictionary.
2. The reading of the word is ambiguous.

If the word is not in the dictionary, it's possible to train a machine learning model or use other methods to predict the reading, but that's pretty difficult. So this time, if a word is an unk we'll just skip converting it and use the raw surface form.

Ambiguous words are more difficult. Some examples of ambiguous words:

- 東: higashi or azuma (or tou)
- 中田: nakada or nakata
- 仮名: kana or kamei
- 網代: amishiro or ajiro
- 最中: saichuu or monaka
- 私: watashi or watakushi
- 日本: nihon or nippon

Usually a reading will be clear from context, but many ambiguous words are proper nouns like the names of people and places, and without knowing which specific entity it's referring to there's no way to be sure of the correct reading. Even worse, there's no way to be sure if the word you're looking at is ambiguous or not just using the tokenizer output.

(Note: Words written the same way but pronounced differently are referred to as 同形異音語 doukei iongo or "heteronyms". They are also common in English, though less so for proper nouns.)

Note: For ambiguous words, deciding their reading could be considered a form of **word sense disambiguation** for common nouns, or **entity linking** for proper nouns. Both are NLP problems with a long history.

So how can we handle ambiguous words if we can't even identify them with certainty? It turns out that their difficulty actually has a silver lining - because even people make mistakes, we can get away with just using the kana UniDic gives us and hope that it's right most of the time. For serious applications replacing the original text with a mistake would be unacceptable, but for our fuseji application, it's not the end of the world if we're wrong occasionally.

Sometimes when you learn about a problem confronting your NLP system, there may not be a solution you're able to implement. In this case, writing a program to disambiguate words would be much more work than the rest of our entire program. But by being aware of the problem, we can consider how failures affect the output of our system, and evaluate whether we should continue with its development, or start over with a design that can work around the problem.

Now that we've settled that, let's change our code to use the kana instead of the surface when censoring words.

```
def fuseji_text(text, ratio=1.0):
    """Given an input string, apply fuseji. """
    out = []
    for node in tagger(text):
        out.append(node.white_space)
        node_text = node.surface if node.is_unk else node.
            feature.kana
        word = fuseji_node(node_text, ratio=0.5) if
            should_hide(node) else node.surface
        out.append(word)
    return "".join(out)


texts = [
    "黒幕の正体はガーランド",
]
```

```
15
16  for text in texts:
17      print(fuseji_text(text))
```

```
1  黒幕の正体は◯ー ラ◯ド
```

And that makes our automatic fuseji program complete. It's not a lot of code, but in building this you learned how to:

1. iterate over the tokens in a text
2. identify parts of speech of interest with example sentences
3. use multiple levels of part of speech tags
4. check if a token is in the dictionary or an unk
5. convert words to their phonetic representation

These are all basic building blocks you can use to build a wide variety of applications.

While our motivation for this program was a simple and playful one, the techniques used here are simple versions of those used in **personally identifying information (PII) removal**, which removes identifying details from documents like medical and legal records so they can be used in audits or analysis without risk to the people they describe.

To learn more about the tokenizer API, consider some ways you might want to extend this application and how you'd make the necessary changes.

- what if you wanted to remove all numbers from a contract, to hide dates or prices?
- what if you wanted to hide a specific list of words, perhaps obscenities, rather than certain parts of speech?
- how would you change the program to replace hard-to-read words with their phonetic versions?

Natural Language Generation and Conversion with Transformer

5.1 Transformer and Text Generation

What is the Transformer?

The Transformer is a neural network architecture published by Devlin et al. in 2017[1]. The core of the architecture is a mechanism called self-attention, which transforms a sequence of input tokens into output representations by focusing on important tokens and taking the weighted sum of the input representations. The technical details of the Transformer are out of the scope of this book—interested readers are referred the wonderful blog post, The Illustrated Transformer[2] by Jay Alammar. For now, you can think of it as a powerful, high-capacity neural network architecture that transforms a set of inputs (token embeddings) into a sequence of representations of the same length, which you can use to solve various NLP tasks such as translation, classification, and language modeling.

The Transformer architecture has taken the NLP field by storm and quickly became the ”de-facto” model of choice for a wide range of tasks. In the remainder of this chapter we'll make heavy use of the Transformer to achieve language generation and conversion.

[1] https://arxiv.org/abs/1706.03762
[2] https://jalammar.github.io/illustrated-transformer/

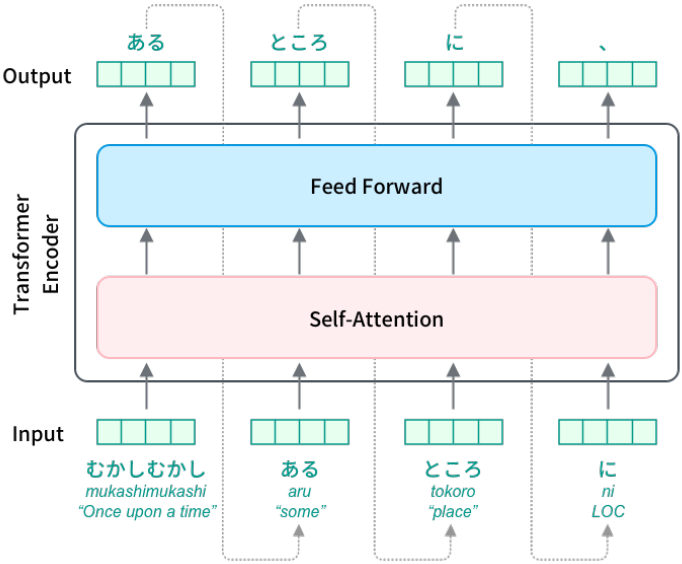

Figure 1: The Transformer architecture—it predicts next tokens in an autoregressive manner.

In this section, we'll use language models to generate Japanese text and solve a QA (question answering) task. A language model is a statistical model that assigns some information (probability) to a given text. Because language models are usually trained on large datasets of naturally occurring text, they can give high probability to sentences that are natural in that language, or ”make more sense”, and give low probability to unnatural sentences.

One type of widely used language model is the autoregressive or causal language model (CLM). CLMs model the probability of a particular input by first decomposing it into a sequence of individual tokens, and then by taking the

product of the individual token probability given the preceeding context.

The figure above shows an illustration of a Transformer-based CLM. When given a context (むかしむかし mukashimukashi "Once upon a time"), it gives you a probability distribution over the set of tokens that could appear next, which may include tokens such as "、" (Japanese comma) and ある aru "some". The model is trained such that it gives high probabilities to sequences of tokens that appear in the training data.

In NLP, language models are traditionally implemented as a statistical model of word n-grams or based on RNNs (recurrent neural networks) such as LSTM (long short-term memory). However, as of this writing (in 2021), most of state-of-the-art language models are implemented based on the Transformer architecture.

Text generation

As discussed in the last section, CLMs are trained to give a probability distribution over the tokens given a context. This also means that you can use a language model to generate new text by producing tokens one at a time.

Because training language models usually require a large amount of training data and compute (dozens or even hundreds of GPUs), for many tasks we just download and use pretrained language models. Popular English language models include GPT-2[3] and GPT-3[4], both developed by OpenAI.

In the remainder of this section, we'll make heavy use of the popular HuggingFace Transformers[5] library, which supports a wide range of Transformer-based pretrained language models including BERT[6] (Chapter 6) and GPT-2.

We'll start by installing and importing libraries and modules necessary for language generation and QA, including Transformers, HuggingFace

[3] https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
[4] https://arxiv.org/abs/2005.14165
[5] https://github.com/huggingface/transformers
[6] https://arxiv.org/abs/1810.04805

datasets[7], as well as SentencePiece[8] (for tokenizing Japanese text) and PyTorch.

```
%%capture
!apt-get install jq
!pip install datasets==1.11.0
!pip install transformers==4.9.0
!pip install sentencepiece==0.1.96
!pip install torch==1.9.0
```

```
from collections import Counter
import json
import logging

from datasets import load_dataset
import torch
from transformers import T5Tokenizer, AutoModelForCausalLM,
    Trainer, TrainingArguments

# supress logging from transformers
logging.getLogger("transformers").setLevel(logging.ERROR)
logging.getLogger("transformers.trainer").setLevel(logging.
    ERROR)
logging.getLogger("datasets").setLevel(logging.ERROR)

_ = torch.manual_seed(42)
```

For our Japanese autoregressive language model, we'll use Rinna[9], an open-source Japanese GPT-2 model developed by rinna Co., Ltd.

The model can be referred to by an identifier `rinna/japanese-gpt2-medium` on HuggingFace Hub (https://huggingface.co/rinna/japanese-gpt2-medium). To use it, you just need to load the pretrained model via the `from_pretrained()` method as below. Remember to initialize a corresponding tokenizer—you need to it to process the input for the model.

```
device = torch.device("cuda" if torch.cuda.is_available()
    else "cpu")

# Load Rinna — a Japanese GPT-2 model
```

[7] https://github.com/huggingface/datasets
[8] https://github.com/google/sentencepiece
[9] https://github.com/rinnakk/japanese-pretrained-models

```
4
5  tokenizer = T5Tokenizer.from_pretrained("rinna/japanese-gpt2-
       medium")
6  tokenizer.do_lower_case = True  # due to some bug of
       tokenizer config loading
7
8  model = AutoModelForCausalLM.from_pretrained("rinna/japanese-
       gpt2-medium").to(device)
```

First, let's try generating Japanese text with the pretrained language model. You can either generate text from scratch, or tell the model to generate a continuation to another piece of text (called a prompt). For now we'll generate text that follows the prompt "むかしむかし、あるところに" mukashimukashi, arutokoroni ("Once upon a time, there was/were").

After tokenizing the input (prompt) with the tokenizer, you can invoke the `model.generate()` method to generate the continuation. The method takes a number of parameters that control the generation process, but the details are not important here. You can decode the results and convert them back into text with `tokenizer.decode()` as below

```
1  inputs = tokenizer("むかしむかし、あるところに、",
       return_tensors="pt", add_special_tokens=False).to(device
       )
```

```
1  result = model.generate(
2      **inputs,
3      do_sample=True,
4      top_p=0.9,
5      temperature=0.8,
6      max_length=100,
7      pad_token_id=2,
8      repetition_penalty=1.2
9  )
```

```
1  tokenizer.decode(result[0])
```

```
1  'むかしむかし、あるところに、ふしぎなキツネの村がありました。
       ある時、キツネがひとりきりで野山をさまよい歩いている
       と、その先には白い小さな森が見えてきました。 「あなたの
       お母さんも、この白い森から生まれてきたんだよ」と教えられ
       た男の子。 「そんな風に育てられたんだねぇ... すごく幸せ
       だよ」と声をかけると、キツネはその言葉を返してくれまし
```

```
        た。</s>'
```

You can see that Rinna was able to make up a plausible sounding story that starts with むかしむかし、あるところに、ふしぎなキツネの村がありました。 ”Once upon a time, there was a mysterious village of foxes.”

5.2 Question answering

Generating Japanese text with a language model is interesting and fun, but you can use language models to solve a much wider range of problems.

One such problem is question answering. Here, we’ll use a language model to answer open-domain trivia questions such as ”Which city is called ’the navel of Hokkaido’ due to its location and is also famous for its lavender fields?” (Can you answer this?)

Here we assume that for each question (e.g., Where’s the prefectual capital of Aichi?) there’s a list of answer candidates (e.g., Sapporo, Sendai, Tokyo, Nagoya, Kyoto, etc.) among which the model needs to choose the correct answer. This means that the format is a multiple-choice question where the model needs to rank the candidates in the order of confidence instead of generating the answer from scratch.

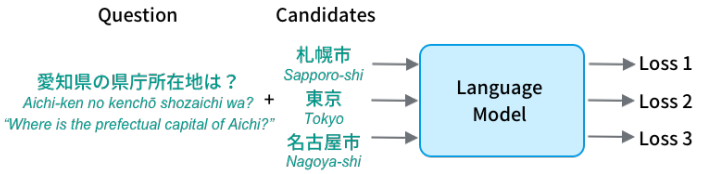

Figure 2: Answering a question using a language model

How can you solve such questions with language model (LMs)? Remember that LMs trained on a large corpus give higher probabilities (or equivalently,

lower losses) to more ”natural” input. As shown in the figure, you can rerank the list of candidates by feeding a sequence like `[question_text] [answer_text]` to the model for a given candidate and by measuring its loss, which roughly corresponds to how ”unexpected” the language model thinks that particular input is. If the continuation of the question and the candidate ”makes sense,” the model will return a smaller loss. You can do this for every choice and pick the choice that has the lowest loss value. The following method `rank_answers` reranks the list of candidates based on the question and the model (the details of the method is not important here).

```
def rank_answers(question, candidates, model, tokenizer, top_n=10):
    """Given a question and a list of answer candidates, rank them based on the language model score
    (negative log likelihood) and return the ranked top N candidates."""
    losses = Counter()
    inputs_question = tokenizer(
        question, return_tensors="pt", add_special_tokens=False
    ).to(model.device)
    labels_question = -100 * torch.ones_like(
        inputs_question["input_ids"], device=model.device
    )
    results = model(**inputs_question, use_cache=True)
    past = results.past_key_values
    for candidate in candidates:
        inputs_candidate = tokenizer(
            candidate, return_tensors="pt", add_special_tokens=True
        ).to(model.device)
        attention_mask = torch.cat(
            (inputs_question["attention_mask"], inputs_candidate["attention_mask"]),
            dim=1,
        )
        results = model(
            input_ids=inputs_candidate["input_ids"],
            attention_mask=attention_mask,
            labels=inputs_candidate["input_ids"],
            past_key_values=past,
        )
        loss = results.loss.detach().item()
        losses[candidate] = -loss
```

```
29        return [a for a, v in losses.most_common(top_n)]
```

Let's try a simple question: Where's the prefectual capital of Aichi? We'll use a list of some prefectual capital cities in Japan as candidates.

```
question = "愛知県の県庁所在地は？"
```

```
answers = ["札幌市", "秋田市", "宇都宮市", "東京", "金沢市",
    "岐阜市", "名古屋市", "大津市", "奈良市", "岡山市", "高
    松市", "佐賀市", "宮崎市"]
```

```
rank_answers(question, answers, model, tokenizer)
```

```
['岐阜市', '名古屋市', '金沢市', '奈良市', '宇都宮市', '秋田
    市', '岡山市', '高松市', '宮崎市', '佐賀市']
```

As you see, the language model ranks 岐阜市 Gifu-shi "Gifu city" as the top candidate, and 名古屋 Nagoya comes second. This suggests that the model encodes some common sense information, but it's not perfect, at least when it comes to Japanese prefectural capitals. But how good is Rinna for answering common sense questions, really? Let's try to evaluate its accuracy more thoroughly below.

Evaluate on the JAQKET dataset

Here we are going to use the JAQKET dataset[10], which is an open-domain question answering dataset developed and distribted by Tohoku University. The dataset includes common sense questions and their answers, where answers and candidates are always drawn from Wikipedia article titles, such as:

- Question: Which city is called "the navel of Hokkaido" due to its location, and is also famous for its lavender fields?
- Answer: Furano
- Candidates: Furano, Nayoro, Mikasa, Makubetsu, Kitami, ...

[10] https://www.nlp.ecei.tohoku.ac.jp/projects/jaqket/

First, let's download, format, and read the datasets (both the train and dev1 portions) so that we can evaluate the language model's quiz answering performance on them.

```
!curl "https://jaqket.s3-ap-northeast-1.amazonaws.com/data/train_questions.json" -s --output train_questions.json
```

```
# delete the "original_answer" keys, which cause some discrepancies between train and dev splits
!jq 'del(.original_answer)' -c train_questions.json > train_questions.nooa.json
```

```
!curl "https://jaqket.s3-ap-northeast-1.amazonaws.com/data/dev1_questions.json" -s --output dev1_questions.json
```

The `evaluate()` method takes a list of questions and answers, evaluates the questions with the model and the tokenizer, and return the number of answers the model got correct.

```
qas = []
with open("dev1_questions.json") as f:
    for line in f:
        qas.append(json.loads(line))
```

```
def evaluate(qas, model, tokenizer, stop_at=None, show_preview=False):
    num_correct = 0
    num_questions = 0
    for qa in qas:
        question = qa["question"]
        candidates = qa["answer_candidates"]
        gold = qa["answer_entity"]
        preds = rank_answers(question, candidates, model, tokenizer)
        is_correct = preds[0] == gold
        if is_correct:
            num_correct += 1
        if show_preview and num_questions < 5:
            print(
                f"Q: {question}, pred: {preds[:5]}, gold: {gold}, is_correct: {is_correct}"
            )
        num_questions += 1

        if stop_at is not None and num_questions == stop_at:
```

```
            break
    print(
        f"Success rate = {100 * num_correct / num_questions}%
            ({num_correct} / {num_questions})"
    )
```

```
evaluate(qas, model, tokenizer, stop_at=100, show_preview=
    True)
```

```
Q: 明治時代に西洋から伝わった「テーブル・ターニング」に起源を
    持つ占いの一種で、50音表などを記入した紙を置き、参加者全
    員の人差し指をコインに置いて行うのは何でしょう?, pred: [
    'テケテケ', '赤い紙、青い紙', 'コックリさん', '小玉鼠（
    妖怪)', 'ヨジババ'], gold: コックリさん, is_correct:
    False
Q: 『non・no』『週刊プレイボーイ』『週刊少年ジャンプ』といえ
    ば、発行している出版社はどこでしょう?, pred: ['実業之日
    本社', '白泉社', '宝島社', '日本文芸社', '幻冬舎'], gold
    : 集英社, is_correct: False
Q: 「パイプスライダー」や「そり立つ壁」などの関門がある、TBS
    系列で不定期に放送されている視聴者参加型のTV番組は何でし
    ょう?, pred: ['最強の男は誰だ!壮絶筋肉バトル!!スポーツマ
    ンNo.1決定戦', '究極の男は誰だ!?最強スポーツ男子頂上決戦
    ', 'SASUKE', '島田紳助がオールスターの皆様に芸能界の厳し
    さ教えますスペシャル!', 'クイズ王最強決定戦～THE OPEN～'
    ], gold: SASUKE, is_correct: False
Q: 東京都内では最も古い歴史を持つ寺院でもある、入口にある「雷
    門」で有名な観光名所は何でしょう?, pred: ['天龍寺（新宿
    区)', '大龍寺（東京都北区)', '大円寺（目黒区)', '源覚寺
    (文京区)', '浅草寺'], gold: 浅草寺, is_correct: False
Q: 「鍋についたおこげ」という意味の言葉が語源であるとされる、
    日本ではマカロニを使ったものが一般的な西洋料理は何でしょ
    う?, pred: ['オムレツ', 'ポテトサラダ', 'ロールキャベツ'
    , 'フレンチトースト', 'スパゲッティ'], gold: グラタン,
    is_correct: False
Success rate = 13.0% (13 / 100)
```

As you can see from the result above, the untuned Rinna model achieves an accuracy of 13% for the first 100 questions in the development set. Each question has 20 candidates, so this accuracy is higher than random chance (which is 1/20 = 5%), although not very impressive. Can we do better?

How to Fine-tune the Language Model for Solving QA

One way to improve Rinna (and any pretrained models for that matter) is to show it a number of examples and optimize its parameters so that it can give higher probability for the correct question-answer pairs. This process is called fine-tuning and is the most common way to adapt a pretrained model to another task.

Below, we'll first load the JAQKET dataset in the JSONL format with the HuggingFace dataset library. Note that we are loading both the training and the dev splits of the dataset for training.

```
dataset = load_dataset('json',
    data_files={'train': 'train_questions.nooa.json',
                'valid': 'dev1_questions.json'})
```

```
Downloading and preparing dataset json/default (download:
    Unknown size, generated: Unknown size, post-processed:
    Unknown size, total: Unknown size) to /home/mhagiwara/.
    cache/huggingface/datasets/json/default-2f5c57ceca4651d1
    /0.0.0/45636811569
    ec4a6630521c18235dfbbab83b7ab572e3393c5ba68ccabe98264...
```

```
{"version_major":2,"version_minor":0,"model_id":""}
```

```
{"version_major":2,"version_minor":0,"model_id":""}
```

```
Dataset json downloaded and prepared to /home/mhagiwara/.
    cache/huggingface/datasets/json/default-2f5c57ceca4651d1
    /0.0.0/45636811569
    ec4a6630521c18235dfbbab83b7ab572e3393c5ba68ccabe98264.
    Subsequent calls will reuse this data.
```

You can visually inspect the instances in the dataset as follows:

```
dataset['train'][0]
```

```
{'qid': 'ABC01-01-0003',
 'question': '格闘家ボブ・サップの出身国はどこでしょう?',
 'answer_entity': 'アメリカ合衆国',
 'answer_candidates': ['アメリカ合衆国',
  'ミネソタ州',
  'オンタリオ州',
```

```
    'ペンシルベニア州',
    'オレゴン州',
    'ニューヨーク州',
    'コロラド州',
    'オーストラリア',
    'ニュージャージー州',
    'マサチューセッツ州',
    'カナダ',
    'テキサス州',
    'ミシガン州',
    'ワシントン州',
    'ニュージーランド',
    'オハイオ州',
    'カリフォルニア州',
    'メリーランド州',
    'イリノイ州',
    'イギリス'],
  'original_question': '格闘家ボブ・サップの出身国はどこでしょう？'}
```

We will fine-tune the model by presenting the `[question_text] [answer_text]` pairs to the language model, so we use the `.map()` method of the dataset to add a new field to each instance with a concatenation of the question and the answer.

```
dataset = dataset.map(
    lambda example: {"text": example["question"] + example["
        answer_entity"]}
)
```

```
{"version_major":2,"version_minor":0,"model_id":"389
    f6bc2b792482787b1bde5bf14737f"}
```

```
{"version_major":2,"version_minor":0,"model_id":"5025
    e0fc274f414abdb2a89581514b31"}
```

Now let's tokenize the concatenated text field. You can use the `.map()` method again to batch process the dataset. Remember to return the **`label`** field as well (which is basically a copy of the `input_ids` field)—the language model is trained by scoring it based on its ability to reproduce the **`label`** (in this case the input sequence) token by token.

```
max_length = 256

```

```python
def tokenize_function(examples):
    inputs = tokenizer(
        examples["text"],
        max_length=max_length,
        padding="max_length",
        truncation=True,
        return_tensors="np",
    )
    labels = inputs.input_ids.copy()
    labels[labels == tokenizer.pad_token_id] = -100
    return {
        "input_ids": inputs["input_ids"],
        "attention_mask": inputs["attention_mask"],
        "labels": labels,
    }

tokenized_dataset = dataset.map(
    tokenize_function, batched=True, batch_size=8, num_proc=4
)
```

The processed input looks like this. Notice that each input is padded with padding tokens (id: 3). This is because the neural network requires every instance in a single batch to have the same length.

```python
print(tokenized_dataset['train'][0]['input_ids'])
```

```
[9, 8355, 149, 9255, 13, 209, 2872, 10, 550, 115, 11, 5964,
     16744, 3017, 886, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
     3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
     3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
     3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
     3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
     3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
     3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3]
```

Now we are ready to fine-tune the model. You need to specify some hyperparameters with `TrainingArguments`, create a `Trainer` instance, and invoke the `.train()` method. Note that this takes a while (about an hour) even on

a fast GPU, and will probably be impossible to train on a CPU.

```
training_args = TrainingArguments(
    output_dir = 'rinna-japanese-gpt2-medium-finetuned',
    num_train_epochs = 3,
    evaluation_strategy = "steps",
    learning_rate=5e-5,
    warmup_steps=1000,
    per_device_train_batch_size = 6,
    eval_steps = 200,
    logging_steps = 200,
    save_strategy = "no",
)
```

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset['train'],
    eval_dataset=tokenized_dataset['valid'],
)
trainer.train()
```

Now let's evaluate the model again. This time we get an accuracy of 51%. Since there's no question overlap between the training and the development datasets, this means that the model didn't get better simply by memorizing the questions it was presented with, but it appears that it rewired the parameters in such a way that it now got better at answering a wide range of common sense questions about Japan.

```
evaluate(qas, model, tokenizer, stop_at=100, show_preview=
    True)
```

Next Steps

You can solve a much wider range of NLP tasks with language models, and it's fun to think how you'd make them solve certain tasks by designing prompts or even fine-tuning if necessary. How would do go about solving the following tasks, for example?

- Translation. Can Rinna translate between, say, Japanese and English?
- Arithmetic. Can Rinna answer simple math questions such as 6+7=?

- Word analogy. Can Rinna answer analogy questions such as Japan is to Yen as USA is to...?

If you need some inspration, the GPT-3 paper[11] has many examples.

[11] https://arxiv.org/abs/2005.14165