Consistency scores in text data *

Ke-Li Chiu *University of Toronto*
Rohan Alexander *University of Toronto*

In this paper we introduce a process to clean the text extracted from PDFs using various methods from natural language processing. Our approach compares originally extracted text with the text generated from, or expected by, these methods using earlier text as stimulus. To guide this process, we introduce the notion of a consistency score, which refers to the proportion of text that is unchanged by the model. This is used to monitor changes during the cleaning process, and to compare the messiness of different texts. The methods that we consider are: n-grams, Word2Vec, GloVe, and the GPT models. We illustrate our process on text from the Canadian Hansard and introduce both a Shiny application and an R package to make our process easier for others to adopt.

Keywords: text-as-data; natural language processing; quantitative analysis.

Introduction

When we think of quantitative analysis, we may like to think that our job is to 'let the data speak'. But this is rarely the case in practise. Datasets can have errors, be biased, incomplete, or messy. In any case, it is the underlying statistical process of which the dataset is an artifact that is typically of interest. In order to use statistical models to understand that process, we typically need to clean and prepare the dataset in some way. This is especially the case when we work with text data. But this cleaning and preparation requires us to make many decisions. To what extent should we correct obvious errors? What about slightly-less-obvious errors? Although cleaning and preparation is a necessary step, we may be concerned about the extent to which have we introduced new errors, and the possibility that we have made decisions that have affected, or even driven, our results.

In this paper we introduce the concept of consistency in a text corpus. Consistency refers to the proportion of words that are able to be forecast by a statistical model, based on preceding words and surrounding context. Further, we define internal consistency as when the model is trained on the corpus itself, and external consistency as when the model is trained on a more general corpus. Together, these concepts provide a guide to the cleanliness and consistency of a text dataset. This can be important when deciding whether a dataset is fit for purpose; when carrying out data cleaning and preparation tasks; and as a comparison between datasets.

To provide an example, consider the sentence, 'the cat in the...'. A child who has read this book could tell you that the next word should be 'hat'. Hence if the sentence

*Thank you to **X, Y and Z** for helpful comments. A Shiny application for interactive and small-scale examples is available: https://kelichiu.shinyapps.io/Arianna/. An R package for larger scale applications is available: **HERE**. Our code and datasets are available: https://github.com/RohanAlexander/consistency_scores_in_text_datasets. Comments on the 20 August 2020 version of this paper are welcome at: rohan.alexander@utoronto.ca.

was actually 'the cat in the bat', then that child would know something was likely wrong. The consistency score would likely be lower than if the sentence were 'the cat in the hat'. After we correct this error, the consistency score would likely increase. By examining how the consistency scores evolve in response to changes made to the text during the data preparation and cleaning stages we can better understand the effect of the changes. Including consistency scores and their evolution when text corpuses are shared allows researchers to be more transparent about their corpus. And finally, the use of consistency scores allows for an increased level of automation in the cleaning process.

We consider various natural language processing methods. These include n-grams, Word2Cec, GloVe, GPT models and BERT. The n-gram approach involves identifying two, three, or more, words that are commonly found together. Here think of a two-gram involving the word 'good' 'good morning'. At scale these can identify missing or unusual words, and work quickly, but they lack nuance. For instance, an equally reasonable two-gram involving the word 'good' is 'good work'. For that reason, we consider pre-trained word embedding models. These include Word2Vec and GloVe, which place each word in a multi-dimensional space such that distance between words can illustrate their relationship. We also consider pre-trained generative models such as GPT-2 and GPT-3 and BERT. GPT-2, GPT-3 and BERT are pre-trained generative unsupervised language models. GPT-2 and GPT-3 are the two generations of the same model that differ in the number of parameters. GPT-2 has 1.5 billion parameters and GPT-3 has more than 175 billion parameters. While GPT-2 is more limited, it can be run on smaller machines, whereas GPT-3 cannot and is accessed via an API. BERT has 340 millions parameters which is smaller than both GPT-2 and GPT-3. BERT also differs from GPT models because its encoding process is bidirectional instead of unidirectional.

We apply our approach to the Canadian Hansard. The Canadian Hansard was digitised by Beelen et al. (2017). This was a process by which PDFs were scanned, put through optical character recognition (OCR), and then corrected to a reasonable extent. The dataset is extensive, fit-for-purpose, and, appropriately, it has been used considerably, for instance Rheault and Cochrane (2020). However, there are many issues with the dataset in terms of reading it. We focus on one particular year - **(Rohan pick one)** - and show that our approach can be used to relatively quickly improve the quality of the dataset in a reproducible and consistent manner.

The remainder of our paper is structured as follows: **(Rohan add structure)**. Additionally, we construct a Shiny app available at: **(Rohan add link)**. That app computes internal and external consistency scores for corpus excerpts, and have developed an R Package that allows our approach to be used on larger datasets, which is available at: **(Rohan add link)**.

Background

Language Model

Given the nature of human languages, some combinations of words tend to occur more frequently than others. Think of 'good', which is more often followed by 'morning', than 'duck'. As such, we could consider English text production as a conditional probability,

$\Pr(w_k|w_1^{k-1})$, where k is the number of words in a sentence, w_k is the predicted word, and w_1^{k-1} is the history of the word occurring in a sequence (Brown et al., 1992). In this way, the generation of some prediction, w_k, is based on the history, w_1^{k-1}. This is the underlying principle of all language models. Essentially, a language model is a probability distribution over sequences of words. The goal of statistical language modeling is to estimate probability distributions over different linguistic units — words, sentences, and even documents (Bengio et al., 2003). However, this is difficult as language is categorical. If we consider each word in a vocabulary as a category, then the dimensionality of language becomes large (Rosenfeld, 2000). The reason that there is such a variety of statistical language models is that there are various ways of dealing with this fundamental problem.

PDF text extraction and n-grams

The Portable Document Format (PDF) is typically used to render documents in a relatively fixed way. PDFs contain a lot of information, but often that information is not able to be analysed directly by statistical language models. The information first needs to be extracted from the PDF, often by optical character recognition (OCR), and then cleaned, prepared, and made into some type of dataset. One issue in this process is that OCR is not perfect. Hence there is a need to correct the output before it is analysed. One way to do this is to correct errors one-by-one and independent of context. We all know that 'tbe' is wrong and should be 'the'. But statistical language models have the advantage of correction that can occur within context (Rosenfeld (2000)). This allows significant improvements, for instance sometimes 'bat' should be 'hat' and sometimes it really should be 'bat'.

The foundation of an n-gram language model is the conditional probability set-up introduced above. An n-gram model is a probabilistic language model that predicts the next word in a sequence of words (Bengio et al. (2003)). The n in n-gram refers to the number of words in that sequence. Consider the following excerpt from *Jane Eyre*: 'We had been wandering'. 'We' is a uni-gram, 'We had' is a bi-gram, 'We had been' is a tri-gram, and 'We had been wandering' is a 4-gram. Notice that the two tri-grams in this excerpt, 'We had been', and 'had been wandering', overlap. The use of n-gram models enables us to assign probabilities to both the next sequence of words and just the next word. For instance, consider the two sentences: 'We had been wandering', and 'We had been wangling'. The former is likely to be more frequently encountered in a training corpus. Thus, an n-gram would assign a higher probability to the next word being 'wandering' than 'wangling', given the sequence 'We had been'.

To predict the next word, we have to take the sequence of preceding words into account, which requires knowing the probability of the sequence of words. The probability of a sequence appearing in a corpus follows the chain rule:

$$\Pr(\text{We,had,been,wandering}) = \Pr(\text{We}) \times \Pr(\text{had} \mid \text{We}) \times \Pr(\text{been} \mid \text{We,had}) \times \Pr(\text{wandering} \mid \text{We,had,bee}$$

However, the likelihood that more and more words will occur next to each other in an identical sequence becomes smaller and smaller, making prediction difficult. Alternatively, we can approximate the probability of a word depending on only the previous word. This is known as the 'Markov assumption' and it allows us to approximate the probability using only the last n words (Brown et al., 1992):

$$\Pr(\text{We,had,been,wandering}) \approx \Pr(\text{We}) \times \Pr(\text{had} \mid \text{We}) \times \Pr(\text{been} \mid \text{had}) \times \Pr(\text{wandering} \mid \text{been}).$$

As a bi-gram model only considers the immediately preceding word, under the Markov assumption, an n-gram model can be reduced to a bi-gram model with n being any number:

$$\Pr(w_n|w_1^{n-1}) \approx \Pr(w_n|w_{n-1})$$

Language models underpinned by n-grams are widely applied in text prediction, spelling-correction, and machine translation (Brown et al., 1992). In our application, we use a tri-gram based model to detect both real-word and non-word errors and correct them with the candidate word that has the highest probability.

However, n-gram models do not take the linguistic structure of language into account. For instance, Rosenfeld (2000, p. 1) discusses language in this context, saying that '...it may as well be a sequence of arbitrary symbols, with no deep structure, intention or thought behind'. The prediction of the next word is based on only a few preceding words and broader context is not taken into account. Hence next-word prediction using n-gram based language models can be limited.

There are a variety of ways to implement an n-gram model within R R Core Team (2019) including using R packages such as `Quanteda` (Benoit et al., 2018), `tidyText` (Silge and Robinson, 2016) and `tm` (Feinerer and Hornik, 2019). We used the Text Analysis Utilities, `tau`, R package Buchta et al. (2019) because of we can separate text-cleaning and n-gram making in two different steps. We want to have control to manually clean the text because we want to add an end-of-sentence token to each sentence int he corpus. To illustrate the use of `tau`, we will first install and load the library, and create a small corpus.

```
install.packages("tau")
```

```
library(tau)
corpus <- "There was no possibility of taking a walk that day. We had been
wandering, indeed, in the leafless shrubbery an hour in the morning; but
since dinner (Mrs. Reed, when there was no company, dined early) the cold
winter wind had brought with it clouds so sombre, and a rain so penetrating,
that further out-door exercise was now out of the question."
```

Our purpose is to construct tri-grams and so we first split the corpus into sentences by period, exponential mark, or question marks.

First, we remove all punctuation and other non-letter characters, and convert all text to lower case.

```
tokenizer <- function(corpus) {
  lines <- vector()
  # Read the corpus line by line
  for (line in corpus) {
    # add two start-of-sentence tokens to the beginning of a sentence
    lines <-c(lines, line)
  }
  lines <- tolower(lines)
  # add end-of-sentence token to the end of a sentence
  lines <- gsub("[.!?]$|[.!?] |$", " </s>", lines)
  tokens <- unlist(strsplit(lines, "[^a-z<>/]"))
  tokens <- tokens[tokens != ""]
  return(tokens)
}
```

We will use the `textcnt()` function to turn the training corpus into tri-grams. The most important argument of this function is *corpus*, as well as *n* to set the number of words in a sequence, *method* to determine which type of counts to compute, and *split* to designate the regular expression pattern for splitting. The function outputs a `textcnt` object, which is essentially a table with each tri-gram and their counts (Buchta et al., 2019).

```
trigrams <- textcnt( tokens, n=3, method = "string",
                     split = "[[:space:][:digit:]]+", decreasing = TRUE)
```

We then manually added an end of sentence token, `</s>`, to the end of each sentence.

```
trigrams <- trigrams[!grepl("</s>", names(trigrams))]
```

At this point it is possible to take other measures around fine-tuning the tri-grams. For instance we may like to remove the tri-grams that contain an end-of-sentence token. This would mean that tri-grams that include a word from a new sentence will be removed. For example, the tri-gram 'day we' will be removed because it has words from two different sentences. Table 1. shows four sample tri-grams from the small sample corpus after this process.

there was no	a rain so	a walk that	an hour in
2	1	1	1

Table 1. Four sample tri-grams generated from the corpus

The tri-grams can be accessed by calling the names of this table object and the counts can be used to calculate probabilities. There are a variety of options around constructing these probabilities. One is Kneser–Ney smoothing, which is the smoothing technique recommended by Chen and Goodman (1999) to improve the accuracy of n-gram models.

Finally, it is helpful to create a data table that contains the first two words and the last of each tri-gram in separate column and the smoothed probability in another column. For instance, Table 2. provides and example of this set-up.

First Words	Last Word	Probability
there was	no	0.6278947

Table2. The smoothed probability table

Word embedding models

Although, from an implementation perspective, there is an inherent numerical representation of language in n-gram models, it is not overly tractable. The numeric representation typically relies on each term being represented in a vector, which can quickly result in quite high-dimensional objects. In contrast, word embedding models represent terms by vectors of real numbers where each vector is a 'feature', hence the final object has considerably lower dimensionality (Bengio et al., 2003).

This approach has a long history, but the implementation of Bengio et al. (2003) has become the foundation for much subsequent work. Here, terms are encoded into feature vectors that represent different aspects of the term, and so each term is represented by a position in the vector space. Terms that are used in similar ways will have similar representations in the vector space. This will be due to their co-occurrence in usage. For example, 'cat' will be closer to 'dog' than to 'car', because we are more likely to talk about cats and dogs, than cats and cars. See how they might be close when considered along a 'pets' dimension, but far when considered along a 'related-to-tigers' dimension.

Terms that are often used in similar contexts will have similar vector representations. Hence, word embedding models can identify similar terms, calculate the similarity between a pair of terms, and find the odd term in a group of terms (Mikolov et al., 2013a). There are a variety of ways to achieve this dimensionality reduction, including Word2Vec and GloVe.

Word2vec

Word2vec is a word embedding model that uses a neural network to learn word associations (Mikolov et al., 2013a) (Mikolov et al., 2013b). It takes a corpus as an input and outputs a set of feature vectors that allow the terms in that corpus to be represented. Mikolov et al. (2013a) proposed two architectures for determining this distributed representation of terms: continuous bag-of-words (CBOW), and continuous skip-gram. Both have an input layer, a projection layer, and an output layer. But CBOW uses several surrounding words, or the context, to predict a target word. On the other hand, continuous skip-gram takes a single word as input to predict its neighbouring words.

To illustrate the difference, consider the quote from Descartes, 'I think therefore I am'. The focus word for which we want to learn the embedding is 'therefore', and the context window is set to two. This means that we consider the two words immediately preceding and proceeding 'therefore'. The CBOW input is 'I, think, I, am', and the output is 'therefore'. In contrast, the continuous skip-gram input is 'therefore', while 'I, think, I, am' is the output. The input layer encodes terms within a context window to one-hot vectors. Using a projection matrix, the projection layer then project the discrete one-hot vectors to lower-dimension continuous vectors. Finally, the output layer is a softmax regression classifier that obtains the posterior distribution of words (Rong, 2014). It is the input and output vectors that differ between the two architectures.

Word2Vec is trained on the Google News dataset (Google, 2013). This contains six billion tokens and the vocabulary size is restricted to the one million most common words (Mikolov et al., 2013a). As this is a large dataset, it provides an accurate statistical pattern of language, as it is used in this dataset, and a comprehensive mapping between terms and vector positions.

To the extent that the relative positions of terms reflects their semantic relationship, Word2Vec will perform well for word analogies. However, terms that have context-specific meaning may not perform as well.

There are a few packages for implementing a word2vec model in R, including `rword2vec` (Chaware, 2020) and `word2vec` (Wijffels, 2020). Both packages use the same algorithm. Another way to implement a word2vec model in R is through Keras which is an API for deep learning written in Python (Chollet et al., 2015) for which the `keras` package provides an R interface (Allaire and Chollet, 2020).

Here, we follow Falbel (2017) to implement a skip-gram model of Word2Vec through Keras. There are three steps in the Keras deep learning framework: 1) prepare the data; 2) define the model; and 3) train and evaluate the model. **(Ke-Li: Can we change this to be a different dataset please?)** Following these steps, we first downloaded the Google News dataset (Google, 2013) and preprocess the text into tokens.

```
tokenizer %>% fit_text_tokenizer(training_corpus)
```

We then define a model that mimics the skip-gram algorithm of word2vec using the function `skipgrams_generator()`.

```
skipgrams_generator <- function(text, tokenizer, window_size, negative_samples) {
  gen <- texts_to_sequences_generator(tokenizer, sample(text))
  function() {
    skip <- generator_next(gen) %>%
      skipgrams(
        vocabulary_size = tokenizer$num_words,
        window_size = window_size,
        negative_samples = 1
      )
    x <- transpose(skip$couples) %>% map(. %>% unlist %>% as.matrix(ncol = 1))
    y <- skip$labels %>% as.matrix(ncol = 1)
    list(x, y)
  }
}
```

The embedding dimensions, the size of the context window and the number of samples are configurable within this function.

(Ke-Li: Add code and steps for embedding layers preparation.)

Finally, we use the `fit_generator()` function to train the model.

```
model %>%
  fit_generator(
    skipgrams_generator(training_corpus, tokenizer, skip_window, negative_samples),
    steps_per_epoch = 10, epochs = 1
  )
```

What's supposed to happen: As a result, a embedding matrix is obtained and serves as the look-up table of the word vectors. **What really happened:** However, the training process was stuck in the operation of `fit_generator()` function. According to Stack Flow this might be an issue of incompatible versions between Keras and Tensorflow. More investigation is needed.

GloVe

The Global Vectors (GloVe) model is a pre-trained word embedding model that emphasises the semantic meaning of words (Pennington et al., 2014). In the Word2Vec model, word representations are learnt locally within the context window. On the other hand, GloVe considers the properties of the corpus globally. As the model considers the context of an entire corpus the co-occurrence probabilities are based on counts for the entire training dataset.

GloVe additionally considers the relative probability of word co-occurrence. For instance, consider the two words 'ice' and 'steam'. GloVe considers their co-occurrence with a variety of probe words, k. For instance k could be 'solid', which is a property of ice; 'gas', which is a property of steam; and 'water', which is related to both.The ratio of co-occurrence probability is:

$$\frac{\Pr(k|\text{ice})}{\Pr(k|\text{steam})}.$$

The ratio between the co-occurrence probability of the 'ice, solid' pair and the 'steam, solid' pair is high because the co-occurrence of 'ice' and 'solid' is higher than the co-occurrence of 'steam' and 'solid'. Conversely, the ratio between the co-occurrence of the 'ice, gas' pair and the 'steam, gas' pair is low. On the other hand, the ratio of the co-occurrence probability of the 'ice, water' pair and the 'steam, water' pair us close to one because water is relevant to both 'ice' and 'steam'.

As Hanretty et al. (2018) point out, although word embedding models can capture semantic meanings of words, "they are context-free and fail to capture higher-level concepts in context, such as polysemous disambiguation, syntactic structures, semantic roles, anaphora". Word embedding models do not represent a homonyms that has difference meanings in different vectors depending on the context. For example, "squash" can be a type of hard shelled vegetable or can be referring to a racket and ball sport, but there is only one series of vectors to represent the word "squash", instead of dynamically changing the vectors depending on the context. Moreover, from our observation, the word embedding models are unable to handle novel words that are not included in the vocabulary of the training corpus; for this reason, we conclude that the generalizability of word embedding models is sub-optimal.

There are a variety of ways to implement GloVe word embedding models in R. One is to load GloVe embeddings into a frozen Keras Embedding layer (Falbel et al., 2018) using the `Keras` R package (Falbel et al., 2020). This means **(Ke-Li: Add what it means please.)** Another way, is to use the `text2vec` package (Dmitriy Selivanov, 2020).

Here we follow **CITE** to implement GloVe using the `keras` R package (Falbel et al., 2018). The blog post is aimed to create a word embedding model for text classification. However, we will only use the guide create a embedding index and skip the steps of text classification. Instead of building a model for word embedding from scratch as we did for the skip-gram model of Word2Vec, here we are merely loading the Glove model and its pre-trained word vectors as an index. The word embedding index is then saved to an R environment.

[As before, we need code and examples here]

Pre-trained transformer language models

In recent years, the use of pre-trained language models has played a significant role in advancing natural language processing (NLP) tasks such as reading comprehension, text generation, and even creative writing (Brown et al., 2020). Pre-trained models remove the need for researchers to prepare training datasets or allot computational resources to the stage of pre-training. Because of this, there has been substantial development in NLP research. In this section we discuss the GPT models from OpenAI and the BERT model from Google.

GPT

OpenAI is an AI research company who openly pursues Artificial General Intelligence — to have machines learn and understand any intellectual tasks as human do (Brundage et al., 2018). OpenAI's Generative Pre-Trained Transformer (GPT) models are pre-trained language models that are built based on the transformer architecture. Proposed by Vaswani et al. (2017) from Google in 2017, the Transformer is a novel network architecture for neural network that outperforms RNN-based and CNN-based models in computational efficiency (Vaswani et al., 2017). Since the emergence of the Transformer model, most of the representative pre-trained models are built based on this architecture (Hanretty et al., 2018). Therefore, we can say that Transformer marks a new era of language models.

As of August 2020, OpenAI has three generations of GPT models. GPT was introduced to the public in June, 2018. In the year 2019, its successor GPT-2 was released in three stages. Not long after, in June 2020, OpenAI announced GPT-3, which is the latest GPT model as of July 2020 (Brown et al., 2020). Since the API of GPT-3 at the moment is only available to a few invited users, public applications of GPT-3 are limited. On the other hand, its predecessor, GPT-2, has enabled researchers made breakthroughs in the past years. Alt et al. (2019) extended GPT-2 to relation extraction, which is a task of identifying the relationship between concepts appeared in text. They had shown a milestone in predicting larger range of distinct relation types with higher confidence (Alt et al., 2019). In the field of speech recognition, data scarcity is often an issue due to the difficulty to collect large amount of data from human for learning (Bird et al., 2020). Bird et al. (2020) employed GPT-2 as part of the solution to generate augmented data for classification. GPT-2 also helped NLP researchers tackling the task of textual paraphrasing. Hegde and Patil (2020) used GPT-2 to build an unsupervised paraphrasing model that is not restricted by domain shift within an article, which is a problem faced by the usual supervised models of paraphrasing.

Most of prior language models are trained in a single domain by supervised learning to perform a single task. OpenAI, on the other hands, train the GPT models to perform many tasks (Radford et al., 2019). GPT-2 is pre-trained on WebText, a comprehensive dataset that is scrapped from millions of webpages and contains total of 40 GB of text (Radford et al., 2019). WebText contains all kinds of textual data: dialogues, comments, translations, questions and answers…etc. Such diversity is critical for GPT-2 to be a generalized model that can perform multiple tasks with any domain boundary. However, GPT-2 has the capacity to adapt to a context and generates custom output with the "fine-tuning" technique (Ziegler et al., 2019). While the pre-training stage is unsupervised and mandatory, fine-tuning is an optional training stage where GPT can learn to perform new tasks in a supervised or reinforced approach such as stylistic continuation and summarization (Ziegler et al., 2019). During the pre-training stage, the model is trained to perform text completion task. Give it a prompt to start and it will generate texts that continues what has been started. In the fine-tuning stage, Radford et al. (2019) provided additional training datasets for GPT-2 to generate the texts with specific sentiment, or be descriptive, or even summarize the prompt text.

OpenAI has made GPT-2 open-sourced through Github with Python and Tensorflow (Abadi et al., 2015). There is also a package in R that wraps the Python code into a R package, created by Keydana and Luraschi (2019). There are three choices of models in terms of parameter sizes — "124M", "355M" and "774M". The R package only permits exploratory experiments with the pre-trained tasks; fine-tuning for new tasks is not available thorough R. To proceed with the implementation of GPT-2 in R, theoretically, we will fine-tune the model through Python and create a wrapper code for R. This could possibility be achieved by employing the `reticulate` R package (Allaire et al., 2017) that allows operation of Python code in R. The practicality of this approach is to be further investigated.

GPT-3 has over 175 billions of parameters (Brown et al., 2020). In the past few years, the growth of parameters of transformer language models expands rapidly. The first generation GPT has 110 million parameters. Released in 2018, GPT-2 has 1.6 billion parameters, and within two years, GPT-3 has grown the size to 175 billion, which is over 100 times than its predecessor. The more parameters there are the better the model can generalize to new tasks; therefore, the increasing amount of parameters accelerates the improvements in text synthesis and downstream NLP tasks (Brown et al., 2020). GPT-3 can learn NLP tasks with just a handful of example; OpenAI refers this ability as few-shot learning, or in-context learning (Brown et al., 2020). As a result, we only need to provide

GPT-3 a few demonstrations for it to learn a new NLP task. For example, OpenAI team trained GPT-3 to generate "news articles". The prompt input is a plausible first sentence of a news story written by human, and the output is an entire news article generated by the model itself. Without the demonstrations, GPT-3 tends to treat the prompts as "tweets" and generate texts that serve as responses or follow-up tweets. The team fed a few previous news articles in the model's context as demonstrations, and the model learned to produce the outputs that adapt to the style and length of news stories (Brown et al., 2020). This few-shot learning ability indicates that the model can learn like human do and would democratize the use of language models to broader fields.

BERT

BERT stands for Bidirectional Encoder Representations from Transformers (Devlin et al., 2018). It was released by researchers at Google AI Language in 2018. The key innovation of BERT is its application of bidirectional training of the Transformer model, which results in a deeper sense of language context and flow. What is bidirectional training? In the context of English language, a sentence is constructed from left to right. The context on the left is used to determine the next word on the right. In this sense, GPT models are unidirectional since the probability of the next word is dependent only on the past. It is also possible to reverse the direction and model the sentence from right to left, and BERT incorporates both directions of left-to-right and right-to-left to model bidirectional contexts.

BERT is pre-trained by using two unsupervised tasks. The first task is Masked Language Model and and second task is Next Sentence Prediction. The bidirectional approach of BERT is alleviated by the "Masked Language Model" (Devlin et al., 2018). The Masked Language Model masks random words in the input corpus which forces the model to predict the original words by "filling in the blank". This training process blends the representation from the left and right context and therefore increases the model's context sensitivity (Nozza et al., 2020). In the Next Sentence Prediction process, the model receives two sentences in sequence as input. 50% of the sentences are paired in the right sequence and the ones from the other half have sentences that are paired randomly (Devlin et al., 2018). The goal of this process is to have BERT evaluate if the second sentence is the next sentence of the first sentence in the original text and sharpen its capacity to predict the likelihood of the next sentence.

BERT was pre-trained on BooksCorpus dataset with 800 million words and English Wikipedia with 2,500 million words (Devlin et al., 2018). Because BERT employs the Transformer architecture with the self-attention mechanism, fine-tuning is also available with one additional output layer and a task-specific input (Devlin et al., 2018). Google AI researchers have fine-tuned BERT for 11 NLP tasks including question-answering and common-sense inference.

BERT implementation in R is also possible through Keras. HG (2018) created a python module called `keras-bert` and the module can be called by R through `reticulate` package (Allaire et al., 2017), which enables the operation of Python modules in R. To start, we install the keras-bert model in Python on the machine and load BERT model into R through `reticulate`. Once `keras-bert` is loaded, we can use it to fine-tune BERT on a task-specific training dataset. Fine-tuning is started by tokenizing the training data. The output of the tokenization is then fed to BERT as the input. The next step is to define the model by configuring its parameters such as batch size, number of epochs and the learning rate. Finally, the model is compiled and we can begin the fine-tuning.

For the task of text cleaning, fine-tuning is not required. The process of text cleaning involves detecting and correcting mis-spelled words. Correcting the incorrect words can be seen as predicting the correct words. The Masked Language Model of BERT can be employed to do such prediction. In the pre-training process, the Masked Language Model would mask random words to do the prediction. However, we can tell the model what we want to "mask" by assigning the words to be encoded as `[MASK]`. Consider the erroneous sentence that has ten words:

> "There was no possibility of taking a wlak that day."

Using a spell-checker, "wlak" is identified as an error and is substituted by `[MASK]`:

> "There was no possibility of taking a [MASK] that day."

The implementation of text correction with BERT in R with the `keras-bert` Python module is not yet completed. For the purpose of demonstrating how BERT predicts the missing words, we will show the implementation process in Python using the `pytorch_pretrained_bert` library. The reason to use `pytorch_pretrained_bert` instead of `keras-bert`is that `pytorch_pretrained_bert` has relatively comprehensive functions that makes the prediction of missing words more straightforward.

After we retrieved the new text data with the error replaced by [MASK], we use the tokenizer function in `pytorch_pretrained_bert` to tokenize the text:

```
tokenized_text = tokenizer.tokenize("There was no possibility of taking a [MASK] that da
tokenized_text
>>> ['there', 'was', 'no', 'possibility', 'of', 'taking', 'a', '[MASK]', 'that', 'day',
```

We then get the index of `[MASK]` within the sentence and save it to a list. Since we only have one [MASK] token, the list has only one element:

```
mask_index = 7
```

`pytorch_pretrained_bert` has a dictionary of tokens generated from the pre-training stage. The tokens include the vocabulary, punctuations and special symbols. Each token in the dictionary is given an index, represented as {token: index}. The function `convert_tokens_to_ids()` encodes the tokens we got previously to their positional indices in the tokens dictionary and collects them into a list:

```
indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)
indexed_tokens
>>> [2045, 2001, 2053, 6061, 1997, 2635, 1037, 103, 2008, 2154, 1012]
```

The next step is to encode the text by its segments. The model encodes a sentence of representing each word in the sentence with the positional index of the sentence in the provided corpus. For example, the example text has only one sentence "There was no possibility of taking a [MASK] that day." Therefore, the position of the sentence is "0", and the encoding for each word in the sentence, including the period is a series of "0" to the length of the sentence:

```
segments_tokens = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The previous two steps give us two set of encoding — `indexed_tokens` that contains the encoding for each token, and `segments_tokens` that contains the encoding for each sentence. These two encoding layers that we feed to BERT Masked Language Model. The model takes tensor objects as inputs, therefore we transformed the two lists into tensor objects beforehands. These two tensor objects has two dimensions. The first dimension is the container, and the second dimension is the length of the token lists.

```
tokens_tensor = torch.tensor([indexed_tokens])
tokens_tensor

segments_tensor = torch.tensor([segments_encodes])
segments_tensor
>>> [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

The encoding layers can be represented as the following:

	'there'	'was'	'no'	'possibility'	'of'	'taking'	'a'	'[MASK]'	'that'	'day'	'.'
token	2045	2001	2053	6061	1997	2635	1037	103	2008	2154	1012
segment	0	0	0	0	0	0	0	0	0	0	0

Table2. The smoothed probability table

We then load the pre-trained BertForMaskedLM and feed the two tensor objects to it. The model then returns a tensor object that has three dimensions. The `predictions` tensor objects contains the prediction score of all the words in the vocabulary.

```
model = BertForMaskedLM.from_pretrained('bert-base-uncased')
predictions = model(tokens_tensor, segments_tensor)
predictions
```

To get the prediction, we will get the element with the highest prediction score and retrieve its index. The retrieved index is then used to find the corresponding token in the tokens dictionary. The predicted word is "walk", which matches with the original sentence in **Jane Eyre**.

```
predicted_word_index = torch.argmax(predictions[0, mask_index]).item()
predicted_word_index
>>> 3328
predicted_word_token = tokenizer.convert_ids_to_tokens([predicted_index])[0]
predicted_word_token
>>> walk
```

Stylized example

(TODO: This stylized example need to be changed and updated to actually use GPT-3 at some point.)

In our application, we deploy OpenAI GPT-3 to generate text and ...

As a stylized example, let's consider the following actual paragraph from Jane Eyre, by Charlotte Bronte:

> There was no possibility of taking a walk that day. We had been wandering, indeed, in the leafless shrubbery an hour in the morning; but since dinner (Mrs. Reed, when there was no company, dined early) the cold winter wind had brought with it clouds so sombre, and a rain so penetrating, that further out-door exercise was now out of the question.

Let's pretend that this text had been created from optical character recognition and that it had the following errors: some 'h' were replaced with 'b'; and some 'd' have been replaced with 'al':

> There was no possibility of taking a walk that day. We had been wandering, indeed, in tbe leafless shrubbery an hour in tbe morning; but since dinner (Mrs. Reeal, when tbere was no company, dined early) the cold winter wind had brought with it clouds so sombre, and a rain so penetrating, tbat further out-door exercise was now out of the question.

Assume a model that is trained to perfectly forecast the next word in Jane Eyre. For this fragment there are 62 words, comprising 5 errors and 57 correct words. So the internal consistency score of this fragment would be: $57/62 = 0.919$. When we recognise and correct the errors, this consistency score would increase to 1.

Similarly, assume a model that is trained on an external data source. This means that it will recognise the the cases where some 'h' were replaced with 'b', but not recognise that 'Reed' has become 'Reeal'. Hence, the external consistency score would be $58/62 = 0.935$.

Data

Various data can be used....

Application

Discussion

Internal validity vs external- one is their own words the other is a general set of words.
In the same way that precision and recall provide important measures…

Appendix

Literature summaries

(These are just here for now - we'll remove later.)

Article: Improving Language Understanding by Generative Pre-Training (2018)

Introduction of a semi-supervised approach for language understanding machine learning tasks. The approach is the combinaiton of unsupervised pre-training and supervised fine-tuning.

Article: Language Models are Few-Shot Learners (2020)

Official paper from OpenAI to introduce OpenAI GPT-3. Emphasizes on the "fewer-shots" and "task-agnostic" aspects of the latest langual model compared to its predecessor GPT-2.

Discussion of how GPT-3 works

(Ke-Li - it would be good for us to go through how it actually works in as much detail as we can manage (in our own words). This is mostly just to force us to learn how it works and may be taken out of the final version of the paper, but it's still an important exercise.)

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Allaire, J. and Chollet, F. (2020). *keras: R Interface to 'Keras'*. R package version 2.3.0.0.

Allaire, J., Ushey, K., Tang, Y., and Eddelbuettel, D. (2017). *reticulate: R Interface to Python*.

Alt, C., Hübner, M., and Hennig, L. (2019). Fine-tuning pre-trained transformer language models to distantly supervised relation extraction. *arXiv preprint arXiv:1906.08646*.

Beelen, K., Thijm, T. A., Cochrane, C., Halvemaan, K., Hirst, G., Kimmins, M., Lijbrink, S., Marx, M., Naderi, N., Rheault, L., et al. (2017). Digitization of the canadian parliamentary debates. *Canadian Journal of Political Science/Revue canadienne de science politique*, 50(3):849–864.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Benoit, K., Watanabe, K., Wang, H., Nulty, P., Obeng, A., Müller, S., and Matsuo, A. (2018). quanteda: An r package for the quantitative analysis of textual data. *Journal of Open Source Software*, 3(30):774.

Bird, J. J., Faria, D. R., Ekárt, A., Premebida, C., and Ayrosa, P. P. (2020). Lstm and gpt-2 synthetic speech transfer learning for speaker recognition to overcome data scarcity. *arXiv preprint arXiv:2007.00659*.

Brown, P. F., Della Pietra, V. J., Desouza, P. V., Lai, J. C., and Mercer, R. L. (1992). Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–480.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Brundage, M., Avin, S., Clark, J., Toner, H., Eckersley, P., Garfinkel, B., Dafoe, A., Scharre, P., Zeitzoff, T., Filar, B., et al. (2018). The malicious use of artificial intelligence: Forecasting, prevention, and mitigation. *arXiv preprint arXiv:1802.07228*.

Buchta, C., Hornik, K., Feinerer, I., and Meyer, D. (2019). *tau: Text Analysis Utilities*. R package version 0.0-21.

Chaware, M. (2020). *rword2vec: R Interface to Google's Word2vec*. R package version 1.1.

Chen, S. F. and Goodman, J. (1999). An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394.

Chollet, F. et al. (2015). Keras. https://github.com/fchollet/keras.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dmitriy Selivanov, Manuel Bickel, Q. W. (2020). *text2vec: Modern Text Mining Framework for R*.

Falbel, D. (2017). Rstudio ai blog: Word embeddings with keras.

Falbel, D., Allaire, J., Chollet, F., RStudio, and Google (2018). pretrained_word_embeddings. https://keras.rstudio.com/articles/examples/pretrained_word_embeddings.html.

Falbel, D., Allaire, J., Chollet, F., RStudio, Google, Tang, Y., Bijl, W. V. D., Studer, M., and Keydana, S. (2020). *keras: R Interface to 'Keras'*.

Feinerer, I. and Hornik, K. (2019). *tm: Text Mining Package*. R package version 0.7-7.

Google (2013). Google news dataset. https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit.

Hanretty, C., Lauderdale, B. E., and Vivyan, N. (2018). Comparing strategies for estimating constituency opinion from national survey samples. *Political Science Research and Methods*, 6(3):571–591.

Hegde, C. and Patil, S. (2020). Unsupervised paraphrase generation using pre-trained language models. *arXiv preprint arXiv:2006.05477*.

HG, Z. (2018). *Keras BERT*.

Keydana, S. and Luraschi, J. (2019). Rstudio ai blog: Innocent unicorns considered harmful? how to experiment with gpt-2 from r.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Nozza, D., Bianchi, F., and Hovy, D. (2020). What the [mask]? making sense of language-specific bert models. *arXiv preprint arXiv:2003.02912*.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9.

Rheault, L. and Cochrane, C. (2020). Word embeddings for the analysis of ideological placement in parliamentary corpora. *Political Analysis*, 28(1):112–133.

Rong, X. (2014). word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*.

Rosenfeld, R. (2000). Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278.

Silge, J. and Robinson, D. (2016). tidytext: Text mining and analysis using tidy data principles in r. *JOSS*, 1(3).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Wijffels, J. (2020). *word2vec: Distributed Representations of Words*. R package version 0.2.1.

Ziegler, D. M., Stiennon, N., Wu, J., Brown, T. B., Radford, A., Amodei, D., Christiano, P., and Irving, G. (2019). Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*.