O'REILLY®

Designing Large Language Model Applications

A Holistic Approach

Early Release
RAW & UNEDITED

Suhas Pai

Designing Large Language Model Applications

A Holistic Approach

Suhas Pai

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Designing Large Language Model Applications

by Suhas Pai

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Editors: Michele Cronin and Nicole Butterfield
- Production Editor: Ashley Stussy
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- December 2024: First Edition

Revision History for the Early Release

- 2023-07-07: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098150501 for release details.

978-1-098-15044-0

[FILL IN]

Chapter 1. LLM Ingredients: Training Data

In Chapter 1, we defined LLMs, ruminated on their strengths and limitations, explored current and potential use cases, and presented the scaling laws that seemingly govern progress in this field. In Chapter 2, we dug deep into the trenches to understand the most significant advance in machine learning in recent times, the Transformer architecture which makes modern LLMs possible. Armed with this knowledge, let's set our sights on utilizing these models to build useful applications!

To set the stage for the rest of this book, in this chapter and the next we will discuss the recipe for pre-training LLMs and the ingredients that go into them in detail. We will also take a journey through the LLM landscape and showcase the different pre-trained models available for our use, both open-source and proprietary. We will classify them according to various criteria including training data domain, architecture type, licensing etc.

But wait, this book is about utilizing pre-trained LLMs to design and build user applications. Why do we need to discuss the nuances of pre-training billion parameter models from scratch, something most machine learning practitioners are never going to do in their lives?

Actually, this information is very important because many of the decisions taken during the pre-training process heavily impact downstream performance. As we will notice in subsequent chapters, failure modes are more easily understandable when you have a comprehension of the training process. Just like we appreciate having ingredients listed on packages at our grocery stores, we would like to know the ingredients that go into making a language model before we use it in serious applications.

> **NOTE**
>
> There is not much information available in the public realm about some of the proprietary LLMs that are accessible only through an API. This book will provide as much information as has been made public. While the lack of information doesn't mean that we should avoid using these models, model transparency is something that you might need to take into your calculus while making a final decision regarding what model to use.

Ingredients of an LLM

Let's start with the ingredients that go into making an LLM.

Broadly speaking, we have:

1. Pre-training data - **What's it trained on?** As the old computer science adage 'Garbage In, Garbage Out' comes back to bite us, we will explore popular pre-training datasets and dig into the various pre-processing steps taken to ensure *high-quality* data is fed to the model. We will also showcase some tools that allow us to probe these datasets and understand how pre-training data composition impacts downstream tasks.

2. Vocabulary and tokenizer - **What's it trained over?** In order to build a model over a language, we have to first determine the vocabulary of the language we are modeling, and rules to break down a stream of text into the right vocabulary units (tokenization). Linguistically, humans process language in terms of meaning-bearing words and sentences. Language models process language in terms of tokens. We will explore the downstream impact when there is a mismatch between the two.

3. Learning objective - **What is it being trained to do?** By pre-training a language model, we aim to imbibe the language model with general skills in syntax, semantics, reasoning and so on, that will hopefully enable it to reliably solve any task you throw at it even if it was not specifically trained on it. We will discuss the various tasks (learning objectives) that pre-trained models are trained on. You might wonder if LLMs are better suited to solving downstream tasks that are similar to the tasks the pre-trained model has been trained to solve. We will test this assumption and discuss the impact various learning objectives have on task performance.

4. Architecture - **What's its internal structure?** As mentioned in Chapter 2, most modern language models are based on the Transformer architecture. We will discuss the various architectural backbones- specifically encoder-only models, encoder-decoder models, and decoder-only models, and the rationale used by organizations training LLMs for their choice of architecture type.

Let's look at how these ingredients fit together in (Figure 1-1):

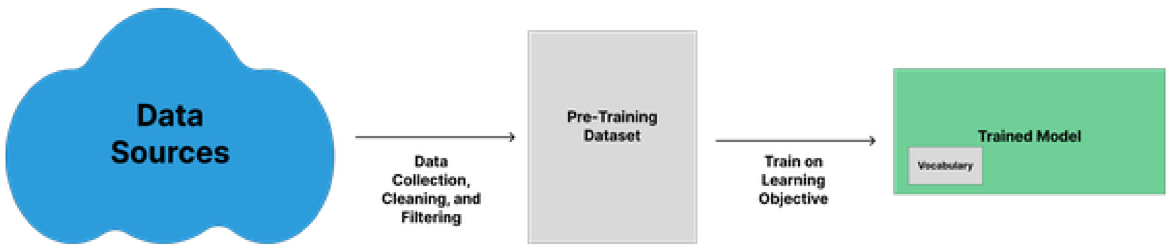

Data Sources
Data Collection, Cleaning, and Filtering
Pre-Training Dataset
Train on Learning Objective
Trained Model
Vocabulary

Figure 1-1. Figure depicting how all the ingredients come together to make an LLM.

The language models trained using the process described in this chapter and the next are called *base models*. Lately, model providers have been augmenting the base model by tuning it on much smaller datasets in order to steer them towards being more aligned with human needs and preferences. Some popular tuning modes are:

- Supervised instruction fine-tuning, so that the model is better at following human instructions.
- RLHF (Reinforcement Learning by Human Feedback), so that the model is better aligned with human preferences.
- Domain-adaptive or task-adaptive continued pre-training, so that the model is better attuned to specific domains and tasks.

to name a few. Based on the specific augmentation carried out, the resulting models are called *instruct models*, *chat models* and so on.

We will cover instruct and chat models in Chapter 6, and domain/task-adaptive pre-training in Chapter 8.

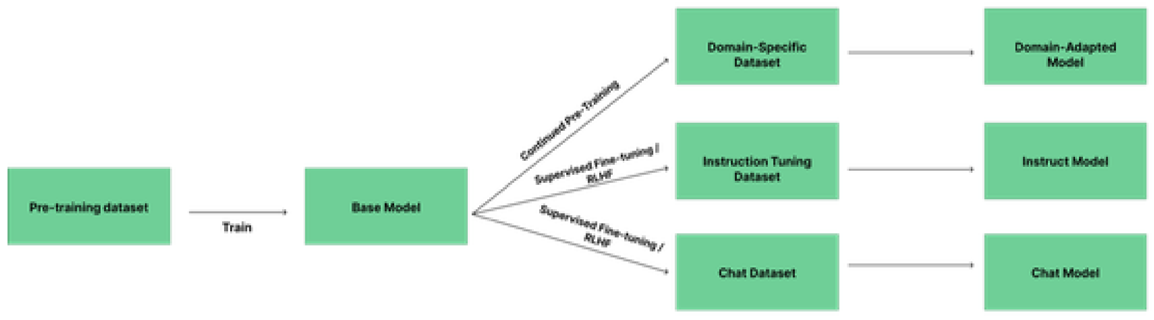


Figure 1-2. Figure showing the relationship between base models and their derivatives.

LLM PRE-TRAINING CHALLENGES

Pre-training an LLM is a very technically challenging task, and requires a lot of computational resources and exceptional technical skills. For example, GPT-4's technical report credits 343 unique contributors, not including the annotators in Kenya who contributed to their RLHF (Reinforcement Learning with Human Feedback) training. Delving into every aspect of pre-training LLMs is an entire book in itself. In this chapter we will not focus on infrastructure or engineering considerations for pre-training LLMs, nor focus on the nuances of distributed and parallel computing. We will instead focus on aspects of the pre-training process that can directly impact your application's behavior and performance.

However, if you are curious to read more about the challenges involved in pre-training LLMs, here are some useful resources to quench your thirst -

- Blog post from Big Science that explains the hardware, types of parallelisms employed, and optimizations used in training BLOOM, an open-source 176B parameter multilingual model.
- Training chronicles(log book) from BLOOM and OPT, which is a 175B parameter LLM released by Meta, documenting the trials and tribulations faced during training, including hardware failures and how to recover from them, training instabilities, loss spikes and the like.
- Video featuring Susan Zhang, the lead author of OPT, who discusses the OPT chronicles in detail.
- The Deep Learning Tuning book by Google, which discusses hyperparameter optimization, multi-host setups, training instabilities and a lot more.

Pre-training data requirements

Although it has been shown that higher capacity models are relatively more sample efficient, in general neural networks are very sample inefficient, meaning they need tons of examples to learn a task. It is infeasible to create such a large supervised dataset with human annotations, hence the predominant means to pre-train language models is using *self-supervised* learning, where the target labels exist within your training inputs.

Using this setup, virtually any type of text is fair game to be included in a pre-training dataset, and theoretically any non-textual signal can be encoded in text and included as part of a pre-training dataset.

From our scaling laws discussion in Chapter 1, we know that most current language models are severely undertrained and can benefit from additional performance gains by just training them longer and on more data. Also, as discussed in Chapter 1, the *consolidation effect* at play in the field raises expectations on what a single language model is expected to do end-to-end. Today a single model is expected to answer factual questions about the world, employ arithmetic and logical reasoning, write code, and come up with creative ideas.

All this means that the data needs for language model pre-training are enormous. Now, the key question is if textual data available in the world actually contains sufficient and relevant signals needed to learn all the skills we want LLMs to learn.

Note that language models that are trained solely on text only have access to the linguistic form i.e the sequence of characters making up a sentence like ‘Walter White tossed the pizza onto the roof’. In order to understand its meaning, the linguistic form has to be mapped to the communicative intent of the writer/speaker. While a section of the research community argues that one cannot learn meaning from form alone, recent language models are increasingly proving otherwise.

In order to have access to the full picture, the linguistic form needs to be grounded to the real world. In the cognitive sciences, grounding is defined as

The process of establishing what mutual information is required for successful communication between two interlocutors

—Chandu et al., Grounding 'grounding' in NLP

Human text is generally very underspecified, with a lot of communicative intent existing outside the textual context, depending on the reader/listener to use their common sense, world knowledge, ability to detect and understand emotional subtext in order to interpret it.

NOTE

It is estimated that only around 12% of information we understand from text is explicitly mentioned in text. There are several theories explaining why we communicate thus, including Zipf's principle of least effort, which states it is "human nature to want the greatest outcome at the least amount of work".

The field of NLP has seen a lot of work in grounding language models to the real world. Multimodal models that combine different modalities like image, video, speech, text are a promising avenue of research, and are likely to see more widespread usage in the coming years. Imagine a model seeing 'pizza' in the training text, but also getting signals on how it looks, how it sounds, and how it tastes!

But do multimodal models really help? Can we achieve the effect of grounding by just feeding the model with massive amounts of diverse text? These are unsolved questions, and there are good arguments in both directions as shown by this debate.

Whether training on massive amounts of text alone can enable language models to learn skills like logical reasoning is another open question. Note that text on the Internet contains a lot of text describing reasoning steps, like theorem proofs, explanations of jokes, step-by-step answers to puzzles and so on. However, there is simply not enough of derivational text going around, which leads us to cover the shortfall by using prompting methods like chain-of-thought (described further in Chapter 5). There is recent evidence that process supervision, where feedback is provided for each step of the problem-solving process, as opposed to outcome supervision, where feedback is provided only on the final solution, helps improve arithmetic reasoning.

A crucial skill that language models have to learn is dealing with the inherently ambiguous nature of language. Following up on the aforementioned Zipf's principle of least effort, ambiguity enables speakers to manage the efficiency-clarity tradeoff in communication. Earlier language models struggled heavily with modeling ambiguity. As an example, I long used this sentence as a canonical example in my NLP talks to highlight ambiguity in language.

"WWE's John Cena surprises Make-A-Wish 7-year-old with cancer."

While GPT-4 seems to get the correct interpretation of this particular sentence, recent work shows that state-of-the-art models like GPT-4 still struggle to deal with ambiguity in general. Whether just scaling up models and data is enough for LLMs to model ambiguity is an open question.

If our only option to resolve all these shortcomings is to scale up dataset sizes, the next question is if we actually have enough data available in the world that is sufficient to enable LLM's to learn these skills. Are we at risk of running out of training data any time soon? There is a misconception in certain quarters of our field that we are. However, lack of raw data is far away from being a bottleneck in training models. For instance, there are billions of publicly available documents accessible by scraping or via a free API that haven't yet made it into most pre-training data sets such as parliamentary proceedings, court judgements, and most SEC filings. Moreover, text generated by language models can be used to self-improve them, albeit with the risk that training on LLM-generated data can be detrimental, as the model deviates from the true distribution of the data.

Of course, one could make a distinction between the volume of available *high-quality* data vs *low-quality* data and claim that it is high-quality data that is close to exhaustion , but what exactly makes data high-quality is a very nuanced question.

NOTE

LLMs are underfit, and are usually trained with just one epoch or less (each training example is fed to the model only once, unless duplicates of that example exist across the dataset). However, in recent times, there is increasing evidence that you can safely train for multiple epochs (at least ~5) without being in danger of overfitting. The GALACTICA model from Meta was trained on 4 epochs, and noted improved performance. Recent work from Muennighoff et al. and Xue et al. provide further evidence on this. Therefore, the impending data-apocalypse has been thwarted even further.

COPYRIGHT ISSUES PERTAINING TO PRE-TRAINING DATASETS

Can LLMs be trained on copyrighted text without the explicit consent of the copyright holder and without attribution? Can LLMs be trained on text that inadvertantly contains sensitive personal information without legal liabilities? These are all fluid legal and moral questions. In the U.S, the ‘fair use’ doctrine has been used to justify training LLMs on copyrighted text. However, this is currently being tested, and as of this book’s writing, a class action lawsuit has been filed against Github, Microsoft, and OpenAI for using code from Github repositories that were published under restrictive licenses for training Github Copilot, a code completion LLM. The AI community will be watching this case with interest. However, all over the world, laws are fast loosening to permit this type of usage and clear legal hurdles for LLM training and adoption.

As LLM usage expands and they become an integral part of the economy, data used to train them becomes more valuable. Reddit and StackOverflow, both of which have been an important source of data in many influential pre-training datasets, have recently announced they will start charging for data access. Expect more such announcements in future.

What are the copyright implications for people and organizations using these language models downstream? We will discuss this in more detail in Chapter 14, where we will provide more background on the various types of software licenses and their degree of permissibility for commercial usage.

Popular pre-training datasets

A lot of text is not freely available in public. This includes data exposed behind paywalled APIs and login screens, and paywalled books and documents, many of whom may not even be digitized. Larger companies like Google and OpenAI can afford to purchase this data - Elon Musk revealed that Open AI had access to the Twitter database, and Google has access to over 40 million books it has scanned and digitized as part of the Google Books project. Domain specific text is often proprietary and available only to large incumbents (for example Bloomberg trained BloombergGPT partly on their proprietary financial data). However, even for models trained by the largest companies, a significant proportion of training data comes from publicly available data sources.

Next, we will cover some of the most popular general purpose pre-training datasets that are being used to train LLMs. While this is not a comprehensive list, most LLMs, including closed-source ones, have at least a large subset of their training data drawn from these sources. We will defer discussion of domain-specific (catered to a particular field like social media, finance, biomedical etc) datasets to Chapter 8.

TIP

Most general purpose LLMs are trained to be a jack-of-all-trades - to be able to solve tasks from a variety of domains. If the data domain for your use case happens to be represented in a pre-training dataset, you will see some performance improvement on your downstream task, even though the data in the pre-training dataset is unlabeled. This means that if you intend to use LLMs for a specific well-defined use case in a particular domain, then domain-specific models could likely be more preferable. You can also perform *continued domain-adaptive or task-adaptive pretraining* to leverage this phenomenon. This will be discussed in detail in Chapter 8.

Common Crawl/C4: The Web is the largest source of openly available textual data. Common Crawl is a non-profit that creates and makes available a snapshot of all web crawl data, updated every month. However, as one could imagine, this is an extremely coarse data set and needs to be significantly cleaned before it is ready to use. Most pre-training datasets have a sizeable portion of their data sources from Common Crawl. Google prepared C4 (Colossal Clean Crawled Corpus), a 750GB dataset after applying a set of pre-processing and filtering steps to a Common Crawl snapshot from 2019 and released the code for it. Dodge et al. used this script to reproduce C4 and have made it publicly available. C4 has been used for training several well-known LLMs including all models from the T5 family.

The Pile: The Pile is a 825 GB dataset from Eluether AI, who focused on publishing a dataset that is drawn from more diverse sources. Diversity of data is important since in-domain unlabeled data in pre-training is helpful for downstream performance on that domain, and diverse data sets also enable generalization to previously unseen tasks and domains. To this end, the data from The Pile comes not only from Common Crawl but also PubMed Central, ArXiv, GitHub, the FreeLaw Project, Stack Exchange, the US Patent and Trademark Office, PubMed, Ubuntu IRC, HackerNews, YouTube, PhilPapers, NIH ExPorter, Project Gutenberg, Wikipedia among others. It is one of the most preferred datasets for open-source LLM models today.

ROOTS: The ROOTS dataset is a 1.61 TB multilingual dataset released by BigScience, the open source collaboration that trained BLOOM, which at the time of release was the largest multilingual language model in the world. A large proportion of ROOTS data comes from web domains and datasets that were marked by volunteers from across the world as being highly relevant.

WebText/OpenWebText/OpenWebText2: These refer to a subset of web text, and are limited to text from pages representing outbound links on Reddit that have at least 3 *karma*, where karma refers to the absolute difference between upvotes and downvotes. The idea is that the wisdom of the crowds will enable only quality links to surface, that contain information that people actually find interesting. Models that have been trained on this data include GPT-2 and GPT-3.

Wikipedia - A full dump of Wikipedia contains valuable encyclopedic text that provides factual knowledge to the model. Wikipedia’s editorial system ensures that the text follows a highly structured format. However, it is not diverse stylistically, with text written in a formal manner. Hence, it is usually combined with a corpus like the BooksCorpus.

BooksCorpus/BooksCorpus2 - Probably the most influential of all pre-training datasets, this dataset was part of the training corpus for well known models like BERT, RoBERTa, GPT-2/3 etc. The BooksCorpus contains over 7,000 free, mostly fiction books written by unpublished authors. It has since been found that several books in the dataset have restrictive copyright licenses. The original corpus is no longer public. 26% of books in the original dataset belonged to the Romance genre. A replication of the BooksCorpus is present in The Pile as BooksCorpus2.

The following table provides a list of some of the most commonly used datasets, their size, year of release, and the means to access them.

Table 1-1. Popular pretraining datasets

Name	Data Source(s)	Size	Year Released	Pu
C4	Common Crawl	750GB	2019	Ye (re ve
The Pile	Common Crawl, PubMed Central, Wikipedia, ArXiv, Project Gutenburg, Stack Exchange, USPTO, Github etc	825GB	2020	Ye
RedPajama	Common Crawl, Github, Wikipedia, arXiv, StackExchange etc	1.2T tokens	2023	Ye
BooksCorpus	Sampled from smashwords.com	74M sentences	2015	Or av an
OpenWebText2	outbound reddit links	65GB	2020	Ye

Name	Data Source(s)	Size	Year Released	Pu
ROOTS	BigScience Catalogue, Common Crawl, Github	1.6T tokens	2022	No av re
RefinedWeb	Common Crawl	5T tokens	2023	Ye su
SlimPajama	Cleaned from RedPajama	627B tokens	2023	Ye

As you can see, most models are trained from the same few datasets. In this chapter, we are limiting our coverage to pre-training datasets for base models. We will cover datasets used to augment base models like instruction tuning datasets, RLHF datasets, prompt datasets etc in Chapter 6.

EXERCISE

Let’s do some sleuthing. Investigate the C4 dataset and explore its characteristics.

- Is your personal data present in C4? Use this tool to find out.
- Consider a domain of your choice (finance, poetry, biomedical etc), catering to your professional and/or personal interests. What are the popular websites for your domain? To find out what proportion or C4’s data comes from those websites, you can use this tool from The Washington Post.(scroll down until you find the tool *the websites in C4’s dataset*.)

Training Data Preprocessing

Once we have collected or procured data, we need to run the data through a preprocessing pipeline in order to create the pre-training dataset. Data preprocessing is the most unglamorous and underappreciated part of the LLM training pipeline, yet perhaps the most important. I would argue that there are a lot of low-hanging gains to be had for LLMs just by focusing more on data pre-processing. As we walk through the data processing pipeline, I hope you come to appreciate the complexity of language text and the difficulty in processing it. Note that since these datasets are enormous, any preprocessing step should also be very efficient (ideally linear time).

Figure 1-3 shows the typical preprocessing steps used to generate a pre-training dataset. The ordering of steps is not fixed, but there are dependencies between some of the steps.

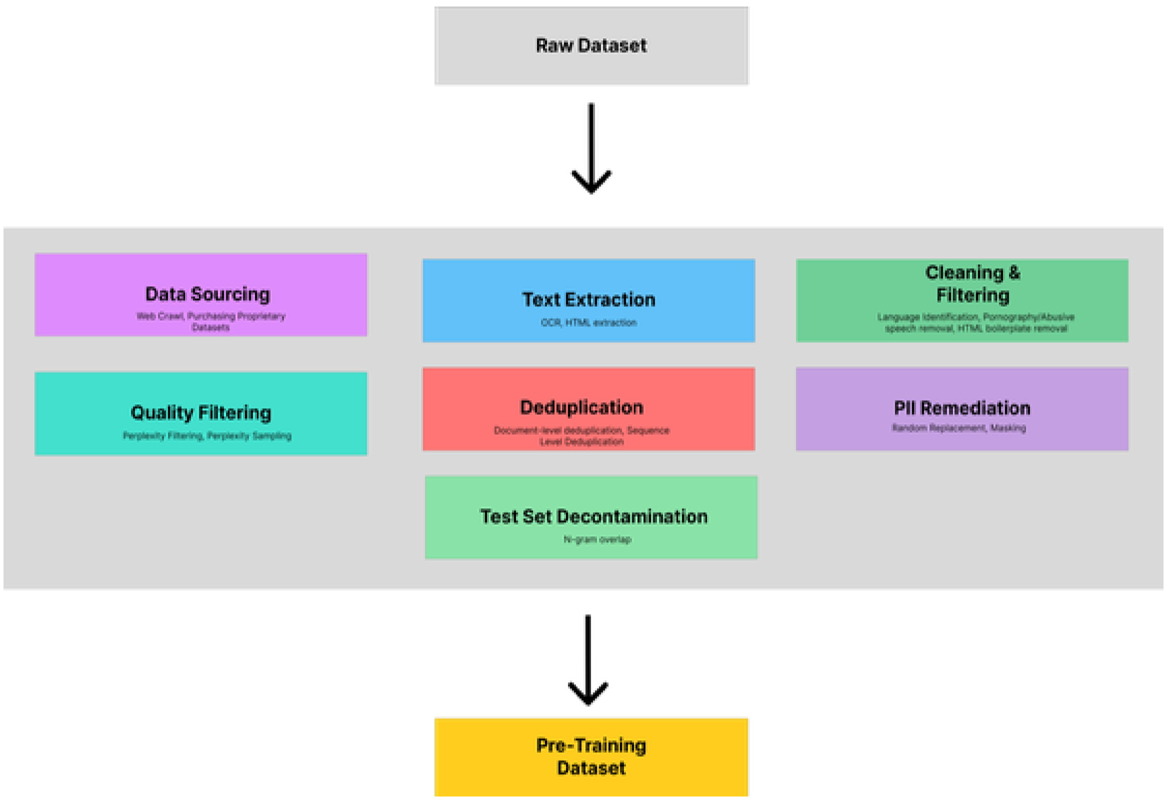

Raw Dataset
Data Sourcing
Web Crawl, Purchasing Proprietary Datasets
Text Extraction
OCR, HTML extraction
Cleaning & Filtering
Language Identification, Pornography/Abusive speech removal, HTML boilerplate removal
Quality Filtering
Perplexity Filtering, Perplexity Sampling
Deduplication
Document-level deduplication, Sequence Level Deduplication
PII Remediation
Random Replacement, Masking
Test Set Decontamination
N-gram overlap
Pre-Training Dataset

Figure 1-3. Data collection and pre-processing pipeline

Let’s go through these steps in detail.

Data filtering and cleaning

A majority of text extracted from HTML files is gibberish, like menu text from websites, boilerplate text, and random web page artifacts. There is a significant amount of pornography, toxic, and hateful language on the Web too. For example, here is how a text sample from an uncleaned version of the C4 dataset looks like:

“Skip to Main Content Skip to Footer Skip to Email Signup Skip to Feedback Form MY REWARDS SIGN OUT SIGN IN & EARN REWARDS 0 Keyboard Controls Welcome to the main navigation. This menu has three levels of product categories. Use and keys to navigate between each category in the current level. Use the key to navigate down a level. Use the key to navigate up a level. Hit the key to be taken to the selected category page. Men What’s Hot New Arrivals Brand That Unites Performance Shop Online Exclusives Express Essentials Vacation Getaway Wedding Tuxedos

Military Trend 9 Pieces / 33 Looks The Edit x Express NBA Collection Express + NBA Fashion NBA Game Changers Suiting & Blazers Find"

How useful do you think this text is for language and task learning?

Data from Common Crawl is made available via both raw HTML and WET (web-extracted text) format. While many dataset creators directly use the WET files, the open source organization Eluether AI noticed that the quality of the WET files left much to be desired, with HTML boilerplate still prominent as seen above. To create The Pile, Eleuther AI thus used the justext library to more reliably remove boilerplate text from HTML documents.

Let us explore the effect of using justext with an example.In your Google Colab or jupyter-lab notebook, try this out -

```
!pip install justext

import requests
import justext

response =
requests.get("https://en.wikipedia.org/wiki/Toronto_Transit_Commi
ssion")
text = justext.justext(response.content,
justext.get_stoplist("English"))
for content in text:
  if content.is_boilerplate:
    print(content.text)
```

The output displays all the boilerplate that is filtered out from a standard Wikipedia article.

```
Jump to content
Main menu
Main menu
Navigation
Main page
Contents
Current events
Random article
About Wikipedia
```

```
Contact us
Donate
Contribute
Help
Learn to edit
Community portal
Recent changes
Upload file
Languages
Language links are at the top of the page across from the title.
Search
Create account
Log in
Personal tools
…
```

justext just so happens to be more aggressive in removing content, but this is generally OK for cleaning pre-trained datasets since there is an abundance of text available. Some alternative libraries used for this task include dragnet, html2text, inscriptis, newspaper, and trafilatura. According to the creators of The Pile, dividing the extraction pipeline across multiple libraries can reduce the risk of the resulting dataset being affected by any bias introduced by one of these libraries.

EXERCISE

Use your favorite news website and open a news article. Use any of the text extraction libraries mentioned, to remove web boilerplate. Is the output desirable on your first try? What kind of additional heuristics might you need?

PRE-TRAINING ON RAW HTML DOCUMENTS

Do we really need to filter out HTML tags from raw HTML documents before pre-training? What if we pre-trained on raw HTML documents instead? This outlandish yet creative idea was implemented by Aghajanyan et al. in their HTLM (Hyper-text Language Model) model. The structured format of HTML enables valuable metadata to be encoded with text. For example, the <title> tags could represent the summary, and the <class> tags could provide category information about the text.

Not all of the HTML is useful for pre-training. For example, CSS isn’t very informative for language learning. Therefore, the creators of HTLM convert the raw HTML into a simplified form, by filtering out iframes, headers, footers, forms etc. This process is called *minification.*

The results presented in their paper show the model is especially good at summarization, because the access to the category tags helps it focus on the salient aspects of the topic under discussion. However, as of this book’s writing, this pre-training paradigm hasn’t caught on yet.

Once text is extracted, rudimentary filtering steps based on heuristics are applied. While the details differ across datasets, here are some of the steps typically performed:

- Boilerplate Removal: Only lines that end with a punctuation, like the period, exclamation and question mark are retained. This ensures that menu text from websites is removed. Only lines with greater than a particular threshold of words and documents with greater than a particular threshold of sentences are retained. The latter helps in modeling long sequences which is an important capability for language models to have. Documents containing *lorem ipsum…* and other boilerplate text are filtered out.

- Non-English text removal: Libraries like *langdetect, langid, fasttext, pycld2* are used to detect the language of the text. For example, C4 retains text that has > 0.99 probability of English as judged by *langdetect.* Note that these libraries can also be used to remove boilerplate and web page artifacts since they give a lower probability of English to those texts.

- SEO text/Spam removal: Documents with a lot of repeated character sequences are removed. Documents with a low proportion of closed class words are removed. Closed class words in English are function words like of, at, the, is etc. If a page is engaged in keyword stuffing and other SEO tricks, then they would have a lower closed class words ratio.
- Pornographic/abusive text removal: Documents containing any words from keyword lists like the "List of Dirty, Naughty, Obscene or Otherwise Bad Words" are removed.

Tools like *langdetect* and *langid* are helpful for speedy determination of the language in which text is written at scale, but how do they deal with code-switched text (text with multiple languages, where oftentimes it is English interspersed with a local language)?

You can try it out yourself! Here is an example for Taglish (Tagalog + English, which is a common mode of communication in the Philippines). In your notebook, run

```
!pip install langdetect

from langdetect import detect_langs()

detect_langs("""Pag-uwi ko galing sa paaralan, sobrang pagod ako
dahil sa dami

ng aking ginawa sa buong araw. Ang traffic din sa kalsada,
nakaka-stress

talaga! Pero nang makarating ako sa aking tahanan, nabuhayan ako
ng loob dahil

sa masarap na amoy ng ulam na inihanda ni nanay. Excited na akong
kumain
kasama ang aking pamilya at i-share ang mga kwento ko tungkol sa
aking mga

kaibigan, guro, at mga natutunan ko sa school. After dinner,
magre-relax muna

ako habang nanonood ng TV, and then magre-review ng lessons bago
matulog. Ito

ang routine ko pag-uwi mula sa school, at masaya ako na dumating
sa bahay namay

naghihintay na pamilya na handang makinig at suportahan ako sa
aking

pag-aaral.""")
```

Output:

```
[tl:0.9999984631271781]
detect_langs("""After a long day at school, pagod na pagod talaga
ako. The

traffic on the way home didn't help, nakakastress na nga! But
upon arriving

home, I felt a sense of relief dahil sa welcoming atmosphere and
the delicious
aroma of the ulam na inihanda ni Mommy. Excited na akong mag-
share ng

experiences ko today with my family during dinner, kasama ang mga
kwento about
my friends, teachers, and interesting lessons sa school. After
eating, it's

time for me to chill while watching some TV shows, and then
review my lessons

bago ako matulog. This is my daily routine pag-uwi galing school,
and I am

grateful na may loving family ako na handang makinig at
supportahan ako sa

aking educational journey.""")
```

Output:

```
[en:0.9999954357601804]
```

The second paragraph would get included in the C4 dataset, as per its filtering criteria (probability of English should be greater than .99). Therefore, even datasets that claim to be English-only routinely contain text in other languages, leading to surprising multilingual behavior during inference. Ever wondered why some monolingual models seem to perform well at machine translation? This is a major reason.

The way *langdetect* is implemented makes it poor at identifying language when short sequences are provided. For example:

```
detect_langs('I love you too.')
```

returns

```
[sk:0.8571379760844766, en:0.14285726700161824]
```

sk refers to Slovak here.

EXERCISE

C4 is an English language dataset, with text getting less than 0.99 probability of English in *langdetect* being removed. However, a lot of non-English data persists in this dataset. If you know a second language, then search for words in that language in C4. In what contexts do these non-English text fragments appear? Could an LLM *learn* these languages using these leftover fragments?

Selecting Quality Documents

While LLM's are trained with the intention of making them a jack-of-all-trades, the Internet is a very vast place and not all data is created equal.

There are many websites whose content one would be hard pressed to find relevancy to any potential downstream task, however imaginative you might be. Moreover, as we have seen earlier, the data cleaning process is far from optimal. A common way of filtering out less *useful* documents from Common Crawl is to build a classifier for quality text. The examples for the positive class are from a dataset known to be useful, like say, Wikipedia, and the examples for the negative class would be random documents from common crawl.

Perplexity for quality selection

Perplexity, an intrinsic evaluation measure for language models, has been used in the data-processing stage for document filtering, notably by the creators of CCNet.

Just like the classifier approach, we select documents from data sources (like Wikipedia) that we deem useful as the positive class. We then train a 5-gram language model using KenLM (a library facilitating training of n-gram language models.) over it. Next, we take the dataset we want to filter, and calculate the perplexity of each paragraph in it over the trained language model. The lower the perplexity, the more similar it is to the positive class. We can then discard documents with high perplexity.

Low perplexity may not always be a good thing. Short and repetitive text can have low perplexity. Note that writing style gets factored into perplexity. If the reference language model is trained over Wikipedia, then documents written in an informal style may receive higher perplexity scores. Therefore, it would be beneficial to have a more involved filtering strategy.

To resolve this, the creators of BERTIN introduced the concept of perplexity sampling. In perplexity sampling, instead of just filtering out low-perplexity text, they utilize perplexity scores in a sampling strategy over their dataset. The sampling strategy is to oversample from the middle part of the perplexity probability distribution.

Exploring perplexity with Wikipedia LMs

Download the file *https://huggingface.co/edugp/kenlm/blob/main/model.py*
After placing the file in your home directory, run this code in a new file

```
from model import KenlmModel
model = KenlmModel.from_pretrained("wikipedia", "en")
model.get_perplexity("She was a shriveling bumblebee, and he was
a bumbling

banshee, but they accepted a position at Gringotts because of
their love for

maple syrup")
```

EXERCISE

Try out sentences in different styles and topics to see how the perplexity varies! In particular get the perplexities of these types of text:

- Social media text, like Twitter
- SEO spam
- Text with a lot of slang

Additionally, you can train a KenLM model on your own domain dataset. Sample a portion of your dataset and train the model using the instructions provided in their Github. You can then take the remaining portion of the dataset, break it into chunks, and calculate the perplexity of each chunk. Which chunks have the highest perplexity? Which chunks have the lowest perplexity? After manually inspecting the results, do you think perplexity sampling is a good measure of quality?

NOTE

According to an analysis of C4, the Internet domain that contributed the largest proportion of text in the dataset was patents.google.com. Over 10 percent of the text from this domain is in fact machine translated, with patents from countries like Japan being translated from Japanese to English. So a significant amount of pre-training data is already not generated by humans!

Propelled by LLM’s, the Internet is slated to see widespread prevalence of AI-generated text. Recognizing whether text was written by a human or an LLM is a non-trivial task, and certainly not feasible at scale. How this would affect future LLM performance is an open research question.

Despite all the data cleaning steps, the resulting dataset is still not going to be perfect at this level of scale. For example, Eleuther AI reported that the boilerplate sentence “select the forum that you want to visit from the selection below” occurs 180k times in the Pile.

Deduplication

So far we have discussed data extraction and cleaning, language identification, and quality filtering. Let’s now explore the most contentious step in the pipeline - deduplication.

We know that web-crawled text is ridden with a lot of duplicates. Duplicates form a non-trivial portion of the training dataset, so any decision taken about them will have a noticeable impact on the ensuing model.

How do we define a duplicate? We will make a distinction between three kinds:

- *Exact Matches*: Two sequences with the same text are exact-match duplicates. They are the easiest to handle.
- *Approximate Matches*: In many cases, there are near-duplicates, where sequences of text are identical except for a few characters. Sometimes these sequences are slightly different only due to HTML text extraction artifacts and other filtering processes.

- *Semantic Duplicates*: Duplicates that semantically convey the same content but using different wordings. This is usually treated as out of scope.

Duplicates can also be categorized based on the granularity at which they occur.

- *Document-level Duplicates*: Duplicate documents are removed during the preparation of most pre-training datasets. However, in some datasets like The Pile, certain subsets (like Wikipedia) are deliberately duplicated, so that they are seen more often by the model.
- *Sequence-level Duplicates*: These are sequences in documents that are repeated across multiple documents. In some cases they can be massively duplicated, like Terms of Service text, copyright notices, website prefaces etc.

To Deduplicate or to not Deduplicate

The jury is still out on the effectiveness or lack thereof of deduplication.

There is evidence that you can train for four epochs without overfitting. This is equivalent to text being duplicated four times. However, there is still a benefit in removing duplicates that are just boilerplate text and occur thousands of times.

On the other hand, here are a few arguments in support of deduplication:

- A small subset of the pre-training dataset is usually kept aside for validation/test. Deduplication can ensure the removal/reduction of overlap between the train and test sets, which is essential for an unbiased evaluation. Without sequence-level deduplication, there is a high likelihood of overlap of common text sequences in the train and test sets.
- Anthropic’s work shows a surprising double descent phenomenon - this means that data that is duplicated only a few times doesn’t negatively impact model performance too much, data that is duplicated

too many times doesn’t negatively impact model performance too much, but in the distribution of duplication frequency, there is a peak in the middle where the damage is maximum.

- Removing duplicate sequences reduces the overall size of the training dataset. However, Lee et al. show that this does not affect the perplexity of the model. Thus, the model can be trained for a shorter period yet with the same benefit.
- Deduplication can also reduce the tendency of the model to memorize its training data. Memorization is closely linked to model overfitting, and thwarts the ability of the model to generalize. While there are many ways to quantify memorization, we will focus on *memorization by generation*, where a model is said to have memorized a sequence if it is capable of generating it verbatim. Lee et al. have shown that models trained on datasets that have been deduplicated at the sequence level generate ten times less verbatim training data.

TIP

One advantage of using models trained on publicly available datasets is that you can search through the dataset to see if the text generated by the model exists verbatim in the dataset. For example, the ROOTS search tool can be used to test generations from the BLOOM model, which was trained on ROOTS.

SECURITY VULNERABILITIES IN LLMS DUE TO MEMORIZATION

Memorization makes language models vulnerable to security and privacy attacks. Two demonstrated types of attacks are:

- *Membership inference attack*: With just closed-box access to a model, a membership inference attack enables an attacker to determine if a sequence of text has been used to train the model or not.

- *Training data extraction attack*: With just closed-box access to a model, the attacker can prompt the model to generate memorized sensitive information. A naive example involves prompting the model with the text 'Suhas Pai's phone number is' and asking the model to provide the continuation, with the hope that it has memorized Suhas's number.

Carlini et al. show that larger models memorize more easily and thus are most susceptible to these types of attacks. However, it is hard to estimate how much data is memorized by the model, as some memorized data is output by the model only when prompted with a delicately prepared prefix of a longer length. This makes models harder to audit for privacy guarantees.

Figure 1-4 demonstrates the flow of a rudimentary training-data extraction attack.

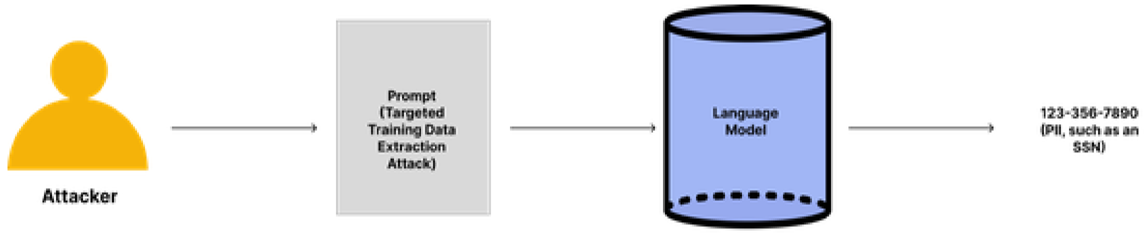

Attacker
Prompt (Targeted Training Data Extraction Attack)
Language Model
123-356-7890 (PII, such as an SSN)

Figure 1-4. Privacy Attacks against LLMs

TIP

Deduplication is computationally intensive, especially when it comes to removing near-duplicates. Some of the efficient algorithms used include MinHash, SimHash, Suffix Array etc.

Removing PII (Personally Identifiable Information)

While deduplication can reduce the likelihood of the model memorizing training data, it is by no means a panacea to the memorization problem. Even information that appears only once in the training set could potentially be memorized (and leaked). While a lot of content in the training data is innocuous (Terms of Service text) and perhaps even desirable to memorize (factual information, like the capital of Canada), memorization of personally identifiable information (PII) is a major concern.

Let us see what PII entails. The formal definition from Cornell Law is -

> *Information that can be used to distinguish or trace an individual's identity, either alone or when combined with other personal or identifying information that is linked or linkable to a specific individual.*

Based on this definition, non-PII can become PII when another piece of information becomes public, which when combined with the non-PII can be used to uniquely identify an individual.

The legal definition of PII varies by jurisdiction. For example, the GDPR (General Data Protection Regulation) in Europe, says

> *Protection should be extended to anything used to directly or indirectly identify a person (or data subject). This may be extended to include characteristics that describe "physical, physiological, genetic, mental, commercial, cultural, or social identity of a person.*

Most open-source models are trained on publicly available datasets. These datasets might contain PII, but one might be tempted to say 'well it is already out in the open, so there is no need for privacy protection'. This argument overlooks the importance of consent and discoverability controls.

For instance, I might have shared my PII on my blog which resides in an obscure corner of the Internet and is not easily discoverable through search engines, but if it ends up being added to a pre-training dataset, it suddenly brings this data into the spotlight, without my consent. This concept is called *contextual integrity* - data should only be shared in the original context in which it was shared.

So ideally, we would like to *detect* PII in the dataset, and then *remediate* it in some fashion, so that the PII is no longer present in the training data or at least not memorizable. The presence of *public-figure PII* adds a layer of complexity to this problem. We would like our model to be able to answer factual questions about public figures like their birth date accurately. The privacy expectations for public figures is lower, showcasing how the values of transparency and openness clash with privacy. Determining who is a public figure and what level of privacy they are entitled to is a complex social and technical challenge.

Data that is considered private includes names, addresses, credit card data, government IDs, medical history and diagnosis data, email IDs and phone numbers, identity and affinity groups the person belongs to (religion, race, union membership), geolocation data and so on.

Attacks can be either targeted or untargeted. In an untargeted attack, the attacker just generates a large body of text using the model, and then runs a membership inference attack to determine text within it that is most likely to be memorized. In a targeted attack, the attacker attempts to recover personal information about a particular individual or a group of individuals. Targeted attacks are more difficult to execute, because while language models are good at memorization, they are bad at *association* - for instance, identifying that an email ID belongs to a person.

EXERCISE

Use the instructions in the ReadMe to run this code for analyzing privacy attacks on LLMs. It goes without saying, but please do not use this in the real world! Running the code and observing the outputs will give you an understanding of the limitations of this type of attack, and the type of data that is typically memorized by an LM.

Additionally, you can play around with Google's Training Data Extraction Challenge and make a submission!

NOTE

Language models are also susceptible to training data poisoning attacks. Since a large portion of training data is sourced from web-crawled text, bad actors have an opportunity to influence the content of the training set. Tramer er al. have shown that one can poison less than 0.1 percent of the training set with data whose effect is to make it easier for other data in the training set to leak more easily.

As LLMs increasingly get used as search engines, the demand for LLM SEO will soon crop up. For example, a company could write content on their web sites in a manner that makes it more likely to be chosen in a pre-training dataset creation process that uses perplexity filtering.

Most pre-training datasets have undergone little to no PII remediation. The Privacy working group (of which I was the co-lead) of the Big Science project that trained the BLOOM model developed a pipeline for PII detection and remediation, which we will discuss next.

Figure 1-5 shows a typical PII processing pipeline.

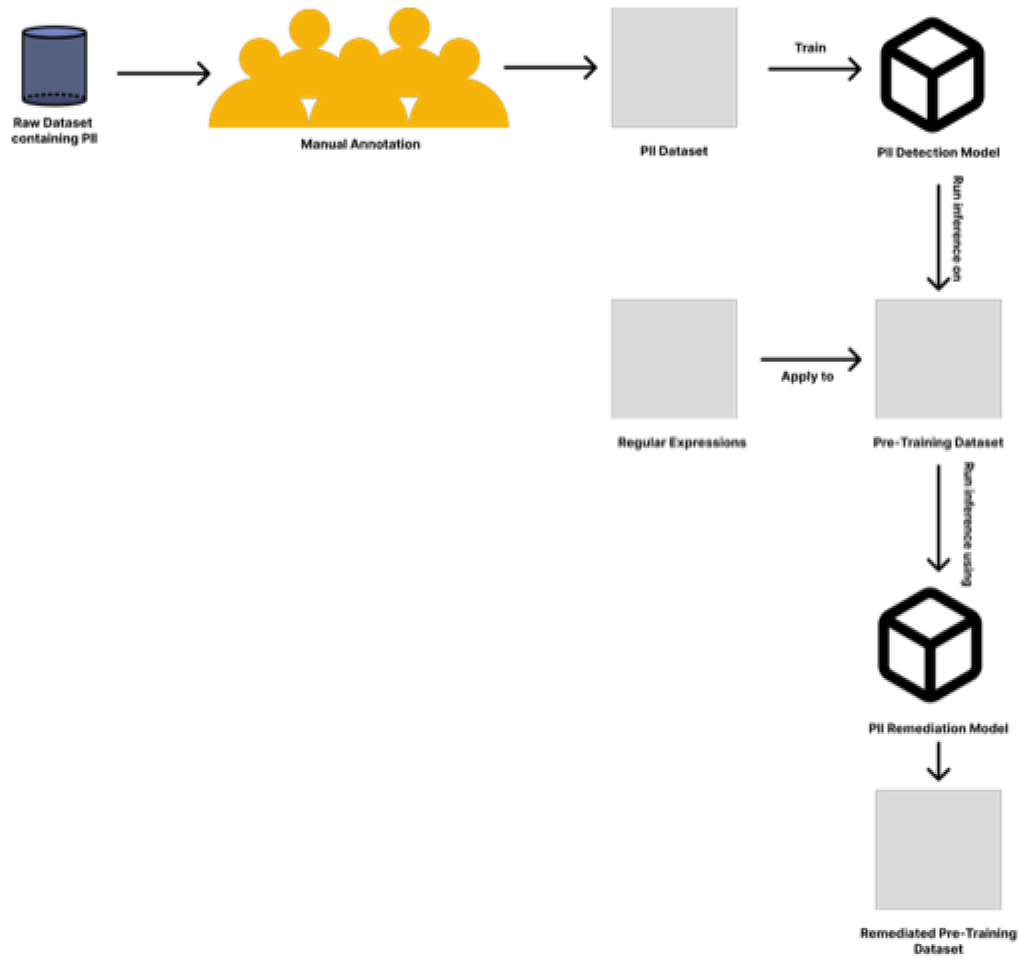


Figure 1-5. PII Processing pipeline

PII Detection

The task of PII detection is similar to the NLP task of Named Entity Recognition, introduced in Chapter 1. However, not all named entities constitute PII. For our task we determined the PII tags to be - PERSON, AGE, NORP (nationality, race, religion, political party affiliation, socio-economic class, union membership), STREET_ADDRESS, CREDIT_CARD, GOVT_ID, EMAIL_ADDRESS, USER_ID, PUBLIC_FIGURE.

We used the PUBLIC_FIGURE tag to identify information about public figures, since we didn’t want to filter them out. We also assigned fictional characters this tag.

Some of the structured tags in this list like emails and government IDs can be identified using regular expressions. For other tags, we annotated datasets which could then be used to train Transformer-based NER-like models. Interestingly, we observed a very high degree of inter-annotator disagreement (same example being annotated differently by different people) that underscored the cultural nuances of the definition of privacy and what constitutes personal information.

Here is the regular expression to detect SSN (U.S Social Security Numbers):

```
ssn_pattern = r"(?!000|666|333)0*(?:[0-6][0-9][0-9]|[0-7][0-6][0-
9]|

[0-7][0-7][0-2])[-\ ](?!00)[0-9]{2}[-\ ](?!0000)[0-9]{4}"
```

Note that detection is not the same as validation. Not all 9 digit numbers of the form XXX—XX-XXXX are SSNs! Validation is the process of checking if a sequence of characters maps to a valid identifier. For example, the Canadian equivalent of SSN, the SIN (Social Insurance Number) contains a checksum digit which can be used to validate it.

```
from stdnum.ca import sin
sin_pattern = re.compile(r"\d{3}[-\ ]\d{3}[-\ ]\d{3}",
flags=re.X)
for match in sin_pattern.findall(text):
    if sin.is_valid(match):
        print(match)
```

The is_valid() function uses the Luhn checksum algorithm to validate if the sequence of digits maps to a valid SIN. The same algorithm is also used to validate credit cards. Here is the regex for detecting credit card numbers.

```
from stdnum import luhn
cc_base_pattern =  r"\b \d (?:\d[ -]?){14} \d \b"
cc_full_pattern = r"""4[0-9]{12}(?:[0-9]{3})? |
                      (?:5[1-5][0-9]{2}|222[1-9]|22[3-9][0-
9]|2[3-6][0-9]{2}|27[01][0-9]|
                                  2720)[0-9]{12} |
                                  3[47][0-9]{13} |
                                  3(?:0[0-5]|[68][0-9])[0-9]{11} |
                                  6(?:011|5[0-9]{2})[0-9]{12} |
                                  (?:2131|1800|35\d{3})\d{11}"""
```

The regular expression for detecting email address is

```
email_pattern = r"[\w\.=-]+ @ [\w\.-]+ \. [\w]{2,3}"
```

EXERCISE

These regular expressions were run on the ROOTS dataset. How effective were they in detecting PII? Find out using the ROOTS search tool. If you search for ‘gmail.com’, you will find that all entries in the search results have been successfully redacted. Alter the spelling a little and see if it still holds true. Can you improve the regular expession?

NOTE

Removing structured PII data while keeping the number of false positives low is hard enough, but detecting and remediating unstructured data is even harder. Due to the complexity of this task and the uncertainty about its impact on the resulting model performance, we decided to not run the Transformer model based PII pipeline over the ROOTS dataset for training the BLOOM model

PII Remediation

Once PII has been detected, it can be remediated. Figure 1-6 depicts one of the remediation schemes.

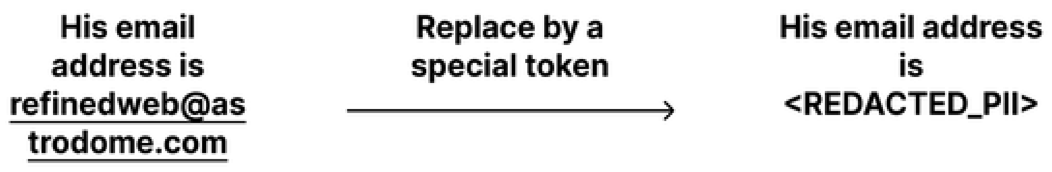

His email address is refinedweb@astrodome.com
Replace by a special token
His email address is <REDACTED_PII>

Figure 1-6. PII Remediation Options

Here is a non-exhaustive list of remediation options:

- *Replace by a special token*: For example, a valid phone number can be replaced by the string <phone number>
- *Replace with a random token of the same entity type*: For example, replace the name ‘Clarietta Richards’ with ‘Natasha Bridges’, or any other name.
- *Replace with a shuffled token*: Entities detected across the dataset can be shuffled.
- *Remove entire document/data source*: If the amount of PII detected in a single document or data source is higher than a specific threshold, it is probably best to remove it. For example, pastebin.com is said to contain a lot of inadvertently placed PII, and is recommended to be not included in training datasets.

Each of these techniques can have a varied effect on downstream performance of the model. How does replacing tokens affect training perplexity? Do downstream tasks like Named Entity Recognition get negatively affected when tuned on the resulting model? How does replacement by special tokens compare to replacement with random tokens? This is a relatively underexplored topic and all these questions are still open.

Faker is an excellent library for facilitating random token replacement. It supports random token generation for a variety of PII types including names, addresses, credit card numbers, phone numbers etc. One danger in using random tokens is that the replacement process can alter the demographic distribution of the dataset - for example, if the replacement names were all or mostly Anglo-Saxon names. Faker has localization support to enable replacement with fake data from the same geography/culture. Let's explore the library in more detail.

```
from faker import Faker
fake = Faker('en_IN')   # Indian locale
Faker.seed(0)
for i in range(5):
   print(fake.aadhaar_id)
```

This code generates 12 digit fake Aadhaar ID's, which are the Indian equivalent of Social Security Numbers. Note that the generated IDs are all invalid, but still follow the same format. Similarly,

```
for i in range(5):
   print(fake.address)
```

generates fake but representative addresses for the selected locale.

NOTE

Removing PII from training datasets is only one of several solutions to prevent data leakage from models. One promising technique is differential privacy, which introduces randomness in the inputs or outputs to provide theoretical guarantees for privacy preservation. In neural networks, differential privacy is implemented using the DP-SGD algorithm, which involves gradient clipping and noise addition at the end of each update. However, differential privacy significantly slows down training, negatively affects model performance, and disproportionately impacts minority groups in the dataset in terms of model utility degradation. Apart from differential privacy, other methods include adversarial training, model unlearning, retroactive censoring, and 'memfree' decoding.

Test Set Decontamination

Test set decontamination is a crucial data preprocessing step that helps improve LLM evaluations. A pre-training dataset is said to be contaminated if it contains data from the benchmark test sets used to evaluate its performance. Contamination can happen if the test datasets were constructed from web text, or if the dataset was uploaded on the Web after creation. There are two types of contamination:[1]

- *Input and Label contamination*: In this setting, both the questions (inputs) and answers (target labels) exist in the pre-training dataset. Heard about how GPT-4 can solve all kinds of exams? While the creators of GPT-4 did spend a lot of effort on removing data contamination, in practice it is really hard to remove everything.
- *Input contamination*: In this setting, only the inputs are present in the pre-training dataset but not the target labels. We will describe the effects of input contamination and how we can leverage it for positive use in Chapter 8 and 9.

Open AI addressed test set contamination in GPT-3 by finding 13-gram overlaps between text in the test/validation set and the train set, and removing 200 characters before and after the matched texts.

DATASET ORDERING

After all data pre-processing stages have been completed, the training process can commence. The order in which the data is fed to the model does matter. The area of study to determine the most optimal order is called curriculum learning. To our knowledge, most models do not go beyond some simple ordering heuristics.

One technique is to start the training with shorter training sequences and then gradually increase the sequence lengths. This can be done by either truncating initial sequences to fit a certain length, or by simply reordering the dataset so that shorter sequences are ordered first.

Researchers have also experimented with introducing more common words to the model first, by replacing rarer words occurring in early training examples with their part-of-speech tag or with hypernyms (for example, the hypernym of magenta is color).

Now that we have discussed all the important data collection and pre-processing steps for preparing a pre-training dataset, let us see how individual datasets differ in terms of the preprocessing steps they have undergone.

TIP

Big Science has developed a visualization tool that helps you understand the effect of various preprocessing functions on the pre-training dataset. Use the Process Pipeline Visualizer to sequentially run through the preprocessing pipeline yourself!

Table 1-2 provides a list of the popular pre-training datasets, and the kind of preprocessing they went through.

Table 1-2. Pretraining Datasets and their Preprocessing Pipeline

Name	Extraction and Cleaning	Quality Filtering	Deduplication	La Ide
C4	Remove pages containing word in blocklist, remove code, remove short lines and pages	-	Deduplication of 3-sentence spans	lan
The Pile	justext library for text extraction	fasttext classifier	Document level, with MinHashLSH	pyc
CCNet	-	Perplexity filtering	Paragraph level deduplication	fast
RedPajama	Ccnet pipeline	Classifier distinguishing between Wikipedia text and random C4 text	Paragraph level deduplication (for Common Crawl)	fast

Name	Extraction and Cleaning	Quality Filtering	Deduplication	La Ide
CleanPajama	low-length filter, NFC normalization	-	MinHashLSH	-
RefinedWeb	URL filtering by blocklists, trafilatura library for text extraction, repetitive content removal	-	Fuzzy document level deduplication with MinHash, Exact sequence-level deduplication	fast
ROOTS	removal of documents with low ratio of closed class words, high ratio of blocklist words, high ratio of character/word repetition	Perplexity filtering	SimHash, Suffix Array	fast

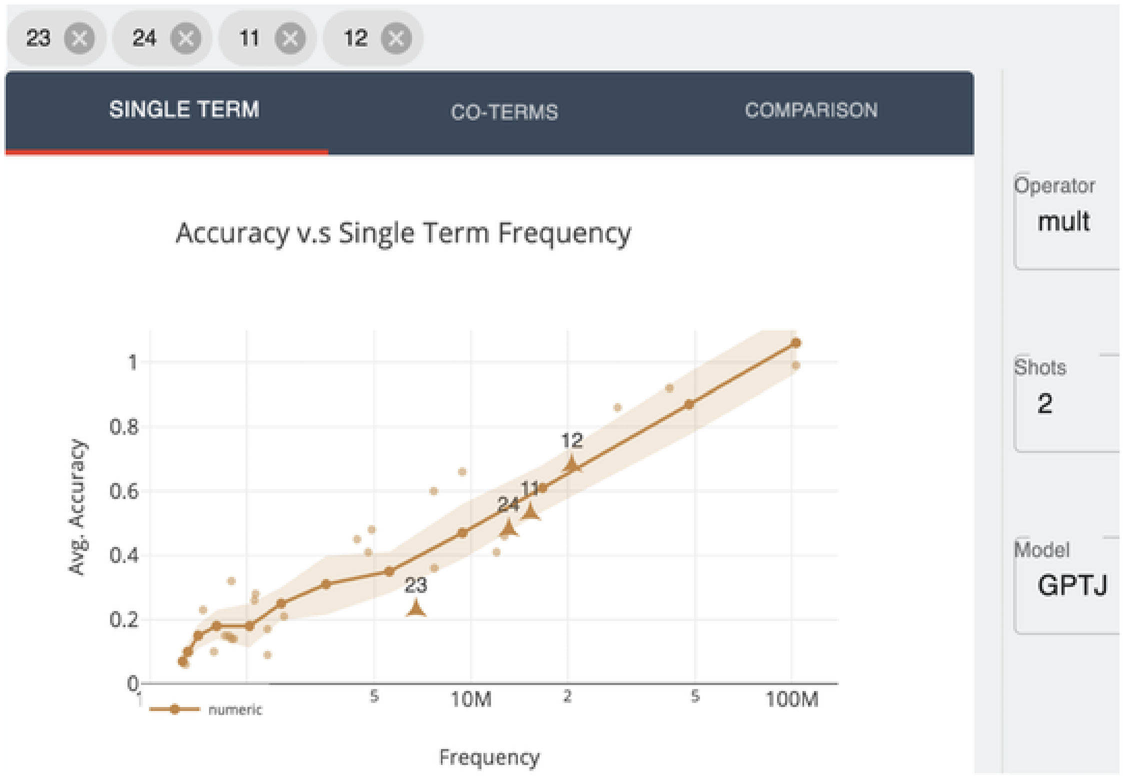
Leveraging Pre-training Dataset Characteristics

How well do LLM’s do on arithmetic and logical reasoning? The prospects of a very large number of use cases depend on the answer being a positive one. We will investigate this question in more detail in Chapter 11.

But for now, I would like you to dwell a moment on this fascinating observation - there is a correlation between a model’s performance on a given input example and the pre-training corpus frequency of the words present in that input.

Razeghi et al. observed this with the GPT-J model - when asked arithmetic questions like addition, subtraction, multiplication etc, the model gets it right sometimes, and wrong other times.If you plot a graph of pre-training frequencies of the numbers versus the performance for arithmetic operations using those numbers, there is a clearly visible trend. The more frequent a number appears in the pre-training dataset, the better the model is at arithmetic operations involving that number.

The effect is most drastic for multiplication tasks. As shown in Figure 1-7, the model is more correct at multiplication operations involving the number 24 than ones involving the number 23, and the frequency of the numbers in the dataset show a large difference between the term frequency for these numbers.



Operator: mult

Shots: 2

Model: GPTJ

Figure 1-7. Plot of average accuracy plotted against term frequency, using the Snoopy tool. Image taken from Razeghi et al.

The authors investigate this phenomenon using three types of frequencies. Consider the input

```
Q: What is 40 times 51? A:
```

The frequencies calculated are

1. Unigram frequency: For example, the number of times the number ‘40’ occurs in the dataset

2. Input term co-occurrence: Two input terms co-occuring within a window size of 5. For the current example, it is (40, 51)

3. Input and output term co-occurrence: Two input terms and the output term co-occurring within a window size of 5. For the current example, it is (40, 51, 2040)

The unigram frequencies alone cause noticeable performance gaps. This phenomenon can be replicated across other types of tasks and datasets as well. This means that Open AI’s technique of finding 13-gram overlaps between text in the training set and in benchmark sets isn’t enough to eliminate input contamination.

If your task is well-defined, doesn’t have drastic data drifts, then input contamination may not really be such a bad thing. You can then leverage frequency statistics to design inputs to the model that are more likely to give the right answer!

TIP

You can explore this phenomenon on your own by using the Snoopy tool. Snoopy is a tool built by Radeghi et al. for analyzing the impact of pre-training term frequencies on model performance. It uses The Pile, the dataset used to train most open-source models including GPT Neo-X, for analysis. You can experiment with a variety of benchmark tasks.

EXERCISE

Using the Snoopy tool, try out different benchmark datasets from the drop down dataset and explore the effect of term frequency (both unigram and co-occurrence) on model accuracy. For which tasks is this phenomenon least prevalent? Why could it be?

Bias and Fairness Issues in Pre-training Datasets

A multitude of ethical questions arise during the productization of large language models. The existence of significant bias and fairness issues in these models often lead to a no-ship condition for a large number of use cases. We will give these issues their due coverage in Chapter 12. For now, in this section we will go through some bias and fairness issues specifically related to the collection and filtering of pre-training data.

The scale of data that LLMs are fed with means that they are not just constructing models of language, but also of the world we inhabit. This gives rise to the question - ‘Do we want to model the world the way it is or do we want to model the world the way we would like it to be?’ The Internet is filled with hate, violence, and abusive language and is often used as an outlet for humanity’s worst impulses. The text in it implicitly encodes long existing biases against groups of people. For example, in The Pile, an analysis of word co-occurrence statistics shows the word ‘radical’ co-occurs with the word ‘Muslim’ substantially more than it does for other religions.

The phenomenon of *bias amplification* makes these problems all the more critical. It has been shown that large language models amplify the biases that are encoded in their pre-training data - they make biased predictions against groups of people at higher rates than what the training data statistics would suggest.

So, can we ‘fix’ our training data such that we can model a world that encodes our values and principles which downstream applications will inherit? There is substantial debate in the research community around this. Opponents argue it is hard to identify and fix all societal biases encoded in the data since there are so many dimensions of bias that intersect in complex ways. Values are not universal and model providers would like to be value-neutral in order to cater to all sections of society

However, as Anna Rogers describes in her paper, this question is already moot. Data curation is already happening, whether we like it or not, and the values and interests of model providers are already being encoded into the models. For example, only a small proportion of available data is ‘selected’ to be part of the pre-training set. This selection process is not value-neutral, even if one might explicitly not think in terms of them.

For example, Wikipedia is one of the more popular datasets used in training LLMs. While this might be a no-brainer to include, let’s explore the implications. Wikipedia is edited by volunteers, a very large proportion of them being men. Since the determination of whether a topic is reputable enough to deserve a Wikipedia page rests with the editors who are largely made up of men, we see disparities like obscure male football players from lower level leagues getting their own pages while a disproportionate number of biography articles about women are slated for deletion.

Similarly, the highly influential WebText dataset is sourced from Reddit outbound links. Reddit is a predominantly male site, with 74% of users being men. Naturally, links posted on Reddit are more likely to be catered to male interests.

Bias can also be introduced during the data filtering stages. Earlier, we noted that keyword lists are often used to filter out pornographic material and abusive text. However, using a naive keyword list is a lazy approach that not only has problems with effectiveness (false negatives), but also inadvertently causes disproportionately filtering out positive text written by or about minority communities, as well as text written in dialects like African-American English and Hispanic-aligned English. The fact that words in English have multiple senses has resulted in certain documents about breastfeeding being filtered out of the C4 dataset.

Overall, whether a word is hateful, abusive, or toxic depends on the social context, the intentions of the reader, and the intended audience. Keyword based methods simply do not capture this nuance. The question of whether it is more effective to handle these issues at the pre-training stage or further downstream is an open area of research. We will explore techniques that can be employed downstream in Chapter 12.

The authors of the Pythia model experimented by replacing masculine pronouns with feminine ones for the last 7 percent of training tokens and noticed a ‘de-biasing’ impact on downstream tasks.

We will further explore bias, fairness, and safety issues and how to integrate these values while designing LLM applications in Chapter 11.

Summary

In this chapter, we outlined the key ingredients of a language model - the pre-training data, the vocabulary and tokenizer, the language objective, and the model architecture. We walked through the steps involved in creating a pre-training dataset in detail, including language identification, text extraction and cleaning, quality filtering, deduplication, PII removal, and test set decontamination. We also provided a list of commonly used pre-training datasets and the steps taken for pre-processing each of them.

Now that you have a good idea about the data side of LLMs, it is time to explore the model side. In the next chapter, we will provide details on the remaining ingredients of the language model - the vocabulary and tokenizer, learning objective, and model architecture.

---

[1] from A Case Study on the Colossal Clean Crawled Corpus, Dodge et al., EMNLP 2021

Chapter 2. LLM Ingredients: Tokenization, Learning Objectives & Architectures

In Chapter 3, we dug into the datasets that are used to train the language models of today. Hopefully this foray has underscored how influential pre-training data is to the resulting model. In this chapter, we will go through the remaining ingredients: vocabulary and tokenization, learning objectives, and model architecture.

Vocabulary and Tokenization

What do you do first when you start learning a new language? You start acquiring its vocabulary, expanding it as you gain more proficiency in the language. Let's define vocabulary here as

All the words in a language that are understood by a specific person

The average native English speaker is said to have a vocabulary ranging between 20,000-35,000 words. Similarly, every language model has its own vocabulary, with most vocabulary sizes ranging anywhere between 5,000 to 500,000 *tokens*.

As an example, let us explore the vocabulary of the GPT Neo-X 20B model. Open the file tokenizer.json and ctrl+f for ‘vocab’. You can see that the words comprising the language model vocabulary don’t entirely look like English language words that appear in a dictionary. These word-like units are called ‘types’, and the instantiation of a type (when it appears in a sequence of text) is called a token.

NOTE

In recent times, and especially in industry, I have hardly heard anyone use the term ‘type’ except in older NLP textbooks. The term token is broadly used to refer to both the vocabulary units and when it appears in a text sequence. We will henceforth use the word ‘token’ to describe both concepts, even though I personally am not the biggest fan of it.

In the vocabulary file, we see that next to each token is a number, which is called the *input id* or the *token index*. The vocabulary size of GPT Neo-X is just above 50,000.

The first few hundred tokens are all single character tokens, starting from special characters, digits, capital letters, small letters, and accented characters. Longer words appear later on in the vocabulary. There are a lot of tokens that correspond to just a part of a word, called a *subword*, like ‘impl’, ‘inated’, and so on.

Let’s Ctrl + F for ‘office’. We get nine results -

```
"Ġoffice": 3906
"Ġofficer": 5908
"Ġofficers": 6251
"ĠOffice": 7454
"ĠOfficer": 12743
"Ġoffices": 14145
```

```
"office": 30496
"Office": 33577
"ĠOfficers": 37209
```

The Ġ character refers to a space before the word. For instance, in the sentence ‘He stopped going to the office’, the space before the letter ‘o’ is considered part of the token. You can see that the tokens are case-sensitive - there is a separate token for ‘office’ and ‘Office’. Most models these days have case-sensitive vocabularies. Back in the day, BERT came with both a cased and an uncased version.

Cased vocabularies are almost always better, especially when you are training on such a huge body of text such that most tokens are seen by the model enough times so as to learn meaningful embeddings for them. For instance, there is a definite semantic difference between ‘web’ and ‘Web’ and it is good to have separate tokens for them.

Let’s search for some numbers. Ctrl+F for ‘93’. There are only three results

```
"93": 4590
"937": 47508
"930": 48180
```

It seems like not all numbers get their own tokens! Where is the token for 934? It is impractical to give every number its own token, especially if you want to limit your vocabulary size to just 50,000. As discussed in Chapter 2, the vocabulary size determines the size of the embedding layer and we do not want to see it become too large. We will discuss the impact of missing tokens later in this section.

Popular names and places get their own token. There is a token representing Boston, Toronto, and Amsterdam but none representing Mesa or Chennai. There is a token representing Ahmed and Donald, but none for Suhas or Maryam.

You might have noticed that tokens like

```
"]);": 9259
```

exist, indicating that GPT Neo-X is also primed to process programming languages.

EXERCISE

Go through the tokenizer.json file and explore the vocabulary in detail. Specifically,

- What are some unexpected tokens you see?
- What are the top ten longest tokens?
- Are there tokens representing words from other languages?

How are vocabularies determined? Surely, there was no executive committee holding emergency meetings burning midnight oil, with members making impassioned pleas to include the number *937* in the vocabulary at the expense of *934*?

Let us revisit the definition of a vocabulary

All the words in a language that are understood by a specific person

Since we want our language model to be an expert at English, we can just include all words in the English dictionary as part of its vocabulary. Problem solved?

Not nearly. What do you do when you communicate with the language model using a word that it has never seen? This happens a lot more often than you think. New words get invented all the time, words have multiple forms - ‘understanding’, ‘understanding’, ‘understandable’ etc, multiple words can be combined into a single word, and so on. Moreover, there are millions of domain-specific words (biomedical, chemistry etc)

THE DEFINITION OF A WORD

What exactly is a word, anyway? It is surprisingly very hard to answer this. Conceptually, you could say that a word is the smallest unit of text that has a self-contained meaning. This is not exactly true. For example, the word ‘snowball’ has components that have self-contained meanings of their own. Algorithmically, you can say that a word is just a sequence of characters separated by white space. This isn’t always true either. For example, the word ‘Hong Kong’ is generally regarded as a single word, even if it is separated by white space. Meanwhile the word ‘can’t’ could potentially be regarded as two or three words, even if there is no white space separating them.

NOTE

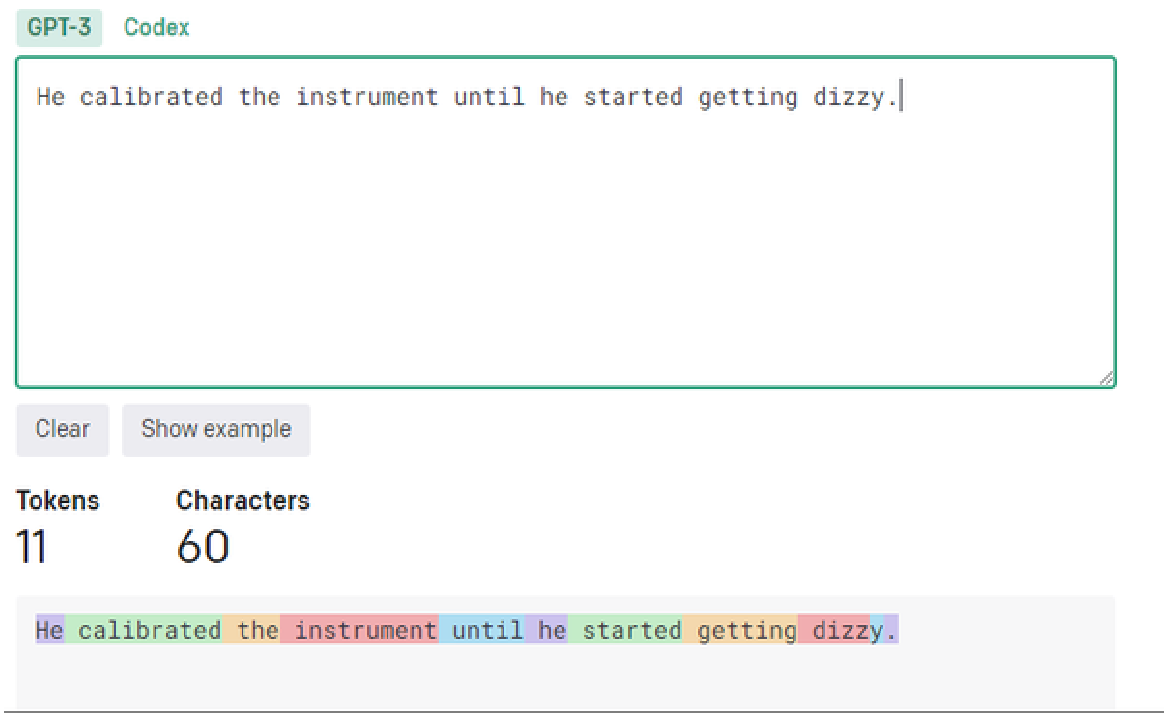
The twitter account ‘NYT first said’ tweets out words when they appear in the New York Times for the first time, excluding proper nouns. An average of 5 new words appear in the American paper of record for the first time each day. On the day I wrote this section, the words were ‘unflippant’, ‘dumbeyed’, ‘dewdrenched’, ‘faceflat’, ‘saporous, and ‘dronescape’. Many of these words might never get added to a dictionary.

A token that doesn’t exist in the vocabulary is called an OOV (Out-of-vocabulary) token. In Chapter 2, we saw how each token is assigned an embedding in the Transformer architecture. The architecture is fixed, and the number of embeddings in the embedding layer equals the size of the vocabulary of the model. Traditionally, OOV tokens were represented using a special <UNK> token. The <UNK> token is a placeholder for all tokens that don’t exist in the vocabulary. All OOV tokens share the same embedding (and encode the same meaning), which is undesirable. Moreover, the <UNK> token cannot be used in generative models. You don’t want your model to output something like

```
‘As a language model, I am trained to <UNK> sequences, and output
<UNK> text’.
```

To solve the OOV problem, one possible solution could be to represent tokens in terms of characters instead of words. Each character has its own embedding, and as long as all valid characters are included in the vocabulary, there will never be a chance of encountering an OOV token. However, there are many downsides to this. The number of tokens needed to represent the average sentence becomes much larger. For example, the previous sentence contains 13 tokens with a word tokenization scheme but 81 tokens with a character tokenization scheme. As seen in Chapter 2, the sequence length of a Transformer is limited, and the expanded number of tokens makes both training and inference slower, and reduces the amount of context that can be provided to a model in zero-shot or few-shot settings. Therefore, character-based tokens cannot be adapted without a significant change to the Transformer architecture. There have been attempts to do this including CANINE, ByT5, CharFormer, which we will discuss later in this section.

So, the middle ground and the best of both worlds (or the worst of both worlds, the field hasn't come to a consensus yet) is using subwords. Subwords are the predominant mode of representing vocabulary units in the language model space today. The GPT Neo-X vocabulary we explored earlier uses subword tokens. Figure 2-1 shows the Open AI tokenizer playground that demonstrates how words are split into their constituent subwords.



He calibrated the instrument until he started getting dizzy.

Tokens
11

Characters
60

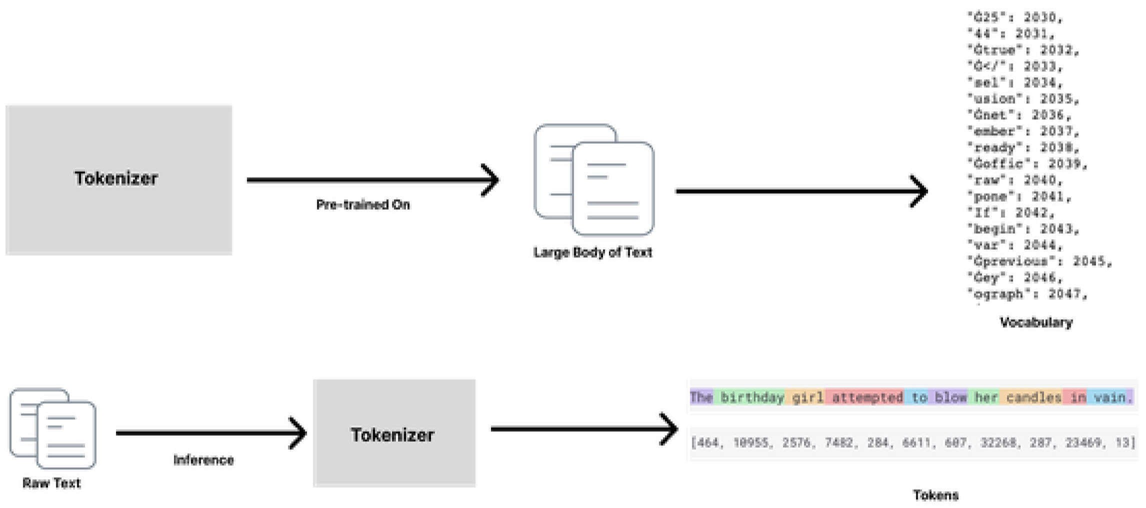
He calibrated the instrument until he started getting dizzy.

Figure 2-1. Subword Tokens

Tokenizer

A tokenizer has two responsibilities -

1. In the tokenizer pre-training stage, the tokenizer is run over a body of text to generate a vocabulary.
2. While processing input during both training and inference, free-form raw text is run through the tokenizer algorithm to break down the text into tokens. Figure 2-2 depicts the roles played by a tokenizer



```
"Ġ25": 2030,
"44": 2031,
"Ġtrue": 2032,
"Ġ</": 2033,
"sel": 2034,
"usion": 2035,
"Ġnet": 2036,
"ember": 2037,
"ready": 2038,
"Ġoffic": 2039,
"raw": 2040,
"pone": 2041,
"If": 2042,
"begin": 2043,
"var": 2044,
"Ġprevious": 2045,
"Ġey": 2046,
"ograph": 2047,
```

```
The birthday girl attempted to blow her candles in vain.
```

```
[464, 10955, 2576, 7482, 284, 6611, 607, 32268, 287, 23469, 13]
```

Figure 2-2. Tokenizer Workflow

When we feed raw text to the tokenizer, it breaks down the text into tokens that are part of the vocabulary, and maps the tokens to their token indices. The sequence of token indices (input ids) are then fed to the language model where they are mapped to their corresponding embeddings. Let us explore this process in detail.

This time, let's experiment with the FlanT5 model. You need a Google Colab Pro or equivalent system to be able to run it.

```
!pip install transformers accelerate sentencepiece
from transformers import T5Tokenizer, T5ForConditionalGeneration


tokenizer = T5Tokenizer.from_pretrained("google/flan-t5-xl")
model = T5ForConditionalGeneration.from_pretrained("google/flan-
t5-xl", device_map="auto")


input_text = "what is 937 + 934?"
encoded_text = tokenizer.encode(input_text)
tokens = tokenizer.convert_ids_to_tokens(encoded_text)
print(tokens)
```

The output is

```
['▁what', '▁is', '▁9', '37', '▁+', '▁9', '34', '?', '</s>']
```

The encoder() function tokenizes the input text and returns the corresponding token indices. The token indices are mapped to the tokens they represent using the convert_ids_to_tokens() function.

As you can see, the Flan-T5 tokenizer doesn't have dedicated tokens for the numbers 937 or 934. Therefore, it splits the numbers into '9' and '37'. The </s> token is a special token indicating the end of the string. The '_' means that the token is preceded by a space.

Let's try another example.

```
input_text = "Insuffienct adoption of corduroy pants is the
reason this

economy is in the dumps!!!"
encoded_text = tokenizer.encode(input_text)
tokens = tokenizer.convert_ids_to_tokens(encoded_text)
print(tokens)
```

The output is

```
['▁In', 's', 'uff', 'i', 'en', 'c', 't', '▁adoption', '▁of',
'▁cord', 'u',

'roy', '▁pants', '▁is', '▁the', '▁reason', '▁this', '▁economy',
'▁is', '▁in',

'▁the', '▁dump', 's', '!!!', '</s>']
```

I had made a deliberate typo with the word 'Insufficient'. Note that subword tokenization is rather brittle with typos. But at least the OOV problem has been dealt with by breaking down the words into subwords. The vocabulary also doesn't seem to have an entry for the word 'corduroy', thus confirming its poor sense of fashion. Meanwhile, there is a separate token for three contiguous exclamation points, which is different from the token that represents a single exclamation point. Semantically, they do convey slightly different meanings.

NOTE

Very large models trained on a massive body of text are more robust to misspellings. A lot of misspellings already occur in the training set. For example, even the rare misspelling '*Insuffienct*' occurs 14 times in the C4 pre-training dataset. The more common misspelling '*insufficent*' occurs over 1100 times. Larger models can also infer the misspelled word from its context. Smaller models like BERT are quite sensitive to misspellings.

If you are using models from Open AI, you can explore their tokenization scheme using the tiktoken library. (no relation to the social media website).

Using tiktoken, let's see the different vocabularies available in the Open AI ecosystem.

```
!pip install tiktoken

import tiktoken
tiktoken.list_encoding_names()
```

The output is

```
['gpt2', 'r50k_base', 'p50k_base', 'p50k_edit', 'cl100k_base']
```

The numbers like 50k/100k are presumed to be the vocabulary size. Open AI hasn't revealed much information about these. Their documentation does state that *cl100k_base* is used by GPT-4 and GPT 3.5 (chatGPT), while *p50k_base* is used by the Codex models, and the Instruct versions of GPT-3.

```
encoding = tiktoken.encoding_for_model("gpt-4")
input_ids = encoding.encode("Insuffienct adoption of corduroy
pants is the

reason this economy is in the dumps!!!")
tokens = [encoding.decode_single_token_bytes(token) for token in
input_ids]
```

The output is

```
[b'Ins', b'uff', b'ien', b'ct', b' adoption', b' of', b' cord',
b'uro', b'y',

b' pants', b' is', b' the', b' reason', b' this', b' economy', b'
is', b' in',

b' the', b' dumps', b'!!!']
```

As you can see there is not much of a difference between the tokenization used by GPT-4 and GPT Neo-X.

EXERCISE

Using tiktoken, find the difference between *p50k_base*, the encoding used for GPT 3.5 (chatGPT), and *cl100k_base*, the encoding used for GPT-4. What are the 50,000 extra tokens in the GPT-4 vocabulary representing?

TIP

While adapting LLM’s to your use case, If you see strange behavior from the model on a subset of your inputs, it is worthwhile to check how they have been tokenized. While you cannot definitively diagnose your problem just by analyzing the tokenization, it is often helpful in analysis. In my experience, a non-negligible amount of LLM failures can be attributed to the way the text was tokenized. This is especially true if your target domain is different from the pre-training domain.

TOKENIZATION-FREE MODELS

As discussed in Chapter 1, the *consolidation effect* has resulted in end-to-end architectures. However, one last hold-out is the tokenization step. You have seen in the code earlier that the tokenization is used as a pre-processing step to prepare the input to be fed into the model. The input to the model is the sequence of token indices and not raw text. But what if we make the model truly end-to-end by removing the tokenization step? Is it possible to directly feed raw text to the model and have it output results?

There have been forays into the world of tokenization-free language modeling, with models like CANINE, ByT5, and CharFormer.

- CANINE accepts Unicode codepoints as input. But there are 1,114,112 possible code points, rendering the vocabulary and resulting embedding layer size infeasible. To resolve this, CANINE uses hashed embeddings so that the effective vocabulary space is much smaller.
- ByT5 accepts input in terms of bytes, so there are only 259 embeddings in the embedding matrix (including a few special tokens), thus reducing the embedding layer size drastically.
- CharFormer also accepts input in terms of bytes, and passes it to a gradient-based subword tokenizer module, that constructs latent subwords.

Tokenization Pipeline

Figure 2-3 depics the sequence of steps performed by a tokenizer.

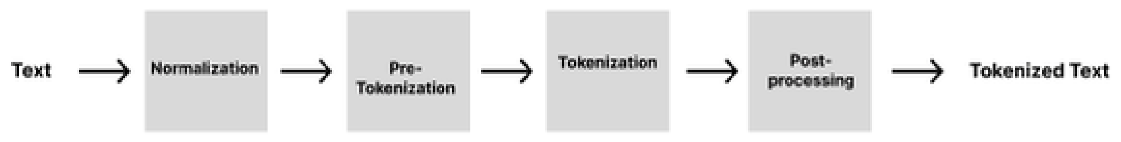


Figure 2-3. HuggingFace Tokenizers Pipeline

If you are using the tokenizers library from HuggingFace, your input text is run through a multi-stage tokenization pipeline. This pipeline is composed of four components -

- Normalization
- Pre-tokenization
- Tokenization
- Post-processing

Note that different models will have different steps executed within these 4 components.

Normalization

Different types of normalization applied include

- Converting text to lowercase (if you are using an uncased model)
- Stripping off accents from characters, like from the word Peña
- Unicode normalization

Let’s see what kind of normalization is applied on the uncased version of BERT:

```
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
print(tokenizer.backend_tokenizer.normalizer.normalize_str('Pédrò
pôntificated at üs:-)')
```

The output is

```
pedro pontificated at us:-)
```

As we see, the accents have been removed and the text has been converted to lowercase.

There isn't much normalization done in tokenizers for more recent models.

Pre-tokenization

Before we run the tokenizer on the text, we can optionally perform a pre-tokenization step. As mentioned earlier, most tokenizers today employ subword tokenization. A common step is to first perform word tokenization and then feed the output of it to the subword tokenization algorithm. This step is called pre-tokenization.

Pre-tokenization is a relatively easy step in English compared to many other languages, since you can start off with a very strong baseline by just splitting text on whitespace. There are outlier decisions to be made - how to deal with punctuation, multiple spaces, numbers etc. In HuggingFace the regular expression

```
\w+|[^\w\s]+
```

is used to split on whitespace.

Let's run the pre-tokenization step of the T5 tokenizer.

```
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-xl")
tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str("I'm
starting to

suspect - I am 55 years old!   Time to vist New York?")
```

The output is

```
[("▁I'm", (0, 3)),
 ('▁starting', (3, 12)),
 ('▁to', (12, 15)),
 ('▁suspect', (15, 23)),
 ('▁-', (23, 25)),
 ('▁I', (25, 27)),
 ('▁am', (27, 30)),
 ('▁55', (30, 33)),
 ('▁years', (33, 39)),
 ('▁old!', (39, 44)),
 ('▁', (44, 45)),
 ('▁', (45, 46)),
 ('▁Time', (46, 51)),
 ('▁to', (51, 54)),
 ('▁vist', (54, 59)),
 ('▁New', (59, 63)),
 ('▁York?', (63, 69))]
```

Along with the pre-tokens (or word tokens), the character offsets are returned.

The T5 pre-tokenizer splits only on whitespace, doesn't collapse multiple spaces into one, does't split on punctuation or numbers. The behavior can be vastly different for other tokenizers.

Tokenization

After the optional pre-tokenization step, the actual tokenization step is performed. Some of the important algorithms in this space are BPE (Byte Pair Encoding), Byte BPE, WordPiece, and Unigram LM. The tokenizer comprises a set of rules that is learned during a pre-training phase over a pre-training dataset. Now let's go through these algorithms in detail.

BPE (Byte Pair Encoding)

This algorithm is the simplest and most widely used tokenization algorithm.

Training stage

We take a training dataset, run it through the normalization and pre-tokenization steps discussed earlier, and record the unique tokens in the resulting output and their frequencies. We then construct an initial vocabulary consisting of the unique characters that make up these tokens. Starting from this initial vocabulary, we continue adding new tokens using *merge* rules. The merge rule is simple - we merge the most frequent consecutive pairs of tokens. The merges continue until we reach the desired vocabulary size.

Let’s explore this with an example. Imagine our training dataset is composed of six words, each appearing just once.

```
‘bat’, ‘cat’, ‘cap’, ‘sap’, ‘map’, ‘fan’
```

The initial vocabulary is then made up of

```
‘b’, ‘a’, ‘t’, ‘c’, ‘p’, ‘s’, ‘m’, ‘f’, ‘n’
```

The frequencies of contiguous token pairs are

```
‘ba’ - 1, ‘at’ - 2, ‘ca’ - 2, ‘ap’ - 3, ‘sa’ - 1, ‘ma’ - 1, ‘fa’
- 1, ‘an’ - 1
```

The most frequent pair is ‘ap’, so the first merge rule is to merge ‘a’ and ‘p’. The vocabulary now is

```
‘b’, ‘a’, ‘t’, ‘c’, ‘p’, ‘s’, ‘m’, ‘f’, ‘n’, ‘ap’
```

The new frequencies are -

```
‘ba’ - 1, ‘at’ - 2, ‘cap’ - 1, ‘sap’ - 1, ‘map’ - 1, ‘fa’ - 1,
‘an’ - 1
```

Now,the most frequent pair is ‘at’, so the next merge rule is to merge ‘a’ and ‘t’.This process continues until we reach the vocabulary size.

Inference stage

After the tokenizer has been trained, it can be used to divide the text into appropriate subword tokens and feed the text into the model. This happens in a similar fashion as the training step. After normalization and pre-tokenization of the input text, the resulting tokens are broken into individual characters and all the merge rules are applied in order. The tokens remaining after all merge rules have been applied are the final tokens which are then fed to the model.

You can open the vocabulary file for GPT Neo-X again and ctrl+f ‘merges’ to see the merge rules. As expected, the initial merge rules join single characters with each other. At the end of the merge list, you can see larger subwords like ‘out’ and ‘comes’ being merged into a single token.

EXERCISE

Implement the BPE algorithm by yourself, using a domain dataset of your choice. What tokens do you end up with and how does it differ from the vocabulary of the popular language models? This also gives you a clue on how effective general-purpose LM’s will be for your use case.

NOTE

Since all unique individual characters in the tokenizer training set will get their own token, it is guaranteed that there will be no OOV tokens as long as all tokens seen during inference in future are made up of characters that were present in the training set. But Unicode consists of over a million code points and around 150,000 valid characters, which would not fit in a vocabulary of size 30000. This means that if your input text contained a character that wasn’t in the training set, that character would be assigned an <UNK> token. To resolve this, a variant of BPE called Byte-level BPE is used. Byte-level BPE starts with 256 tokens, representing all the characters that can be represented by a byte. This ensures that every Unicode character can be encoded just by the concatenation of the constituent byte tokens. Hence it also ensures that we will never encounter an <UNK> token. GPT-n models use this tokenizer.

WordPiece

WordPiece is similar to BPE, so we will highlight only the differences.

Instead of the frequency approach used by BPE, WordPiece uses the maximum likelihood approach. The frequency of the token pairs in the dataset is normalized by the product of the frequency of the individual tokens. The pairs with the resulting highest score are then merged.

```
score = freq(a,b)/(freq(a) * freq(b))
```

This means that lower frequency terms are joined first.

In WordPiece, merge rules are not used. Instead, for each pre-tokenized token in the input text, the tokenizer finds the longest subword from the vocabulary in the token and splits on it. For example, if the token is ‘understanding’ and the longest subword in the dictionary within this token is ‘understand’, then it will be split into ‘understand’ and ‘ing’.

Postprocessing

The final stage of the tokenizer pipeline is the postprocessing stage. This is where model specific special tokens are added. Common tokens include [CLS] or the classification token used in many language models, and [SEP], a separator token used to separate parts of the input.

THE CURIOUS CASE OF SOLIDMAGIGOLDKARP.

There are weird tokens that end up being part of a language model’s vocabulary, due to the way the tokenization algorithms work. One such token is ‘SolidMagiGoldkarp’, representing a now-deleted Reddit user who was one of the site’s most active posters because of his quest to count to infinity. This was a token in the GPT-2 tokenizer. The same tokenizer was used in GPT-3 models but the pre-training dataset of the model had changed, so now a token existed for SolidMagiGoldkarp but there was no signal in the pre-training dataset to learn from. This leads to some anomalous and hilarious behavior in GPT-N models.

EXERCISE

Token archaeology is a new hobby for many LLM enthusiasts. This involves finding rare tokens in the vocabulary of language models, and unearthing its origin. This is not just fun and games though, as knowing the origin of rare tokens can give you an insight into the characteristics of the pre-training dataset. Using tiktoken, find some rare vocabulary terms in GPT-3.5 or GPT-4's vocabulary. Can you figure out their origins?

Special Tokens

Depending on the model, there are a few special tokens that are added to the vocabulary to facilitate processing. These tokens include

- <PAD> - to indicate padding, in case the size of the input is lesser than the maximum sequence length.
- <EOS> - to indicate the end of the sequence. Generative models stop generating after outputting this token.
- <UNK> - to indicate an OOV term

As we have seen, if our data is domain-specific like healthcare, scientific literature etc, tokenization from a general-purpose tokenizer will be unsatisfactory. GALACTICA by Meta introduced several domain specific tokens in their model and special tokenization rules

- [START_REF] and [END_REF] for wrapping citations.
- <WORK> token to wrap tokens that make up an internal working memory, used for reasoning and code generation
- Numbers are handled by assigning each digit in the number its own token

- [START_SMILES], [START_DNA], [START_AMINO], [END_SMILES], [END_DNA], [END_AMINO] for protein sequences, DNA sequences, and amino acid sequences respectively.

NOTE

Why is the vocabulary size so large? Surely, having a smaller vocabulary size would be more convenient as the size of the embedding matrix would be smaller. However, the smaller the vocabulary, the more number of tokens needed to represent a sequence, which would make the model slower in both training and inference.

Learning Objectives

Now that we have discussed the pre-training dataset and vocabulary, let us move on to the next ingredient of the language model - the Learning Objective. Language models are pre-trained in a self-supervised manner. The scale of data we need to train them makes it prohibitively expensive to perform supervised learning, where (input, output) examples need to come from humans. Instead, we use a form of training called self-supervision, where the data itself contains the target labels. The goal of self-supervised learning is to learn a task which acts as a proxy for learning the syntax and semantics of a language, as well as skills like reasoning, arithmetic and logical manipulation, and other cognitive tasks, and (hopefully) eventually leading up to general human intelligence. How does this work?

For example, let’s take the canonical language modeling task - predicting the next word that comes in a sequence. Consider the sequence

```
'Tammy jumped over the'
```

and the language model is asked to predict the next token. The total number of possible answers is the size of the vocabulary. There are a lot of valid continuations to this sequence - like (hedge, fence, barbecue, sandcastle etc), but there are many continuations to this sequence that would violate English grammar rules like (is, of, the). During the training process, after seeing billions of sequences, the model will know that it is highly improbable for the word *the* to be followed by the word *is* or *of*, regardless of the surrounding context. Thus, you can see how just predicting the next token is such a powerful tool - in order to correctly predict the next token you can eventually learn more and more complex functions that you can encode in your model connections. However, whether this paradigm is all we need to develop general intelligence is an open question.

Self-supervised learning objectives used for pre-training LLMs can be broadly classified (non-exhaustively) into three types:

- FLM (Full Language Modeling)
- MLM (Masked Language Modeling)
- PrefixLM (Prefix Language Modeling)

Let’s explore these in detail.

Full Language Modeling

Figure 2-4 shows the canonical FLM objective at work

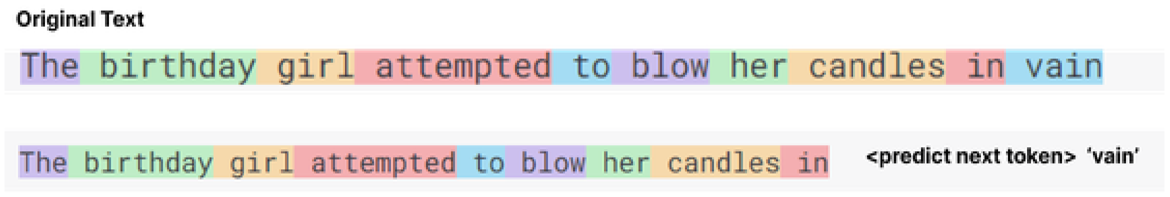

Original Text
The birthday girl attempted to blow her candles in vain
The birthday girl attempted to blow her candles in
<predict next token> 'vain'

Figure 2-4. Full Language Modeling

This is the canonical language modeling objective of learning to predict the next token in a sequence.This is currently the simplest and most common training objective, used by GPT-4 and a vast number of open-source models. The loss is computed for every token the model sees, i.e every single token in the training set that is being asked to be predicted by the language model provides a learning signal for the model, making it very efficient.

Let us explore an example, using the GPT-Neo model.

Suppose we continue pre-training the GPT-Neo model from its publicly available checkpoint, using the full language modeling objective. Let's say the current training sequence is

```
'Language models are ubiquitous'
```

You can run this code

```
import torch
from transformers import AutoTokenizer, GPTNeoForCausalLM


tokenizer = AutoTokenizer.from_pretrained("EleutherAI/gpt-neo-
1.3B")
model = GPTNeoForCausalLM.from_pretrained("EleutherAI/gpt-neo-
1.3B")


input_ids = tokenizer("Language models are", return_tensors="pt")
gen_tokens = model.generate(**input_ids, max_new_tokens =1,

output_scores=True, return_dict_in_generate=True)
output_scores = gen_tokens["scores"]
scores_tensor = output_scores[0]
sorted_indices = torch.argsort(scores_tensor[0], descending=True)
[:20]


for index in sorted_indices:
    token_id = index
    token_name = tokenizer.decode([token_id.item()])
    token_score = scores_tensor[0][index].item()
    print(f"Token: {token_name}, Score: {token_score}")
```

This code tokenizes the input text *Language models are* and feeds it to the model by invoking the generate() function. The function predicts the continuation, given the sequence 'Language models are'. It outputs only one token and stops generating because max_new_tokens is set to 1. The rest of the code enables it to output the top 20 list of tokens with the highest score, prior to applying the softmax at the last layer.

The top 20 tokens with the highest prediction score are

```
Output: Token:  a, Score: -1.102203369140625
Token:  used, Score: -1.4315788745880127
Token:  the, Score: -1.7675716876983643
Token:  often, Score: -1.8415470123291016
Token:  an, Score: -2.4652323722839355
Token:  widely, Score: -2.657834053039551
Token:  not, Score: -2.6726579666137695
Token:  increasingly, Score: -2.7568516731262207
Token:  ubiquitous, Score: -2.8688106536865234
Token:  important, Score: -2.902832508087158
Token:  one, Score: -2.9083480834960938
Token:  defined, Score: -3.0815649032592773
Token:  being, Score: -3.2117576599121094
Token:  commonly, Score: -3.3110013008117676
Token:  very, Score: -3.317342758178711
Token:  typically, Score: -3.4478530883789062
Token:  complex, Score: -3.521362781524658
Token:  powerful, Score: -3.5338563919067383
Token:  language, Score: -3.550961971282959
Token:  pervasive, Score: -3.563507080078125
```

Every word in the top 20 seems to be a valid continuation of the sequence. The ground truth is the token ‘ubiquitous’, which we can use to calculate the loss and initiate the backpropagation process for learning.

As an another example, consider the text sequence

```
'I had 25 eggs. I gave away 12. I now have 13'
```

Run the same code as previously, except for this change.

```
input_ids = tokenizer("'I had 25 eggs. I gave away 12. I now
have", return_tensors="pt")
```

The top 20 output tokens are:

```
Token:  12, Score: -2.3242850303649902
Token:  25, Score: -2.5023117065429688
Token:  only, Score: -2.5456185340881348
Token:  a, Score: -2.5726099014282227
Token:  2, Score: -2.6731367111206055
```

```
Token:  15, Score: -2.6967623233795166
Token:  4, Score: -2.8040688037872314
Token:  3, Score: -2.839219570159912
Token:  14, Score: -2.847306728363037
Token:  11, Score: -2.8585362434387207
Token:  1, Score: -2.877161979675293
Token:  10, Score: -2.9321107864379883
Token:  6, Score: -2.982785224914551
Token:  18, Score: -3.0570476055145264
Token:  20, Score: -3.079172134399414
Token:  5, Score: -3.111320972442627
Token:  13, Score: -3.117424726486206
Token:  9, Score: -3.125835657119751
Token:  16, Score: -3.1476120948791504
Token:  7, Score: -3.1622045040130615
```

The correct answer has the 17th highest score. A lot of numbers appear in the top 10, showing that the model is more or less random guessing the answer, which is not surprising for a smaller model like GPT-Neo

The Open AI API provides the 'logprobs' parameter that allows you to specify the number of tokens along with their log probabilities that need to be returned. This is available for GPT-3, but not yet for ChatGPT. The tokens returned are in order of their log probabilities.

```
import openai
openai.api_key = <Insert your Open AI key>


openai.Completion.create(
  model="text-davinci-003",
  prompt="I had 25 eggs. I gave away 12. I now have ",
  max_tokens=1,
  temperature=0,
  logprobs = 10
)
```

This code calls the older 'text-davinci-003' (GPT-3) model, asking it to generate a maximum of one token.The output is

```
"top_logprobs": [
          {
            "\n": -0.08367541,
            " 13": -2.8566456,
            "____": -4.579212,
            "_____": -4.978668,
            "________": -6.220278
          }
```

GPT-4 is pretty confident that the answer is 13, and rightfully so. The rest of the top probability tokens are all related to output formatting.

TIP

During inference, we don’t necessarily need to generate the token with the highest score. There are several *decoding strategies* that allow you to generate more diverse text. We will discuss these strategies in Chapter 4.

EXERCISE

Ask the text-davinci-003 model to solve individual crossword clues in the Washington Post Daily Crossword. You may have to iterate with the prompt. A good start would be ‘Solve this crossword and answer in one word. The clue is <X> and it is a <Y> letter word. The answer is ‘. Set max_tokens = 3 to account for formatting tokens. Analyze the logprobs output. Is it dangerously close to getting it right/wrong? How many clues does it answer correctly?

Prefix Language Modeling

Prefix LM is similar to the FLM setting. The difference is that FLM is fully causal, i.e in a left-to-right writing system like English, tokens do not attend to tokens to the right (future). In the prefix LM setting, a part of the text sequence, called the prefix, is allowed to attend to future tokens in the prefix. The prefix part is thus non-causal. For training prefix LMs, a random prefix length is sampled, and the loss is calculated over only the tokens in the suffix.

Masked Language Modeling

Figure 2-5 shows the canonical MLM objective at work

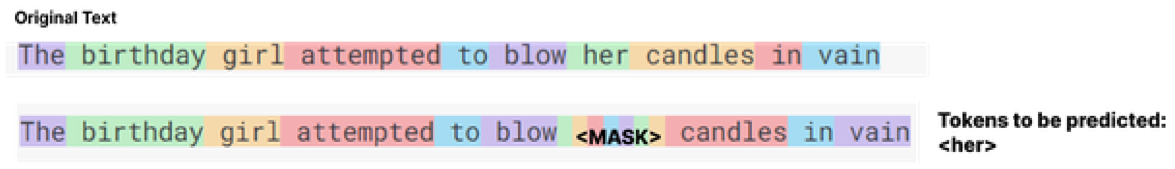

Original Text
The birthday girl attempted to blow her candles in vain
The birthday girl attempted to blow <MASK> candles in vain
Tokens to be predicted:
<her>

Figure 2-5. Masked Language Modeling in BERT

In the MLM setting, rather than predict the next token in a sequence, we ask the model to predict masked tokens within the sequence. In the most basic form of MLM implemented in the BERT model, 15% of tokens are randomly chosen to be masked and are replaced with a special mask token, and the language model is asked to predict the original tokens.

T5

The T5 model creators used a modification of the original MLM objective. In this variant, 15% of tokens are randomly chosen to be removed from a sequence. Consecutive dropped-out tokens are replaced by a single unique special token called the *sentinel token*. The model is then asked to predict and generate the dropped tokens, delineated by the sentinel tokens.

As an example, consider this sequence

```
'Tempura has always been a source of conflict in the family due
to unexplained reasons'
```

Let’s say we drop the tokens ‘has’, ‘always’, ‘of’, ‘conflict’. The sequence is now

```
'Tempura <S1> been a source <S2> in the family due to unexplained
reasons'
```

with S1, S2 being the sentinel tokens. The model is expected to output

```
‘<S1> has always <S2> of conflict <E>’
```

The output sequence is terminated by another sentinel token indicating the end of the sequence.

Generating only the dropped tokens and not the entire sequence is computationally more efficient and saves training time. Note that unlike in Full Language Modeling, the loss is calculated over only a small proportion of tokens (the masked tokens) in the input sequence.

Let’s explore this on HuggingFace

```
from transformers import T5Tokenizer, T5ForConditionalGeneration

tokenizer = T5Tokenizer.from_pretrained("t5-3b")
model = T5ForConditionalGeneration.from_pretrained("t5-3b")

input_ids = tokenizer("Tempura <extra_id_0>  been a source
<extra_id_1> in the
family due to unexplained reasons",
return_tensors="pt").input_ids
targets = tokenizer("<extra_id_0> has always <extra_id_1> of
conflict

<extra_id_2>", return_tensors="pt").input_ids
loss = model(input_ids=input_ids, labels=labels).loss
```

The targets can be prepared using a simple templating function.

EXERCISE

Play around with different masking strategies. Specifically,

- Change the masking rate. What happens if you mask 30% or 50% of tokens?
- Change the masking strategy. Can you do better than random masking? What heuristics would allow you to mask tokens that would contribute more towards learning?

More generally, masked language modeling can be interpreted as a *denoising autoencoder*. You corrupt your input by adding noise(masking, dropping tokens), and then you train a model to regenerate the original input. BART takes this to the next level by using 5 different types of span corruptions:

- Random token masking ala BERT. Figure 2-6 depicts the corruption and denoising steps.

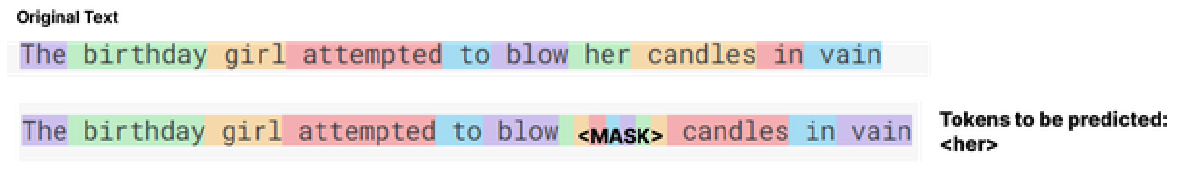

Original Text
The birthday girl attempted to blow her candles in vain
The birthday girl attempted to blow <MASK> candles in vain
Tokens to be predicted:
<her>

Figure 2-6. Random token masking in BART

- Random token deletion. The model needs to predict the positions in the text where tokens have been deleted. Figure 2-7 depicts the corruption and denoising steps.

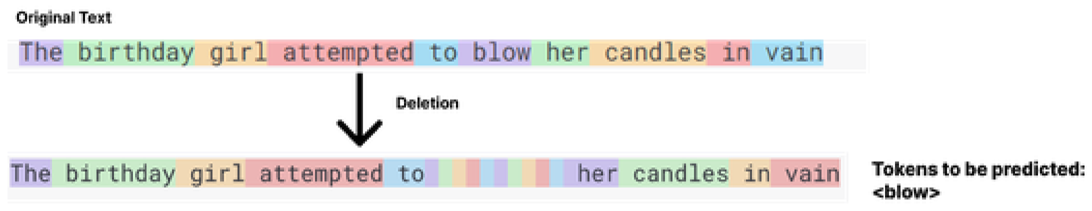

Original Text
The birthday girl attempted to blow her candles in vain
Deletion
The birthday girl attempted to her candles in vain
Tokens to be predicted:
<blow>

Figure 2-7. Random token deletion in BART

- Text spans are sampled from text, with span lengths coming from a Poisson distribution. This means 0 length spans are possible. The spans are deleted from the text and replaced with a single mask token. Therefore the model now has to also predict the number of tokens deleted. Figure 2-8 depicts the corruption and denoising steps.

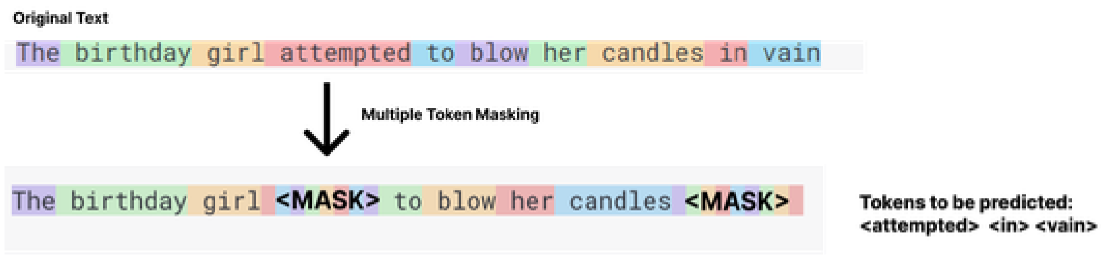

Original Text
The birthday girl attempted to blow her candles in vain
Multiple Token Masking
The birthday girl <MASK> to blow her candles <MASK>
Tokens to be predicted:
<attempted> <in> <vain>

Figure 2-8. Span masking in BART

- Sentences in the input document are shuffled.The model is taught to arrange them in the right order. Figure 2-9 depicts the corruption and denoising steps.

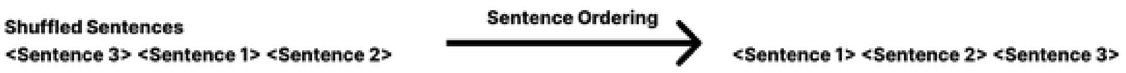

Shuffled Sentences
<Sentence 3> <Sentence 1> <Sentence 2>
Sentence Ordering
<Sentence 1> <Sentence 2> <Sentence 3>

Figure 2-9. Document shuffling objective in BART

- The document is rotated so that it starts from an arbitrary token. The model is trained to detect the correct start of the document. Figure 2-10 depicts the corruption and denoising steps.

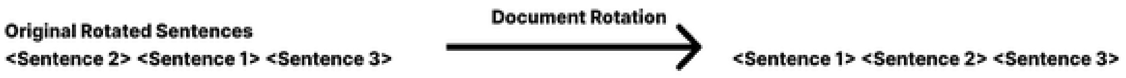

Original Rotated Sentences
<Sentence 2> <Sentence 1> <Sentence 3>
Document Rotation
<Sentence 1> <Sentence 2> <Sentence 3>

Figure 2-10. Document rotation objective in BART

Which learning objectives are better?

It has been shown that models trained with FLM are better at generation, and models trained with MLM are better at classification tasks. However, it is inefficient to use different language models for different use cases. The consolidation effect continues to take hold, with the introduction of UL2, a new paradigm that combines the best of different learning objective types in a single model.

UL2

UL2 mimics the effect of PLMs, MLMs, and PrefixLMs in a single paradigm called *Mixture of Denoisers*.

The denoisers used are -

- R-Denoiser - This is similar to the T5 span corruption task. Spans between length 2-5 tokens are replaced by a single mask token. Figure 2-11 depicts the workings of the R-denoiser.

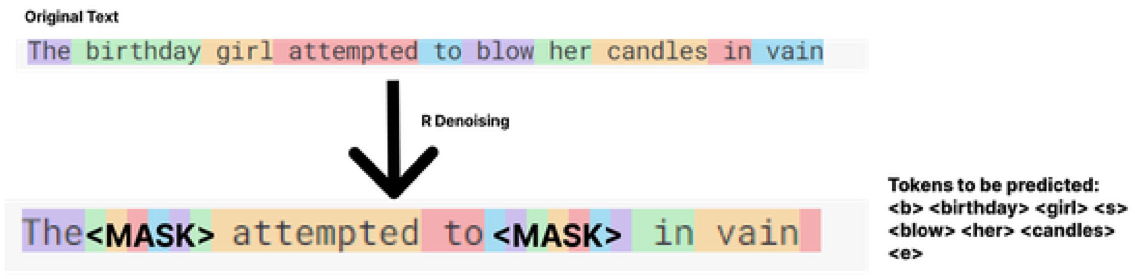

Original Text
The birthday girl attempted to blow her candles in vain
R Denoising
The<MASK> attempted to <MASK> in vain
Tokens to be predicted:
<b> <birthday> <girl> <s>
<blow> <her> <candles>
<e>

Figure 2-11. UL2's R-Denoiser

- S-Denoiser - Similar to prefix LM, the text is divided into a prefix and a suffix. The suffix is masked, while the prefix has access to bidirectional context. Figure 2-12 depicts the workings of the S-denoiser.

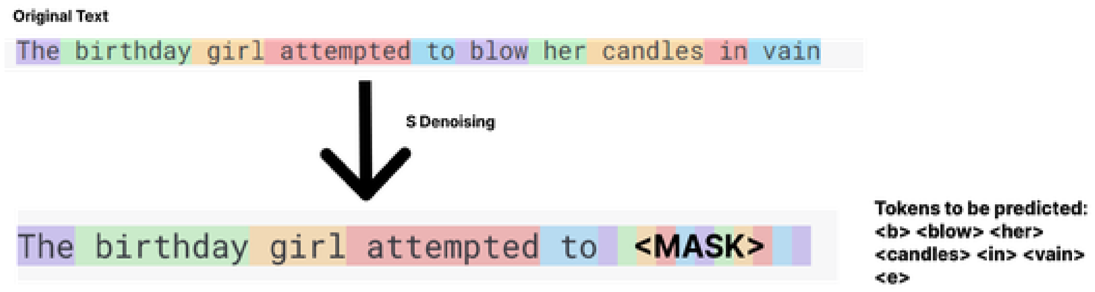

Original Text
The birthday girl attempted to blow her candles in vain
S Denoising
The birthday girl attempted to <MASK>
Tokens to be predicted:
<b> <blow> <her>
<candles> <in> <vain>
<e>

Figure 2-12. UL2's S-Denoiser

- X-Denoiser - This stands for extreme denoising, where a large proportion of text is masked (often over 50%). Figure 2-13 depicts the workings of the X-denoiser.

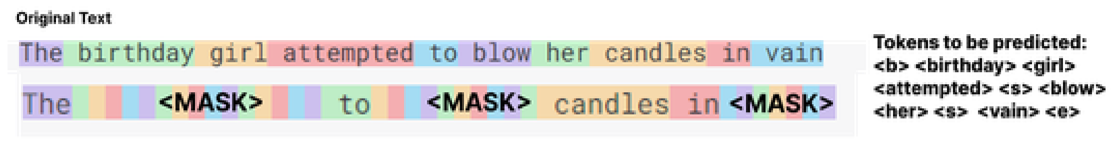


Figure 2-13. UL2's X-Denoiser

Architecture

After covering the pre-training dataset, tokenization, and the learning objective, the final piece of the puzzle is the model architecture itself.

As mentioned in Chapter 2, most modern language models are based on the Transformer architecture. Recall that the original Transformer architecture is made up of an encoder and a decoder. In practice, there are three major types of architecture backbones used:

- Encoder-only
- Encoder-Decoder
- Decoder-only

Encoder-only architectures

Encoder-only architectures were all the rage when Transformer-based language models first burst on the scene. Iconic language models from yesteryears (circa 2018) that use encoder-only architectures include BERT, RoBERTa, etc.

There aren't really many encoder-only LLM's being trained these days. Some reasons are:

- It is relatively harder to train them.
- The masked language modeling objective typically used to train them provides a learning signal in only a small percentage of tokens (the masking rate), thus needing a lot more data in order to reach the same level of performance as decoder-only models.

- For every downstream task, you need to train a separate task specific head, making usage inefficient.

The creators of UL2 recommend that encoder-only models should be considered obsolete. While I personally wouldn't go that far, I generally agree with the arguments made above against using encoder-only LLMs. However, if you already have a satisfactory pipeline for your use case built around encoder-only models, I would say if it ain't broke, why fix it?

If you still want to explore encoder-only models, here are some rules of thumb you can follow.

- RoBERTa performs better than BERT most of the time, since it is trained a lot longer on more data, and adopts best practices learned after the release of BERT.
- DeBERTa is currently regarded as the most well performing encoder-only model, and also the largest one available (1.5B parameters)
- The distilled versions of encoder-only models like DistillBERT etc, are not too far off from the original models in terms of performance, and should be considered if you are operating under resource constraints.

Several embedding models are built from encoder-only models. For example, perhaps one of the most important libraries in the field of NLP, considered the Swiss Army Knife of NLP tools, *sentence-transformers*, still provides encoder-only model based embedding models that are very widely used. '*all-mpnet-base-v2*', based on an encoder-only model called MPNet, and fine- tuned on several task datasets, is still competitive with much larger embedding models.

Encoder-Decoder Architectures

This is the original architecture of the Transformer, as it was first proposed. The T5 series of models uses this architectural type.

In encoder-decoder models, the input is text and the output is also text. A standardized interface ensures that the same model and training procedure can be used for multiple tasks. The inputs are handled by an encoder, and the outputs by the decoder.

Decoder-only Architectures

A majority of LLMs trained today use decoder-only models. Decoder-only models came into fashion starting from the original GPT model from Open AI. Decoder-only models excel at zero shot and few shot learning.

Decoder models can be causal and non causal. Non causal models have bidirectionality over the input sequence, while the output is still autoregressive (you cannot look ahead)

> **TIP**
>
> While the field is still evolving, there has been some compelling evidence for the following results:
>
> - Decoder-only models are the best choice for zero-shot and few-shot generationization
> - Encoder-decoder models are the best choice for multi-task fine tuning.
>
> The best of both worlds is to combine the two - Start with auto-regressive training, and then in an adaptation step, pre-train further with a non-casual setup using a span corruption objective.

Putting it all Together

The recipe for training each model is slightly different. As we have seen, at every step of the way there are a multitude of high-impact decisions to be made.

I often get this question from NLP practitioners - ‘Hey, I am tackling this <insert niche usecase> problem, what language model do you think I should use? There are hundreds of pre-trained models available out there and I have no idea how to choose among them.’ Truth be told, there are dozens of factors that can impact your choice, and sometimes it may not even be the most immediate or right question to ask. In subsequent chapters I will demonstrate how you can navigate tradeoffs and make an informed decision regarding your choice of model, and the various ways you can utilize them in your tasks.

NOTE

Depending on your task, the exact choice of pre-trained model used may not be as important as other data-related choices you need to make. Even in the era of GPT-4, your domain expertise and data cleaning skills are crucial for building successful applications. That being said, throughout the book, we will showcase scenarios where the choice of model can play a crucial role.

Summary

In this chapter, we discussed vocabularies and tokens, and delved into the different tokenization algorithms currently used. We also discussed the tasks that a language model is pre-trained on, and how they are a proxy to learning syntax and semantics. We also discussed the various architectural backbones of the Transformer.

Now that we know the recipe and ingredients behind LLMs, we will next learn how to utilize them to solve our own tasks. We will discuss techniques like fine-tuning, in-context learning, and zero-shot learning. We will also show how to evaluate LLMs for our use cases, and how to select the right model that suits our needs.

About the Author

Suhas Pai is an experienced machine learning researcher, having worked in the tech industry for over a decade. Currently he is the co-founder, CTO, and ML Research Lead at Bedrock AI, a Y-Combinator backed NLP startup operating in the financial domain. At Bedrock AI, Suhas invented several novel NLP techniques and LM based architectures that fully power the core features of Bedrock AI's products. Suhas is also the co-chair of the Privacy Working Group at Big Science for the BLOOM language model project, which when released was the world's largest open-source multilingual language model.

Suhas is active in the ML community, being the Chair of the TMLS (Toronto Machine Learning Summit) ML conference in 2022, as well as the Chair of the TMLS NLP conference in 2021 and 2022. He is also the NLP lead at Aggregate Intellect, an ML community research organization, where he previously wrote a weekly AI newsletter summarizing and contextualizing the latest papers in ML (audience: 3k+) and hosted a weekly seminar to walk through recent NLP papers.