入門 日本語自然言語処理 無料サンプル版

この PDF には、無料サンプルとして節が二つ含まれています。この 2 節は基本的に完成していますが、本書の執筆が続くにつれ、内容が更新される場合があることをご了承ください。

本書についての詳細や予約情報はホームページ japanesenlp.com[1] にてご確認ください。

[1] https://www.japanesenlp.com/index-ja.html

2.1 fugashi の紹介

この章では、日本語の形態素解析器 MeCab のラッパーである fugashi と、辞書の unidic-lite を使い、日本語の形態素解析の基礎について学びます。

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

以上の表は fugashi と UniDic の出力の例です。単語分割のみならず、各単語について様々な情報も含まれています。以上で表示されている情報もまた UniDic の情報の一部でしかありません。

事前準備

まず fugashi とその辞書をインストールする必要があります。

fugashi は昔からある、人気の形態素解析器 **MeCab** のラッパー [^wrapper_ja] です。Cython を用いて MeCab の C API を Python から使えるようにした上に、Python から便利に使えるように細かい変更が加えられています。

unidic-lite は、UniDic 2.1.2 をベースに変更を加えた形態素解析用の辞書です。UniDic のバージョンとしてはやや古いですが、情報の質に問題はなく、後のバージョンと比べてデータ量も少なく、扱いやすいという特徴があります。PyPI の **unidic** パッケージを使えば最新版の UniDic も利用できますが、辞書の見出し語数が大量に増えたせいで環境構築が少しばかり複雑なので、本チュートリアルでは使用しません。

本書執筆時での fugashi の最新版は 1.1.0、unidic-lite は 1.0.8 です。unidic-lite は純粋にデータのみから構成されており、どの環境でも利用できます。fugashi は OSX・Linux・Win64 の「wheel」を提供していますので、そのいずれの環境であれば他の事前準備は不要です。(他の環境ではソースからビルドする必要があります。もしビルド中に何か問題がありましたら、お気軽に issue を立ててください[1]。)

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

これで fugashi がインストールされましたので、正しくインストールされているか確認するために一度シェルから実行してみましょう。まずは、`fugashi -O wakati` を実行し適当な文章を入力してみましょう。文章を入力した後に改行を入力すると、入力文がスペース区切りで返ってきます。出力の確認が終わったら `CTRL+D` でプログラムを終了できます。出力は以下のようになります。

```
!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
```

ここで使ったオプション `wakati` は「分かち書き」のことを指しています。「分かち書き」とは、子供向けの本や、低解像度の画面ゆえにひらがなのみで書かれた文章などで見られる、スペース区切りで書かれた日本語の文章のことです。本来、分かち書きは文節をスペースで区切るのが普通ですが、MeCab での分かち書きは単語 (形態素) 単位で文章を区切ります。

それではコードから fugashi を使ってみましょう。コードでは主に辞書の情報を管理する `Tagger` オブジェクトを使います。`Tagger` を入力分に適応すると

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

Node のリストが返ってきます。Node のテキストは surface 属性に格納され、辞書のさまざまな情報は feature 属性に格納されています。

```
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
```

日本語の言語処理では、原文の表記そのままのことを指して「表層」（英：surface）と呼ぶことが多いです。これはもともと言語学の用語で、語形変化や表記の違いなども含めた、原文のままの「表層形式」の他に、辞書の見出しなどで使われる、標準化した「語彙形式」（英語：lexical form）があります。

基本的な単語分割の処理の紹介は以上です。次の節では、もう少し高度な形態素解析の応用を紹介し、その後、更に高度な使い方などを説明します。

形態素解析を使ってみよう　自動伏せ字プログラムの実装

伏せ字とは、文章の一部の文字を、他の文字に置き換え、単語の一部を隠すことを指します。個人名や商品名の明言を避けるため、ネタバレを避けるため、検索避けのため、など、様々な目的に使われます。

今回は、映画などの新作作品のネタバレを避けるために、自動的に伏せ字を適

用するプログラムを書きたいとしましょう。単に入力文章の一部の文字を伏せたいだけであれば、文字をランダムに置換することもできますが、品詞などに基づいてどこを伏せるかを決めると、結果がより自然になります。UniDic の細かい品詞情報と単語分割の結果を活用し、まずは固有名詞を伏せてみましょう。

```
from fugashi import Tagger
from random import sample

tagger = Tagger()


def fuseji_node(text, ratio=1.0):
    """分かち書きの結果ノードを受け取り、文字列の一部をランダムに◯で置き換える """
    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):
    """入力文を受け取り、適切なところを伏せ字に置き換える """
    out = []
    for node in tagger(text):
        # 純粋な日本語のテキストはスペースを含まないが、英語のテキストと混ぜる場合などに必要になる。
        # スペースはそれ自体でノードにならないので、ここで明示的に追加する必要がある。
        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
```

このプログラムはこのままでもそれなりに使えますが、改善の余地はまだ残っ

ています。場合によっては固有名詞だけでなく、一般名詞や動詞を伏せたいときもあります。

伏せたい品詞はどうやって特定すれば良いでしょうか。品詞表から調べられることも可能ですが、例文をいくつか実際に解析してみて、その結果を見る方が楽で効果的です。実際データを処理してみると、どんなケースで伏せ字プログラムがうまくいくかどうかがより深く理解できます。

- 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

fugashi の解析結果の品詞は、属性 `node.pos`[2] を見ることによって確認できます。この品詞体系は UniDic のものに基づいており、４つのレベルから構成されます。各レベルは、`node.feature.pos1`, `node.feature.pos2`, ... を使って参照できます。`node.pos`は４つ全てのレベルの情報を一つの文字列にまとめた属性です。全てのレベルに情報があるとは限らず、情報がない場合、そのレベルは `*`になります。

例文の品詞タグはシェルから確認できます。上で試したのとは異なり、`-O wakati` を指定しない場合、各行ごとに、品詞情報などを含む単語の情報が表示されます。

```
!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
```

[2] `pos`は「part of speech (品詞)」の略です。

```
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
```

未知語を伏せ字に変換する

様々な文を使って形態素解析を試してみると、映画や書籍のキャラクターの名前など、辞書に含まれていない単語に出くわします。上の例文中の「マジルテ」(架空の場所の名前) もその一つです。ネタバレを避けるために、これらの単語も積極的に伏せたいので、この場合品詞ではなく「辞書にないこと」に基づいて伏せ字の対象にします。これら「辞書にない単語」は「未知語」、英語では「unk」(unknown の略) と呼ばれます。fugashi では`node.is_unk` の属性を見ることで、その単語が未知語かどうかが確認できます。

これらの例文を確認していくと、該当する品詞が分かってきます。動詞については、重要なものとそうでないものを品詞だけでは区別することができないので、除外しない方が良いでしょう。逆に、固有名詞は全部伏せ字対象にした方が良さそうです。一般名詞は動詞と同様、伏せ字にしたいものとそうでないものがあるので、とりあえず対象外にしましょう。

伏せ字対象の条件がやや複雑化しているので、ここで一旦関数にまとめましょう。

```
def should_hide(node):
    """与えられたノードを伏せ字の対象にするかどうかを返す """
    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):
    """与えられた文字列に対して、伏せ字を適用する """
    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
```

読みを使って単語の一部を伏せ字にする

この時点で、このプログラムはどんな文章でもそれなりに伏せ字を適用することができます。ただし、伏せ字対象となった単語の全ての文字をただ単に伏せるのは少し味気が無いですね。一部の文字だけを伏せ、読み手が単語の残りを少しだけ推測できるようにすればさらに面白いと思われます。

しかし、一部の文字だけを伏せると問題が発生します。漢字の単語だと、文字一つだけで元の単語が分かってしまうこともあり、これも面白くありません。これを回避するために、まず単語を読み仮名に変換し、その読みの一部を伏せればどうでしょう。そうすれば、一部の文字を伏せなくても、そう簡単に原文がバレることはないはずです。

UniDic には読み情報もちゃんと収録されているので、この変換に使うことができます。UniDic では、全ての項目に対して `kana` フィールドがあり、表層をこれに置き換えることによって読み仮名に変換できます。(これとは別に `pron`

フィールドもありますが、これは一般的な読み仮名ではなく、棒引き仮名遣いを使った表記のものです。）

注意点として、読み情報は UniDic に収録されいる単語にしか使えないことと、収録されている単語であっても、読み方が文脈に照らし合わせていつも正しいとは限らないことがあります。読み仮名が正しくない場合は大きく分けて 2 つあります。

3. 未知語の場合
4. 読み方が曖昧な単語

未知語の場合、機械学習などを使って読み仮名を判定するプログラムを書くことは可能ですが、簡単ではありません。したがって今回は、未知語はあきらめて表層をそのまま使います。

同形異音語（形は同じでも読み方が曖昧な単語）は未知語よりも対応が難しいです。同形異音語の例には以下のようなものがあります。

- 東: ひがし、あずま、とう
- 中田: なかだ、なかた
- 仮名: かな、かめい
- 牧場: ぼくじょう、まきば
- 網代: あみしろ、あじろ
- 日本: にほん、にっぽん

文脈から読み方が分かる場合も多いですが、同形異音語の多くは人名や地名などの固有名詞なので、どの人・場所に対して使われているかを知らないと、読みが判別できないこともあります。更に大変なことに、MeCab の出力を見るだけでは、ある単語の読みが曖昧かどうかは分かりません。

同形異音語の読みの判別は、一般名詞の場合は**語義の曖昧性解消** (word sense disambiguation)、固有名詞の場合は**エンティティ・リンキング** (entity linking) のタスクであると捉えることもできます。これらのタスクは自然言語処理の分野では昔から多くの研究が行われてきた課題です。

同形異音語の読みがきちんと判定できないのであれば、どう処理すれば良いでしょう。実はその曖昧性自体がある種の救いになっています。読み方が曖昧な

単語だと、人間でも読みを間違えることがあるので、プログラムが間違った読みを出しても読み手がある程度分かってくれます。読み間違いが許されないような応用も数多くありますが、今回の伏せ字プログラムでは、これはあまり気にしなくても特に問題にはなりません。

NLP システムの開発中に問題に出くわした場合、その問題の解決方法を実装するのが現実的ではない場合があります。例えばこの伏せ字プログラムの場合、同形異音語の読みの曖昧性を解消するプログラムを書くことは可能ですが、おそらく解決に必要な作業量は、伏せ字プログラムの他の部分を上回ってしまうでしょう。ただし、問題を正しく認識していれば、うまくいかない場合が出力にどういう影響を及ぼすかを考え、元のプログラムを改善する方法を閃くこともあります。場合によっては開発を断念せざるを得ないこともありますが、逆に問題を最初から回避する仕組み導入できる場合もあります。

それでは早速、読み仮名に変換するところを実装しましょう。

```
def fuseji_text(text, ratio=1.0):
    """入力文に伏せ字を適用する """
    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",
]

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

```
黒幕の正体は◯ー ラ◯ド
```

これで今回の伏せ字プログラムは完成となります。行数は決して多くはありませんが、これを書く過程で、下記の機能の使い方を紹介しました。

1. 文章の単語を一つずつ処理する方法

2. 例文を使って目的の品詞を特定する方法
3. 品詞の構造の扱い
4. 未知語の判別
5. 読み仮名変換

これらの機能はどちらも基礎的な処理なので、組み合わせによってさまざまなプログラムを作ることができます。

今回の伏せ字プログラムは単純で遊び的なものに過ぎませんが、ここで使った技術をさらに発展させ、実用的な個人情報漏洩防止ツールの実装も可能です。このようなツールは、カルテのような医療情報や契約書などの法的文書を監査や解析に出す前に、個人情報を特定・削除するために広く使われています。

MeCab の API を更に深く理解するために、下記の場合、どうやってこの伏せ字プログラムを変更するか考えてみましょう。

- 契約書から日付や金額などの数字を消す
- 品詞によってではなく、禁止語など特定の単語を伏せる
- 難読語を読み仮名に変換する

トランスフォーマーを用いた自然言語生成と変換

5.1 トランスフォーマーとテキスト生成

トランスフォーマーとは

トランスフォーマーは、Devlin らによって 2017 年に開発された[1]ニューラルネットワークのアーキテクチャです。このアーキテクチャのコアとなるのは、自己注意機構 (self-attention) と呼ばれるメカニズムで、重要なトークンに注目し、入力表現の重み付き和を取ることによって、入力トークンの系列を出力表現へと変換します。トランスフォーマーの技術的詳細については本書では詳しく解説しません。興味のある読者の方は、Jay Almmar 氏の素晴らしいブログ記事 The Illustrated Transformer[2] を参照されると良いでしょう。ここでは、入力（トークンの埋め込み）を、同じ長さの表現系列に変換してくれる非常に大きな容量を持った強力なニューラルネットワーク・アーキテクチャと考えておけば十分でしょう。この表現系列を、翻訳や、分類、言語モデリングなど様々な NLP タスクを解くために使うことができます。

トランスフォーマーは NLP 業界に衝撃を与え、たちまち、様々なタスクにおいて「デファクト」として選択されるモデルになりました。本章の残りでは、このトランスフォーマーを活用し、言語生成および変換を実現します。

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

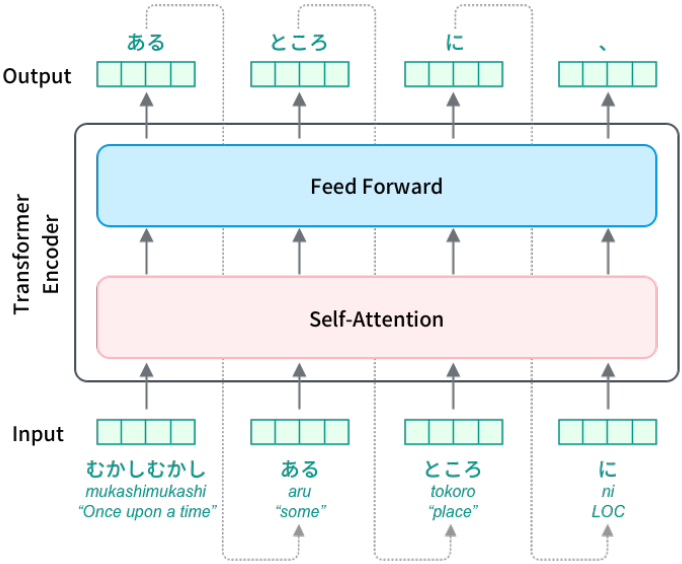

図 1: トランスフォーマーのアーキテクチャ。次のトークンを自己回帰的に予測します

本節では、言語モデルを使って日本語のテキストを生成し、質問応答 (QA) のタスクを解いてみます。言語モデルとは、入力のテキストに対して何らかの情報 (確率) を計算する統計的なモデルのことを指します。言語モデルは通常、大規模コーパス（自然言語テキストのデータセット）から学習されるので、意味の通る文、すなわち、その言語において「自然な」文に高い確率を付与し、不自然な文に低い確率を付与します。

最も広く使われている言語モデルの種類に、因果 言語モデル (causal language model; CLM) があります。CLM は 自己回帰的言語モデル とも呼ばれ、ある入力の確率を、個別のトークンの系列に分解し、文脈を考慮したトークンごとの確率の積としてモデル化します。

上の図は、トランスフォーマーに基づく CLM を表しています。「むかしむかし」という文脈を与えると、次に出現するトークンの確率分布（「、」や「ある」などが含まれている）を計算します。モデルの学習は、訓練データに現れるトークン系列に高い確率を付与するように進みます。

NLP では、言語モデルは単語 N グラムの統計モデル、もしくは、LSTM (long short-term memory) のようなリカレントニューラルネットワーク (recurrent neural network; RNN) によって実装するのが伝統的に行われてきました。しかし、本書執筆時点 (2021 年) では、最先端の言語モデルの多くが、トランスフォーマーのアーキテクチャに基づいて実装されています。

テキスト生成

上で見たように、CLM は、文脈が与えられたとき、トークンの確率分布を与えるように学習されます。これは、トークンを 1 個ずつ選ぶことによって、言語モデルを使って新しいテキストを生成することができることを意味します。

言語モデルの訓練には通常、大規模な訓練データと計算量 (何十個、何百個もの GPU) が必要になるため、多くのタスクでは、訓練済みの言語モデルをダウンロードして使うのが一般的です。よく使われる英語の言語モデルとして、OpenAI によって開発された GPT-2[3] と GPT-3[4] があります。

本節では、深層 NLP モデルの開発用に非常に人気のあるライブラリである HuggingFace Transformers[5] を中心に使います。このライブラリは、BERT[6] (第 6 章で解説) や GPT-2 など、トランスフォーマーベースの訓練済み言語モデルを多数サポートしています。

以下ではまず、言語生成と質問応答に必要なライブラリとモジュールをインストールし、インポートします。これらには、Transformers と、datasets[7]、日本語テキストをトークン化する SentencePiece[8]、そして PyTorch があります。

[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
[7] https://github.com/huggingface/datasets
[8] https://github.com/google/sentencepiece

```
%%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

# transformers のログ出力を制限
logging.getLogger("transformers").setLevel(logging.ERROR)
logging.getLogger("transformers.trainer").setLevel(logging.
    ERROR)
logging.getLogger("datasets").setLevel(logging.ERROR)

_ = torch.manual_seed(42)
```

日本語の自己回帰的言語モデルとして、オープンソースの日本語 GPT-2 モデルである Rinna[9] (rinna 株式会社によって開発) を使います。

このモデルは、HuggingFace Hub にて、`rinna/japanese-gpt2-medium` という識別子を使って参照できます。使うには、以下のように `from_pretrained()` メソッドから訓練済みモデルをロードするだけです。また、対応するトークナイザー (Tokenizer) を初期化するのを忘れないようにしましょう。モデルの入力となるテキストを処理するのに必要になります。

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

# 日本語 GPT-2 モデルの Rinna をロードする

tokenizer = T5Tokenizer.from_pretrained("rinna/japanese-gpt2-
    medium")
tokenizer.do_lower_case = True  # due to some bug of
    tokenizer config loading

```

[9] https://github.com/rinnakk/japanese-pretrained-models

```
8  model = AutoModelForCausalLM.from_pretrained("rinna/japanese-
       gpt2-medium").to(device)
```

まずはじめに、訓練済みモデルを使って日本語のテキストを生成してみましょう。ゼロからテキストを生成することもできますし、プロンプトと呼ばれる他のテキストに継続する形で生成することもできます。以下では、「むかしむかし、あるところに」というプロンプトに続くテキストを生成してみます。

以下のように、入力（プロンプト）をトークン化した後は、`model.generate()` メソッドを使って継続するテキストを生成します。`generate` メソッドは多くのパラメータを取りますが、ここでは重要ではないので割愛します。結果は、`tokenizer.decode()`を使ってデコードし、テキストに戻すことができます。

```
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>'
```

この通り、「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。」から始まる、それらしいストーリーを「りんな」を使って生成することができました。

5.2 質問応答

言語モデルを使って日本語を生成してみるのはそれだけでも興味深く楽しいのですが、言語モデルを使って、さらに色々な問題を解くことができます。

そのような問題の一つに、質問応答があります。ここでは、言語モデルを使って、「北海道の中心に位置することから「北海道のへそ」を名乗る、ラベンダーで有名な都市はどこ?」などのオープンドメインのトリビア的な質問に答えてみましょう（答えは分かりますか ?）

ここでは、例えば、「愛知県の県庁所在地はどこ ?」のような各質問に対し、例えば「札幌」「仙台」「東京」「名古屋」「京都」のような答えの候補のリストが用意されており、モデルはその中から正しい答えを選ぶものとしましょう。このフォーマットは、実際には多肢選択問題で、答えをゼロから生成するのではなく、モデルが自信のある順に候補を並び替える必要があります。

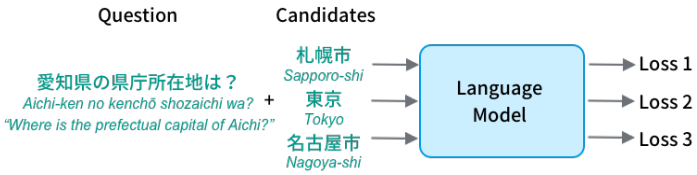

図 2: 言語モデルを使って質問に答える

言語モデルを使ってこのような質問に答えるにはどうしたら良いでしょうか。大規模なコーパスで学習された言語モデルは、より「自然な」入力に対して高い確率（つまり、低い損失関数の値）を返すことを思い出してください。上の図に示したように、候補のリストを並び替えるには、ある候補について、[質問文] [候補] という系列をモデルに入力し、損失関数の値（言語モデルが、その入力をどのぐらい「意外だと思っているか」に対応する）を計算すれば可能です。この質問文と候補の連続が意味を成すものであれば、モデルはより小さい損失関数の値を返します。これを、全ての候補に対して実行し、損失関数の値が一番小さくなるものを選べば良いのです。以下の `rank_answer` メソッ

ドは、候補のリストを、質問とモデルに基づき並び替えます（メソッドの詳細についてはあまり重要ではありません）。

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

まず、「愛知県の県庁所在地は？」という簡単な質問を試してみましょう。日本の都道府県庁所在地の都市のリストを候補として使います。

```
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']
```

見てのとおり、言語モデルは「岐阜市」をトップの候補に選び、正解である「名古屋」は 2 番目です。ここから、言語モデルは、完璧ではないにしろ、ある程度の常識、少なくとも日本の県庁所在地に関する何らかの情報を保持していることが分かります。実際、りんなは常識問題にどのぐらい上手く答えられるのでしょうか？以下では、正解度をもう少しきちんと評価してみましょう。

JAQKET データセットを用いた評価

ここでは、東北大学によって開発・配布されているオープンドメインの質問応答データセットである JAQKET データセット[10] を使います。このデータセットには、以下のように常識問題とその答えが含まれており、答えと候補は必ず Wikipedia 記事のタイトルに対応するようになっています:

- 質問: 北海道の中心に位置することから「北海道のへそ」を名乗る、ラベンダーで有名な都市はどこ?
- 答え: 富良野市
- 候補: 富良野市, 名寄市, 三笠市, 幕別町, 北見市, ...

まず、データセット (訓練データと開発データの両方) をダウンロード、フォーマットし、読み込んで、言語モデルのクイズ回答能力を測定できるようにしましょう。

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

```
# "original_answer" のキーを削除（訓練と開発セットの間で不一致の原因となるため）
!jq 'del(.original_answer)' -c train_questions.json > train_questions.nooa.json
```

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

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

evaluate() メソッドは、質問と答えのリストを受け取り、モデルとトークナイザーを使って質問を評価し、モデルがそのうち何問、正しく回答できたかを返します。

```
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
2   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
3   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
4   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
5   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
6   Success rate = 13.0% (13 / 100)
```

上の結果から分かるように、微調整していないりんなのモデルは、開発セットの最初の 100 問のうち、13% に正しく答えられることができました。各質問には 20 個の候補が含まれているので、この正解率はランダムな確率 (1/20 = 5%) よりも高いですが、あまり素晴らしいとは言えません。これをより改善することは可能でしょうか？

言語モデルを微調整して質問応答を解くには

りんななど言語モデルを改善する一つの方法は、多くの例を提示し、それらの質問と解答のペアに対して、より高い確率が与えられるように、言語モデルのパラメータを最適化することです。このプロセスは、「微調整」(fine-tuning) と呼ばれており、訓練済み言語モデルを他のタスクに適応するときに使う最も一般的な方法です。

以下ではまず、HuggingFace の dataset ライブラリを使い、JSONL 形式の JAQKET データセットを読み込みます。訓練用に、訓練データ (train) と開発

データ (dev) の両方をロードしている点に注意してください。

```
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.
```

以下のようにして、データセットのインスタンスを目視でチェックすることもできます。

```
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？'}
```

[質問文] [答え] というペアを言語モデルに提示して微調整するので、データセットの .map() メソッドを使い、質問文と答えを連結し、新しいフィールドとして追加します。

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

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

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

ここで、この連結したテキストのフィールドをトークン化しましょう。もう一度 .map() メソッドを使い、データをバッチで処理します。戻り値として、input_ids フィールドを基本的にそのままコピーした **label** フィールドを返すのを忘れないようにしましょう。言語モデルは、この入力と同じラベルを、トークンごとに再現できるかどうかに基づいて最適化されます。

```
max_length = 256

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
)
```

処理された入力は、以下のようになります。入力が、パディング用のトークン(id: 3)によって埋められている点に注意してください。これは、ニューラルネットワークに何らかのデータを入力する時には、バッチ内のインスタンスが全て同じ長さになる必要があるからです。

```
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]
```

さて、モデルを微調整する準備が整いました。ここでは、`TrainingArguments`を使ってハイパーパラメータを指定し、`Trainer`インスタンスを作成し、`.train()`メソッドを実行します。これには、高速なGPUを使ってもある程度の時間(1時間程度)かかることに注意してください。CPUでは訓練さえできないかもしれません。

```
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()
```

ここで、もう一度モデルを評価してみましょう。今度は、51% の正解率を達成することができました。訓練用と開発用データセットの間には同じ問題は含まれていないので、モデルは単に提示された問題を丸暗記したのではなく、モデルがパラメータを調節して、日本語の常識問題にある程度うまく解答ができるようになったと考えられます。

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

次のステップ

言語モデルを使って、もっと様々な NLP タスクを解くことができます。プロンプトを設計したり、必要に応じて微調整したりして、どうやったらタスクを解くようにできるかを考えるのも面白いでしょう。例えば、以下のタスクを解くにはどうしたら良いでしょうか？

- 翻訳。りんなを使って、例えば、日本語と英語の翻訳をすることはできるでしょうか？
- 演算。りんなは、6+7=? のような簡単な算数の問題に答えることができるでしょうか？
- 単語の類推。日本 → 円、アメリカ →? のような類推問題に答えることができるでしょうか？

もしヒント等が必要であれば、GPT-3 の論文[11] にこのような例がたくさん載っています。

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