令和 6 年度茨城大学大学院理工学研究科情報工学専攻

修士学位論文

入力単語の制限による日本語 CommonGen タスクの拡張

所属　情報工学専攻

著者　鈴木　雅人 (23NM726L)

指導教員　新納浩幸教授

令和 7 年 2 月 3 日（月）

令和 6 年度茨城大学大学院理工学研究科情報工学専攻　修士学位論文

入力単語の制限による日本語 CommonGen タスクの拡張

著者

鈴木　雅人 (23NM726L)

指導教員

新納浩幸教授

論文要旨

自然言語処理には，常識推論と呼ばれる分野が存在する．常識推論とは，言語モデルが推論を行う際に我々が持つ常識のような膨大な背景知識に基づいた出力を行うことである．

言語モデルの常識推論能力を測るタスクとして CommonGen と呼ばれる，いくつかの単語を入力として，それらを含む常識的，自然な文を出力が存在する．

しかし，高性能な大規模言語モデル（LLM）の出現により、言語モデルによる文生成の能力は大きく向上した．それに伴い、高性能な LLM の出現以前に利用されていた文生成能力を測るタスクは現在の LLM の能力を測る指標としては適切なものでなくなってしまったものが多い．そして，CommonGen タスクもその一つである．

本研究では、文生成タスクの CommonGen における入力単語を特定の分野に絞った専門用語にすることで、さらに高い文生成能力を求められるタスクにする試みを行った．更に試験的に作成したデータセットを用いて、LLM による生成文の評価と人手評価を行い、LLM の文生成能力と自動評価の可能性について調査した．

Master's Thesis in Scholastic 2024, Major in Computer and Information Sciences,
Granduate School of Science and Engineering, Ibaraki University

Extending Japanese CommonGen Tasks by Restricting Input Words

Author : Masato Suzuki (23NM726L)

Adviser : Prof. Hiroyuki Shinnou

Abstract

In natural language processing, there is a field called common sense reasoning. Common sense reasoning means that when a language model performs inference output based on a large amount of background knowledge, such as our own common sense.

As a task to measure a language model's common sense reasoning ability, we use CommonGen, which takes a few words as input and produces a set of common sense, natural sentences containing those words. CommonGen takes a few words as input and outputs common sense, natural sentences containing those words.

However, with the advent of high-performance large-scale language models (LLMs), the ability of language models to generate sentences has greatly improved. As a result, many of the tasks used to measure sentence generation ability before the advent of high-performance LLMs are no longer appropriate as a measure of current LLM ability. The CommonGen task is one such example.

In this study, we attempted to make the CommonGen sentence generation task a task that requires an even higher level of sentence generation ability by using specialized terms focused on specific fields as input words in the CommonGen task. Furthermore, we investigated the sentence generation ability of LLM and the possibility of automatic evaluation by evaluating the generated sentences by LLM and manually evaluating them using a test dataset.

目次

第 1 章　序論　5

第 2 章　関連研究　7
- 2.1　常識推論を取り扱うタスク 7
- 2.2　文生成の手法 10
- 2.3　CommonGen 18

第 3 章　CommonGen タスクの妥当性の検証　21

第 4 章　実験　25
- 4.1　テストデータの作成 25
- 4.2　LLM による文生成 25
- 4.3　人手による評価 26
- 4.4　ChatGPT-4o による評価 26

第 5 章　考察　29
- 5.1　専門用語に限定する手法の効果の考察 29
- 5.2　人手と機械による評価結果の考察 30

第 6 章　結論　32

参考文献　34

付録　36
- A　プログラム 36

第 1 章

序論

自然言語処理には常識推論と呼ばれる問題があり，我々人間が持つ常識と呼ばれるような巨大な背景知識に矛盾しないように推論できるかという重要な問題である．それは人間が持っている常識は膨大な量であり，それをどのように記述し，そこからどのように推論を行えばよいかが現在であっても未解決だからである．このような難問に取り組むには，適切なマイルストーンとなるタスクを考案することが重要である．考案されたタスクの解決法を考えることで，複雑な問題の構造が解明されてゆき，その解決法も改善されていくからである．そのような常識推論能力を測るデータセットはいくつも考案されている．Alon らの公開した CommonSenseQA [1]，Maarten らの SocialIQA [2]，Keisuke らの WinoGrande [3]，Kazumasa らの KUCI [4]，Rowan らの SWAG [5]，Rowan らの HellaSWAG [6]，などがある．しかしこれらはいずれも選択式の常識推論タスクである．

そのなかで Lin らは CommonGen という新しい常識推論のタスクを提案した [7]. CommonGen は，数個のキーワードを入力し，それらキーワードを用いて妥当な文を生成するという制約付き文生成のタスクである．

例えばキーワードが「彼，犬，餌」なら「彼が犬に餌をあげる」などといった文を生成するのが目標である．「犬が彼に餌をあげる」は文法的には正しくても常識的にはおかしな文であり，そのような文を生成しないために常識推論が必要とされている．

しかし，CommonGen タスクを始めとする様々なタスクは，高性能な LLM の出現により，人工知能において解決可能なタスクとなったものも少なくない．

本研究では，まず ChatGPT と T5 を利用したベースラインシステムで同一のテストデータにて文を生成し，現在の LLM に対する日本語 CommonGen タスクの妥当性を検証した．

その結果をもとに現在の LLM に見合った難易度のタスクに拡張するために，既存の CommonGen タスクに入力単語の制限を加える手法を提案し，拡張法の効果について検証と考察を行った.

第2章

関連研究

2.1 常識推論を取り扱うタスク

2.1.1 CommonsenseQA

CommonsenseQA [1] は，一般的な前提知識を必要とする質問応答型タスクである．データセットは質問文に対して選択肢となる5つの単語から質問文に該当する単語を選ぶ形となっている．

質問文を作る際に，元となる単語が決められており，その単語が質問文に含まれる．そして選択肢となる単語の5つの内4つはConceptNetにより，元となる単語と関係がある単語が抽出されている．

この作成方法により，選択肢は一定の関係を持つ単語となっており，質問文のもつ情報を常識的な知識として持つ単語を選ぶ必要がある．例として論文内では，”Where on a river can you hold a cup upright to catch water on a sunny day?”という質問文と”waterfall, bridge, valley, pebble, mountain”の5つの単語の選択肢が示されている．この質問文では滝が川の上にあり，水が落ちてくるという常識的背景を捉えなければ正解することができないようになっている．

2.1.2 Social IQA

Social IQA [2] は社会的に一般な状況を提示し，質問と3つの選択肢から，適切な回答を選ぶタスクである．

例として，”Tracy had accidentally pressed upon Austin in the small elevator and it was awkward.”という状況が示され，”Why did Tracy do this?”という質問が提示され，”(a) get very close to Austin”，”(b) squeeze into the elevator”，”(c) get flirty with Austin”の 3 つの選択肢が示され，a, b, c のいずれかで答えるものが示されている．

この例では，質問の”this”が何を指すのか，原因と結果の関係を理解できているかといったことを試しており，正解は b のエレベーターに乗ったためとなっている．

Social IQA では，不正解の選択肢として質問と類似の質問において正解となるような選択肢を用意するようクラウドワーカーに指示しており，この構造により不正解の回答の不自然さを減らしている．

2.1.3　WinoGrande

WinoGrande [3] は 2 つの文章とその文章中にある代名詞に対して 2 つの単語の選択肢が与えられ，各文章に代名詞の指す正しい単語を選択するタスクである．

例としては，”The trophy doesn’t fit into the brown suitcase because it’s too large.”，”The trophy doesn’t fit into the brown suitcase because it’s too small.”の二つの文が示され，それぞれの文に出現する”it”に ”trophy / suitcase” のどちらが入るかを選択するものが示されている．

この例では，何かに物を入れる際，入れ物と入れるもの大小関係の常識が試されており，初めの文が”trophy”, 後の文が”suitcase”となるのが正解となっている．

データセットを作成するにあたって，注意深く設計されたクラウドソージングの手続きと AfLite アルゴリズムと呼ばれる人間が認知できる単語の関連性を機械が認知できる埋め込み表現に取り入れる仕組みを用いて統計的なバイアスを減らす取り組みが重要なステップだと位置づけた．

人間によるテスト結果と言語モデルによるテスト結果を比較し，データセットの有用性を示すとともに，バイアスによって我々が機械の常識推論能力を過大評価している可能性があるという懸念を示し，統計的なバイアスの軽減が重要であることを主張した．

2.1.4　KUCI

KUCI [4] は日本語において中断されている文章とそれに続く蓋然的な関係を持つ文章を 4 つの選択肢から 1 つ選択するタスクである．

例として，「電池の減りはやはり早いので、」という文と，「a. 実際の半導体製造装置

は実現しません」,「b. 今回は期間限定でのお届けになります」,「c. 原子炉を手動停止する」,「d. 充電用に USB ケーブル買います」の 4 つの選択肢があるという問題が示されている．この例では，電池の減りが早いならば，どうなるのが最も自然かを試しており，正解は d の選択肢となっている．

コーパスからの自動抽出とクラウドソージングを組み合わせることで拡張性があり，低バイアスで低コストな常識推論データセットの構築方法を提案した．人間が高い精度で解くことができるものの言語モデルではそれなりに低い精度となることを示していることやデータセットの解析によりバイアスが少ないことを確認している．

2.1.5　SWAG

SWAG [5] は「ある場面でのビデオキャプション文」と 4 つの「次の場面でのキャプション文」の選択肢となる文章を提示し，選択肢から本物の次の場面でのキャプション文を選択するタスクである．

例として ”On stage, a woman takes a seat at the piano. She”, という文と ”a) sits on a bench as her sister plays with the doll.” ”b) smiles with someone as the music plays.”, ”c) is in the crowd, watching the dancers.”, ”d) nervously sets her fingers on the keys.” の 4 つの選択肢が示されており，ステージ上でピアノの前に座った女性がとる常識的な行動は，どれに当たるのかを試していている．この例では，d の「緊張しながら，鍵盤の上に指を置く」が正解となっている．

既存の多くのデータセットに見られるアノテーションの不自然さや人間によるバイアスといった課題に対処すべく，文体分類器のアンサンブル学習を繰り返し行い，それらを用いてデータのフィルタリングを行う Adversarial Filtering を提案し，積極的な Adversarial Filtering の導入により，人間のテスト結果では高い正解率を誇るものの当時の言語モデルでは苦戦することを確認した．

2.1.6　HellaSwag

HellaSwag [6] は SWAG を元としてさらに難しい不正解の選択肢を導入する Adversarial Filtering の強化や元となるビデオキャプションの厳選などを行い，人間と機械とでテスト結果の差が縮まっていた SWAG を改善したものである．

人間にとって些細な変化が最先端のモデルではしばしば誤分類に繋がる重要な要素で

あることがあることを示しており，敵対的な手法によるデータセットの改善が有効であると主張した．

2.2　文生成の手法

2.2.1　単語 N-gram

単語 N-gram とは任意の単語数 N で文章を分割する手法であり，代表的なものとして 1 単語で分割する uni-gram，2 単語で分割する bi-gram，3 単語で分割する tri-gram などがある．

日本語では，単語は明確に分割されていないため，意味を持つ表現要素の最小単位の形態素で分割する．例として bi-gram で「今日は良い天気ですね」という文章を分割すると，”今日は”,”は良い”,”良い天気”,”天気です”,”ですね”というように分割できる (図 2.1)．

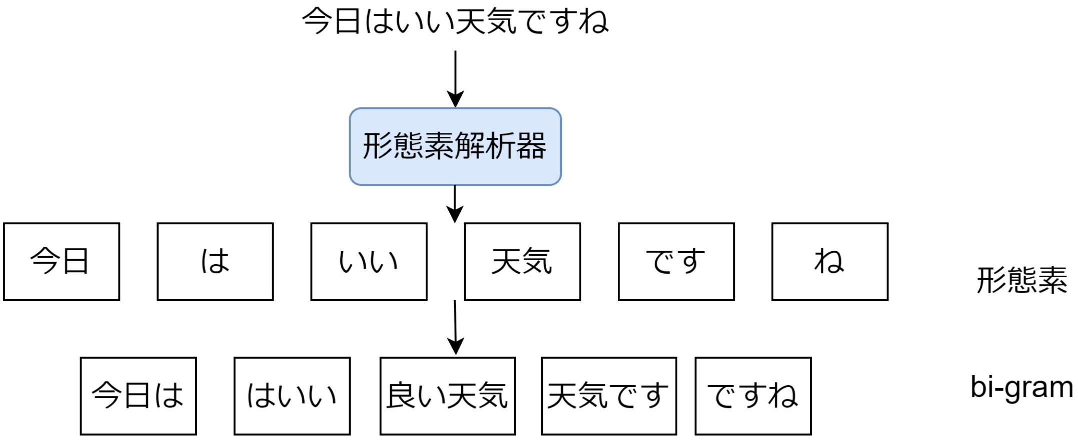

図 2.1: bi-gram の例

これを利用して一定以上の長さの文章で，単語の組み合わせの N-gram が出現する数を集計することで，ある単語の次に出現する単語の確率を出す手法が考えられ，計算機により次の単語を予測していくことで文章を生成するような言語モデルが考案された．

2.2.2　Neural Network

Neural Network(以下 NN) は，入力に対して多数の線形変換と非線形変換を行うことで線形分離不可能な問題も解くことができる数理モデルである．NN は多数のパーセプ

トロンで構成されており，各パーセプトロンはそれぞれ固有の重みを持つ．図 2.2 にパーセプトロンの例を示す.

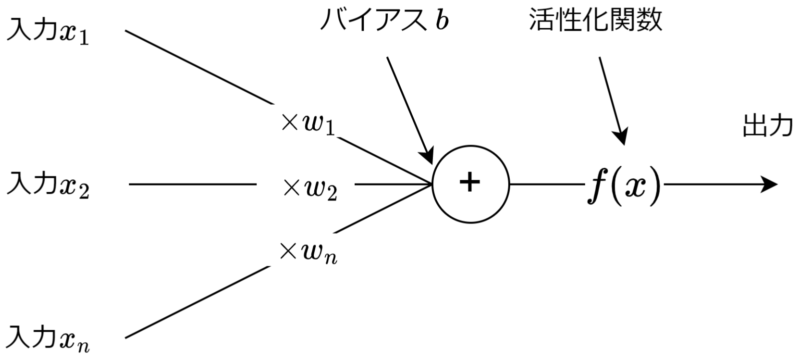

図 2.2: パーセプトロンの構造

全体としては入力層，中間層，出力層とよばれる 3 層で構成されており，入力層で入力となる何らかの値を受け取り，中間層に渡す.

中間層では各入力にそれぞれ固有の重みを積算し，バイアスと呼ばれる固定値を加算，得られた値を調整するため活性化関数にかけたものを出力として，次に接続されたパーセプトロンに入力する.

出力層では，中間層の出力を入力として，同様の操作で出力を得る．図 2.3 に中間層が 1 層の NN の例を示す.

このモデルでは，入力と正解となる出力の組である教師データを与え，モデルの出力と正解となる出力の誤差を損失関数により評価し，モデルの出力が正解に近づくように各パーセプトロンの重みを適切に調整することで様々なタスクの精度を向上させることができる.

2.2.3 Recurrent Neural Network

前述した NN では，時系列データと呼ばれる順番や時系列が重要なデータを扱えない問題があった．そこで Recurrent Neural Network (以下 RNN) では， 中間層を 1 層にして，時系列データの前の入力から得られた出力を, 次の入力とともに接続して再び同じ中間層に入力する再帰的な構造になっている.

これにより最終層では疑似的に全ての時系列を考慮した出力ができる．RNN では NN で扱うことができなかった文生成のタスクを扱うことができるようになった．図 2.4 に

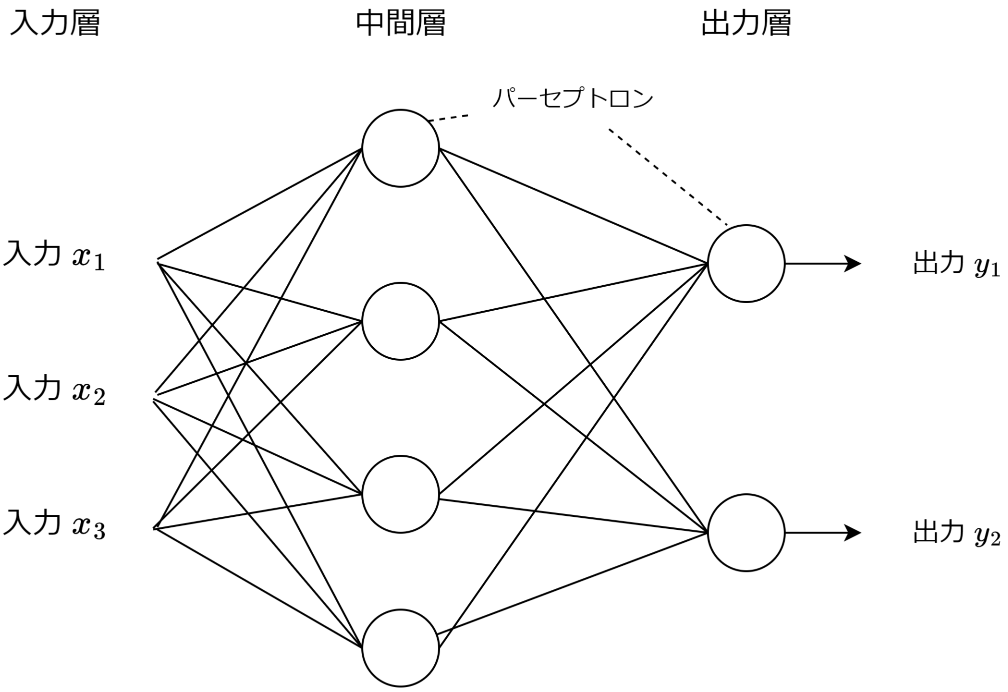

図 2.3: Neural Network の構造例

RNN 中間層を表した図を示す.

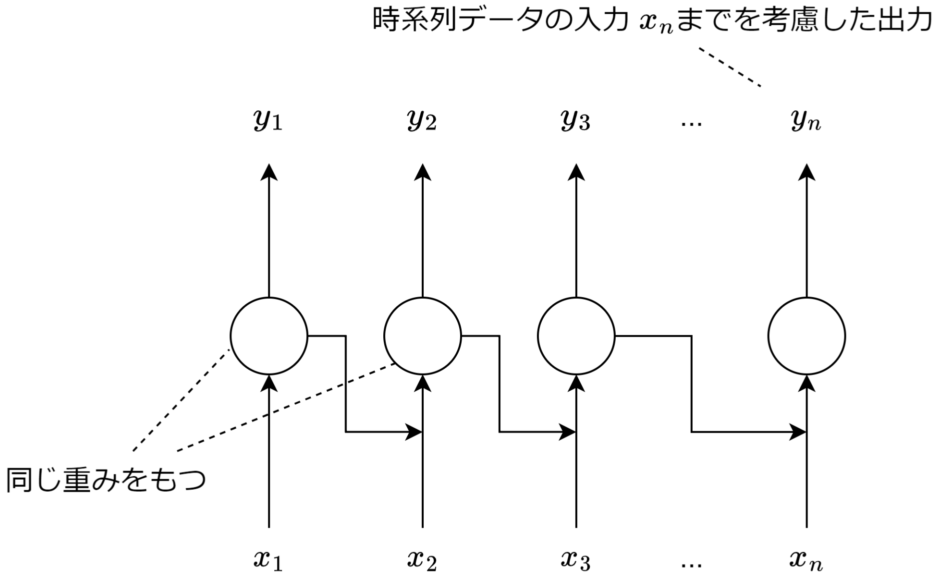

図 2.4: RNN の中間層

2.2.4　LSTM

前述の RNN では，シーケンシャルデータを扱えるようになったが，勾配消失問題を抱えていた．これは時系列データのサイズが大きくなってしまうと再帰回数が増え，学習を適切に進めるための勾配が非常に小さくなり，重みを更新できなくなってしまうという問題である．

これを改善するためにLSTM(Long Short Term Memory) が RNN の一種として考案された．LSTM では，LSTM block と呼ばれるメモリと忘却ゲートと入力ゲート，出力ゲートの三つのゲートを導入することにより RNN よりも長期の記憶を可能としている．そのため文生成のような長期の時系列データを扱うタスクにおいても，大きく性能を向上させた．

2.2.5　Transformer

Transformer [8] は 2017 年に発表された深層学習モデルであり, 翻訳タスクにおいて当時最良の BLUE スコアを達成した．当時主流だった RNN や CNN では並列計算が不可能だったが，それらを使用しておらず，Attention 機構のみを利用した Encoder-Decoder モデルである，そのため並列計算が可能で訓練にかかる時間が少ないという点でも優れていた．

Transformer で使用された Attention 機構は，ある入力トークンを他の全ての入力トークンとの関連性に基づいて重み付けを行うことができ，これにより，文脈などを考慮したベクトルを獲得することができる．Attention 機構のみを利用した構造が優れており，この後の多くの言語モデルにおいてその一部，または全体を参考にしたモデルが多く出現した．Transformer の構造を図 2.5 に示す．

Multi-Head Attention は，複数の Self-Attenion 機構を並べ, 表現力を高める構造である．Add & Norm は深層学習の典型的手法である Skip-connection と Layer Normalization を表している．Skip-connection は以前の層の出力を入力に加算する手法であり，Layer Normalization は正規化の手法である．Feedforward Network は 順方向のみのニューラルネットワークの一種である．Masked Multi-Head Attention は Multi-Head Attention に現在の入力よりも先のトークンを隠す仕組みを加えたものである．

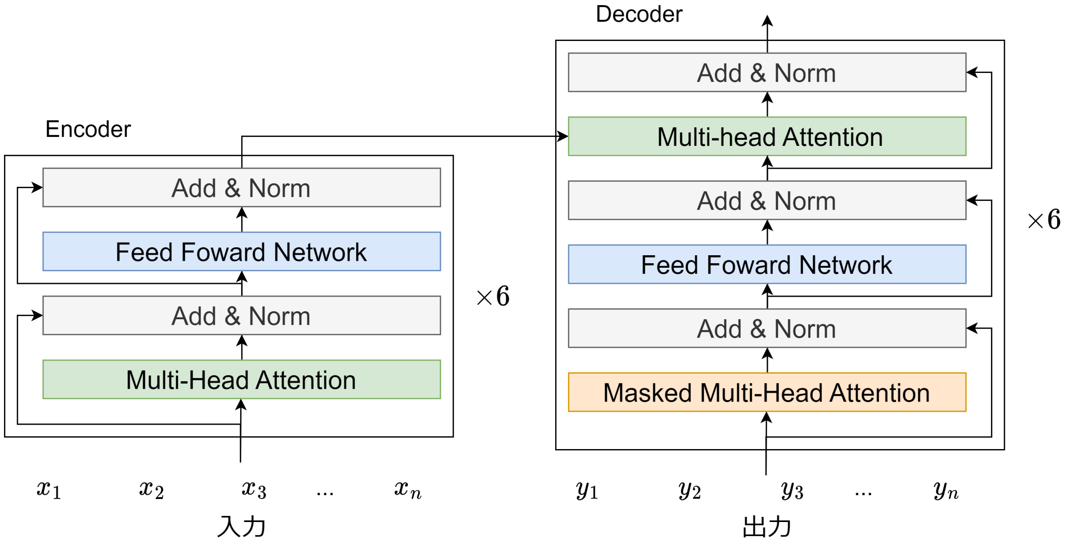

図 2.5: transformer の構造

2.2.6　BERT

BERT [9] は Bidirectional Encoder Representations from Transformers の略で 2018 年に Google から公開されたモデルである．テキストデータの一部のトークンを MASK する　(隠す) 手法により，事前学習と呼ばれる低コストな教師なし学習を大規模に行っている．

事前学習済みのモデルに対し，fine-tuning と呼ばれる，それぞれのタスクに転用するための比較的小規模の学習を行うだけで，複数のタスクにおいて，当時の State of the art を達成し，高い注目を集めた．

構造としては，Transformer の encoder 部分をベースとしたもの（図 2.7）を 12 層重ねた形で構成されている（図 2.6）出力のトークンから必要なトークンを選び，分類器といった各タスクに対応したものに接続することで，様々なタスクに対応でき，汎用的なモデルであるという特徴がある．

2.2.7　GPT2

GPT2 [10] は，Generative Pretrained Transformer 2 の略であり，2019 年に発表された事前学習モデルである．

GPT2 は，当時の他の事前学習モデルよりも大規模なデータセットにてより大きな言

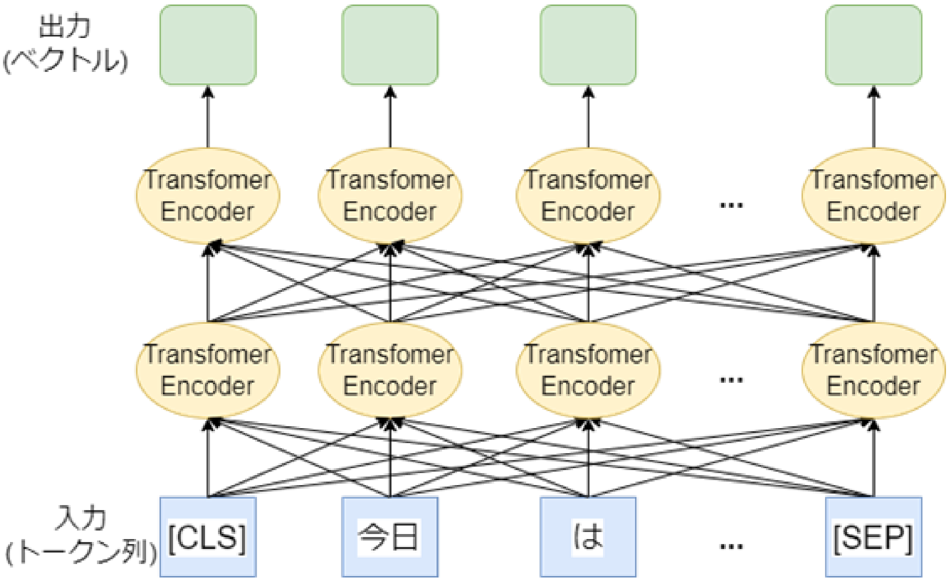

図 2.6: BERT の構造

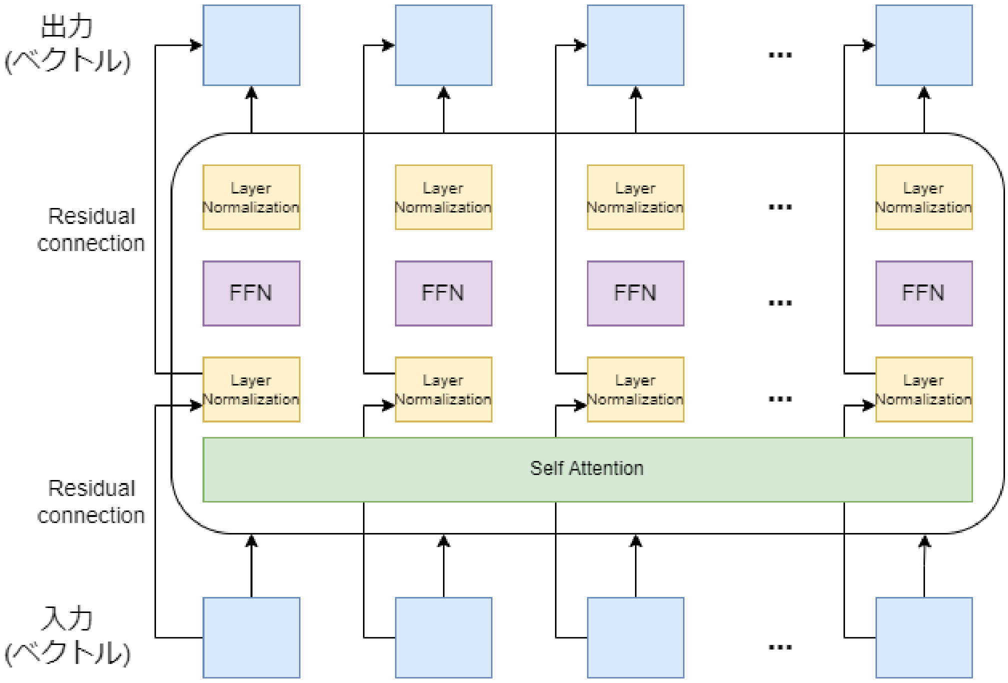

図 2.7: transformer-encoder の構造

語モデルの事前学習を行うことにより，教師データが少量，または一切存在しない状態で汎用的に利用できるモデルを構築することを目的としている．

構成は Transformer の decoder 部分をベースとしている．Transformer の decoder 部分では，以前の情報は MASK されるため推論や文生成などと相性が良く，高い性能を持つ．

2.2.8　T5

T5 [11] は Text to Text Transfer Transformer の省略であり，2020 年に Google が発表した事前学習モデルである．

T5 の特徴としては入力，出力共にテキスト形式で行うため，質問とその回答という形式のテキストに fine-tuning することにより，翻訳や要約，分類タスクなど一つの事前学習モデルを様々なタスクに転用することが容易であるという特徴がある．この特徴を表したイメージ図を図 2.8 に示す．

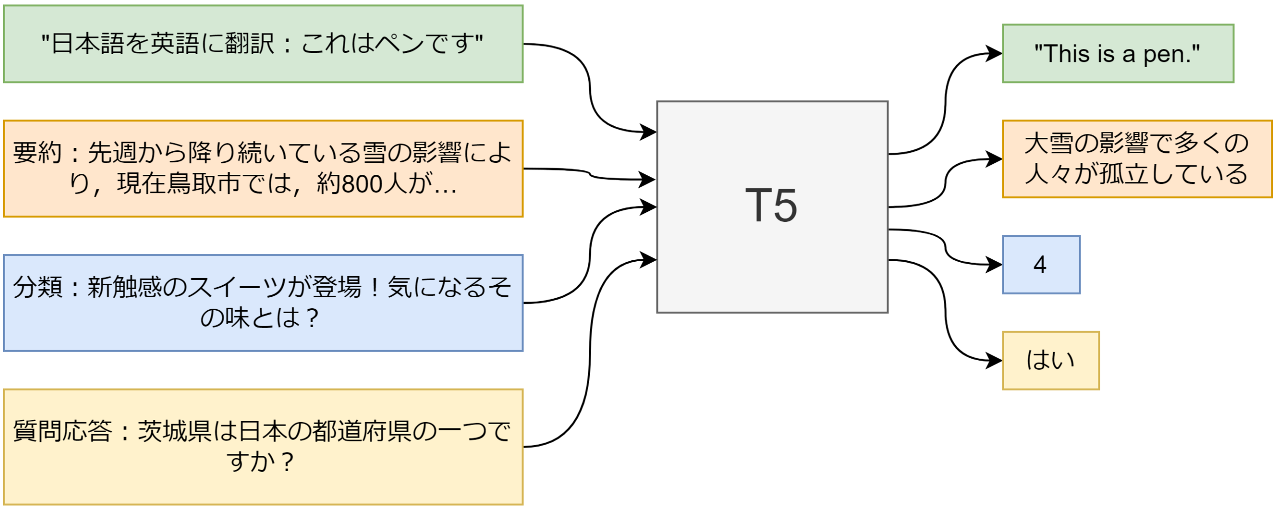

図 2.8: T5 の特徴

GLUE や SuperGLUE で当時の State of the art を達成しており，高性能なモデルとして知られている．モデルの構造は Transformer Encoder-Decoder モデルをベースとしている．

2.2.9　LLM

LLM（Large Language Model）は日本語で大規模言語モデルと呼ばれ，主に Transformer に基づいた数十億から数兆規模に渡る大規模な学習パラメータを持つモデルを，大量の学習データを用いて事前学習を行った言語モデルを指す用語である．

BERT のベースモデルのパラメータ数が約 1.1 億なのに対して[1]ChatGPT（GPT-3.5

---

[1] 本稿では BERT などの fine-tuning を必要とするモデルは事前学習モデルとして扱い，大規模言語モデルとは別の分類とする．

版 ）では約 3550 億と大きく増加しており，現状の言語モデルでは，パラメータ数を大規模化し，大量な学習データで十分に学習することで言語モデルの性能が大きく向上することが示された．

これにより一つのモデルで文生成や質問応答，翻訳，プログラミングなど言語に関する幅広いタスクに対して高品質な出力を行うことができる．

また fine-tuning を必要としないモデルも多く，タスクに対する学習データが存在しなくとも，プロンプトによる指示のみでタスクに対応した出力を得られる zero-shot 学習でいくつかの例示を与える，few-shot やタスクに応じた少量の学習データでモデルのパラメータを調整する fine-tuning を行ったモデルと遜色ない出力を得ることができることも特徴の一つとして挙げられる．

2.2.10　ChatGPT

OpenAI により公開された LLM の一つである．モデルは，Transformer に基づいており，いくつかのバージョンが存在する．GPT-3.5 版では 3550 億のパラメータ，さらに高性能な GPT-4 版や GPT-4o 版では公開されていない．

公開当初，様々なタスクにおいて zero shot 学習で，当時優れた言語モデルであった BERT などに代表される事前学習モデルを超える性能を出したことで大きく注目を集め，専門的な知識を必要とせずとも対話形式で簡単に言語モデルを扱えることから一般的にも使われるようになり，その後の言語モデルに多くの影響を与えた．

2.3 CommonGen

Lin ら [7] は常識推論能力を測る制約付き文生成タスクである CommonGen を提唱し，データセットを公開した．常識推論を取り扱うタスクはいくつか存在するもののその殆どが選択肢の中から正解を選ぶ識別タスクであり，この論文では生成タスクに発展させた．

このタスクは，いくつかのオブジェクトや動作を表す単語を集めたコンセプトセットを用意し，コンセプトセットに含まれる単語を全て使用した日常的な光景として妥当な文を生成するタスクである．

例として論文中には，”dog, frisbee, catch, throw” というコンセプトセットが提示され，望ましい生成例として，”A dog leaps to catch a thrown frisbee.”，や”The dog catches the frisbee when the boy throws it.” のような文が示されている．

このタスクのデータセットを公開し，人間といくつかの当時の最先端の言語モデルでのベースラインを示し，SPICE 指標にて最も良かった言語モデルにおいても ”dog catches a frisbee and throws it to a dog” のようなありえない文を生成してしまうことを課題として挙げた．

2.3.1 日本語における CommonGen を扱った研究

鈴木ら [12] [13] は，Lin　らの CommonGen を参考に日本語により CommonGen のベースラインシステムとデータセットを作成し，それをもとに日本語においても同様に CommonGen タスクが機能するかの検証を行った．

データセットは，Web 上に公開されているキャプションに関するデータセットから 3 単語を抜き出す手法で自動作成されたデータセットと著者らが独自に考えた単語を入力とするテストデータの 2 つがある

作成されたベースラインシステムは，ウェブ上で公開されている日本語 T5 事前学習済みモデル*2 をもとに，作成されたデータセットを用いて fine-tuning されたモデルである．

実験では，自動作成したテストデータと独自に作成したテストデータのそれぞれで生

---

*2 `https://huggingface.co/sonoisa/t5-base-japanese/discussions`

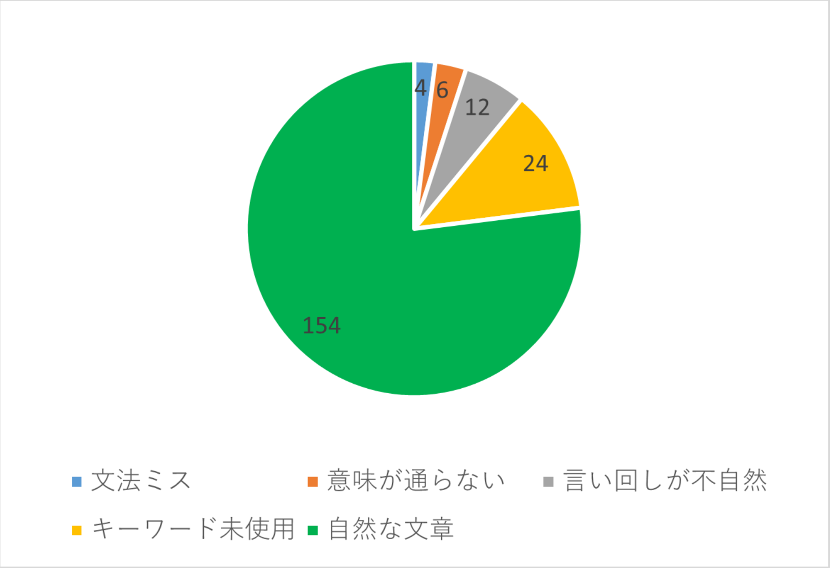

図 2.9: キャプション文から自動作成したテストデータ（200 組）の評価結果

成文の評価を 5 段階に分けて，人手により評価を行われている．

自動作成したテストデータの実験結果は，テストデータの約 8 割は 5 段階中最も良い自然な生成文と判定され，ほぼ妥当な文を生成できており，日本語においても CommonGen タスクは可能であると示された．

また妥当でないと評価された文の中でも文章の自然さを測る上では評価されていない「キーワード未使用」を除くと最も割合が高いのは言い回しの不自然さを含む文であり，日本語においても常識推論能力が求められると著者らは主張した．

独自に作成したテストデータの実験結果からは，入力単語間の関連性や単語の具体性が生成のしやすさに関係しているのではないかということが示されている．

また検証の中で入力単語の中にハブとなる単語を追加する手法を提案し，実装したシステムを使用することにより，生成文の内，入力単語を含まない文を減らすことができ，改善できることを示した．

表 2.1: 独自に作成されたテストデータのの生成例

キーワード	生成文
先生, 生徒, 教科書	生徒と先生と生徒が教科書を見ている
ドイツ, フランス, 首脳	フランスの首脳部はこっちを向いて笑っている
コンビニ, ペットボトル, おつり	コンビニのおつりにペットボトルが置いてある
犬, 散歩, リード	犬がリードをつけて散歩している
鍵, 金庫, 貴重品	金庫に貴重品と鍵がおかれている
夏, 海, コップ	夏の海辺にコップがたくさんある
ズボン, 洗濯機, パソコン	洗濯機の上にパソコンとズボンをはいた男性が座っている
食器, スポンジ, スイッチ	食器の横にスイッチとスポンジがある
野球, 観客, 本	野球の観客が本を読んでいる
警察, 手錠, 時計	警察の手錠の前に時計がある
紙, 神社, 道路	道路に神社と紙が貼られている
音楽, 配送, パン	配送中のパンに音楽がかかっている
空港, 健康, 漫画	空港で健康に気を使いながら漫画を読んでいる
お墓, 海岸, ボール	海岸にお墓の前にボールが置いてある
テーブル, デート, 動画	テーブルの上でデートの動画が放送されている
醤油, 山頂, イヤホン	山頂でイヤホンをして，醤油を垂らしている
亀, 鉛筆, クリップ	亀のクリップが鉛筆に刺さっている
財布, 昆虫, 壺	財布の上に昆虫の壺がある
歯, 電気, マグネット	歯とマグネットで電気を点けている
鎖, 溝, ラップ	溝のあいたラップに鎖が刺さっている

第 3 章

CommonGen タスクの妥当性の検証

前述した ChatGPT は，様々な NLP タスクにおいて高水準である．そこで 日本語 CommonGen を扱った研究 [13] で示された T5 モデルのベースラインシステムと比較し，構築した日本語 CommonGen のデータセットに対する ChatGPT の性能評価を行った．

本実験で利用したのは OpenAI が公開した ChatGPT[1] である．本実験でのプロンプトは，3 つの単語を使った一文を作成するよう指示した．その具体例を表 3.1 に示す．

表 3.1: ChatGPT に対するプロンプトの例

: ユーザ
各行の 3 つの単語を使った一文を作成してください
「野球選手」と「ボール」と「飛ぶ」
「ピンク」と「履く」と「子供」
「停る」と「まつ」と「プロペラ機」
. . .
:ChatGPT
野球選手がボールを打ち、ボールが飛ぶ。
子供がピンクの靴下を履く。
プロペラ機が停まり、私たちは出発を待つ。

本実験では，上述したテストデータの内，ChatGPT（GPT3.5 版）から得られた出力

---

[1] `https://openai.com/blog/chatgpt`

を前述した 5 段階で評価し，日本語 CommonGen 用に我々が構築した T5 モデルの出力と比較した．

自動作成したテストデータの T5 モデルの結果を図 3.1 に，ChatGPT（GPT-3.5 版）の結果を図 3.2 に示す．

また独自に作成したテストデータに対する T5 モデルの結果を図 3.3 に，ChatGPT（GPT-3.5 版）の結果を図 3.4 に示す．

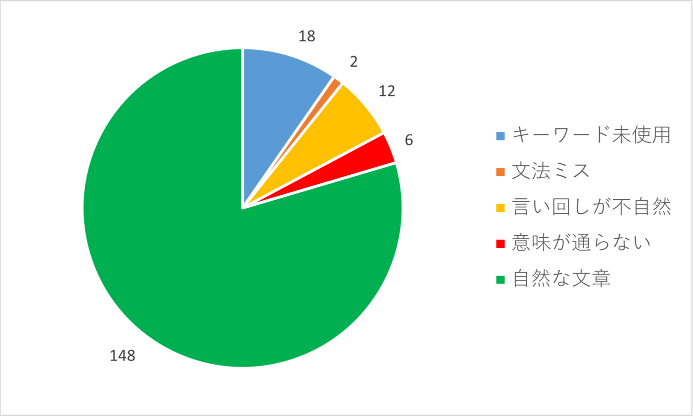

図 3.1: 自動作成テストデータの T5 モデルによる出力の評価結果

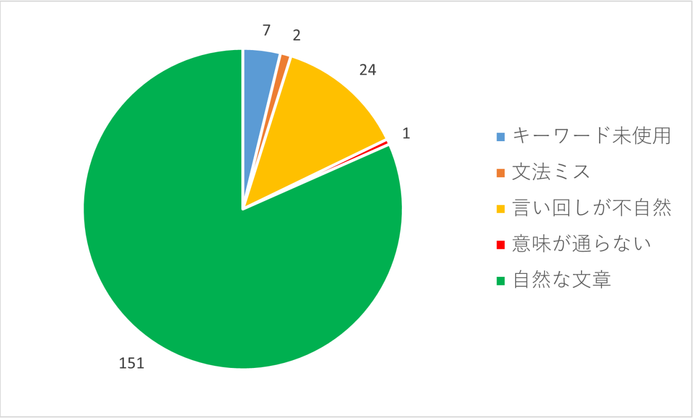

図 3.2: 自動作成テストデータの ChatGPT（GPT3.5 版）による出力の評価結果

自動作成したテストデータに対する結果（図 3.1 と図 3.2）を見ると，ChatGPT（GPT-3.5 版）は，キーワード未使用と評価される生成文は減少したものの不自然な生成文は増えており，自然な生成文の割合は T5 モデルとほとんど変わらなかった．

一方，独自に作成したテストデータの結果（図 3.3 と図 3.4）を見ると，ChatGPT（GPT-3.5 版）は T5 モデルを用いた場合よりもかなり良い評価が得られた．

また各テストデータにおける ChatGPT（GPT-3.5 版）による出力の評価結果の内，自

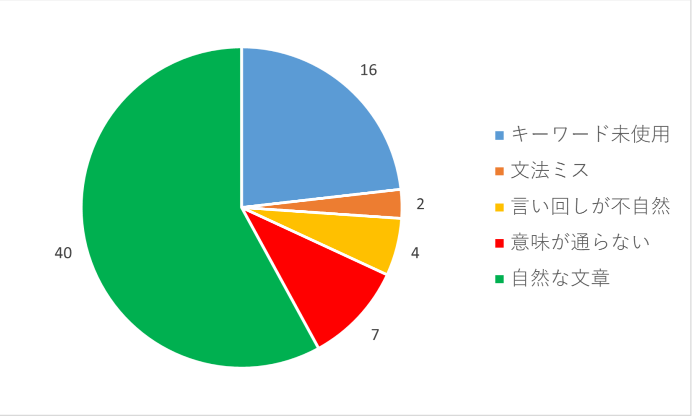

図 3.3: 独自に作成したテストデータの T5 モデルによる出力の評価結果

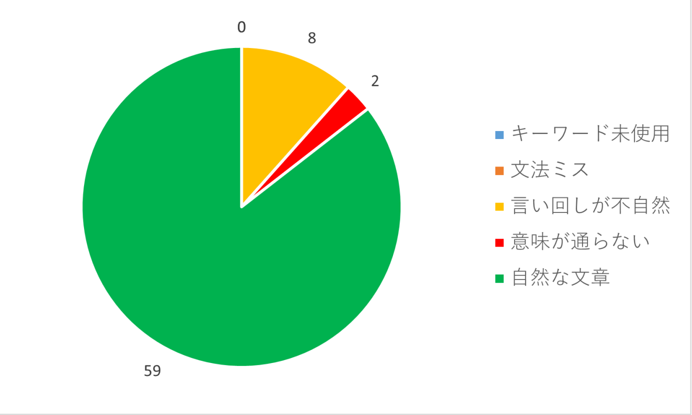

図 3.4: 独自に作成したテストデータの ChatGPT（GPT3.5 版）による出力の評価結果

然な文との評価を得られなかった入力キーワードにおいて GPT-4 版の ChatGPT を利用して，生成を行い，生成文を同様に 5 段階評価した．評価結果を図 3.5 に示す．また，その際の独自のデータセットにおける T5 モデル，ChatGPT（GPT-3.5 版），ChatGPT（GPT-4 版）の生成例を付録に示す．

ChatGPT（GPT-4 版）は ChatGPT（GPT-3.5 版）が自然に生成できなかったキーワード群においても各データセットのほとんどで自然な生成文を出力できており，かなり性能が高いことが分かる．

また ChatGPT（GPT3.5 版）による出力の評価結果の内，自然な文と評価したもの以外の入力キーワードにおいて 5 つの候補を出すよう指示を行い，再生成を行った．

その結果 5 つの候補の内，いくつの候補が自然な文にあたるかの分布を表したものを図 3.6 に示す．5 候補全てで自然と評価されたものだけでも約 4 割あり，5 候補の内自然な候補の方が多いキーワード群は 6 割を超えている．

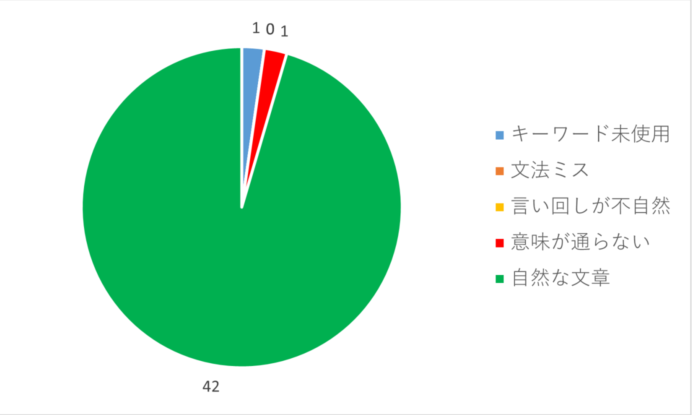

図 3.5: ChatGPT（GPT-3.5 版）の自然でない生成に用いられたキーワード群を利用した ChatGPT（GPT-4 版）による出力の評価結果

このことから ChatGPT（3.5 版）であっても，ある程度生成に幅が存在し，たとえ一度文生成が上手くいかなくとも再生成によって生成文の改善が見られることが分かる．

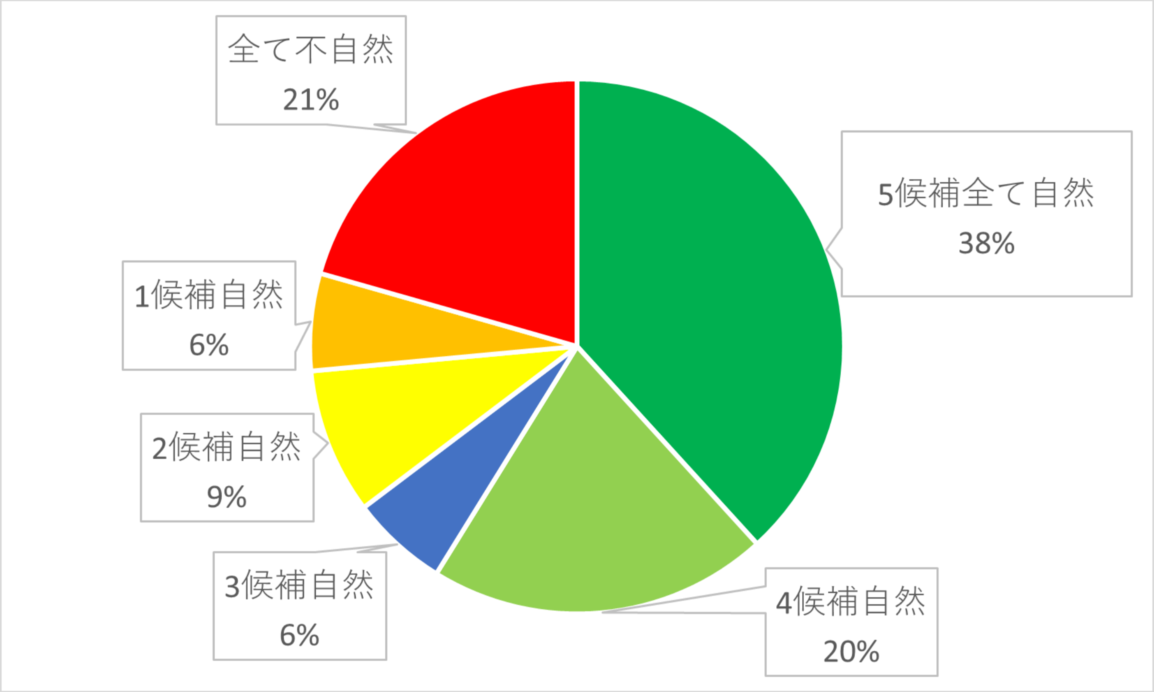

図 3.6: ChatGPT（GPT-4）による出力の評価結果

第 4 章

実験

CommonGen における生成難易度を上げる手法の提案を行う．

4.1　テストデータの作成

本研究では，文生成タスクである CommonGen を現在の LLM に見合ったタスクにするために入力単語を一般的な単語から専門的な単語に変更する．また生成能力を測るために，テストデータの難易度が段階的になるよう設定する．

専門な単語の分野は，情報，哲学，経済分野の単語として，表 4.1 のような組み合わせで設定した．表 4.1 中の番号が若いものほど広い分野の専門用語が出現し，難易度が高いことが分かる．各パターン 10 個のテストデータ，計 30 個のテストデータを作成した．

表 4.1: 分野別の単語組み合わせ

	単語 1	単語 2	単語 3
パターン 1	情報	哲学	経済
パターン 2	情報	情報	経済
パターン 3	情報	情報	情報

4.2　LLM による文生成

データセットが現在の LLM においても有効か確認するため claude 3.5 sonet，gemini 1.5，command R+ の三つの LLM を用いて各 LLM において 4.1 で作成したテストデー

タを入力として生成を行った．生成に用いたプロンプトを表 4.2 に示す．

表 4.2: LLM による文生成のためのプロンプト例

以下の 3 単語を含む一般的な文を作成してください

単語

セキュリティ, 道徳, 買収

ルール

- 3 単語を必ず含む
- なるべく簡潔な文
- 意味が分かり、かつ常識的な文

4.3　人手による評価

各 LLM が生成した文を人手によって比較評価した．評価方法は同一の入力から得られた各 LLM の出力をランダムに並び替えたものを，1 から 3 まで順番を付けるよう指示した．これを計 30 個，16 名に行ってもらった．この評価の結果を図 4.1 に示す．

全体の評価結果を見ると，各 LLM の性能差はあまり大きくは出ていないことが分かる．特徴的と言える点は，Claude 3.5 sonet の 2 位評価と Commnad R+ の 3 位評価が多い点が挙げられる．

4.4　ChatGPT-4o による評価

ChatGPT-4o を用いて各 LLM が生成した文の評価を行った．

まず専門用語の組み合わせについて検証を行うため， ChatGPT-4o に Claude 3.5 sonet が生成した文を 10 点満点で評価を行わせ，表 4.1 で示した専門用語の組み合わせにより難易度の変化が起きているかを確認した．

プロンプトを 4.3 に，評価結果を表 4.4 に示す．評価結果から情報分野のみで統一された場合では，評価の平均点が高く情報，哲学，経済の組み合わせと情報 2 単語，経済 1 単語の組み合わせでは，あまり差ががが出ていないことが分かる．

また評価を行うために同一の入力を与えた際の各 LLM が生成した 3 つの文を入力し，

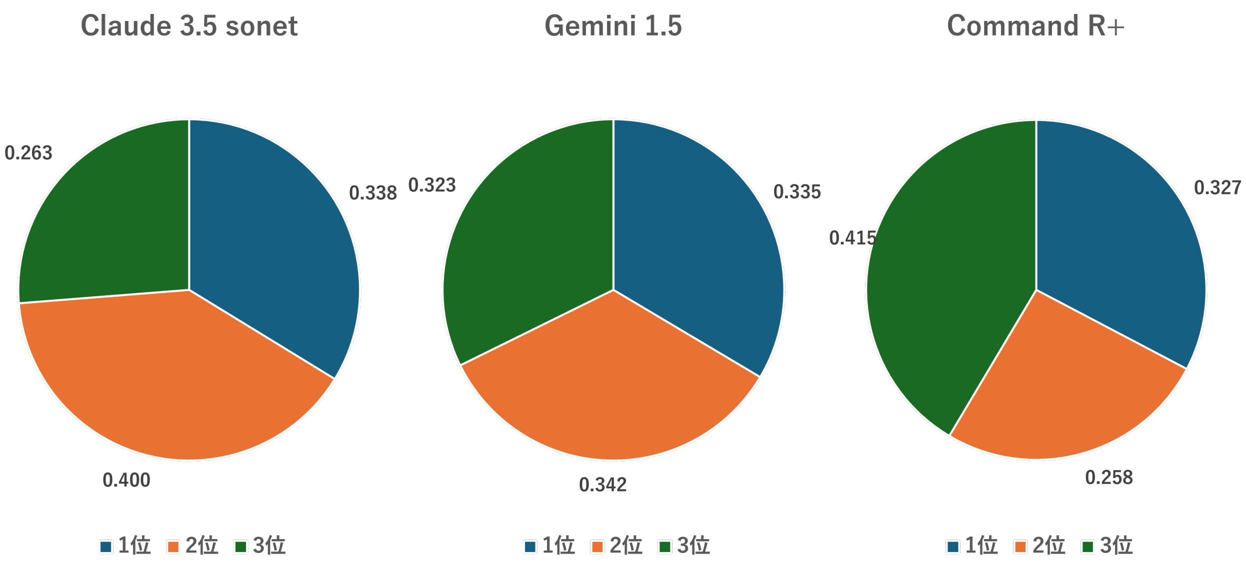

図 4.1: 人手による評価で各 LLM が 1 位から 3 位と評価された割合

表 4.3: ChatGPT-4o による文の妥当性の評価のためのプロンプト例

文の自然さについて 10 点満点中何点か
評価してください
また下の 3 つの点に注意してください
・文全体で意味が通っているか
・常識的におかしな点がないか
・単語の意味に誤りがないか

企業買収の際には、セキュリティと道徳の両面を
考慮することが重要だ。

順位をつけさせた．プロンプトの例を表 4.5 に示す．なお，プロンプトに入力される生成文はランダムに並び替えている．実験結果を表 4.6 に示す．

実験結果は人手評価と比較してはっきりと性能差をつけており， ChatGPT-4o は Claude 3.5 sonet の出力が最もよく，ついで Gemini 1.5，最後に Command R+ という順で評価したことが分かる．

表 4.4: 単語組み合わせ別の評価値

専門用語組み合わせ	評価平均点
情報，哲学，経済	8.4
情報，情報，経済	8.5
情報，情報，情報	10

表 4.5: ChatGPT-4o による文評価のためのプロンプト例

以下の 1 から 3 の文を評価して順番を付けてください
評価の基準は自然な文かつ意味の分かる文であるかどうかです

1. 非同期的に実行される MMT システムは、誤謬推理を検出して修正するメカニズムを組み込むことで、より強固で正確なものとなる。
2.MMT の批評家は、非同期的な経済効果を考慮しないことを誤謬推理の一例として指摘する。
3.MMT 理論は、非同期的な経済政策を通じて、外生的な経済ショックに対応することを提案している。

表 4.6: ChatGPT4-o による生成文の比較結果

	1 番	2 番	3 番
claude 3.5 sonet	16	12	2
gemini 1.5	8	16	2
command R+	2	6	22

第 5 章

考察

5.1 専門用語に限定する手法の効果の考察

本研究にて提案した入力単語を専門的な用語に制限することにより，文生成の難易度を上げる手法の効果について考察する．

生成文の妥当性に関しては，主観による評価となってしまうことを注意が必要である．各 LLM による生成文は，一見するともっともらしい文が多い．

しかし，専門用語を多用する関係で文全体を通してみると意味がかなりあいまいになっており，間違っている文ではないが，的外れな文や無理に単語を使用しているような文が散見される．改善すべき点がある文を生成することができるという点においては，タスクとして現在の LLM に見合ったものにすることができたと考える．

また，専門用語の組み合わせに関しても 4.4 から組み合わせによる難易度の上昇効果が確認できており，複数の専門用語の組み合わせは生成難易度の上昇に効果があると考えられる．

ただ，本研究で入力とした単語には，専門用語であるため，従来の CommonGen タスクと比べ正解文の評価が人間にも難しく，本研究で提案する手法による改変を加えた CommonGen タスクでは，一般的な知識に対する文生成能力を測るタスクとしての効果は疑問点が残り，常識の範囲をどう設定するかという問題は顕在化している．

5.2　人手と機械による評価結果の考察

人間による評価結果から分かる特徴として，Claude 3.5 sonet の 3 位評価が少なく，Command R+ の 3 位評価が多いことが分かる．このことから，3 つの LLM の内，妥当な文の生成能力においては Claude 3.5 sonet が最も高く，Command R+ の能力が低いと考えられる．

この傾向は，ChatGPT-4o による評価とも相関があり，機械においても文の自然さの評価ができているのではないかと考えられる．そのため今後は，LLM による生成文の自動評価という手法は実用的であり，自動評価手法の確立が難しい他の文生成タスクなどにおいても，有効なのではないかと考える．

しかし同時に ChatGPT-4o による評価では，はっきりと差が出ているにもかかわらず人手による評価では，差が大きく見られなかった点に関しても考察を行う．

人手による評価は、全体の割合では差が見られなかったものの質問ごとの比較では，ある程度割れている結果が出ており，感覚の個人差により差が出ていないというわけではないと考えられる．

中には，ChatGPT-4o と人手評価で正反対の評価結果となる場合も確認されており，どのような要素が人手評価と機械による評価において優劣をつけるのかを今後分析する必要があると考える．

表 5.1: 各 LLM の生成文と人手評価，機械評価例

生成した LLM	生成文	1 位	2 位 （人手）	3 位	順位 （LLM）
claude 3.5 sonet	企業買収の際には、セキュリティと道徳の両面を考慮することが重要だ。	0.37	0.50	0.12	2
gemini 1.5	企業買収においては、セキュリティと倫理に配慮することが重要である。	0.43	0.37	0.18	1
command R+	セキュリティ企業の買収には、道徳的な問題がつきまとう。	0.18	0.12	0.68	3
claude 3.5 sonet	IR チームは、機密情報の処理に関するジレンマに直面することがある。	0.12	0.50	0.37	3
gemini 1.5	IR 担当者は、情報処理における倫理的ジレンマに直面することが多い。	0.56	0.37	0.06	2
command R+	IR 処理におけるジレンマは、ノイズ除去と特徴抽出のバランスにある。	0.31	0.12	0.56	1
claude 3.5 sonet	MMT の批評家は、非同期的な経済効果を考慮しないことを誤謬推理の一例として指摘する。	0.12	0.25	0.62	3
gemini 1.5	MMT 理論は、非同期的な経済政策を通じて、外生的な経済ショックに対応することを提案している。	0.31	0.43	0.25	1
command R+	非同期的に実行される MMT システムは、誤謬推理を検出して修正するメカニズムを組み込むことで、より強固で正確なものとなる。	0.56	0.31	0.12	2

第 6 章

結論

本研究では，現在の高性能な LLM において，日本語 CommonGen 形式のタスクがどの程度効果的なのかを検証するため T5 のベースラインシステムと ChatGPT（ GPT-3.5 版と GPT-4 版）で同一の入力から得られた出力を比較した．その結果，ChatGPT（ GPT-3.5 版 ）では，T5 の出力よりも優れた結果が得られ， ChatGPT （ GPT-4 版）では，ほとんどの出力で，自然と評価される結果が得られ，通常の CommonGen 形式のタスクは，高性能な LLM にとって妥当なタスク出ないことが示された．そこでタスクを現在の LLM に見合ったものに拡張する手法として，入力単語を専門的な用語に限定する手法を提案する．提案手法に基づいたテストデータを用いて，3 つの LLM に文を生成させ，それぞれの生成文を人手による評価と ChatGPT-4o による自動評価を行い，それぞれの評価を比較した．比較結果は，3 位となる評価が最も多い LLM と　3 位評価が最も少ない LLM が一致しており機械と人手とで一定の相関が見られた．しかし同時に LLM による評価でははっきりと差が見られたのに対し，人手評価には差が大きくは見られなかったことに関して今後検証が必要であると考えられる．また提案手法に関しては，主観による評価とはなるが，現在の LLM においても有効な難易度に変更できる手法ではないかと結論付ける．

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員の新納浩幸教授に深く感謝申し上げます．また研究活動に取り組むにあたって，様々な助言や知識を頂き，自身の研究に協力して頂いた新納研究室の皆さまにも深く感謝の意を表します．

参考文献

[1] Talmor Alon, Herzig Jonathan, Lourie Nicholas, and Berant Jonathan. Commonsenseqa: A question answering challenge targeting commonsense knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4149–4158. Association for Computational Linguistics, 2019.

[2] Sap Maarten, Rashkin Hannah, Chen Derek, Le Bras Ronan, and Choi Yejin. Social iqa: Commonsense reasoning about social interactions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 4463–4473. Association for Computational Linguistics, 2019.

[3] Sakaguchi Keisuke, Le Bras Ronan, Bhagavatula Chandra, and Choi Yejin. Winogrande: An adversarial winograd schema challenge at scale, 2019.

[4] Omura Kazumasa, Kawahara Daisuke, and Kurohashi Sadao. A method for building a commonsense inference dataset based on basic events. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2450–2460. Association for Computational Linguistics, 2020.

[5] Zellers Rowan, Bisk Yonatan, Schwartz Roy, and Choi Yejin. Swag: A large-scale adversarial dataset for grounded commonsense inference. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 93–104. Association for Computational Linguistics, 2018.

[6] Zellers Rowan, Holtzman Ari, Bisk Yonatan, Farhadi Ali, and Choi Yejin. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–

4800. Association for Computational Linguistics, 2019.

[7] Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. CommonGen: A constrained text generation challenge for generative commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1823–1840. Association for Computational Linguistics, 2020.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc., 2017.

[9] 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, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[10] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[11] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, Vol. 21, No. 140, pp. 1–67, 2020.

[12] 鈴木 雅人, 新納 浩幸. 日本語 CommonGen の試作と入力単語間の関連性からの考察. 自然言語処理研究会 (第 253 回), 2022.

[13] 鈴木 雅人, 新納 浩幸. 日本語 CommonGen における入力キーワード群のハブ単語の自動追加による生成文の改善. 言語処理学会 (第 29 回), 2023.

付録

A　プログラム

T5 のベースラインシステムを訓練するプログラムのソースコードを A.2 に示す.

ソースコード A.1: train.py

```

import argparse
import glob
import os
import json
import time
import logging
import random
import re
from itertools import chain
from string import punctuation

import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
import pytorch_lightning as pl

from transformers import (
    AdamW,
    T5ForConditionalGeneration,
    T5Tokenizer,
    get_linear_schedule_with_warmup
)

# 事前学習済みモデル
PRETRAINED_MODEL_NAME = "sonoisa/t5-base-japanese"
```

```

# 転移学習済みモデル
MODEL_DIR = "model"

# 乱数シードの設定
def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

set_seed(42)

# 利用有無 GPU
USE_GPU = torch.cuda.is_available()

# 各種ハイパーパラメータ
args_dict = dict(
    data_dir="data", # データセットのディレクトリ
    model_name_or_path=PRETRAINED_MODEL_NAME,
    tokenizer_name_or_path=PRETRAINED_MODEL_NAME,

    learning_rate=3e-4,
    weight_decay=0.0,
    adam_epsilon=1e-8,
    warmup_steps=0,
    gradient_accumulation_steps=1,

    # max_input_length=64,
    # max_target_length=512,
    # train_batch_size=8,
    # eval_batch_size=8,
    # num_train_epochs=10,

    n_gpu=1 if USE_GPU else 0,
    early_stop_callback=False,
    fp_16=False,
    opt_level=’01’,
    max_grad_norm=1.0,
    seed=42,
```

```
)

class TsvDataset(Dataset):
    def __init__(self, tokenizer, data_dir, type_path, input_max_len=512, target_max_len=512):
        self.file_path = os.path.join(data_dir, type_path)

        self.input_max_len = input_max_len
        self.target_max_len = target_max_len
        self.tokenizer = tokenizer
        self.inputs = []
        self.targets = []

        self._build()

    def __len__(self):
        return len(self.inputs)

    def __getitem__(self, index):
        source_ids = self.inputs[index]["input_ids"].squeeze()
        target_ids = self.targets[index]["input_ids"].squeeze()

        source_mask = self.inputs[index]["attention_mask"].squeeze()
        target_mask = self.targets[index]["attention_mask"].squeeze()

        return {"source_ids": source_ids, "source_mask": source_mask,
                "target_ids": target_ids, "target_mask": target_mask}

    def _make_record(self, text, keywords):
        # タスク用の入出力形式に変換する。CommonGen
        input = f"{keywords}"
        target = f"{text}"
        return input, target

    def _build(self):
        with open(self.file_path, "r", encoding="utf-8") as f:
            for line in f:
                line = line.strip().split("\t")
                assert len(line) == 2
```

```
                assert len(line[0]) > 0
                assert len(line[1]) > 0

                text = line[0]
                keywords = line[1]

                input, target = self._make_record(text, keywords)

                tokenized_inputs = self.tokenizer.batch_encode_plus(
                    [input], max_length=self.input_max_len, truncation=True,
                    padding="max_length", return_tensors="pt"
                )

                tokenized_targets = self.tokenizer.batch_encode_plus(
                    [target], max_length=self.target_max_len, truncation=True,
                    padding="max_length", return_tensors="pt"
                )

                self.inputs.append(tokenized_inputs)
                self.targets.append(tokenized_targets)

# トークナイザー（）モデルの読み込み SentencePiece
tokenizer = T5Tokenizer.from_pretrained(PRETRAINED_MODEL_NAME, is_fast=True)

# テストデータセットの読み込み
train_dataset = TsvDataset(tokenizer, args_dict["data_dir"], "train.tsv",
                           input_max_len=16, target_max_len=24)

for data in train_dataset:
    print("A.␣入力データの元になる文字列")
    print(tokenizer.decode(data["source_ids"]))
    print()
    print("B.␣入力データ（の文字列がトークナイズされたトークン列）AID")
    print(data["source_ids"])
    print()
    print("C.␣出力データの元になる文字列")
    print(tokenizer.decode(data["target_ids"]))
```

```
    print()
    print("D.␣出力データ（の文字列がトークナイズされたトークン列）CID")
    print(data["target_ids"])
    break

class T5FineTuner(pl.LightningModule):
    def __init__(self, hparams):
        super().__init__()
        self.hparams = hparams

        # 事前学習済みモデルの読み込み
        self.model = T5ForConditionalGeneration.from_pretrained(hparams.model_name_or_path)

        # トークナイザーの読み込み
        self.tokenizer = T5Tokenizer.from_pretrained(hparams.tokenizer_name_or_path, is_fast=True)

    def forward(self, input_ids, attention_mask=None, decoder_input_ids=None,
                decoder_attention_mask=None, labels=None):
        """順伝搬"""
        return self.model(
            input_ids,
            attention_mask=attention_mask,
            decoder_input_ids=decoder_input_ids,
            decoder_attention_mask=decoder_attention_mask,
            labels=labels
        )

    def _step(self, batch):
        """ロス計算"""
        labels = batch["target_ids"]

        # All labels set to -100 are ignored (masked),
        # the loss is only computed for labels in [0, ..., config.vocab_size]
        labels[labels[:, :] == self.tokenizer.pad_token_id] = -100

        outputs = self(
            input_ids=batch["source_ids"],
```

```
            attention_mask=batch["source_mask"],
            decoder_attention_mask=batch[’target_mask’],
            labels=labels
        )

        loss = outputs[0]
        return loss

    def training_step(self, batch, batch_idx):
        """訓練ステップ処理"""
        loss = self._step(batch)
        self.log("train_loss", loss)
        return {"loss": loss}

    def validation_step(self, batch, batch_idx):
        """バリデーションステップ処理"""
        loss = self._step(batch)
        self.log("val_loss", loss)
        return {"val_loss": loss}

    def test_step(self, batch, batch_idx):
        """テストステップ処理"""
        loss = self._step(batch)
        self.log("test_loss", loss)
        return {"test_loss": loss}

    def configure_optimizers(self):
        """オプティマイザーとスケジューラーを作成する"""
        model = self.model
        no_decay = ["bias", "LayerNorm.weight"]
        optimizer_grouped_parameters = [
            {
                "params": [p for n, p in model.named_parameters()
                            if not any(nd in n for nd in no_decay)],
                "weight_decay": self.hparams.weight_decay,
            },
            {
                "params": [p for n, p in model.named_parameters()
                            if any(nd in n for nd in no_decay)],
                "weight_decay": 0.0,
            },
```

```
        ]
        optimizer = AdamW(optimizer_grouped_parameters,
                          lr=self.hparams.learning_rate,
                          eps=self.hparams.adam_epsilon)
        self.optimizer = optimizer

        scheduler = get_linear_schedule_with_warmup(
            optimizer, num_warmup_steps=self.hparams.warmup_steps,
            num_training_steps=self.t_total
        )
        self.scheduler = scheduler

        return [optimizer], [{"scheduler": scheduler, "interval": "step", "frequency": 1}]

    def get_dataset(self, tokenizer, type_path, args):
        """データセットを作成する"""
        return TsvDataset(
            tokenizer=tokenizer,
            data_dir=args.data_dir,
            type_path=type_path,
            input_max_len=args.max_input_length,
            target_max_len=args.max_target_length)

    def setup(self, stage=None):
        """初期設定（データセットの読み込み）"""
        if stage == ’fit’ or stage is None:
            train_dataset = self.get_dataset(tokenizer=self.tokenizer,
                                             type_path="train.tsv", args=self.hparams)
            self.train_dataset = train_dataset

            val_dataset = self.get_dataset(tokenizer=self.tokenizer,
                                           type_path="dev.tsv", args=self.hparams)
            self.val_dataset = val_dataset

            self.t_total = (
                (len(train_dataset) // (self.hparams.train_batch_size * max(1, self.hparams.n_gpu)))
```

```
                // self.hparams.gradient_accumulation_steps
                * float(self.hparams.num_train_epochs)
            )

    def train_dataloader(self):
        """訓練データローダーを作成する"""
        return DataLoader(self.train_dataset,
                          batch_size=self.hparams.train_batch_size,
                          drop_last=True, shuffle=True, num_workers=4)

    def val_dataloader(self):
        """バリデーションデータローダーを作成する"""
        return DataLoader(self.val_dataset,
                          batch_size=self.hparams.eval_batch_size,
                          num_workers=4)

# 学習に用いるハイパーパラメータを設定する
args_dict.update({
    "max_input_length": 16, # 入力文の最大トークン数
    "max_target_length": 24, # 出力文の最大トークン数
    "train_batch_size": 8,
    "eval_batch_size": 8,
    "num_train_epochs": 10,
    })
args = argparse.Namespace(**args_dict)

train_params = dict(
    accumulate_grad_batches=args.gradient_accumulation_steps,
    gpus=args.n_gpu,
    max_epochs=args.num_train_epochs,
    precision= 16 if args.fp_16 else 32,
    amp_level=args.opt_level,
    gradient_clip_val=args.max_grad_norm,
)

# 転移学習の実行（を利用すればエポック分程度）GPU110
model = T5FineTuner(args)
trainer = pl.Trainer(**train_params)
trainer.fit(model)

```

```
# 最終エポックのモデルを保存
# model.tokenizer.save_pretrained(’trained_model’)
# model.model.save_pretrained(’trained_model’)

model.tokenizer.save_pretrained(’test’)
model.model.save_pretrained(’test’)
```

訓練したモデルをテストするプログラムのソースコードを A.2 に示す.

ソースコード A.2: train.py

```
import argparse
from hashlib import new
import gensim
import os
import textwrap
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import T5ForConditionalGeneration, T5Tokenizer
from tqdm.auto import tqdm
from sklearn import metrics
import MeCab
import random

# データの（chiVePATHkv:）KeyedVectors
model_path = "chive-1.2-mc15_gensim/chive-1.2-mc15.kv"

# モデルの読み込み
wv = gensim.models.KeyedVectors.load(model_path)

# 分散表現を用いた単語の発見
def findkeyword(inputtext):
    keywords = []
    result = []
    words = [i for i in inputtext.split(’,’)]
    for word in words:
        try:
            keywords.append(wv[word])
        except KeyError:
            print(word+’は未知語です’)
    match = wv.most_similar(positive=keywords, topn=6)
    for i in match:
```

```
        if i[0] not in words:
            result.append(inputtext+’,’+i[0])
        if len(result) >= 3:
            break
    return result

# 生成結果から単語を発見する
def findkeyword2(str, keyword):
    # 追加キーワード候補
    word_list = []
    m = MeCab.Tagger()
    # 文章を形態素解析
    words = m.parse(str).split(’\n’)
    # 既に決められたキーワードを分割
    keywords = keyword.split(’,’)

    # 文章の形態素からキーワード候補の発見
    for word in words:
        spl = word.split(’\t’)
        if len(spl) >= 2:
            word = spl[0]
            pos = spl[-1].split(’,’)
            if ((’名詞’ in pos) and (’接尾’ not in pos) and (’数’ not in pos)) or (’形容詞’ in pos) or (’連体詞’ in pos):
                pass
            elif ("動詞" in pos) and ("自立" in pos):
                if ’サ変’ in pos or ’する’ in pos or ’ある’ in pos or ’いる’ in pos or ’なる’ in pos:
                    continue
                else:
                    # 基本形に設定
                    word = pos[6]
            else:
                continue
            if word not in keywords:
                word_list.append(word)
            print(word, pos)

    # 追加キーワード候補からランダムに選出
    if len(word_list) >= 1:
        sample = random.sample(word_list, 1)
```

```
        keyword = keyword + ',' + sample[0]
    else:
        print("ERROR追加キーワードが見つかりません:")
        print(keyword, ':', str)

    return keyword

# 事前学習済みモデル
PRETRAINED_MODEL_NAME = "sonoisa/t5-base-japanese"

# 転移学習済みモデル
MODEL_DIR = "trained_model"

# の利用有無 GPU
USE_GPU = torch.cuda.is_available()


class TsvDataset(Dataset):
    def __init__(self, tokenizer, data_dir, type_path, input_max_len
        =512, target_max_len=512):
        self.file_path = os.path.join(data_dir, type_path)

        self.input_max_len = input_max_len
        self.target_max_len = target_max_len
        self.tokenizer = tokenizer
        self.inputs = []
        self.targets = []

        self._build()

    def __len__(self):
        return len(self.inputs)

    def __getitem__(self, index):
        source_ids = self.inputs[index]["input_ids"].squeeze()
        target_ids = self.targets[index]["input_ids"].squeeze()

        source_mask = self.inputs[index]["attention_mask"].squeeze()
        target_mask = self.targets[index]["attention_mask"].squeeze
            ()

```

```
        return {"source_ids": source_ids, "source_mask": source_mask,
                "target_ids": target_ids, "target_mask": target_mask}

    def _make_record(self, text, keywords):
        # タスク用の入出力形式に変換する。CommonGen
        input = f"{keywords}"
        target = f"{text}"
        return input, target

    def _build(self):
        with open(self.file_path, "r", encoding="utf-8") as f:
            for line in f:
                line = line.strip().split("\t")
                assert len(line) == 2
                assert len(line[0]) > 0
                assert len(line[1]) > 0

                text = line[0]
                keywords = line[1]

                input, target = self._make_record(text, keywords)

                tokenized_inputs = self.tokenizer.batch_encode_plus(
                    [input], max_length=self.input_max_len, truncation=True,
                    padding="max_length", return_tensors="pt"
                )

                tokenized_targets = self.tokenizer.batch_encode_plus(
                    [target], max_length=self.target_max_len, truncation=True,
                    padding="max_length", return_tensors="pt"
                )

                self.inputs.append(tokenized_inputs)
                self.targets.append(tokenized_targets)


args_dict = dict(
    data_dir="data", # データセットのディレクトリ
```

```
    model_name_or_path=PRETRAINED_MODEL_NAME,
    tokenizer_name_or_path=PRETRAINED_MODEL_NAME,

    learning_rate=3e-4,
    weight_decay=0.0,
    adam_epsilon=1e-8,
    warmup_steps=0,
    gradient_accumulation_steps=1,

    # max_input_length=64,
    # max_target_length=512,
    # train_batch_size=8,
    # eval_batch_size=8,
    # num_train_epochs=10,

    n_gpu=1 if USE_GPU else 0,
    early_stop_callback=False,
    fp_16=False,
    opt_level=’01’,
    max_grad_norm=1.0,
    seed=42,
)

# 学習に用いるハイパーパラメータを設定する
args_dict.update({
    "max_input_length": 16, # 入力文の最大トークン数
    "max_target_length": 24, # 出力文の最大トークン数
    "train_batch_size": 8,
    "eval_batch_size": 8,
    "num_train_epochs": 10,
})
args = argparse.Namespace(**args_dict)

# トークナイザー（）SentencePiece
tokenizer = T5Tokenizer.from_pretrained(MODEL_DIR, is_fast=True)

# 学習済みモデル
trained_model = T5ForConditionalGeneration.from_pretrained(MODEL_DIR)

# の利用有無 GPU
USE_GPU = torch.cuda.is_available()
```

```
if USE_GPU:
    trained_model.cuda()

# テストデータの読み込み
test_dataset = TsvDataset(tokenizer, args_dict["data_dir"], "test.tsv",
                          input_max_len=args.max_input_length,
                          target_max_len=args.max_target_length)

test_loader = DataLoader(test_dataset, batch_size=8, num_workers=4)

trained_model.eval()

inputs = []
outputs = []
targets = []

for batch in tqdm(test_loader):
    input_ids = batch['source_ids']
    input_mask = batch['source_mask']
    if USE_GPU:
        input_ids = input_ids.cuda()
        input_mask = input_mask.cuda()

    output = trained_model.generate(input_ids=input_ids,
                                    attention_mask=input_mask,
                                    max_length=args.max_target_length,
                                    repetition_penalty=10.0, # 同じ文の繰り返し（モード崩壊）へのペナルティ
                                    )

    output_text = [tokenizer.decode(ids, skip_special_tokens=True,
                                    clean_up_tokenization_spaces=False)
                   for ids in output]
    target_text = [tokenizer.decode(ids, skip_special_tokens=True,
                                    clean_up_tokenization_spaces=False)
                   for ids in batch["target_ids"]]
    input_text = [tokenizer.decode(ids, skip_special_tokens=True,
```

```
                                   clean_up_tokenization_spaces=False)
                  for ids in input_ids]

    inputs.extend(input_text)
    outputs.extend(output_text)
    targets.extend(target_text)

m = MeCab.Tagger()

key_lost_result = []

#キーワード未使用を検出して保存
with open(’result/result.txt’, mode=’w’) as fr, open(’result/keylost.txt’, mode=’w’) as fk:
    for i, (output, target, input) in enumerate(zip(outputs, targets, inputs)):
        inputs = input.split(’,’)
        wakati = m.parse(output).split(’\n’)
        out_words = []
        for out_word in wakati:
            spl = out_word.split(’\t’)
            if len(spl) >= 2:
                pos = spl[-1].split(’,’)
                if ’動詞’ in pos:
                    out_words.append(pos[6])
                else:
                    out_words.append(spl[0])

        for key in inputs:
            if not key in out_words:
                fk.write(str(i))
                fk.write("\t" + input + ’\t’)
                fk.write("\t" + output + ’\t’)
                fk.write("\t" + target + ’\t\n’)
                key_lost_result.append(input)
                break

        fr.write(str(i))
        fr.write("\t" + input)
        fr.write("\t" + output)
```

```
        fr.write("\t" + target + '\n')

mytestdata = [
    #キーワード抜け自動生成テストデータ ()
    "表情ショットミス,,",
    "本沢山なる 1,,",
    "人女食事 7,,",
    "こちら向こう走る,,",
    "カップケーキハケソース,,",
    "携帯写真見える,,",
    "周り両手模様,,",
    "揚げる凧描く,,",
    "草原くちばし足,,",
    "赤色掃除和式,,",
    "入道雲空覆う,,",
    "座る肘灰色,,",
    "冷蔵庫棚電子レンジ,,",
    "柱一つ取り付ける,,",
    "人人女性 1,2,",
    "ケーキホールくま,,",
    "一口並ぶきる,,",
    "走る架かる川,,",

    # 文法自動作成テストデータ ()
    "ドクロかぶる寝る,,",
    "しまうま並ぶ鼻,,",
    "引きシマウマ草原,,",
    "加えるフリスビー子犬,,",

    # 言い回し自動作成テストデータ ()
    "野球バッター試合,,",
    "横入る蛇口,,",
    "方向塔動く,,",
    "子供立つ湖,,",
    "座る道端植木,,",
    "守備者走塁,,",
    "こちら路線バス連結,,",
    "トイレウォシュレット閉まる,,",
    "キリン居る木,,",
    "スキー板ステッキ雪,,",
    "犬壁座る,,",
```

```

    # 意味自動作成テストデータ ()
    "くちばし目鷲,,",
    "塔の前なる赤信号,,",
    "花嫁ハミガキキス,,",
    "衣装隠すテニス,,",
    "ぶら下がる信号機手前,,",
    "写真大都会背,,",
]



newtestdata = []

# 分散表現を用いてキーワードを追加する
# for words in mytestdata:
# newtestdata.append(words)
# for newkeywords in findkeyword(words):
# newtestdata.append(newkeywords)


MAX_SOURCE_LENGTH = args.max_input_length # 入力される記事本文の最大トークン数
MAX_TARGET_LENGTH = args.max_target_length # 生成されるタイトルの最大トークン数

def test(testdata):
    # 推論モード
    trained_model.eval()

    result = []
    for i, inp in enumerate(testdata):
        inputs = [inp]
        batch = tokenizer.batch_encode_plus(
            inputs, max_length=MAX_SOURCE_LENGTH, truncation=True,
            padding="longest", return_tensors="pt")

        input_ids = batch['input_ids']
        input_mask = batch['attention_mask']
        if USE_GPU:
            input_ids = input_ids.cuda()
            input_mask = input_mask.cuda()
```

```

        # 生成処理を行う
        outputs = trained_model.generate(
            input_ids=input_ids, attention_mask=input_mask,
            max_length=MAX_TARGET_LENGTH,
            # temperature=1.0, # 生成にランダム性を入れる温度パラメータ
            # num_beams=10, # ビームサーチの探索幅
            # diversity_penalty=1.0, # 生成結果の多様性を生み出すためのペナルティパラメータ
            # num_beam_groups=10, # ビームサーチのグループ
            # num_return_sequences=10, # 生成する文の数
            repetition_penalty=8.0, # 同じ文の繰り返し（モード崩壊）へのペナルティ
        )

        # 生成されたトークン列を文字列に変換する
        generated_bodies = [tokenizer.decode(ids, skip_special_tokens=True,
                                           clean_up_tokenization_spaces=False)
                            for ids in outputs]

        # 生成された文章を出力する
        tmp = []
        with open(’result/result.txt’, mode=’a’) as f:
            f.write(str(i))
            f.write("\t" + inp)
            tmp.append(inp)
            f.write("\t" + generated_bodies[0] + ’\n’)
            tmp.append(generated_bodies[0])
        result.append(tmp)
    return result

result = test(mytestdata)
newtestdata2 = []
for i in result:
    newtestdata2.append(findkeyword2(i[1],i[0]))

test(newtestdata2)
```