令和 3 年度茨城大学工学部情報工学科卒業研究論文

複数の BERT モデルを利用した Data Augmentation

所属 情報工学科

著者 高萩恭介 (18T4056G)

指導教員 新納浩幸教授

令和 4 年 1 月 31 日 (月)

令和 3 年度茨城大学工学部情報工学科卒業研究論文

複数の BERT モデルを利用した Data Augmentation

著者

高萩恭介 (18T4056G)

指導教員

新納浩幸教授

論文要旨

Data Augmentation は，機械学習における訓練データに何らかの変換を施し，変換後の訓練データを元の訓練データに追加することでデータ数を増やす手法のことである．この Data Augmentation を利用することで，訓練データの数が少ない場合に機械学習モデルの性能を向上させることができる．

自然言語処理で利用される簡単な Data Augmentation の手法の 1 つに，文中の単語をそれに類似する単語に置き換えるというものがある．しかし，類似単語の取得とタスクを解くモデルに同じ事前学習済みモデルを利用する場合，この手法による効果は期待できない．なぜなら，Data Augmentation によって得られる類似単語の知識は，既にタスクを解くモデルに組み込まれていると考えられるからである．

本論文では，類似単語の取得とタスクを解くモデルの構築に異なる事前学習済みモデルを利用する手法を提案する．この場合，取得した類似単語の知識は，タスクを解くモデルには含まれていないと考えられるため，Data Augmentation の効果が期待できる．

実験では，事前学習済みモデルの一種である BERT を利用した．そして，livedoor ニュースコーパスを利用した文書分類タスクに対して，提案手法による Data Augmentation を試み，提案手法の有効性を示した．

また，実験で使用した 2 種類の BERT について，それぞれを利用して同条件で類似単語の取得を行った．これによって，2 種類の BERT が持つ知識がどの程度異なっているかについて調査した．さらに実験での訓練データの量を 2 倍に増やして再度実験を行った．その結果，提案手法による Data Augmentation は，タスクにおける訓練データが小規模な場合のみに有効であることが確認できた．

Bachelor's Thesis in Scholastic 2021,

Department of Computer and Information Sciences, Ibaraki University

Data Augmentation using Multiple BERT models

Author

Kyosuke Takahagi (18T4056G)

Adviser

Prof. Hiroyuki Shinnou

Abstract

In machine learning, data augmentation is a method of increasing the amount of data by applying some modification to the training data and adding the modified training data to the original training data. By using this data augmentation, the performance of machine learning models can be improved when the number of training data is small.

An easy data augmentation method in natural language processing (NLP) is to replace words in a sentence with similar words. However, this is not effective when using pretrained models such as BERT models, because knowledge of similar words is included in a BERT model.

In this paper, we propose to use the masked language model of a different kind of BERT model from the one used for task processing to obtain similar words. In this case, we can use the knowledge of similar words that are not included in the BERT model that is used for task processing. Thus, we expect that the proposed data augmentation method will be effective.

We conducted an experiment in which we applied the proposed data augmentation method to a document classification task using the Livedoor news corpus. The results demonstrate the effectiveness of the proposed data augmentation method.

In addition, we used two BERT models used in the above experiment and obtained similar words under the same conditions for each BERT model. By doing so, we investigated how much the knowledge of each BERT model differed. Furthermore, we doubled the amount of training data used in the above experiment and ran the experiment again. As a result, we confirmed that the proposed method is effective only when the training data in the task is small.

目次

第 1 章 序論 8

第 2 章 関連研究 11
2.1 画像処理における Data Augmentation 11
2.2 自然言語処理における Data Augmentation 12
2.3 BERT 14
2.4 TF-IDF 17
2.5 MeCab 18

第 3 章 提案手法 20
3.1 BERT を用いた類似単語の生成 20
3.2 マスクする単語と置換単語の決定 21

第 4 章 実験 24
4.1 実験設定 24
4.2 実験結果 27

第 5 章 考察 30
5.1 追加実験 1（2 つの BERT の類似単語の重なり） 30
5.2 追加実験 2（訓練データの量の影響） 33

第 6 章 結論 35

参考文献 37

付録 39

A プログラムリスト 39

表目次

4.1 カテゴリと対応するラベル，各データに含まれる各カテゴリのテキスト数 25
4.2 学習に東北大版 BERT を用いたモデルの正解率 27
4.3 学習にストックマーク版 BERT を用いたモデルの正解率 28

5.1 0 から 5 までの各一致数にそれぞれ該当する単語数 31
5.2 学習に東北大版 BERT を用いたモデルの正解率 (訓練データ 2 倍) 34

図目次

1.1 少しの変換によって文の意味が変わる例 9

2.1 Google 翻訳を利用した逆翻訳の例 13
2.2 統一的な Data Augmentation アプローチ 14
2.3 BERT の構造 15
2.4 MeCab の実行結果の例 19

3.1 本研究で提案する Data Augmentation の方法 21
3.2 TF-IDF を用いたマスク対象単語の選択方法 22

4.1 文書分類モデルの図 26
4.2 学習に東北大版 BERT を用いたモデルの正解率 28
4.3 学習にストックマーク版 BERT を用いたモデルの正解率 29

5.1 0 から 5 までの各一致数にそれぞれ該当する単語数 32
5.2 学習に東北大版 BERT を用いたモデルの正解率 (訓練データ 2 倍) . . . 34

第 1 章

序論

Data Augmentation とは，基本的にラベル付きのデータに対してラベルをそのままにし，データに簡易な変換を施したものを新規のラベル付きデータとして元のデータに加える手法である．機械学習を用いて何らかのタスクを解く時，そのタスクに応じた訓練データが必要である．一般的に訓練データの数が多いほどタスクを高い精度で解くことができる．そのため，Data Augmentation によって訓練データの数を人工的に増やし，機械学習モデルの性能向上を図る研究が多く行われている．

画像処理の分野では，一般的に Data Augmentation の有効性が高いため，教師あり学習を行う際に標準的に利用されている [1]．一方自然言語処理の分野では，Data Augmentation の有効性は低く，研究についても画像処理ほど盛んではない．これは，自然言語処理で扱うデータが離散的であることが原因とされる．データが離散的であると，そのデータを変換して新しいデータを生成する際に不変性を維持することが難しくなる．例えば，図 1.1 のように離散データである文に対して少し変換を施しただけでも，元の文が表す意味と大きく変わってしまうことがある．

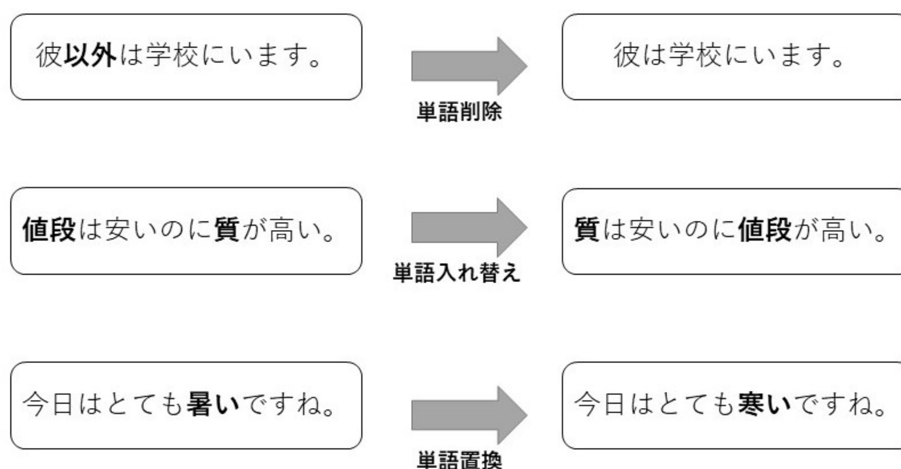

図 1.1: 少しの変換によって文の意味が変わる例

このように，自然言語処理の分野において，Data Augmentation を利用して機械学習モデルの性能を向上させることは困難である．しかし有効な Data Augmentation の手法もいくつか存在する．そのうちの一つが，文中の単語をその単語の類似単語に置き換える手法である [2]．この手法は，事前学習済みモデルである BERT の Masked Language Model を利用することで実現可能である．ただし BERT [3] のような事前学習済みモデルを，類似単語取得とタスクを解くモデルの両方に利用する場合，この手法は有効でないと考えられる．なぜなら，BERT から取得した類似単語の知識は，既にタスクを解くモデルに組み込まれていると考えられるからである．

そこで本論文では，タスクを解くモデルに利用する BERT とは異なる BERT を使って類似単語を得る手法を提案する．この場合，タスクを解くモデルは，モデル自身に利用した BERT とは異なる BERT がもつ類似単語の知識を学習によって獲得できる．したがって，タスクを解くモデルの性能向上が期待できる．

今回用いるタスクは文書分類とし，データセットには livedoor ニュースコーパス[*1] を利用した．また，BERT モデルは，東北大学の乾・鈴木研究室が公開している’bert-base-japanese’[*2]とストックマーク株式会社が公開しているモデル[*3]の 2 種類を利用した．

実験では，提案手法によって拡張した訓練データを含む計 3 パターンの訓練データを

[*1] https://www.rondhuit.com/download.html#ldcc
[*2] https://github.com/cl-tohoku/bert-japanese
[*3] https://qiita.com/mkt3/items/3c1278339ff1bcc0187f

用意し，それぞれを利用して分類モデルを構築した．そして，各分類モデルを使ってテストデータを分類し，その分類精度を比較した．その後，類似単語取得とタスクを解くモデルに利用した BERT の種類を入れ替えて同様の実験を行った．結果は 2 回とも，提案手法によって拡張した訓練データを利用して構築した分類モデルが最高性能となり，提案手法の有効性を示すことができた．

考察では，2 つの追加実験を行った．まず，本実験で使用した 2 種類の BERT を利用して同条件で類似単語の取得を行い，得られた類似単語同士を比較した．その結果，各 BERT が持つ知識がどの程度異なっているかについて確認できた．次に，本実験での訓練データの量を 2 倍に増やして再度実験を行った．その結果，提案手法による Data Augmentation は，タスクにおける訓練データが小規模な場合のみに有効であることが確認できた．

第 2 章

関連研究

2.1 画像処理における Data Augmentation

画像処理の分野では，ニューラルネットワークの過学習を回避するためのテクニックとして，Data Augmentation が標準的に用いられている．一般に，学習に利用するデータの数が十分である場合に過学習が起きることは少なく，医療画像解析のような利用できるデータセットの量が少ない場合に過学習が問題になることが多い．このような，データ量が限られているような領域で Data Augmentation を利用し，学習に利用するデータ数を人工的に増やすことで，過学習を回避することが可能となる．

画像処理の Data Augmentation 手法は，一般にワーピングとオーバーサンプリングのどちらかに分類される [1]．ワーピングは，画像の幾何学的変換および色変換などによって，単一の画像のある特徴を変化させて新しい画像を生成する方法であり，既存の画像のラベルを保持したまま変換を行う．オーバーサンプリングは，画像の混合や特徴空間の拡張によって新しい画像生成する方法である．

Data Augmentation は，当初はラベル付きデータに対する簡易な小さな変換から始まったが，現在は様々な方向に裾野を広げている．例えば，ラベル無しデータに対して何らかのラベルを付与して訓練データを増やす手法である半教師あり学習は Data Augmentation の一種と見なすことができる．また GAN などの生成系ニューラルネットワークの手法も訓練データを追加する形になっており，これらも Data Augmentation の一種と見なすことができる [2]．さらに，Mixup と呼ばれる手法も提案されている．これは，2 つの訓練データとそのデータがもつラベルを各々線形結合し，新たな訓練データを作成する Data Augmentation の手法である [4].

2.2 自然言語処理における Data Augmentation

2.2.1 背景

画像処理の分野で活発に研究が行われている Data Augmentation であるが，自然言語処理の分野においてはまだまだ未開拓である．また，これまでに行われている研究も画像処理分野での Data Augmentation の研究に対する二次的なものが多い．

自然言語処理における Data Augmentation が，画像処理分野に比べて盛んではないのは，一般に自然言語処理で扱うデータが離散的であることが原因と考えられる．データが離散的であると，変換を行う際に不変性を維持することが難しくなる．例えば，離散データである文に対して，文中の単語の順番を入れ替える，別の単語に置き換えるといった処理を行っただけでも，文の意味が通らなくなることや，元の文が表す意味と大きく変わってしまうことがある．

このように，自然言語処理において有効な Data Augmentation 手法を生み出すことは，画像処理と比べると困難である．ただしいくつかの試みは行われており，有効な手法も提案されている．

2.2.2 EDA

EDA (Easy Data Augmentation) とは，Wei らによって提案された言語モデルや外部データを必要としない簡単な Data Augmentation の手法である [5]．具体的には，以下の 4 つの手法を用いて文を変換することで，元の文と似た文を生成する．

Synonym replacement（同義語置換）

文中の単語をランダムに選択し，選んだ単語をランダムな同義語に置き換える．これを n 回繰り返す．ただしストップワードは除く．

Random deletion(ランダム削除)

文中の各単語についてそれぞれ確率 p で削除する．

Random swap（ランダム入れ替え）

文中の 2 つの単語をランダムに選択し．選んだ単語同士の位置を入れ替える．これを n 回繰り返す．

Random insertion（ランダム挿入）

文中のランダムな単語の同義語をランダムに選び，文中のランダムな位置に挿入する．これを n 回繰り返す．ただしストップワードは除く．

上の手法において，ある単語に対する同義語の取得には Wordnet という英語の概念辞書を利用している．また，手法の説明にあるストップワードとは，何らかの処理を行う上で処理対象外となる単語のことである．一般的には，頻繁に登場する単語や存在の有無が文意に大きな影響を与えない単語などがストップワードとなる．

2.2.3 逆翻訳

逆翻訳とは，図 2.1 のように，言語 A で記述された文書を言語 B に翻訳し，その翻訳文を言語 A に再び翻訳し直すことである．図 2.1 では，Google 翻訳を利用して日本語を英語に翻訳した後，再び日本語に翻訳し直している．この逆翻訳を利用した Data Augmentation を行う試みは多くなされている [6] [7].

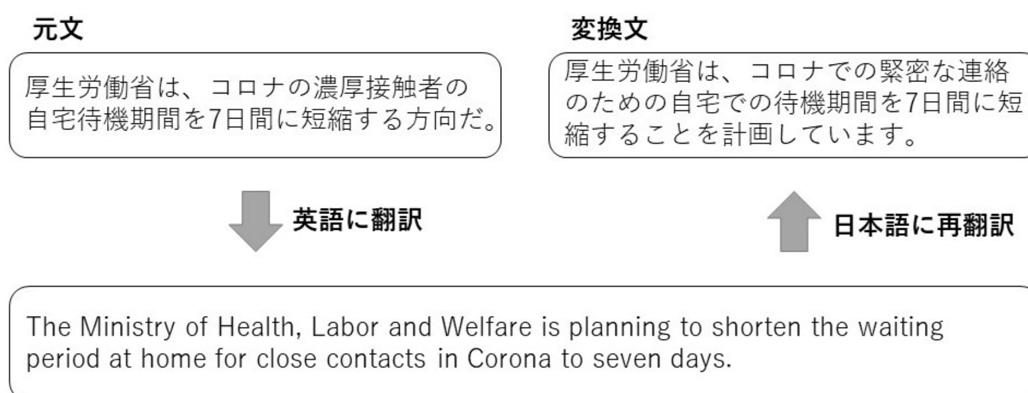

図 2.1: Google 翻訳を利用した逆翻訳の例

2.2.4 自然言語処理における Mixup

Gun らは画像処理分野で用いられている Mixup を応用し，文の埋め込み表現あるいは単語の埋め込み表現を混合する Sentence Mixup や Word Mixup を提案している [8]. Chen らは BERT のある層を混合する TMix に半教師あり学習を併用した MixText を提案している [9].

2.2.5　事前学習済みモデルを使った Data Augmentation

Kumar らは transformer をベースとする事前学習済みモデルを使った Data Augmentation 手法を提案している [10]．具体的には，GPT-2，BERT，BART の 3 つの事前学習済みモデルを使って Data Augmentation を行い，各手法について様々な観点から比較を行っている．

この研究では，異なる 3 つの事前学習済みモデルを使った Data Augmentation の性能について平等に比較を行うために，図 2.2 のような統一的な Data Augmentation アプローチを提案している．

また，事前学習済みモデルに対して，ファインチューニングを行う際にクラスラベルをテキスト列に前置することで，Data Augmentation に効果的な条件を与える方法を示している．

アルゴリズム1: Data Augmentation approach

入力: 訓練データセット D_{train}
事前学習済みモデル $G \in \{AE, AR, Seq2Seq\}$

1. Fine-tune G using D_{train} to obtain G_{tuned}
2. $D_{sybthetic} \leftarrow \{\}$
3. foreach $\{x_i, y_i\} \in D_{train}$ do
4. Synthesize s examples $\{\hat{x}_i, \hat{y}_i\}_p^1$ using G_{tuned}
5. $D_{synthetic} \leftarrow D_{synthetic} \cup \{\hat{x}_i, \hat{y}_i\}_p^1$
6. end

図 2.2: 統一的な Data Augmentation アプローチ

2.3　BERT

2.3.1　BERT の概要

BERT [3] は Bidirectional Encoder Representations from Transformers の略で, 2018 年に Google が発表した言語モデルである．BERT は当時，多くの自然言語処理タスク

において，最先端のモデルの大きく上回るスコアを記録した．BERT は，Attention という方法を用いることで，文脈に応じた処理を行うことができる．また，学習は，事前学習とファインチューニングの 2 段階で行われている．

2.3.2　BERT の構造

BERT は，トークン列をベクトル化したものを入力として受け取り，入力列の各トークンに対応するベクトルを出力するモデルである．BERT は図 2.3 のようにいくつかの層から構成される．それぞれの層は各トークンに対応するベクトルを出力し，次の層はそれを受けて，新たに各トークンに対応するベクトルを出力する．

BERT の各層には， Transformer というモデルの Encoder（Transformer Encoder）が用いられている．Transformer Encoder は，おもに Multi-Head Attention と Feed-forward Network という要素で構成されている．そして各層では，Attention(注意機構) という方法を用いて，各トークンの情報を処理するときに他のトークン情報を直接参照する処理を行う．このとき，それぞれのトークンの情報にどの程度注意を払うかは，それぞれのトークンに応じて適応的に決定する．

BERT は，Attention を用いることで，離れた位置にあるトークンの情報も適切に取り入れることができる．そのため，より深く文脈を考慮したトークンの分散表現を獲得することが可能である．また，層内でのそれぞれのトークンに対する出力は独立に計算できる．つまり，並列化による高い計算効率が実現可能である．

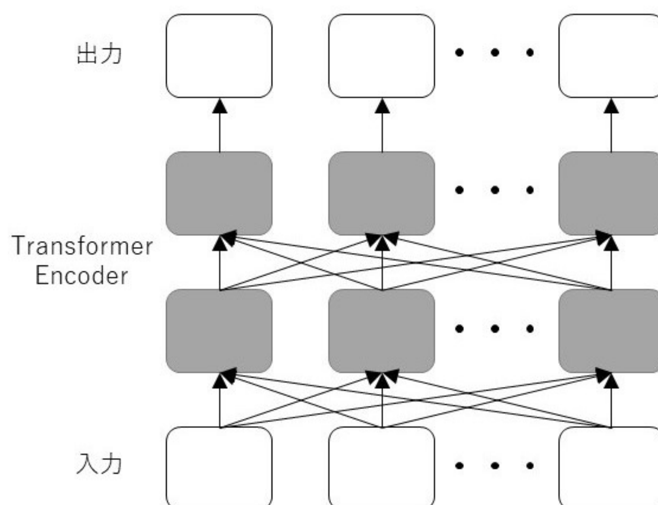

図 2.3: BERT の構造

2.3.3　事前学習

BERT は事前学習を行うことで，汎用的な言語のパターンを学習することができる．事前学習には大量のラベルなしデータが利用され，そのタスクには以下の 2 つが用いられる．

Masked Language Modeling

Masked Language Modeling 文章中のある単語を周りの単語から予測するタスクである．具体的には次の手順で行われる．

1. ランダムに選ばれた 15% のトークンを [MASK] という特殊トークンに置き換える．
2. 1 で置き換えた文章を BERT に入力し，[MASK] の位置に元々あったトークンを予測する．

つまり，[MASK] に置き換えられたトークンを，[MASK] を含む文章に対するラベルとして扱うことで，その入出力関係を学習する．[MASK] を含む文章から [MASK] に入るトークンを予測する言語モデルを Masked Language Model という．

Next Sentence Prediction

Next Sentence Prediction は，2 つの文の関係性を理解するために用いられるタスクである．そのために，事前学習時には，BERT に常に 2 つの文のペアが入力される．具体的な手順を以下に示す．

1. 2 つの文を BERT に入力する．入力された 2 文が連続した文である確率は 50% である．
2. 1 で入力された 2 文が連続しているかどうかを判定する．

つまり，「連続している」「連続していない」を入力された 2 文に対するラベルとして扱うことで，その入出力関係を学習する．

2.3.4　ファインチューニング

ファインチューニングとは，個別のタスクのラベル付きデータを用いて，BERT がそのタスクに特化するように学習を行うことである．BERT で個別のタスクを解くとき，そのタスクに応じて BERT に分類器などを接続するなどして，そのタスクに特化したモデルを作る．つまり，自然言語処理タスクにおいて，BERT は特徴抽出器のような働きをする．ファインチューニングは次の手順で行う．

1. モデルのパラメータの初期値を設定する．
   - BERT のパラメータの初期値…事前学習で得たパラメータ
   - BERT に接続された分類器のパラメータの初期値…ランダムな値
2. ラベル付きデータを用いて，BERT と分類器の両方のパラメータを学習する．

このように，事前学習で得られたパラメータを初期値とすることで，比較的少数の訓練データからでも高い性能のモデルを構築できる．

2.4　TF-IDF

文字列をプログラムで扱う場合，何からの方法によりコンピュータで計算可能な形式に変換する必要がある．機械学習の分野では，文を特徴量という「元の文が持つ特徴が反映されたベクトル」に変換する手法が用いられていており，この手法を特徴抽出という．

IF-IDF は，特徴抽出手法の 1 つであり，ある文書中に含まれる各単語を，「その単語が文書内でどれくらい重要か」を示す数値として表すことができる．このときの数値は，そのまま単語の IF-IDF という．TF-IDF は，TF と IDF という 2 つの値を掛け合わせることによって求められる．具体的には以下の手順で計算される．

1．TF を計算する

TF（Term Frequency）は単語頻度である．具体的には，「ある文」における「ある単語」の出現回数を表す．文 d における単語 t の TF は次のように求められる．

$\mathrm{TF}(t,d) =$ 文 d に登場する単語 t の数

2．DF を計算する

DF（Document Frequency）は文書頻度である．具体的には，「全ての文」に対す

る「ある単語を含む文」の割合を表す．単語 t の DF は次のように求められる．

$\mathrm{DF}(t) = \frac{t\ \text{を含む文の総数}}{\text{文の総数}}$

3．IDF を計算する

IDF(Inverse Document Frequency) は，DF の逆数の対数のことである．対数を取るのは，値の変化を緩やかにするためである．

$\mathrm{IDF}(t) = \log\left(\frac{1}{\mathrm{DF}(t)}\right) = \log\left(\frac{\text{文の総数}}{t\ \text{を含む文の数}}\right)$

上の式を見て分かるように，多くの文に登場する単語の IDF は小さくなる．

4．TF と IDF を掛ける

1. で求めた TF と 3. で求めた IDF を掛け合わせた値が TF-IDF になる．

$\mathrm{TF\text{-}IDF}(t,d) = \mathrm{TF}(t,d) \cdot \mathrm{IDF}(t)$

2.5　MeCab

自然言語で書かれた文をプログラムで扱う場合，まず文を単語に分割して各単語を数値化するという処理を行うのが一般的である．この処理によって，文をベクトルと見なせるようになるため，プログラムでの扱いが容易になる．

英語のように初めから単語の間にスペースがある言語に関しては，ほとんどの場合，文を単語に区切る処理をする必要はない．しかし，日本語のように単語間にスペースがない言語は，まず文中の単語の境界を判定し，文を単語に分解する必要がある．この文を単語に分割することを分かち書きという．

MeCab(和布蕪)[*1]は，日本語用の形態素解析エンジンであり，これを使うことで日本語を分かち書きすることができる．形態素解析とは，文を意味を持つ最小の単位である形態素に分割し，それぞれの形態素の品詞などを判別する処理のことである．実際に MeCab を使って日本語の文を形態素解析した結果を図 2.4 に示す．

MeCab による形態素解析は，様々な単語の情報が格納されたデータベースである辞書に基づいて行われる．この辞書には様々な種類があり，用途に応じて利用する辞書を選ぶことができる．

[*1] http://taku910.github.io/mecab/

```
:~$ mecab
今日はいい天気ですね。
今日	名詞,副詞可能,*,*,*,*,今日,キョウ,キョー
は	助詞,係助詞,*,*,*,*,は,ハ,ワ
いい	形容詞,自立,*,*,形容詞・イイ,基本形,いい,イイ,イイ
天気	名詞,一般,*,*,*,*,天気,テンキ,テンキ
です	助動詞,*,*,*,特殊・デス,基本形,です,デス,デス
ね	助詞,終助詞,*,*,*,*,ね,ネ,ネ
。	記号,句点,*,*,*,*,。,。,。
EOS
```

図 2.4: MeCab の実行結果の例

第 3 章

提案手法

3.1 BERT を用いた類似単語の生成

BERT モデルは Masked Language Modeling というタスクによって事前学習されており，BERT モデル自体に，マスクされた単語を推定する仕組みである Masked Language Model(MLM）が含まれている．MLM は BertForMaskedLM と呼ばれる機構を用いて BERT モデルから取り出すことができる．

本研究ではまず，訓練データのテキスト中から単語を選択し，その単語に対応する類似単語を MLM を用いて生成する．そして，選択した単語を生成した類似単語に置き換え，置き換え後のテキストを訓練データに追加する．具体的な Data Augmentation の手順を以下及び図 3.1 及び以下に示す．

1. 訓練データ中の特定のテキストを，トークナイザーを用いて単語列に分割する．
2. BERT モデルの語彙リストファイル vocab.txt に基づいて，単語列を ID 列に変換する．
3. マスクしたい単語に対応する ID を MASK トークンの ID に置き換え，置き換え後の ID 列を MLM に入力する．
4. MLM の出力として，ID 列に対応する各単語の位置に出現する単語の分布が得られるため，そこから MASK の位置に出現する確率が高い単語を上位 5 つ取り出す．
5. 取り出された 5 つの単語から，置換に最も適した単語を選択する．
6. 元のテキストの MASK の位置にある単語を選択した単語で置き換える．

7. 置き換え後のテキストを訓練データに追加する．

訓練データの数が設定した数に達するまで上記の手順を繰り返す．なお，本研究の提案手法では，上記の Data Augmentation に利用する BERT モデルは，タスクを解くモデルに利用する BERT モデルと別種のものを利用することとする．

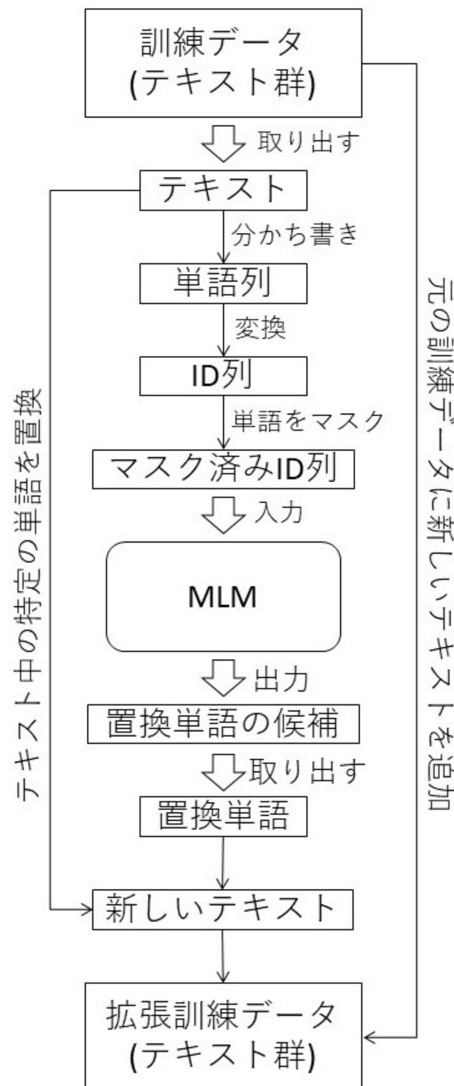

図 3.1: 本研究で提案する Data Augmentation の方法

3.2　マスクする単語と置換単語の決定

Data Augmentation によって新しいテキストを生成するためには，元の訓練データから文とその文中のマスクすべき単語を適切に選択することが必要となる．本研究では，TF-IDF を用いてマスクする単語を選択する．具体的な選択方法を以下及び図 3.2 に

記す．

1. Python の機械学習ライブラリである scikit-learn の TfidfVectorizer を用いて訓練データのテキストを TF-IDF ベクトルに変換する．ここでの TfidfVectorizer のトークナイザーは，Data Augmentation に利用する BERT のトークナイザーである．
2. TF-IDF ベクトルを参照し，訓練データ中の全単語を TF-IDF 値の大きい順に並べ替える．
3. 未選択の単語の中から，TF-IDF 値が最も高い単語を選択する．選択した単語が以下の条件を満たすならば，その単語をマスクする．
   - 単語が名詞である．
   - 単語が BERT モデルの vocab.txt に存在する．
4. 手順 3. に戻る．

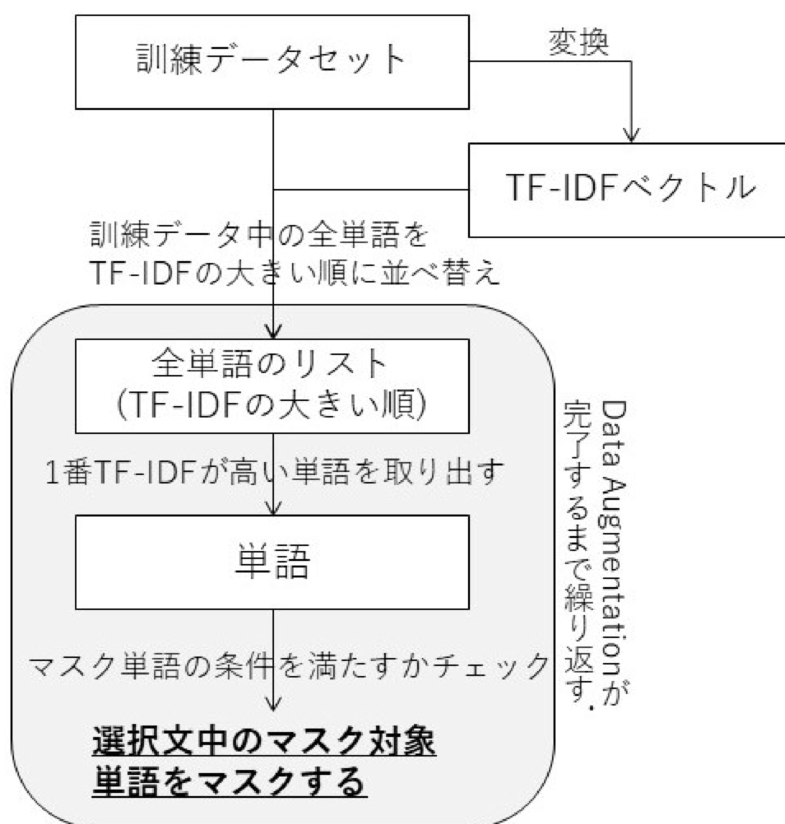

図 3.2: TF-IDF を用いたマスク対象単語の選択方法

さらに，マスクされた単語を置き換える際に，5 つの候補単語の中から最も置換に適した単語を決定する必要がある．このときの具体的な決定方法を以下に記す．

1. 未選択の単語の中から，文中のマスクされた単語の位置に出現する確率が最も高い単語を選択する．
2. 選択された単語が以下の条件を完全に満たす場合，その単語を置換単語に決定する．それ以外の場合は手順 1 に戻る．
   - 単語が名詞である．
   - 単語が BERT モデルの vocab.txt に存在する．
   - 単語がマスクされた単語と同じものである．
   - 選択された単語がマスクされた単語を含んでおらず，かつマスクされた単語が選択された単語を含んでいない．

第 4 章

実験

4.1 実験設定

4.1.1 実験で使用する BERT モデル

本実験では，Data Augmentation とタスクを解くモデルで別の BERT モデルを利用するために，2 つの BERT モデルを用意した．1 つは東北大学の乾・鈴木研究室が公開している BERT モデル”bert-base-japanese”（以下「東北大版 BERT」という），もう 1 つはストックマーク社が公開している BERT モデル（以下「ストックマーク版 BERT」という）である．

東北大版 BERT は，事前学習に日本語 Wikipedia を用いている．また，トークナイザーには MeCab と WordPiece を使用している．MeCab の辞書は Unidic 2.1.2 である．

ストックマーク版 BERT は，ビジネスニュース向けの BERT モデルで，事前学習に日本語ビジネスニュース記事を用いている．また，トークナイザーは MeCab であり，辞書は NEologd を使っている．

4.1.2 実験で使用するデータセット

本実験のタスクは文書分類とする．文書分類モデルの学習・評価に用いるデータセットには，livedoor ニュースコーパスを利用する．livedoor ニュースコーパスは，NHN Japan 株式会社が運営する「livedoor ニュース」の中からニュース記事を収集し，可能な限り HTML タグを取り除いて作成したものである．このコーパスには，9 つのカテゴリのニュース記事が格納されている．

このコーパスの各カテゴリに対してラベルを割当て，コーパスから各ラベルのテキストを同数取り出す．取り出し方はランダムとする．そして，取り出したテキストとラベルの組から，訓練データ，検証データ，テストデータの 3 つを作成する．このときのテキストのカテゴリとそれに対応するラベル，各データに含まれる各カテゴリのテキスト数は，表 4.1 の通りである．

表 4.1 における訓練データが baseline となる．この訓練データに対して，3 章に記した Data Augmentation の手法を適用し，拡張訓練データを作成する．作成するものは，東北大版 BERT の MLM を利用した拡張訓練データ (以下「拡張訓練データ (東北大)」という) と，ストックマーク版 BERT の MLM を利用した拡張訓練データ (以下「拡張訓練データ (ストックマーク)」という) の 2 種類である．データ数は 2 種類とも 370 個であり，その内訳は元のデータが 270 個で Data Augmentation で新たに生成したデータが 100 個である．

表 4.1: カテゴリと対応するラベル，各データに含まれる各カテゴリのテキスト数

ラベル	カテゴリ	訓練	検証	テスト
0	独女通信	30	30	30
1	IT ライフハック	30	30	30
2	家電チャンネル	30	30	30
3	livedoor HOMME	30	30	30
4	MOVIE ENTER	30	30	30
5	Peachy	30	30	30
6	エスマックス	30	30	30
7	Sports Watch	30	30	30
8	トピックニュース	30	30	30
	合計	270	270	270

4.1.3　実験で使用する文書分類モデル

本実験では，訓練データを評価するために，BERT を利用した文書分類モデルを作成する．ここで作成するモデルは，ニューラルネットワークに東北大版 BERT を利用した

ものとストックマーク版 BERT を利用したものの 2 種類である．

作成する文書分類モデルは図 4.1 のように BERT 層と識別のための全結合層から構成される．具体的には，まずモデルに 512 次元以下の ID 列が入力され，BERT 層で 768 次元ベクトルに変換される．この 768 次元ベクトルは入力 ID 列の先頭の CLS トークンに対する BERT の出力である．次に 768 次元ベクトルが全結合層で 9 次元ベクトルに変換され，出力される．9 次元ベクトルの各要素はデータの各ラベルに対応している．つまり，9 次元ベクトルの最大要素のインデックスが予測結果のラベルとなっている．

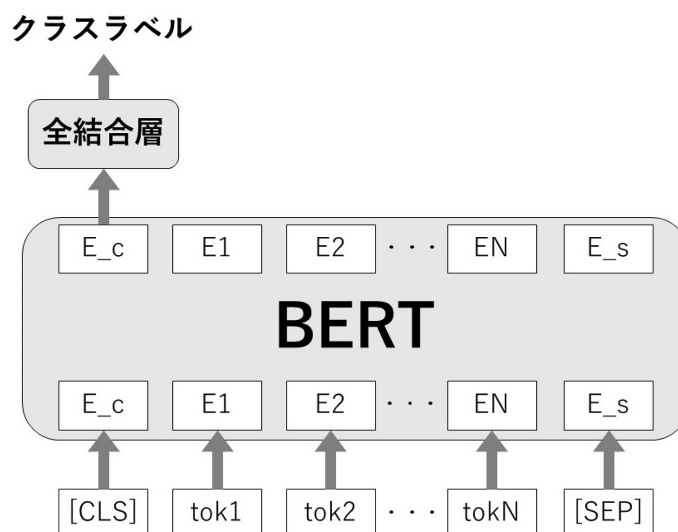

図 4.1: 文書分類モデルの図

4.1.4　学習時の設定

文書分類モデルを学習するときの設定を以下に記す.

- 最適化アルゴリズム: 確率的勾配降下法 (SGD)
- 損失関数: クロスエントロピー
- バッチサイズ: 2

エポック数はアーリーストッピングを用いて決定している．アーリーストッピングは機械学習において過学習を防ぐ効果をもつ手法である．具体的には，まず学習が 1 エポック終了するごとに，その段階で構築されたモデルを検証データにより評価し，検証データに対する正解率を出す．そして，正解率が 5 エポック連続でそのときまでの最高値を下

回った時点で学習を中断する．学習が中断された場合，それまでの最高値となる正解率を出したモデルを最終的なモデルとする．アーリーストッピングによる学習の中断がなかった場合は，30 エポックで学習を終了させる．

4.1.5　評価用モデルの選出方法

本実験で用いた学習プログラムでは，モデルの全結合層の重みの初期値が毎回異なる．そのため，学習に同じ訓練データを使っても，構築されるモデルは毎回異なる．そこで，同じ学習プログラム・訓練データを使って 5 つのモデルを構築し，その中で検証データでの正解率が一番高いモデルを評価用のモデルとする．

4.2　実験結果

4.2.1　文書分類モデルに東北大版 BERT を用いた場合

まずニューラルネットワークの一部に東北大版 BERT を使用したプログラムを用いて実験を行った．ここでは baseline の訓練データ，拡張訓練データ (東北大)，拡張訓練データ (ストックマーク) を用いて計 3 つのモデルを構築した．そして構築した各モデルを用いてテストデータの分類を行い，その正解率を出した．その結果を表 4.2 及び図 4.2 に示す．

表 4.2 に示すように，拡張訓練データ (ストックマーク) を用いたモデルの正解率は baseline の訓練データを用いたモデルの正解率に比べて 0.0074 高かった．また拡張訓練データ (東北大) を用いたモデルの正解率は，baseline の訓練データを用いたモデルの正解率に比べて 0.0148 低かった．

表 4.2: 学習に東北大版 BERT を用いたモデルの正解率

訓練データ	正解率
baseline	0.8629
拡張訓練データ (東北大)	0.8481
拡張訓練データ (ストックマーク)	0.8703

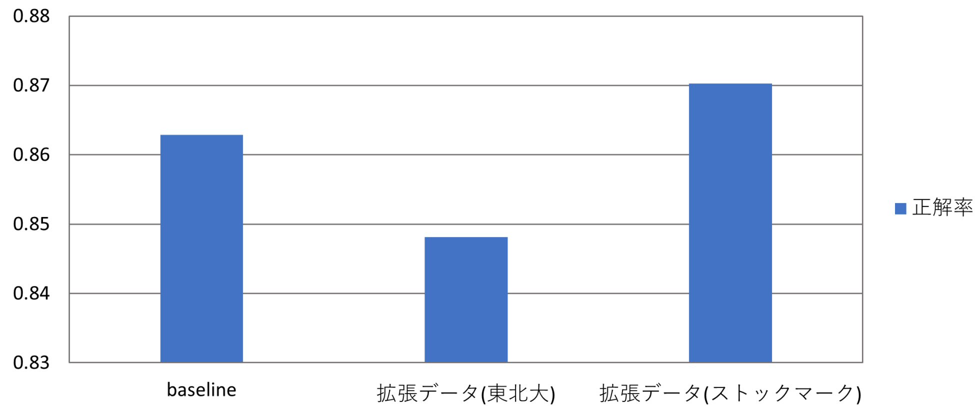

図 4.2: 学習に東北大版 BERT を用いたモデルの正解率

4.2.2　文書分類モデルにストックマーク版 BERT を用いた場合

次にニューラルネットワークの一部にストックマーク版 BERT を使用したプログラムを用いて実験を行った．ここでは baseline の訓練データ，拡張訓練データ (東北大)，拡張訓練データ (ストックマーク) を用いて計 3 つのモデルを構築した．そして構築した各モデルを用いてテストデータの分類を行い，その正解率を出した．その結果を表 4.3 及び図 4.3 に示す．

表 4.3 に示すように，拡張訓練データ (東北大) を用いたモデルの正解率は baseline の訓練データを用いたモデルの正解率に比べて 0.0296 高かった．また拡張訓練データ (ストックマーク) を用いたモデルの正解率は，baseline の訓練データを用いたモデルの正解率に比べて 0.0259 高かった．さらに拡張訓練データ (東北大) を用いたモデルの正解率は，拡張訓練データ (ストックマーク) を用いたモデルの正解率に比べて 0.0037 高かった．

表 4.3: 学習にストックマーク版 BERT を用いたモデルの正解率

訓練データ	正解率
baseline	0.8333
拡張訓練データ (東北大)	0.8629
拡張訓練データ (ストックマーク)	0.8592

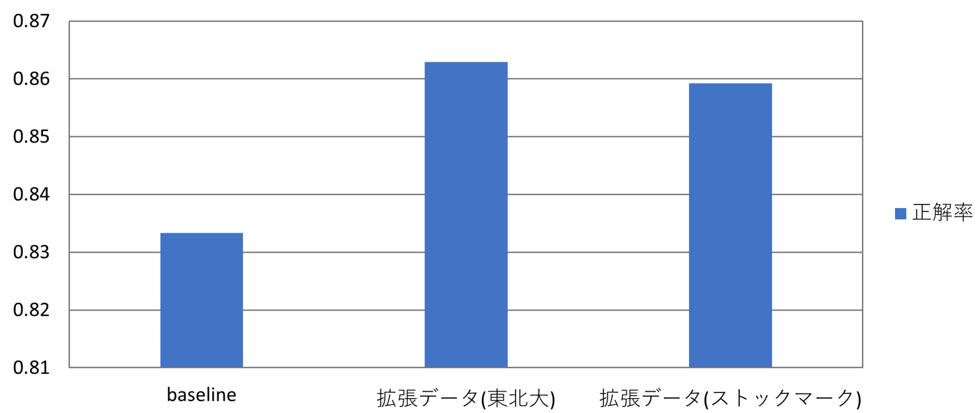

図 4.3: 学習にストックマーク版 BERT を用いたモデルの正解率

第 5 章

考察

5.1 追加実験 1（2 つの BERT の類似単語の重なり）

5.1.1 実験の目的

東北大版 BERT とストックマーク BERT では事前学習に用いたデータが異なるため，それぞれの BERT の MLM を利用して同じ条件で類似単語の予測を行っても結果が異なる．ここでの 2 つの BERT の類似単語の予測結果の違いは，2 つの BERT がもつ知識の違いであると捉えることができる．つまり，提案手法では 2 つの BERT の知識を同時に利用することが可能になり，効果を発揮したと考えられる．ここでは，2 つの BERT モデル間で同じ文章中の同じ単語をマスクした場合の類似単語の予測結果がどの程度異なるかを確認するために，追加の実験を行うこととする．

5.1.2 実験設定

まず，2 つの BERT モデルを用いて訓練データ中のいくつかの単語に対する類似単語の予測を行った．そしてその予測結果から，2 つの BERT における類似単語の重なり具合を調査した．具体的な手順を以下に示す．

1. baseline 訓練データの各テキストを東北大版 BERT の tokenizer で単語に分割する．
2. 手順 1 で得られた全ての単語の中から，東北大版 BERT とストックマーク版 BERT の両方の語彙リストに含まれる 100 個の名詞を，TF-IDF 値が大きい順に選択する．

3. 選択された 100 個の単語一つ一つについて，セクション 3 に記載された方法を用いて，2 つの BERT の類似単語を上位 5 つずつ取得する．
4. 選択された各 100 単語について，東北大版 BERT とストックマーク版 BERT それぞれから得られた 5 つの類似単語同士を比較し，類似単語の一致数を調べる．

5.1.3　実験結果

上記の実験を行った後，TF-IDF 値が大きい順に単語を選択した場合の効果を調べるために，単語の選択方法をランダムに変更して再度実験を行った．2 パターンの実験結果をまとめたものを表 5.1 及び図 5.1 に示す．

ここでは，表 5.1 の結果に基づいて類似単語の一致率を計算した．TF-IDF 値の大きい順に選んだ単語 100 個における 2 つの BERT の類似単語は平均 5 個中 1.64 個（32.8%）重なっていた．ランダムに選んだ単語 100 個における 2 つの BERT の類似単語は平均 5 個中 0.53 個 (10.6%) 重なっていた．

単語を TF-IDF の高い順に選んだ場合とランダムに選んだ場合の結果を比較すると，ランダムに選んだ場合の方が類似単語の重なり具合が悪いことが分かる．このような結果になったのは，ランダムに単語を選んだ場合に，ストックマーク版 BERT による類似単語取得をうまく行えなかったことが原因であると考えられる．具体的には，類似単語として「は」,「を」,「の」,「、」,「。」などを多く取得してしまった．これらの単語は明らかに今回選択した単語に対する類似単語ではなかった．そこで今回は，単語を TF-IDF の大きい順に選んだ場合の結果のみについて考察を行うこととする．また，訓練データのテキスト群の分かち書きに使うトークナイザーを東北大版 BERT からストックマーク版 BERT のトークナイザーに変更して，この章の実験を再度行ったが，変更前の実験と同じような結果が得られた．

表 5.1: 0 から 5 までの各一致数にそれぞれ該当する単語数

	0	1	2	3	4	5	合計
TF-IDF	13	33	34	17	3	0	100
ランダム	68	19	7	4	2	0	100

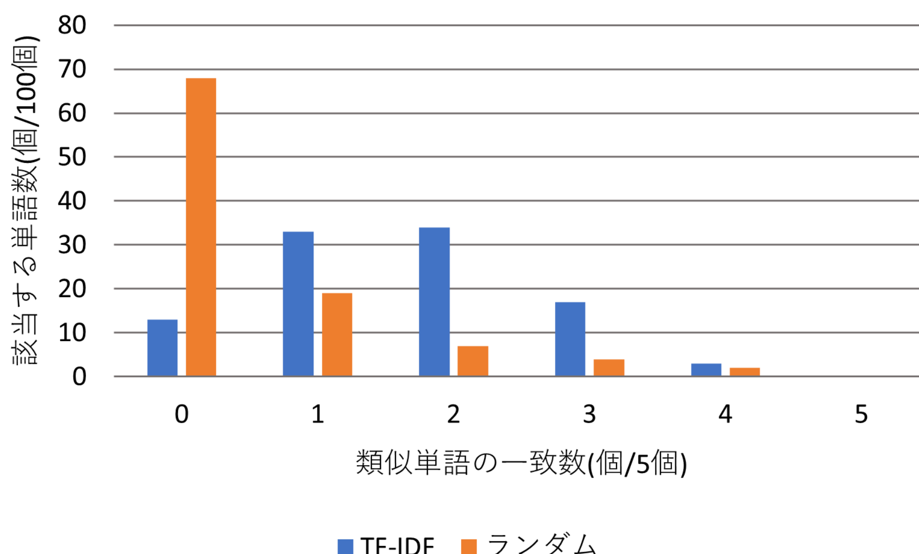

図 5.1: 0 から 5 までの各一致数にそれぞれ該当する単語数

5.1.4　結果に関する考察

ここまでの結果より，今回の訓練データのテキスト中に含まれる TF-IDF 値が大きい単語に対する類似単語は，東北大版 BERT とストックマーク版 BERT で 32.8 %程重なっていることが分かった．ここでの重なった類似単語の知識は，東北大版 BERT とストックマーク版 BERT の両方に含まれると考えられる．一方でそれ以外の重なっていない類似単語の知識は，それぞれの BERT に固有の知識であると考えることができる．本研究での提案手法は，この類似単語に含まれる片方の BERT に固有の知識が訓練データの知識とタスクを解くモデルに用いたもう片方の BERT の知識に加わったことで，効果を発揮したと考えられる．

今回は，提案手法を行うために東北大版 BERT とストックマーク版 BERT を利用した．しかし，これらの BERT 以外にも様々な BERT が存在する．そこで，BERT 同士の知識があまり重ならないような 2 種類の BERT のペアを選択して，今回と同じように Data Augmentation を行えば，今回以上の精度向上が期待できる．また，Data augmentation に使う BERT の種類を 1 種類ではなく多種類に増やすことによっても，精度の向上が期待できる．

5.2　追加実験 2（訓練データの量の影響）

5.2.1　実験の目的

提案手法のような簡易な Data Augmentation は，元の訓練データが少ない場合には効果があるが，データ量が十分である場合には効果がないと考えられている [5]．ここでは，この点を確認するために，4 章で行った実験を訓練データのサイズを 2 倍にして行うこととする．

5.2.2　実験設定

この実験で使用する各種データは以下の通りである．

訓練データ (baseline)　データ数：540（各ラベル 60 個ずつ）

拡張訓練データ (ストックマーク)　データ数：740（baseline+ 拡張文 200 個）

検証データ　データ数:270 個（4 章の実験の検証データと同じもの）

テストデータ　データ数:270 個（4 章の実験のテストデータと同じもの）

上記の baseline の訓練データの内容は 4 章の実験における baseline の訓練データと完全に別のものとしている．また，baseline の訓練データに対して，3 章に記した Data Augmentation の手法を適用して拡張訓練データ (ストックマーク) を作成している．

機械学習モデルの構築には，4 章で説明したニューラルネットワークに東北大版 BERT を利用した文書分類プログラムを用いる．学習時の設定・評価用モデルの選出方法については 4 章の実験と同様である．

5.2.3　実験結果

ここからは上記の実験を行った結果について示す．タスクを解くモデルに東北大版 BERT を利用した学習プログラムを利用して，baseline の訓練データと拡張訓練データ (ストックマーク) のそれぞれに対応する文書分類モデルを構築した．その後，構築した 2 つのモデルを使ってテストデータの識別を行い，それぞれについて正解率を出した．このときの結果を表 5.2 及び図 5.2 に示す．表 5.2 から分かるように，正解率は baseline の訓練データと拡張訓練データ (ストックマーク) で変わらないという結果になった．

表 5.2: 学習に東北大版 BERT を用いたモデルの正解率
(訓練データ 2 倍)

訓練データ	正解率
baseline	0.8888
拡張訓練データ (ストックマーク)	0.8888

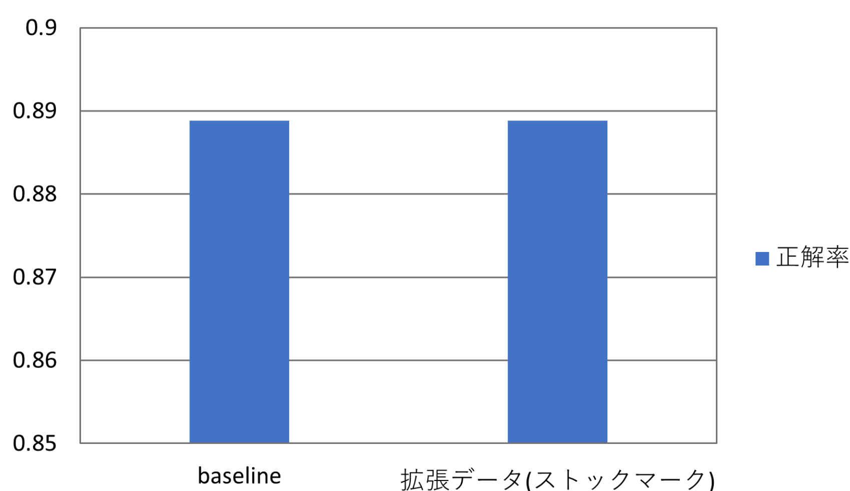

図 5.2: 学習に東北大版 BERT を用いたモデルの正解率
(訓練データ 2 倍)

5.2.4　結果に関する考察

この結果から，本論文で提案した Data Augmentation の手法は，訓練データのサイズが小さい場合には効果を発揮するが，サイズが大きくなると効果がなくなることを確認できた．つまりこの手法は，適用するタスクにおいて訓練データの数が十分ではない場合には適しているが，十分な数の訓練データがあるときには実用性は期待できない．

今後はデータの量が中規模であっても効果が出るような Data Augmentation の手法を考案したい．そこでは複数の BERT を利用するというアイデアが使えると考えている．

第 6 章

結論

自然言語処理分野の Data Augmentation は，画像処理と比べて効果が表れにくいが，文中の単語を類似単語に置き換える手法は，訓練データが少ない場合の文書分類タスクにおいて有効である．しかし，BERT モデルを用いて類似単語を取得する場合，類似単語の知識は既にモデルに含まれているため，この手法による文書分類モデルの性能向上は期待できない．

本論文では，2 つの BERT モデルを用いた Data Augmentation の手法を提案した．提案手法では，タスクを解くモデルに用いる BERT モデルとは異なる BERT モデルの MLM を用いて名詞を類似名詞に置き換えることにより，訓練データを拡張する．その結果，2 つの BERT モデルのうち 1 つの BERT モデルに特有の類似単語の知識が，訓練データとタスクを解くモデルに用いるもう 1 つの BERT モデルの知識に追加されるため，提案手法が効果を発揮する．

4 章の実験では，livedoor ニュースコーパスを用いた文書分類タスクに東北大版 BERT とストックマーク版 BERT を用いた提案手法を適用した．その結果，提案手法の有効性を示すことができた．今後は複数の BERT モデルを用いた別の Data Augmentation の手法を考案したい．

謝辞

本研究を進めるにあたり，多くのご指導を頂いた指導教官の新納浩幸教授に感謝申し上げます．また，研究室での活動を通じて，多くの知識や示唆を頂いた新納研究室の皆さまに感謝致します．

参考文献

[1] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, Vol. 6, No. 1, pp. 1–48, 2019.

[2] Steven Y Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. A Survey of Data Augmentation Approaches for NLP. *arXiv preprint arXiv:2105.03075*, 2021.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186. Association for Computational Linguistics, June 2019.

[4] H Zhang, M Cisse, Y Dauphin, and D Lopez-Paz. mixup: Beyond empirical risk minimization. iclr 2018. *arXiv preprint arXiv:1710.09412*, 2017.

[5] Jason Wei and Kai Zou. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196*, 2019.

[6] Mengzhou Xia, Xiang Kong, Antonios Anastasopoulos, and Graham Neubig. Generalized data augmentation for low-resource translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 5786–5796, Florence, Italy, July 2019. Association for Computational Linguistics.

[7] Jiaao Chen, Yuwei Wu, and Diyi Yang. Semi-supervised models via data augmentationfor classifying interactive affective responses. *arXiv preprint arXiv:2004.10972*, 2020.

[8] Hongyu Guo, Yongyi Mao, and Richong Zhang. Augmenting data with mixup for sentence classification: An empirical study. *arXiv preprint arXiv:1905.08941*, 2019.

[9] Jiaao Chen, Zichao Yang, and Diyi Yang. MixText: Linguistically-informed interpolation of hidden space for semi-supervised text classification. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2147–2157, Online, July 2020. Association for Computational Linguistics.

[10] Varun Kumar, Ashutosh Choudhary, and Eunah Cho. Data augmentation using pre-trained transformer models. In *Proceedings of the 2nd Workshop on Life-long Learning for Spoken Language Systems*, pp. 18–26, Suzhou, China, December 2020. Association for Computational Linguistics.

付録

A プログラムリスト

本研究で使用した主要なプログラムは以下の通りである.

A.1 baseline の訓練データから拡張訓練データを作成するプログラム

A.2 各訓練データを使って，文書分類モデルの学習を行うプログラム

A.3 テストデータを使って，作成した文書分類モデルの評価を行うプログラム

ソースコード A.1: mk-aug-train.py

```
1  import pickle
2  import re
3  import numpy as np
4  import copy
5  import itertools
6  import MeCab
7  import torch
8
9  from transformers import BertJapaneseTokenizer,BertForMaskedLM,
       BertConfig
10 from transformers.models.bert_japanese import
       tokenization_bert_japanese
11 from sklearn.feature_extraction.text import TfidfVectorizer
12
13 # データ拡張クラス
14 class DataAugmentater:
15
16     def __init__(self,tknz,MaskedLM,vocab,mecab,N):
17
18         self.tknz = tknz
```

```
19            self.MaskedLM = MaskedLM
20            self.vocab = vocab
21            self.mecab = mecab
22            self.N = N # 生成する文の数
23
24        # 1行目がTF-IDF値，2行目が文番号，3行目が単語の2次元配列を作る
25        def make_tfidf_table(self,texts,labels):
26
27            #textsの tfidfを計算する
28            vectorizer = TfidfVectorizer(tokenizer=self.tknz.tokenize)
29            tfidf = vectorizer.fit_transform(texts) #textsを tf-
                  idf行列に変換
30            feature_name = vectorizer.get_feature_names() #vocabの単語リスト
                  (辞書順)
31
32            #tfidf・文番号・単語リストを作成する
33            result = [] #tfidfのリスト
34            sent_n = [] #文番号のリスト
35            word_list = [] #単語のリスト
36            result = list(itertools.chain.from_iterable(tfidf.toarray()))
37            for i in range(len(texts)):
38                for j in range(len(feature_name)):
39                    sent_n.append(i)
40                    word_list.append(feature_name[j])
41
42            #numpy配列に変換
43            result = np.array(result)
44            sent_n = np.array(sent_n)
45            word_list = np.array(word_list)
46
47            #配列同士を結合し，tableを作成
48            table = np.stack([result, sent_n, word_list])
49
50            #tableを tfidf値で降順ソート
51            table_t = np.array(table).T
52            table_t = sorted(table_t, key=lambda x: x[0], reverse=True)
53            table = np.array(table_t).T
54
55            return table
56
57        # 単語が適切なものかチェックする(mask単語選択時＆予測単語選択時)
```

```
58     def check1(self,word):
59
60         #単語が名詞かどうか判定する
61         node = self.mecab.parseToNode(word)
62         hinsi = node.next.feature.split(",")[0]
63
64         if (hinsi != "名詞"): #単語が名詞じゃない
65             return False
66         elif not (word in self.vocab): #単語が vocab にない
67             return False
68         else:
69             return True
70
71     # 単語が適切なものかチェックする(予測単語選択時のみ)
72     def check2(self,j,maskword):
73
74         if j == maskword: #元の単語だった
75             return False
76         elif (j in maskword) or (maskword in j): #被ってる
               (例:オーダーとオーダーメイド)
77             return False
78         else:
79             return True
80
81     # 拡張文を生成する
82     def _generate(self,maskword,masksen):
83
84         # 選んだ単語がmask する単語として適しているかチェック
85         if self.check1(maskword) == False:
86             return False
87
88         # マスク単語をid に変換
89         maskid = self.tknz.encode(maskword)[1]
90
91         #マスク単語がある文を id 列に変換
92         maskids = self.tknz.encode(texts[masksen])
93         if len(maskids) > 512:
94             maskids = maskids[:512]
95
96         #id 列のマスク単語部分を mask する
97         for j in range(len(maskids)):
```

```
98            if maskids[j] == maskid:
99                maskids[j] = 4
100               break
101
102       #マスクした部分を予測する
103       if not (self.tknz.mask_token_id in maskids):
104           return False
105       mskpos = maskids.index(self.tknz.mask_token_id)
106       x = torch.LongTensor(maskids).unsqueeze(0)
107       a = self.MaskedLM(x)
108       b = torch.topk(a[0][0][mskpos],k=5)
109       ans = self.tknz.convert_ids_to_tokens(b[1])
110
111       #置き換える単語を選び，拡張文を作る
112       for replace_word in ans:
113           if self.check1(replace_word) and self.check2(replace_word,
                  maskword):
114               aug_text = texts[masksen].replace(maskword,
                      replace_word)
115               print("文",masksen, ":", self.tknz.decode(maskid),"→
                      ",replace_word)
116               return aug_text
117
118       return False
119
120   # 拡張文リストを生成する
121   def generate(self,texts,labels):
122
123       table = self.make_tfidf_table(texts,labels)
124
125       aug_texts = copy.deepcopy(texts) #拡張文のリスト
126       aug_labels = copy.deepcopy(labels) #拡張文のラベルのリスト
127
128       count = 0
129
130       for word,sen_n in zip(table[2],table[1]):
131
132           # word_id と sen_n を float 型から int 型に変換
133           sen_n = int(sen_n)
134
135           # 拡張文を生成する
```

```
136             aug_text = self._generate(word,sen_n)
137             if aug_text != False:
138                 aug_texts.append(aug_text)
139                 aug_labels.append(labels[sen_n])
140                 count += 1
141                 #print("\r",count,"/",self.N,end="")
142 
143             # 拡張文をN 個作ったらループを抜ける
144             if count >= self.N:
145                 break
146 
147         return aug_texts,aug_labels
148 
149 # train.tsv を読み込み，ラベルのリスト labels と文のリスト texts を作る
150 texts = [] # 文のリスト
151 labels = [] # ラベルのリスト
152 
153 with open('train.tsv','r',encoding='utf-8') as f:
154     for line in f: #train.tsv を読み込み，1 行ずつ line に代入
155         line = line.rstrip() #rstrip()で line の中のスペースを除去
156         result = re.match('^(\d+)\t(.+?)$', line) #
            line が正規表現パターンにマッチするかチェック
157         labels.append(int(result.group(1))) #正規表現パターンのグループ
            1(ラベル)をリストydata に追加
158         sen = result.group(2) #正規表現パターンのグループ
            2(文章)をsen に代入
159         texts.append(sen)
160 
161 # 東北大版BERT を使ってデータ拡張する
162 tknz = BertJapaneseTokenizer.from_pretrained('cl-tohoku/bert-base-
        japanese')
163 
164 MaskedLM = BertForMaskedLM.from_pretrained('cl-tohoku/bert-base-
        japanese')
165 
166 tohoku_vocab = []
167 with open('東北大版 BERT/vocab.txt','r',encoding='utf-8') as f:
168     for line in f: #train.tsv を読み込み，1 行ずつ line に代入
169         line = line.rstrip() #rstrip()で line の中のスペースを除去
170         tohoku_vocab.append(line)
171 
172 data_augmentater = DataAugmentater(tknz, MaskedLM, tohoku_vocab, MeCab
```

```
    .Tagger("-Ochasen"),100)
173 aug_texts, aug_labels = data_augmentater.generate(texts, labels)
174
175 with open('tohoku_train.tsv','w',encoding='utf-8') as f:
176     for label,text in zip(aug_labels,aug_texts):
177         data = str(label) + "\t" + text
178         f.write(data)
179         f.write('\n')
180
181 # ストックマーク版BERTを使ってデータ拡張する
182 tknz = BertJapaneseTokenizer(vocab_file='ストックマーク版
    BERT/vocab.txt',
183                             do_lower_case=False,do_basic_tokenize=
                                False)
184 tknz.word_tokenizer = tokenization_bert_japanese.MecabTokenizer()
185
186 config = BertConfig.from_json_file('ストックマーク版
    BERT/bert_config.json')
187 MaskedLM = BertForMaskedLM.from_pretrained('ストックマーク版
    BERT/pytorch_model.bin',config=config)
188
189 stockmarks_vocab = []
190 with open('ストックマーク版BERT/vocab.txt','r',encoding='utf-8') as f:
191     for line in f: #train.tsvを読み込み，1行ずつlineに代入
192         line = line.rstrip() #rstrip()でlineの中のスペースを除去
193         stockmarks_vocab.append(line)
194
195 data_augmentater = DataAugmentater(tknz, MaskedLM, stockmarks_vocab,
    MeCab.Tagger("-Ochasen"),100)
196 aug_texts, aug_labels = data_augmentater.generate(texts, labels)
197
198 with open('stock_train.tsv','w',encoding='utf-8') as f:
199     for label,text in zip(aug_labels,aug_texts):
200         data = str(label) + "\t" + text
201         f.write(data)
202         f.write('\n')
```

ソースコード A.2: train.py

```
1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3
4 import torch
```

```
5  import torch.nn as nn
6  import torch.optim as optim
7  import torch.nn.functional as F
8  from torch.utils.data import Dataset, DataLoader
9  from torch.nn.utils.rnn import pad_sequence
10 from transformers import BertModel,BertConfig
11 import numpy as np
12 import pickle
13 import sys
14
15 class EarlyStopping:
16     """earlystopping クラス"""
17
18     def __init__(self, patience, verbose, path):
19         """引数：最小値の非更新数カウンタ、表示設定、モデル格納 path"""
20
21         self.patience = patience #設定ストップカウンタ
22         self.verbose = verbose #表示の有無
23         self.counter = 0 #現在のカウンタ値
24         self.best_score = None #ベストスコア
25         self.early_stop = False #ストップフラグ
26         self.val_acc_max = 0 #前回のベストスコア記憶用
27         self.path = path #ベストモデル格納 path
28
29     def __call__(self, val_acc, model):
30         """
31         特殊 (call)メソッド
32         実際に学習ループ内で最小 loss を更新したか否かを計算させる部分
33         """
34         score = val_acc
35
36         if self.best_score is None: #1Epoch 目の処理
37             self.best_score = score #1
                   Epoch 目はそのままベストスコアとして記録する
38             self.checkpoint(val_acc, model) #記録後にモデルを保存してス
                   コア表示する
39         elif score < self.best_score: # ベストスコアを更新できなかった場合
40             self.counter += 1 #ストップカウンタを +1
41             if self.verbose: #表示を有効にした場合は経過を表示
42                 print(f'EarlyStopping␣counter:␣{self.counter}␣out␣of␣{
                       self.patience}') #現在のカウンタを表示
```

```
                      する
43              if self.counter >= self.patience: #設定カウントを上回ったら
                    ストップフラグを True に変更
44                  self.early_stop = True
45          else: #ベストスコアを更新した場合
46              self.best_score = score #ベストスコアを上書き
47              self.checkpoint(val_acc, model) #モデルを保存してスコア表示
48              self.counter = 0 #ストップカウンタリセット
49
50      def checkpoint(self, val_acc, model):
51          '''ベストスコア更新時に実行されるチェックポイント関数'''
52          if self.verbose: #表示を有効にした場合は、前回のベストスコアから
                どれだけ更新したか？を表示
53              print(f'Validation accuracy increased ({self.val_acc_max:.6
                    f} --> {val_acc:.6f}).  Saving model ...')
54          torch.save(model.state_dict(), self.path) #ベストモデルを指定し
                た path に保存
55          self.val_acc_max = val_acc #その時の loss を記録する
56
57  #DataLoader
58  class MyDataset(Dataset):
59      def __init__(self, xdata, ydata):
60          self.data = xdata
61          self.label = ydata
62      def __len__(self): #label の長さを返す
63          return len(self.label)
64      def __getitem__(self, idx): #data と label の idx 番目の要素を返す
65          x = self.data[idx]
66          y = self.label[idx]
67          return (x,y)
68
69  #バッチの形をデフォルトから変更
70  def my_collate_fn(batch):
71      images, targets= list(zip(*batch))
72      xs = list(images)
73      ys = list(targets)
74      return xs, ys
75
76  #学習用のモデルの定義
77  class DocCls(nn.Module):
78      def __init__(self,bert):
79          super(DocCls, self).__init__()
```

```
80            self.bert = bert
81            self.cls=nn.Linear(768,9)
82            nn.init.xavier_normal_(self.cls.weight)
83        def forward(self,x1,x2):
84            #print(x1.size(),x2.size())
85            #attention_mask →モデルが注意を払うべきトークンの判別に利用
86            #1が注意を払うべきトークン、0が埋め込み
87            bout = self.bert(input_ids=x1, attention_mask=x2)
88            bs = len(bout[0])
89            h0 = [ bout[0][i][0] for i in range(bs)]
90            h0 = torch.stack(h0,dim=0)
91            output = self.cls(h0)
92            return output
93
94    #検証用のモデルの定義
95    class DocCls2(nn.Module):
96        def __init__(self,bert):
97            super(DocCls2, self).__init__()
98            self.bert = bert
99            self.cls=nn.Linear(768,9)
100       def forward(self,x):
101           bout = self.bert(x)
102           bs = len(bout[0])
103           h0 = [ bout[0][i][0] for i in range(bs)]
104           h0 = torch.stack(h0,dim=0)
105           return self.cls(h0)
106
107   #学習
108   def train_net(dataloader, net, optimizer, criterion, device):
109
110       i, lossK = 0, 0.0
111
112       net.train()
113       for xs, ys in dataloader: #
              xs は要素が単語 id 列のリスト, ys はラベルのリスト(要素数=バッチサイズ)
114           xs1, xmsk = [], [] #要素が
                  tensor のリスト→[tensor([]),tensor([]),...tensor([])]
115           for k in range(len(xs)):
116               tid = xs[k]
117               xs1.append(torch.LongTensor(tid))
118               xmsk.append(torch.LongTensor([1] * len(tid)))
```

```
119
120         #pad_sequece:長さの違うテンソルを与えると，短いものの末尾にゼロ埋めを
                施して次元を揃えてくれる関数
121         #例...xs1[0]の長さ 480,xs1[1]の長さ 403だったら,両方長さ 480に揃えてく
                れる
122         #[tensor([]),tensor([]),...tensor([])]をtensor([[],[],...[]])に
                直してくれる
123         xs1 = pad_sequence(
124             xs1,
125             batch_first=True #(seq_len, batch, input_size)→ (batch,
                    seq_len, input_size)
126             ).to(device)
127         xmsk = pad_sequence(
128             xmsk,
129             batch_first=True
130             ).to(device)
131         ys = torch.LongTensor(ys).to(device)
132
133         outputs = net(xs1,xmsk)
134
135         loss = criterion(outputs, ys)
136         lossK += loss.item()
137
138         optimizer.zero_grad()
139         loss.backward()
140         optimizer.step()
141         i += 1
142
143     return net
144
145 #検証
146 def val_net(xval, yval, net, device):#, earlystopping):
147
148     real_data_num, ok = 0, 0 #テストデータの数，正解数
149
150     #テスト時は以下の 2行を追加（微分値の計算を行わないようにする）
151     net.eval()
152     with torch.no_grad():
153         for i in range(len(xval)):
154             x = torch.LongTensor(xval[i]).unsqueeze(0).to(device)
155             ans = net(x)
```

```
156             #ansの行の，最大要素のインデックスを返す
157             #item()で、
                    Pythonの組み込み型（この場合は整数 int）として要素の値を取得できる

158             ans1 = torch.argmax(ans,dim=1).item()
159
160             if (ans1 == yval[i]): #予測結果と正解ラベルが一致したら
161                 ok += 1
162             real_data_num += 1
163
164     return ok/real_data_num
165
166 # 以下main
167
168 #torch.manual_seed(1) #ネットワーク重みの初期値を固定
169
170 #コマンドラインから引数を受け取る
171 argvs = sys.argv
172 argc = len(argvs)
173
174 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu"
        )
175
176 with open(argvs[1],'br') as fr:
177     xtrain = pickle.load(fr)
178 with open(argvs[2],'br') as fr:
179     ytrain = pickle.load(fr)
180
181 with open('xval.pkl','br') as fr:
182     xval = pickle.load(fr)
183 with open('yval.pkl','br') as fr:
184     yval = pickle.load(fr)
185
186 #dataloaderを作成
187 batch_size = 2
188 train_dataset = MyDataset(xtrain,ytrain) #MyDatasetのインスタンスを生成
189 train_dataloader = DataLoader(
190     train_dataset,
191     batch_size=batch_size, #バッチサイズ (一度に取り出すデータの数)
192     shuffle=True, #データの順序がシャッフルされる
193     collate_fn=my_collate_fn #自作の collate_fn を使用
```

```
194     )
195
196 #学習済みの日本語 BERT モデルを読み込む
197 #東北大版
198 #config = BertConfig.from_json_file('東北大版BERT/config.json')
199 #bert = BertModel.from_pretrained('東北大版BERT/pytorch_model.bin',
        config=config)
200 #bert = BertModel.from_pretrained('cl-tohoku/bert-base-japanese')
201
202 #ストックマーク版
203 config = BertConfig.from_json_file('ストックマーク版
        BERT/bert_config.json')
204 bert = BertModel.from_pretrained('ストックマーク版
        BERT/pytorch_model.bin',config=config)
205
206 # model generate, optimizer and criterion setting
207 net = DocCls(bert).to(device)
208 net2 = DocCls2(bert).to(device)
209
210 optimizer = optim.SGD(net.parameters(),lr=0.001)
211 criterion = nn.CrossEntropyLoss()
212 epochs = 30
213
214 earlystopping = EarlyStopping(patience=5, verbose=True, path=argvs[3])
215
216 flag = 0
217 for ep in range(epochs):
218     #学習
219     net = train_net(
220         dataloader=train_dataloader,
221         net=net,
222         optimizer=optimizer,
223         criterion=criterion,
224         device = device
225         )
226     print("{}␣epoch␣finished".format(ep))
227
228     #学習したモデルを検証用モデルに読み込む
229     outfile = "tmp.model"
230     torch.save(net.state_dict(),outfile)
231     net2.load_state_dict(torch.load("tmp.model"))
232
```

```
233     #検証
234     accuracy = val_net(
235         xval=xval,
236         yval=yval,
237         net=net2,
238         device = device
239         #earlystopping=earlystopping
240     )
241
242     #early stopping
243     earlystopping(accuracy, net) #call メソッド呼び出し
244     if earlystopping.early_stop: #ストップフラグが
            True の場合、break で for ループを抜ける
245         print("Early␣Stopping!")
246         break
```

ソースコード A.3: test.py

```
1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3
4 import torch
5 import torch.nn as nn
6 import torch.optim as optim
7 import torch.nn.functional as F
8 from torch.utils.data import Dataset, DataLoader
9 from torch.nn.utils.rnn import pad_sequence
10 from transformers import BertModel, BertConfig
11 import numpy as np
12 import pickle
13 import sys
14
15 #コマンドラインから引数(モデル)を受け取る
16 argvs = sys.argv
17 argc = len(argvs)
18
19 #学習済みの日本語 BERT モデルを読み込む
20 #東北大版
21 #config = BertConfig.from_pretrained('cl-tohoku/bert-base-japanese')
22 #bert = BertModel(config=config)
23 #ストックマーク版
24 config = BertConfig.from_json_file('ストックマーク版
```

```
    BERT/bert_config.json')
25 bert = BertModel.from_pretrained('ストックマーク版
    BERT/pytorch_model.bin',config=config)
26
27 #GPUを利用する
28 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu"
    )
29
30 # Data Setting
31 with open('xtest.pkl','br') as fr:
32     xtest = pickle.load(fr)
33 with open('ytest.pkl','br') as fr:
34     ytest = pickle.load(fr)
35
36 # Define model
37 class DocCls(nn.Module):
38     def __init__(self,bert):
39         super(DocCls, self).__init__()
40         self.bert = bert
41         self.cls=nn.Linear(768,9)
42     def forward(self,x):
43         bout = self.bert(x)
44         bs = len(bout[0])
45         h0 = [ bout[0][i][0] for i in range(bs)]
46         h0 = torch.stack(h0,dim=0)
47         return self.cls(h0)
48
49 # model generate
50 net = DocCls(bert).to(device)
51 #保存したモデルを呼び出して使う
52 net.load_state_dict(torch.load(argvs[1]))
53
54 # Test
55
56 real_data_num, ok = 0, 0 #テストデータの数，正解数
57
58 #テスト時は以下の2行を追加（微分値の計算を行わないようにする）
59 net.eval()
60 with torch.no_grad():
61     for i in range(len(xtest)):
62         x = torch.LongTensor(xtest[i]).unsqueeze(0).to(device)
```

```
63         ans = net(x)
64         #print(ans.size())
65         #ans の行の，最大要素のインデックスを返す
66         #item()で、
               Python 組み込み型（この場合は整数 int）として要素の値を取得できる
67         ans1 = torch.argmax(ans,dim=1).item()
68
69         if (ans1 == ytest[i]): #予測結果と正解ラベルが一致したら
70             ok += 1
71         real_data_num += 1
```