El Shorouk Academy

Higher Institute of Computers & Information Technology

Department of Computer Science

Graduation Project

Project Title: Question Answering system

Project No: 1

Team Members:

Noha Ahed Ebrahim Ahmed Gad

Mohaned Ashraf Abdo Hassan EL-Haddad

Abdelrahman Mohamed Abu Alyazid

Mohamed Abd Alazez Hashim

Mohamed Ahmed Mohamed Ahmed

Academic Supervisor: Dr. Ahmed Kaboudan

Assistant Supervisor: Eng. Asmaa elsaid

Academic year

June 2021

Dedication

While we are taking our last steps in the last academic life we must pause and return to those years when we spent our time with our dear teachers at the University who gave us a lot of effort to send a new generation to our nation. We offer our deepest thanks and gratitude, appreciation, and love to those who carried a message in the most sacred of life and who paved the road of science and knowledge to us before we leave to all our distinguished teachers. A special and extreme gratitude we give to our final year project academic supervisor, Dr. Ahmed Kaboudan for his patience, encouragement, and guidance, whose contribution in stimulating suggestions and encouragement. For his surplus effort to help us to coordinate our project Furthermore, we would also like to sincerely thank Eng. Asmaa elsaid who taught us optimism to move forward and stood by our side when we lost our way and provided us with assistance, facilities, ideas, information, and guidance during the stages of this work. We sincerely thank each person who provides us help. The services we got from the Management Information Systems department at El Shorouk Academy may be that they don't feel their role, but they must get great thanks from us. Without Those people we would not achieve anything from what we have right now. All our thanks and appreciation to them.

Abstract

Stack overflow is a website that is visited by millions of users for different purposes like asking questions, giving solutions and others who are looking for solutions which are our target in our project. Most researchers say that most visitors look for the code of solution without caring about the explanation of the code. So, we decided to make AI that can figure the code of user by only asks about its example if i want to search for program that count from 1 to 10 it returns to us the code that do this but we only targeting python language in our project. Which can save users time to look all over for answers with high accuracy and the most important thing is that it is all for free.

Acknowledgments

We would like to express our deepest appreciation to all those who provided us the possibility to complete this project A special thanks and gratitude we give to Dr. Ahmed Kaboudan, for his patience, encouragement, and guidance. Furthermore, we would also like to acknowledge with much appreciation the crucial role of TA. Asmaa Elsaid, for his patience, encouragement, and help. We must appreciate the guidance given by other supervisors as well as the panels especially in our project presentation that has improved our presentation skills thanks to their comment and advice.

Contents

1 Introduction

1.1 Project definition

Programmers all over the world face a lot of issues while their everyday coding session, they waste a lot of time trying to find the best solutions and solve this issue maybe more than the time of coding.

We implement a question answering system to help them find a solution for their problems easily by entering a question to the system and the result will be a long answer and a short answer.

The short answer is the code solving this problem as most of the programmers are seeking for the code without any details.

while the long answer will be the code and the text describing the code and the solution.

Our system is using semantic search to find solutions related to the meaning of the question, so it is not searching only for solutions have the exact keywords of the question as its one of modern search techniques.

1.2 Project objectives

- Our project intends to help programmers to find the most suitable solutions for issues facing them, by implementing a question answering system that you can ask a question about the issue facing you and the question answering system will find a suitable solution for this issue.
- Extract best answer from a huge number of documents.
- Not only the best answer but also a short one.

51

Update 01/12/2020: This issue re-emerged lately, (apparently) caused once again by some changes on the Google translation API.

A solution is being discussed (again) in this Github issue. Although there is not a definitive solution yet a Pull Request seem to be solving the problem: https://github.com/ssut/py-googletrans/pull/237.

While we wait for it to be approved it can be installed like this:

```
$ pip uninstall googletrans
$ git clone https://github.com/alainrouillon/py-googletrans.git
$ cd ./py-googletrans
$ git checkout origin/feature/enhance-use-of-direct-api
$ python setup.py install
```

Original Answer:

Apparently it's a recent and widespread problem on Google's side. Quoting various Github discussions, it happens when Google sends you directly the raw token.

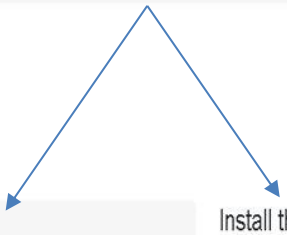
It's being discussed right now and there is already a pull request to fix it, so it should be resolved in the next few days.

For reference, see:

https://github.com/ssut/py-googletrans/issues/48 <-- exact same problem reported on the Github repo https://github.com/pndurette/gTTS/issues/60 <-- seemingly same problem on a text-to-speech library https://github.com/ssut/py-googletrans/pull/78 <-- pull request to fix the issue

To apply this patch (without waiting for the pull request to be accepted) simply install the library from the forked repo https://github.com/BoseCorp/py-googletrans.git (uninstall the official library first):

```
$ pip uninstall googletrans
$ git clone https://github.com/BoseCorp/py-googletrans.git
$ cd ./py-googletrans
$ python setup.py install
```



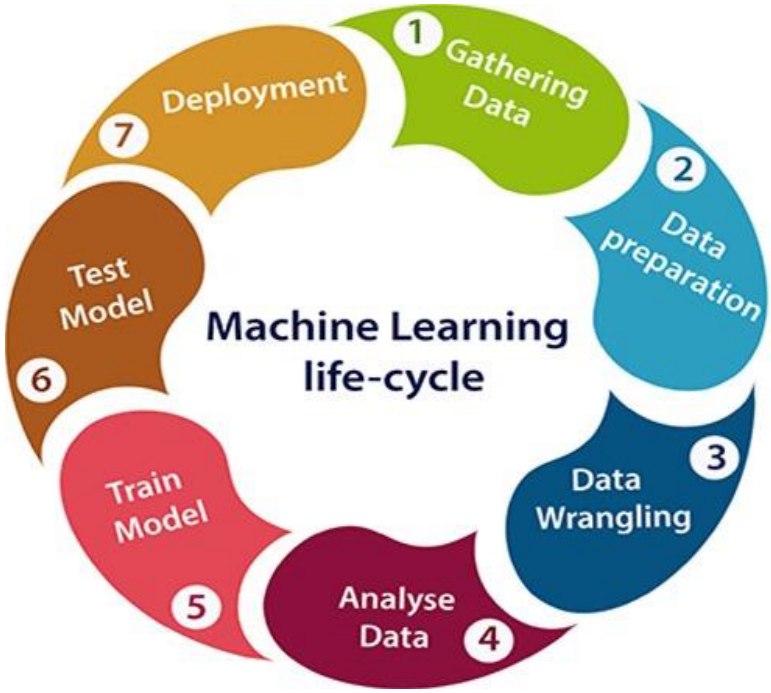

```
$ pip uninstall googletrans
$ git clone https://github.com/alainrouillon/py-googletrans.git
$ cd ./py-googletrans
$ git checkout origin/feature/enhance-use-of-direct-api
$ python setup.py install
```


Install the alpha version like this:

```
pip install googletrans==3.1.0a0
```

1.3 Machine learning Project Life cycle

Machine learning has given the computer systems the abilities to automatically learn without being explicitly programmed. But how does a machine learning system work? So, it can be described using the life cycle of machine learning. Machine learning life cycle is a cyclic process to build an efficient machine learning project. The main purpose of the life cycle is to find a solution to the problem or project.



Machine learning life cycle involves seven major steps, which are given below:

1. Gathering Data

Data Gathering is the first step of the machine learning life cycle. The goal of this step is to identify and obtain all data-related problems.

By performing the above task, we get a coherent set of data, also called as a **dataset**. It will be used in further steps.

2. Data preparation

After collecting the data, we need to prepare it for further steps. Data preparation is a step where we put our data into a suitable place and prepare it to use in our machine learning training.

In this step, first, we put all data together, and then randomize the ordering of data.

This step can be further divided into two processes:

- Data exploration
- Data pre-processing

3. Data Wrangling

Data wrangling is the process of cleaning and converting raw data into a useable format. It is the process of cleaning the data, selecting the variable to use, and transforming the data in a proper format to make it more suitable for analysis in the next step. It is one of the most important steps of the complete process. Cleaning of data is required to address the quality issues.

It is not necessary that data we have collected is always of our use as some of the data may not be useful. In real-world applications, collected data may have various issues, including:

- Missing Values
- Duplicate data
- Invalid data
- Noise

So, we use various filtering techniques to clean the data.

It is mandatory to detect and remove the above issues because it can negatively affect the quality of the outcome.

4. Data Analysis

Now the cleaned and prepared data is passed on to the analysis step.

5. Train Model

Now the next step is to train the model, in this step we train our model to improve its performance for better outcome of the problem.

We use datasets to train the model using various machine learning algorithms. Training a model is required so that it can understand the various patterns, rules, and features.

6. Test Model

Once our machine learning model has been trained on a given dataset, then we test the model. In this step, we check for the accuracy of our model by providing a test dataset to it.

Testing the model determines the percentage accuracy of the model as per the requirement of project or problem.

7. Deployment

The last step of machine learning life cycle is deployment, where we deploy the model in the real-world system.

If the above-prepared model is producing an accurate result as per our requirement with acceptable speed, then we deploy the model in the real system. But before deploying the project, we will check whether it is improving its performance using available data or not. The deployment phase is similar to making the final report for a project.

1.4 Project Plan

1	Plans	Begins	Ends
2	Studying Machine learning	Oct 18	Oct 29
3	Studying Deep learning	Nov 1	Nov 26
4	Midterm Exams (First term)	Nov 29	Dec 3
5	Studying NLP	Dec 6	Jan 14
6	System analysis	Jan 17	Jan 28
7	UML design	Jan 31	Feb 4
8	System architecture selection	Feb 7	Feb 11
9	Dataset selection	Feb 14	Feb 18
10	Dataset understanding	Feb 21	Feb 25
11	Final exams (First term)	Feb 28	Mar 25
12	Preparation of first term presentation	Mar 28	Apr 13
13	Presentation Discussion	Apr 14	Apr 14
14	Data preprocessing	Apr 18	Apr 29
15	Model training & development	May 2	Jun 17
16	Midterm Exams (Second term)	May 16	May 23
17	Project Documentation	May 24	Jun 20
18	Preparation of Final presentation	Jun 20	Jul 11
19	Final Exams (Second term)	Jun 17	Jun 28
20	Graduation Project Discussion	Jul 11	Jul 11

1.5 Project Phases

- Overview about question answering systems
- Learning machine learning
- Learning deep learning
- Learning NLP
- Deep Searching and studying different QA systems
- Studying and researching about Retrieval and reader
- Selecting the dataset
- Data analysis & pre-processing
- Model implementation
- Model training
- Model evaluation

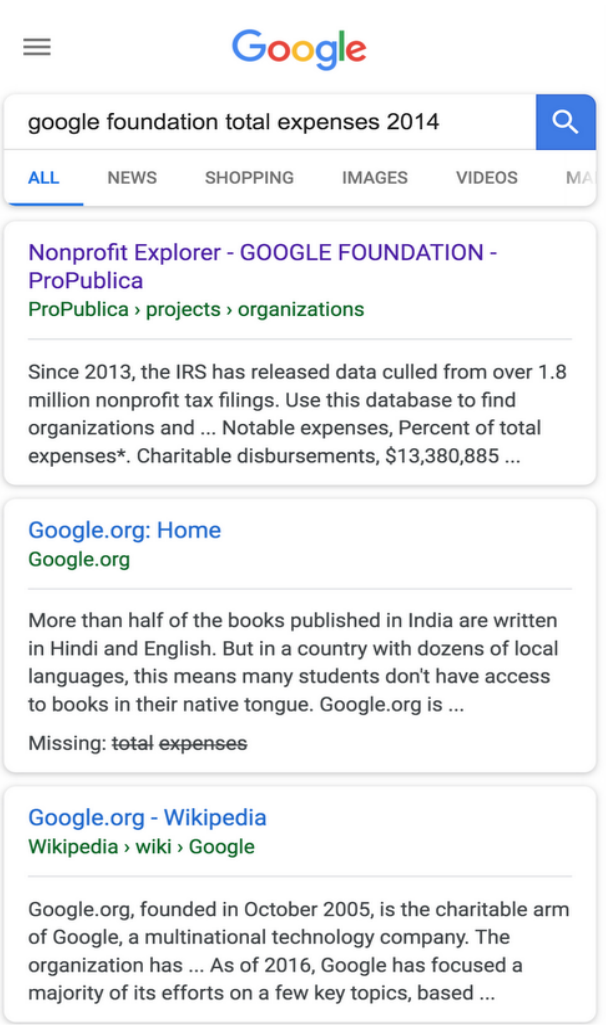
1.6 Technology and Tools

2 Introduction to Question Answering System

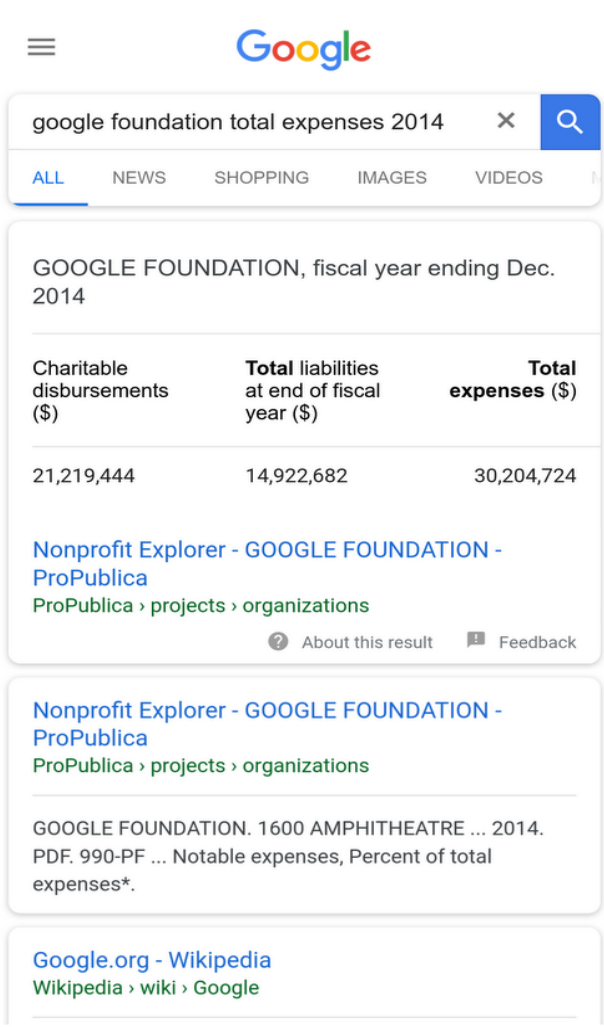
2.1 Introduction to search

2.1.1 Conventional search vs Modern search

Conventional

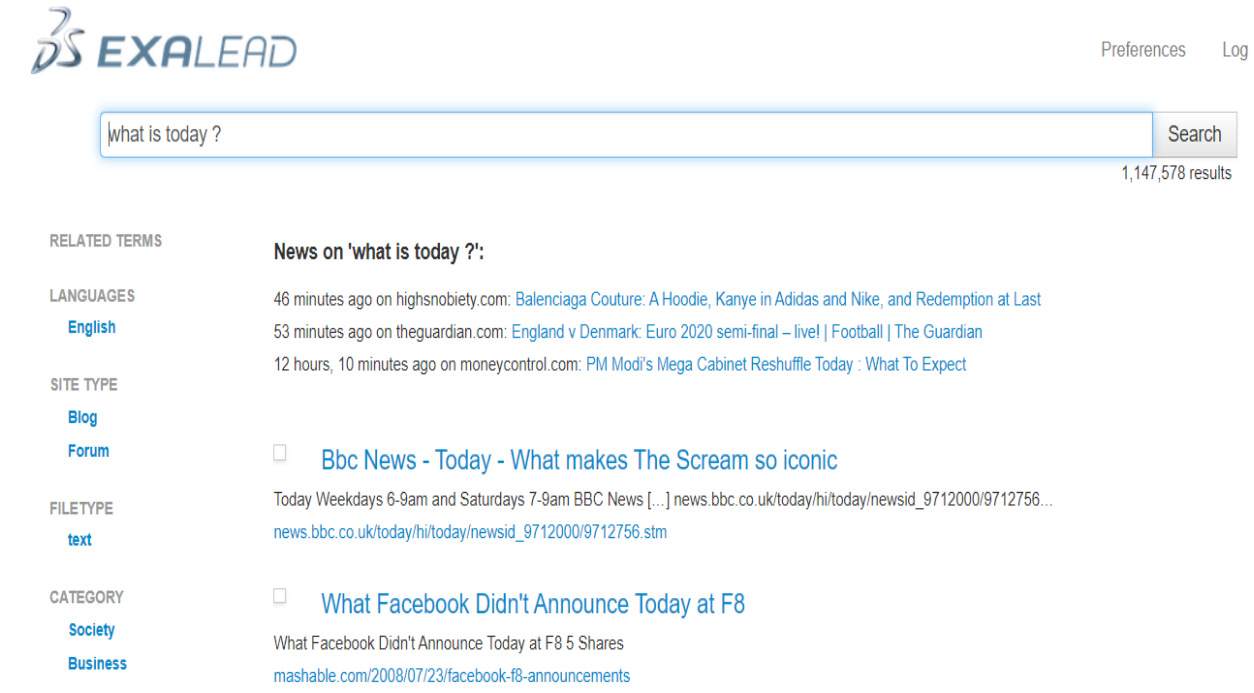


Modern

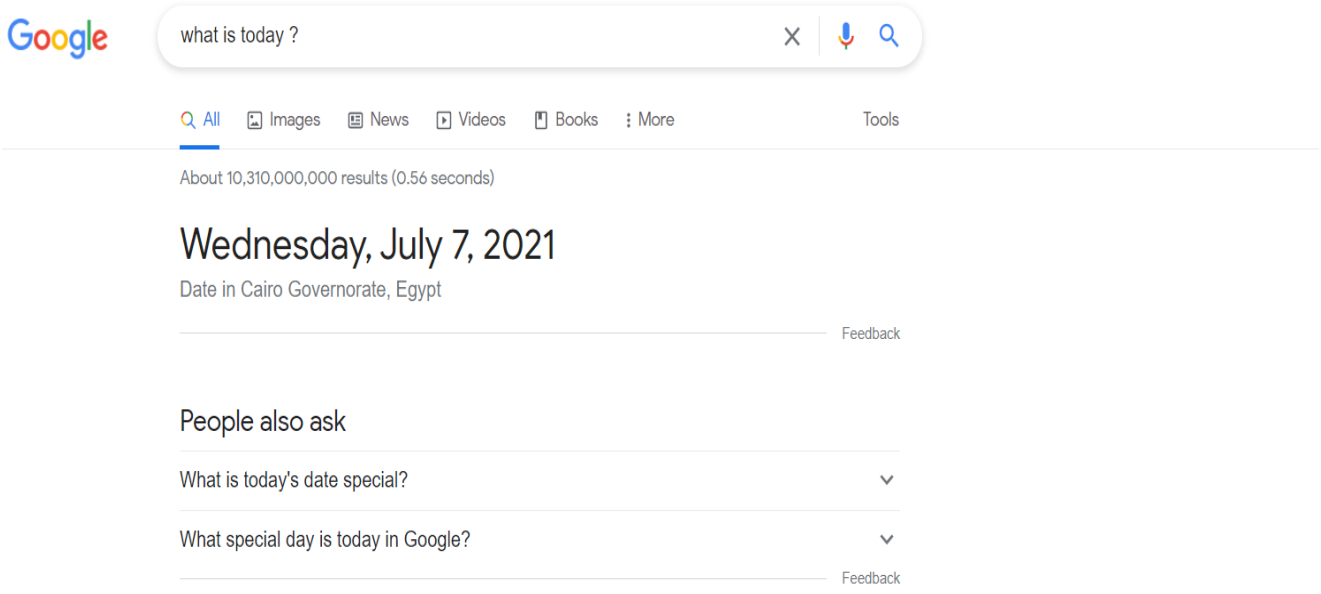


2.1.2 Syntax VS Semantic

Search by syntax



Search by semantic



2.2 Natural language processing background

Natural language processing (NLP) is an area of computer science and artificial intelligence concerned with the interaction between computers and humans in natural language.

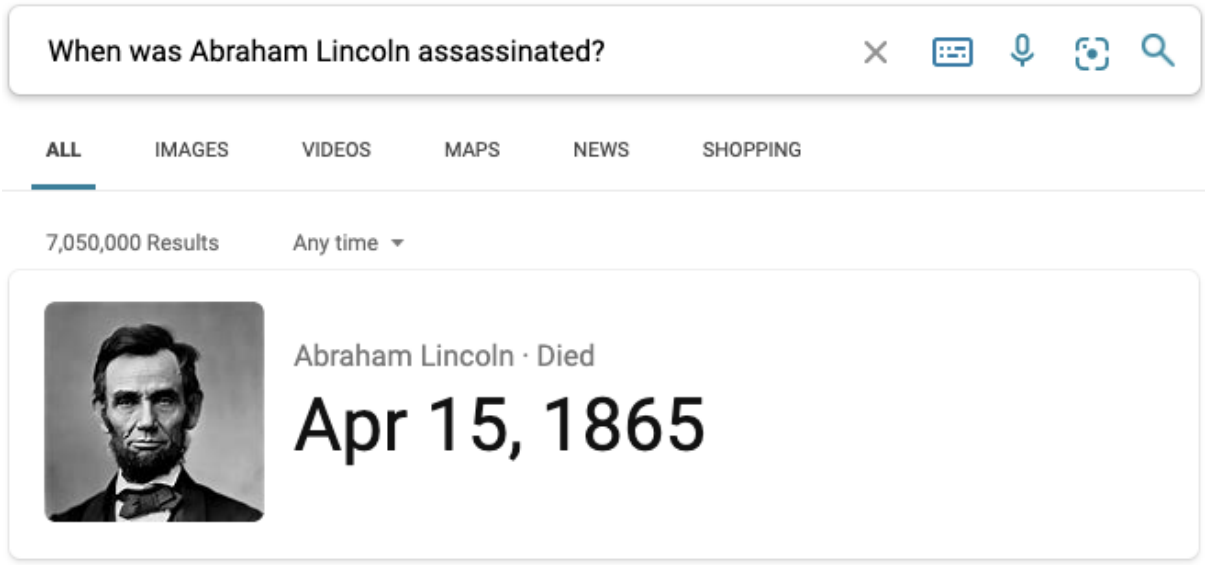
NLP is a branch of artificial intelligence that deals with analyzing, understanding and generating the languages that humans use naturally in order to interface with computers in both written and spoken contexts using natural human languages instead of computer languages.

QA systems

Question Answering(QA) system is a system that gives appropriate answers to questions expressed in natural languages such as English, Chinese, and so on. For example, suppose a user asks, "When was Abraham Lincoln assassinated?" In this case, the question answering system is expected to return "Apr 15, 1865". Below is an example of question-answer pairs.

Question	Answer
Where is the Louvre Museum located?	in Paris, France
What's the abbreviation for limited partnership?	L.P.
What are the names of Odin's ravens?	Huginn and Muninn
What currency is used in China?	the yuan
What kind of nuts are used in marzipan?	almonds
What instrument does Max Roach play?	drums
What's the official language of Algeria?	Arabic
How many pounds are there in a stone?	14

A question answering system will help you find information efficiently. We use search engines to search for relevant documents when we look for some information on the Web. However, because they show you documents, you must read the documents and decide whether they contain the information you need. It's a bother. Thus, commercial search engines have a question answering feature so that you can find information efficiently.



2.3 Question answering system types

2.3.1 Open domain VS Closed domain

Open domain

- Domain independent QA systems can answer any query from any corpus.
- +Covers wide range of queries.
- -Lower accuracy than closed domain QA.

Closed domain

- Domain specific QA systems are limited to specific domain.
- +Higher accuracy than open domain QA.
- -Limited coverage over the possible queries.

2.3.2 Open book vs Closed book

Open book

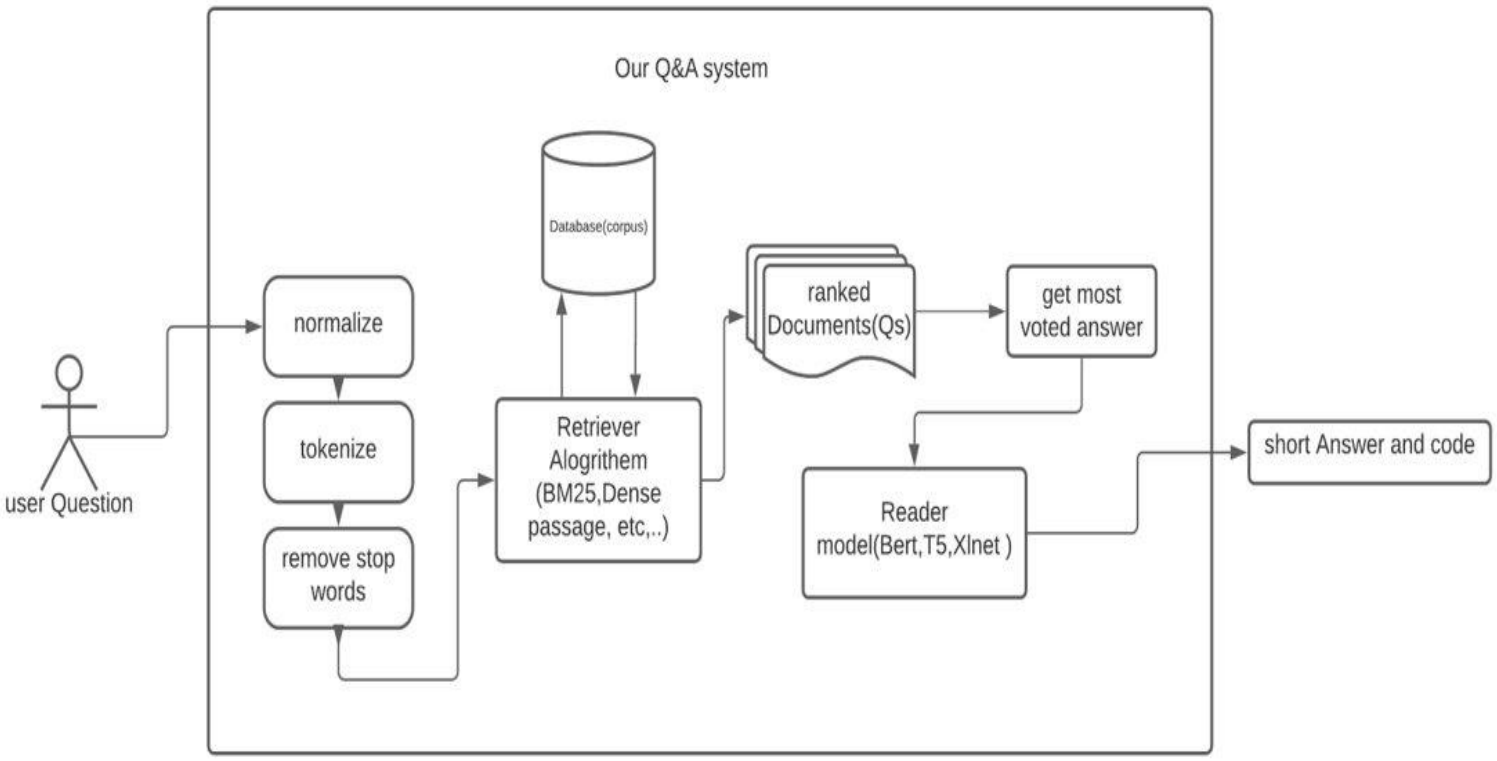
If a language model has no context or is not big enough to memorize the context which exists in the training dataset, it is unlikely to guess the correct answer. In an open-book exam, students are allowed to refer to external resources like notes and books while answering test questions

Closed book

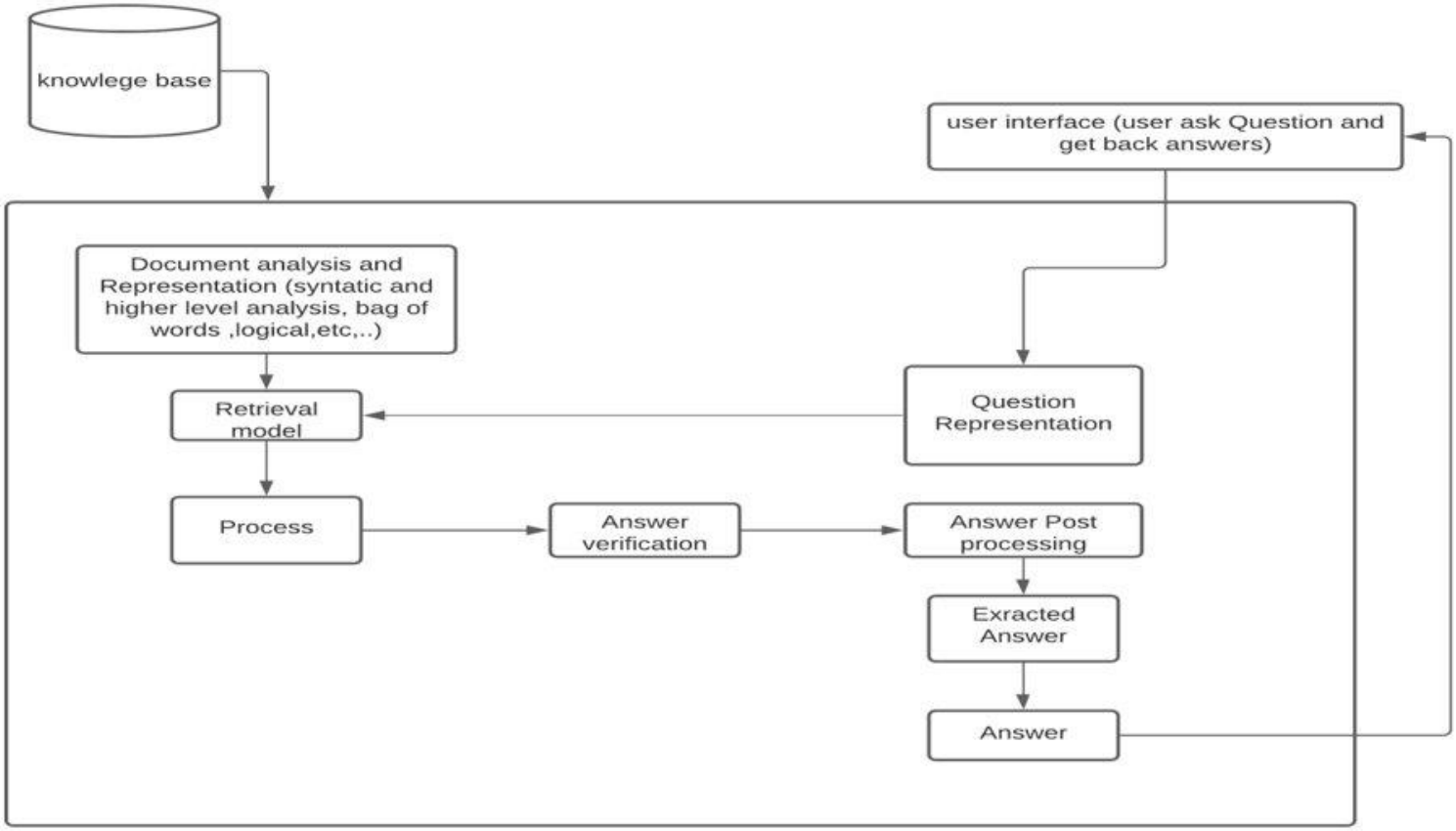
Big language models have been pre-trained on a large collection of unsupervised textual corpus. Given enough parameters, these models can memorize some factual knowledge within parameter weights. Therefore, we can use these models to do question-answering without explicit context, just like in a closed-book exam

2.4 System Description

2.4.1 System Use Case



2.4.2 System Architecture



2.4.3 Hardware Requirements

A Nvidia GPU with at least 12 GB of RAM for training.

We Used

- NVIDIA Tesla P100 PCIe 16 GB.
- NVIDIA Tesla V100 SXM2 16 GB

At Least 8GB of RAM

2.4.4 Software Requirements

Python 3.6+ and the used libraries.

3 Data

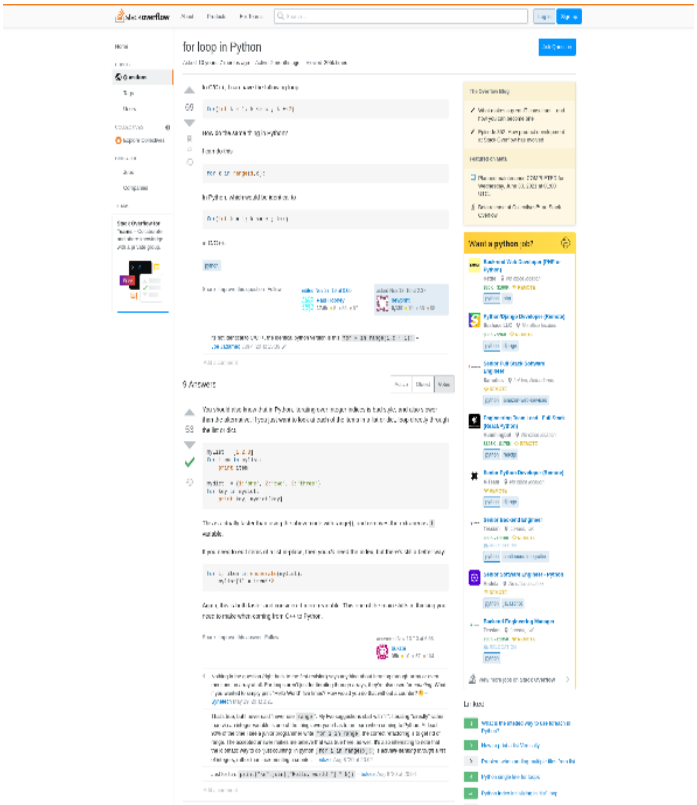
3.1 Our Dataset

Our data is about Question and Answer in Python from Stack Overflow from 2008 – 2016 form Kaggle which was collected by using Stack Overflow Big Query API.

3.2 What is Stack Overflow?

Stack Overflow is a question-and-answer website for professional and enthusiast programmers. It is the flagship site of the Stack Exchange Network, created in 2008 by Jeff Atwood and Joel Spolsky It features questions and answers on a wide range of topics in computer programming. It was created to be a more open alternative to earlier question and answer websites such as Experts-Exchange. Stack Overflow was sold to Prosus, a Netherlands-based consumer internet conglomerate, on 2 June 2021 for $1.8 billion.The website serves as a platform for users to ask and answer questions, and, through membership and active participation, to vote questions and answers up or down similar to Reddit and edit questions and answers in a fashion similar to a wiki.Users of Stack Overflow can earn reputation points and "badges"; for example, a person is awarded 10 reputation points for receiving an "up" vote on a question or an answer to a question, and can receive badges for their valued contributions, which represents a gamification of the traditional Q&A website. Users unlock new privileges with an

increase in reputation like the ability to vote, comment, and even edit other people's posts.



As of March 2021, Stack Overflow has over 14 million registered users, and has received over 21 million questions and 31 million answers. Based on the type of tags assigned to questions, the top eight most discussed topics on the site are: JavaScript, Java, C#, PHP, Android, Python, jQuery, and HTML. Stack Overflow also has a Jobs section to assist developers in finding their next opportunity. For employers, Stack Overflow provides tools to brand their business, advertise their openings on the site, and source candidates from Stack Overflow's database of developers who are open to being contacted.

Advantages of Stack Overflow:

1. **Learning:** You will learn a lot of things everyday by new questions and answers posted there on stack overflow. If you cannot answer any questions, still you can see the answers and make notes from there. There are questions that you cannot answer, and when you see others' answers, you learn something new. There are questions that you research before answering, so you improve your knowledge. Why do people ask questions that they can also research? Because you have a higher level of general programming knowledge that allows you to look at the right place and find the information. Even if you already know the answer, you get a better

understanding when writing it down. Answering questions on Stack Overflow can be compared to reading blogs that discuss common technical issues.

2. **Self-Evaluation:** You see where you stand – if others agree with your answers (and upvote them), that makes you feel confident in the topics that you follow. Which is a good thing (if it's not overconfidence). Questions that people ask show you areas that you probably didn't know about, but that are important. That way you get a more adequate view of where you stand with your programming knowledge.
3. **Improve Communication Skills:** Writing good answers doesn't necessarily mean you are a good programmer. But it certainly means you are a good communicator. And that's a very important skill. It might even be better to be a good programmer and a good communicator than to be a slightly better programmer that can't communicate his ideas.
4. **Improving Analytical Skills and Problem-Solving Ability:** You get to know what the questions is about. In many cases the questions are not well-phrased or miss important information. You have to analyze what information is needed, or what the poster has meant.
5. **Improve Your Speed:** For some questions you must act fast, otherwise other answers will appear before yours and get all the up votes. Well, sometimes you need to act fast in your programming career. Production issue, unhappy customers. If you can analyze and solve the problem in 5 minutes it will be way better than 10 minutes, let alone an hour.

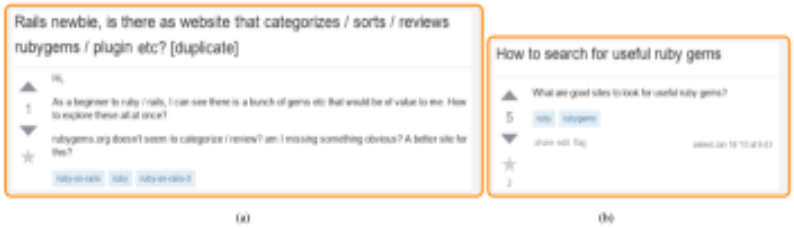
Problems of Stack Overflow:

Stack Overflow is filled with a lot of silly / obsolete and repeated questions.

Answers to complex questions are voted up less as compared to the answers to the simple questions.

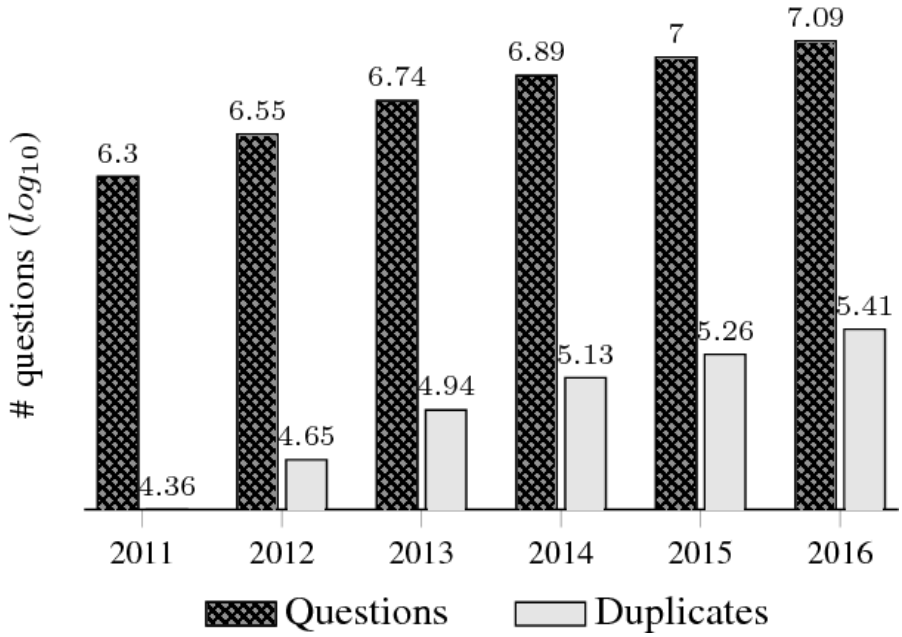
Many questions on Stack Overflow remain unanswered.

It's time consuming. You have to spend a lot of time building your reputation there on Stack Overflow.



How to overcome repeated question in Stack Overflow:

Duplicate questions make Stack Overflow site maintenance harder, waste resources that could have been used to answer other questions, and cause developers to unnecessarily wait for answers that are already available. To reduce the problem of duplicate questions, Stack Overflow allows questions to be manually marked as duplicates of others. Since there are thousands of questions submitted to Stack Overflow every day, manually identifying duplicate questions is difficult work. Thus, there is a need for an automated approach that can help in detecting these duplicate questions (DupPredictor/Rep/-T).



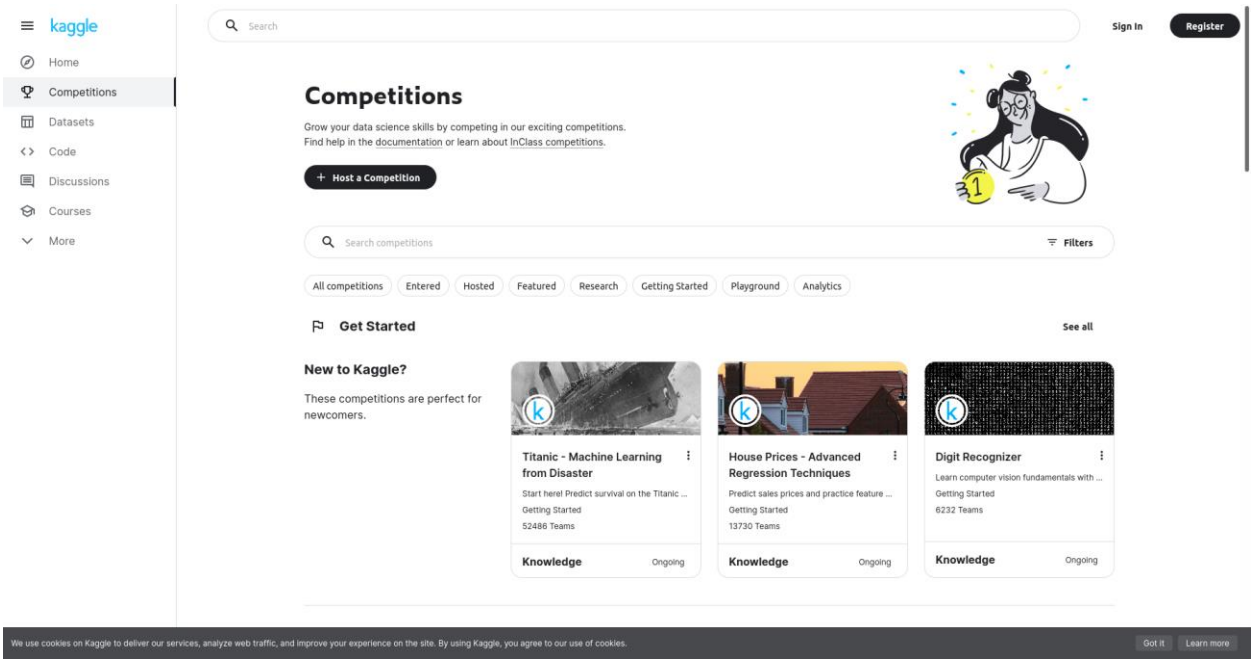
DupPredictor that takes a new question as input and detects potential duplicates of this question by considering multiple factors. DupPredictor extracts the title and description of a question and also tags that are attached to the question. These pieces of information (title, description, and a few tags) are mandatory information that a user needs to input when posting a question. DupPredictor then computes the

latent topics of each question by using a topic model. Next, for each pair of questions, it computes four similarity scores by comparing their titles, descriptions, latent topics, and tags. These four similarity scores are finally combined together to result in a new similarity score that comprehensively considers the multiple factors.

Approach	R@20	R@10	R@5
DupPredictor	0.638	0.533	0.423
DupPredictorRep	0.433	0.353	0.282
DupPredictorRep-T	0.427	0.352	0.278

3.3 What's Kaggle?

Kaggle, a subsidiary of Google LLC, is an online community of data scientists and machine learning practitioners. Kaggle allows users to find and publish data sets, explore and build models in a web-based data-science environment, work with other data scientists and machine learning engineers, and enter competitions to solve data science challenges.



Kaggle got its start in 2010 by offering machine learning competitions and now also offers a public data platform, a cloud-based workbench for data science, and Artificial Intelligence education. Its key personnel were Anthony Goldbloom and Jeremy Howard. Nicholas Gruen was the founding chair succeeded by Max Levchin. Equity was raised in 2011 valuing the company at $25 million. On 8 March 2017, Google announced that they were acquiring Kaggle.

3.4 Data Analysis

What does our dataset consist of ?

Questions table

Answers table

Tags table

3.4.1 Question table

It is the table which we have all data about the questions which has been asked in stack overflow and this table we have 6 column:

- Id : which represents the Question id which we will need.
- Score : which represents how this question was useful to others or not.
- CreationDate : which represents when did it when this question was asked.
- OwnerUserId : which represents the user id who asked the question.
- Title : which represents the title of the question.
- Body : which gives more description about the question can have code including it.

	Id	OwnerUserId	CreationDate	Score	Title	Body
0	469	147.0	2008-08-02T15:11:16Z	21	How can I find the full path to a font from it...	<p>I am using the Photoshop's javascript API t...
1	502	147.0	2008-08-02T17:01:58Z	27	Get a preview JPEG of a PDF on Windows?	<p>I have a cross-platform (Python) applicatio...
2	535	154.0	2008-08-02T18:43:54Z	40	Continuous Integration System for a Python Cod...	<p>I'm starting work on a hobby project with a...
3	594	116.0	2008-08-03T01:15:08Z	25	cx_Oracle: How do I iterate over a result set?	<p>There are several ways to iterate over a re...

3.4.2 Answer table

It’s the table which we have all data about the answers which has been answered in stackoverflow and this table we have 5 column:

- Id : which represents the answer id.
- CreationDate : which represents when did it when this question asked.
- PanentId : which represents the id of the question in the question table.
- Score : which represents how this answer was useful to others or not.

• Body : which gives more description about the Answer can have code include it.

	Id	OwnerUserId	CreationDate	ParentId	Score	Body
0	497	50.0	2008-08-02T16:56:53Z	469	4	<p>open up a terminal (Applications->Utilit...
1	518	153.0	2008-08-02T17:42:28Z	469	2	<p>I haven't been able to find anything that d...
2	536	161.0	2008-08-02T18:49:07Z	502	9	<p>You can use ImageMagick's convert utility f...
3	538	156.0	2008-08-02T18:56:56Z	535	23	<p>One possibility is Hudson. It's written in...
4	541	157.0	2008-08-02T19:06:40Z	535	20	<p>We run <a href="http://buildbot.net/trac">B...

3.4.3 Tag table

It’s the table which we have all data about tags for each questions has been asked in

stackoverflow and this table we have 2 column:

• Id : which represents the Question id which we will need.

• Tag : which represents how this question was useful to others or not.

	Id	Tag
0	469	python
1	469	osx
2	469	fonts
3	469	photoshop

3.4.4 Data preparation

First thing we faced was the size of the dataset. We found that we have 607282 questions and 987122 answers, so we try to minimize those dataset by doing some statistics.

```
len(Answers)
```

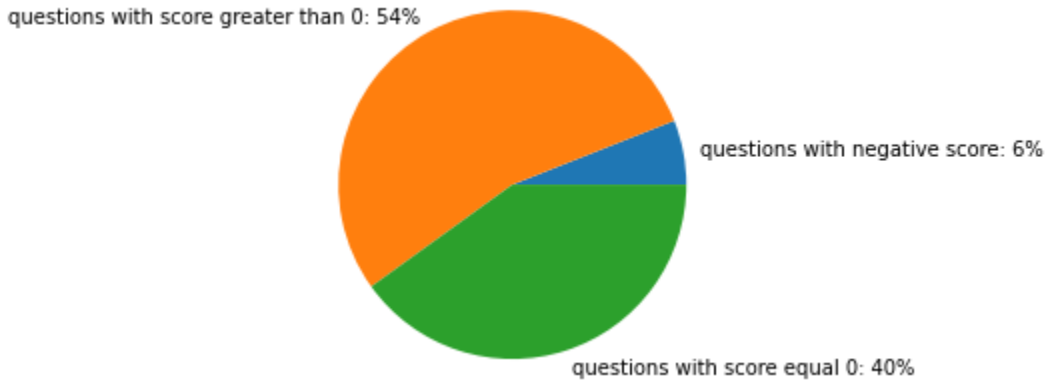
```
987122
```

```
len(Questions)
```

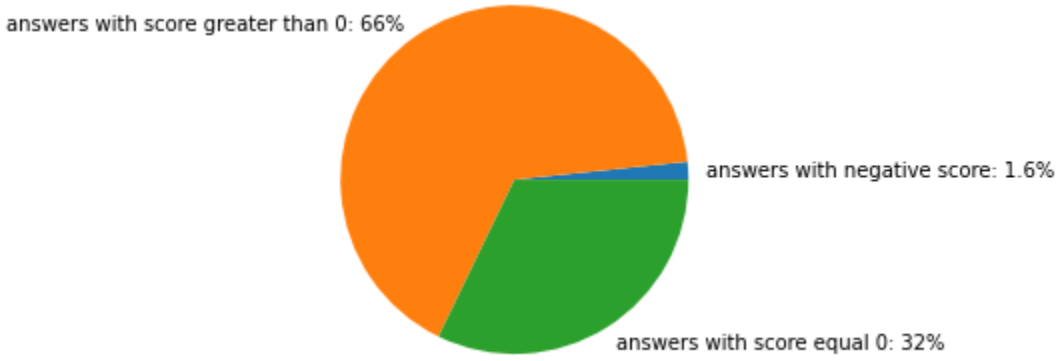
```
607282
```

3.4.4.1 Removing Negative data

We try first the Question which has a score less than 0.
We found that 6% of the data is less than zero , 40% of the data is equal to zero and 54% of the data is higher than the zero as represented in the chart.

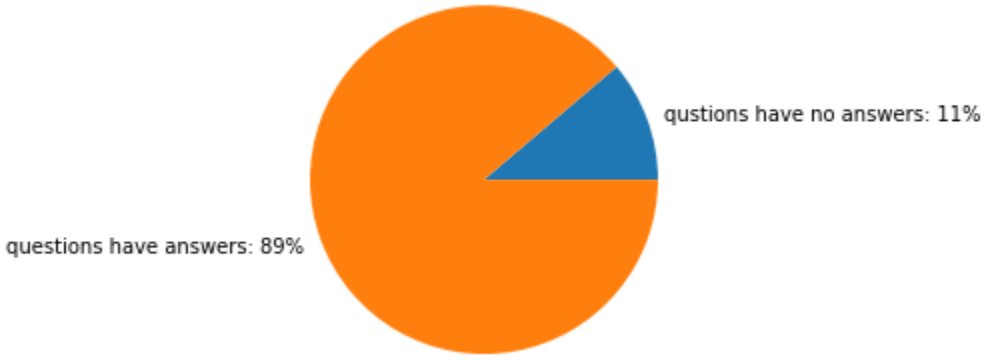


And also, we found that in Answers we have 1.6% ~ 2 % less than zero , 32% is equal to zero and 66% is higher than zero as represented in the chart.

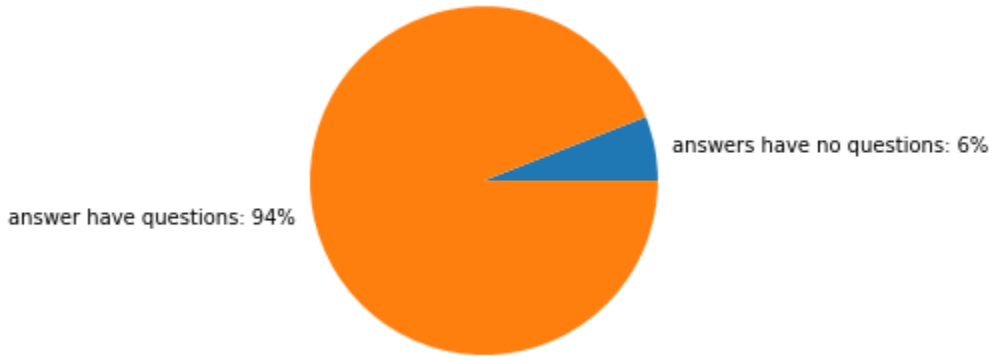


So, we decided to remove negative from both answers and questions data to minimize data and improve the result. The questions get minimized from 607282 to 570972 and answers from 987122 to 970907.

Then we decided to check if there are questions that have no answers which figure out that 11% of the data has no answer and 89% of data have answers which are represented in the following chart so we reduce the data and increase the efficiency.



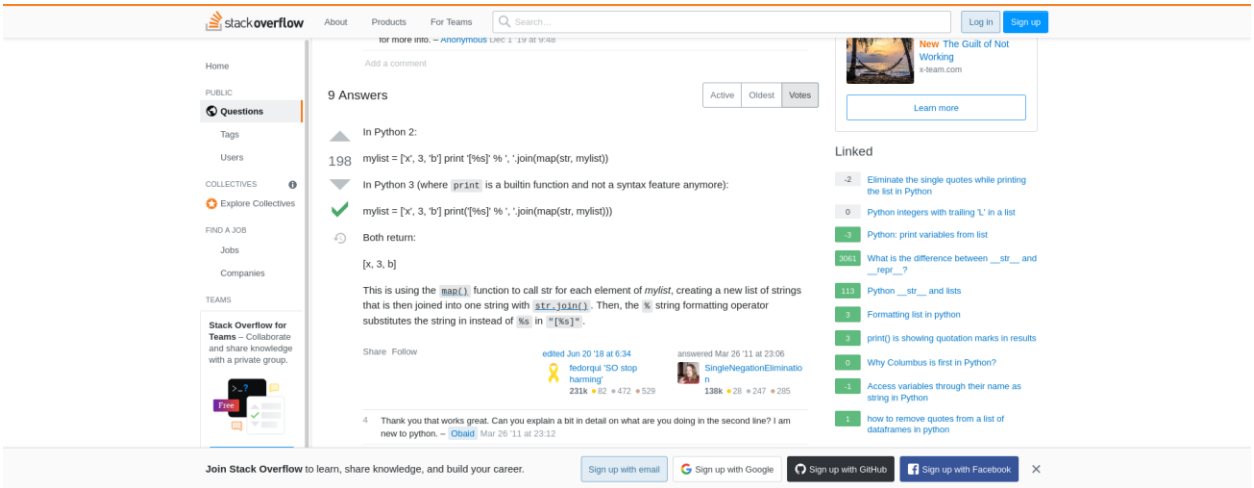
So removed it to so the size of the dataset of questions which have no question to become 502112 form originally 607282 then we figure out that we have answer have 537060 unique answer so begin to check how many answer that may not have question, so we figure out that we have 6% of answers have no question and 94% have questions as represented in the chart.



By removing those answers, we finally have 913099 from originally 987122 so we finally have 502112 questions and 913099 answers.

3.4.4.2 Remove HTML Tags

Then we figure out that we need to remove html tags from the answers body without split pre tags because we figure out that some of codes answers not between pre tags or code tags and vice versa others use those tags in order to ensure that something for example:



So, we decided to remove those html tags using Beautiful soup.

What is beautiful soup?

It is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work.

Example image before using beautiful soup:

```
 <p>The canonical way is to use the built-in cursor iterator.</p>

<pre><code>curs.execute('select * from people')
for row in curs:
    print row
</code></pre>

<hr>

<p>You can use <code>fetchall()</code> to get all rows at once.</p>

<pre><code>for row in curs.fetchall():
    print row
</code></pre>

<p>It can be convenient to use this to create a Python list containing the values returned:</p>

<pre><code>curs.execute('select first_name from people')
names = [row[0] for row in curs.fetchall()]
</code></pre>

<p>This can be useful for smaller result sets, but can have bad side effects if the result set is large.</p>

<ul>
<li><p>You have to wait for the entire result set to be returned to
your client process.</p></li>
<li><p>You may eat up a lot of memory in your client to hold
the built-up list.</p></li>
<li><p>It may take a while for Python to construct and deconstruct the
list which you are going to immediately discard anyways.</p></li>
</ul>

<hr>

<p>If you know there's a single row being returned in the result set you can call <code>fetchone()</code> to get the single row.</p>

<pre><code>curs.execute('select max(x) from t')
maxValue = curs.fetchone()[0]
</code></pre>

<hr>

<p>Finally, you can loop over the result set fetching one row at a time.  In general, there's no particular advantage in doing this over using the iterator.</p>

<pre><code>row = curs.fetchone()
while row:
    print row
    row = curs.fetchone()
</code></pre>
```

Example image after using beautiful soup:

```
 The canonical way is to use the built-in cursor iterator.


You can use  to get all rows at once.

It can be convenient to use this to create a Python list containing the values returned:

This can be useful for smaller result sets, but can have bad side effects if the result set is large.

You have to wait for the entire result set to be returned to
your client process.
You may eat up a lot of memory in your client to hold
the built-up list.
It may take a while for Python to construct and deconstruct the
list which you are going to immediately discard anyways.


If you know there's a single row being returned in the result set you can call  to get the single row.


Finally, you can loop over the result set fetching one row at a time.  In general, there's no particular advantage in doing this over using the iterator.
```

3.4.5 Pre-processing

3.4.5.1 Question Table

We begin to make a function to remove symbols and stop words from the title. After that use a tokenizer coming with our transformer and add a new column named by NoStopwordsTitle and then remove both body column and title column.

Before applying:

	Id	OwnerUserId	CreationDate	Score	Title	Body
0	469	147.0	2008-08-02T15:11:16Z	21	How can I find the full path to a font from its display name on a Mac?	I am using the Photoshop's javascript API to find the fonts in a given PSD. \n\n Given a font name returned by the API, I want to find the actual physical font file that that font name corresponds to on the disc. \n\n This is all happening in a python program running on OSX so I guess I'm looking for one of: \n\n \n - Some Photoshop javascript \n - A Python function \n - An OSX API that I can call from python \n \n
1	502	147.0	2008-08-02T17:01:58Z	27	Get a preview JPEG of a PDF on Windows?	I have a cross-platform (Python) application which needs to generate a JPEG preview of the first page of a PDF. \n\n On the Mac I am spawning sips. Is there something similarly simple I can do on Windows? \n
2	535	154.0	2008-08-02T18:43:54Z	40	Continuous Integration System for a Python Codebase	I'm starting work on a hobby project with a python codebase and would like to set up some form of continuous integration (i.e. running a battery of test-cases each time a check-in is made and sending nag e-mails to responsible persons when the tests fail) similar to CruiseControl or TeamCity. \n\n I realize I could do this with hooks in most VCSes, but that requires that the tests run on the same machine as the version control server, which isn't as elegant as I would like. Does anyone have any suggestions for a small, user-friendly, open-source continuous integration system suitable for a Python codebase? \n
3	594	116.0	2008-08-03T01:15:08Z	25	cx_Oracle: How do I iterate over a result set?	There are several ways to iterate over a result set. What are the tradeoff of each? \n
4	683	199.0	2008-08-03T13:19:16Z	28	Using 'in' to match an attribute of Python objects in an array	I don't remember whether I was dreaming or not but I seem to recall there being a function which allowed something like, \r\n\r\n<pre><code>foo in iter_attr(array of python objects, attribute name)</code></pre>\r\n\r\n I've looked over the docs but this kind of thing doesn't fall under any obvious listed headers

After applying:

	Id	OwnerUserId	CreationDate	Score	NoStopwordsTitle	BobyTokened
0	469	147.0	2008-08-02T15:11:16Z	21	[How, I, find, full, path, font, display, name, Mac]	[I, am, using, the, Photoshop, ', s, javascript, API, to, find, the, fonts, in, a, given, PSD, ., Given, a, font, name, returned, by, the, API, ,, I, want, to, find, the, actual, physical, font, file, that, that, font, name, corresponds, to, on, the, disc, ., This, is, all, happening, in, a, python, program, running, on, OSX, so, I, guess, I, ', m, looking, for, one, of, :, Some, Photoshop, javascript, A, Python, function, An, OSX, API, that, I, can, call, from, python]
1	502	147.0	2008-08-02T17:01:58Z	27	[Get, preview, JPEG, PDF, Windows]	[I, have, a, cross, -, platform, (, Python,), application, which, needs, to, generate, a, JPEG, preview, of, the, first, page, of, a, PDF, ., On, the, Mac, I, am, spawning, sip, ##s, ., Is, there, something, similarly, simple, I, can, do, on, Windows, ?]
2	535	154.0	2008-08-02T18:43:54Z	40	[Continuous, Integration, System, Python, Code, ##base]	[I, ', m, starting, work, on, a, hobby, project, with, a, python, codebase, and, would, like, to, set, up, some, form, of, continuous, integration, (, i, ., e, ., running, a, battery, of, test, -, cases, each, time, a, check, -, in, is, made, and, sending, nag, e, -, mails, to, responsible, persons, when, the, tests, fail,), similar, to, CruiseControl, or, TeamCity, ., I, realize, I, could, do, this, with, hooks, in, most, VCS, ##es, ,, but, that, requires, that, the, tests, run, on, the, same, machine, as, the, version, control, server, ,, which, isn, ', t, as, elegant, ...]
3	594	116.0	2008-08-03T01:15:08Z	25	[cx, Oracle, How, I, iterate, result, set]	[There, are, several, ways, to, iterate, over, a, result, set, ., What, are, the, tradeoff, of, each, ?]
4	683	199.0	2008-08-03T13:19:16Z	28	[Using, in, match, attribute, Python, objects, array]	[I, don, ', t, remember, whether, I, was, dreaming, or, not, but, I, seem, to, recall, there, being, a, function, which, allowed, something, like, ,, foo, in, iter, _, attr, (, array, of, python, objects, ,, attribute, name,), I, ', ve, looked, over, the, docs, but, this, kind, of, thing, doesn, ', t, fall, under, any, obvious, listed, headers]

3.4.5.2 Answer Table

We begin to remove html tags using beautifulsoup as referenced before and then use tokenizer which comes with our transformer and add a new column named by BodyNoHtml and then remove the body column.

Before applying:

	Id	OwnerUserId	CreationDate	ParentId	Score	Body

0	497	50.0	2008-08-02T16:56:53Z	469	4	open up a terminal (Applications->Utilities->Terminal) and type this in: \r\n\r\n<pre><code>locate InsertFontHere
						</code></pre>\r \n\r\n This will spit out every file that has the name you want. \r \n\r\n Warning: there may be alot to wade through.

1	518	153.0	2008-08-02T17:42:28Z	469	2	I haven't been able to find anything that does this directly. I think you'll have to iterate through the various font folders on the system: <code>/System/Library /Fonts</code>, <code>/Library/Fonts</code>, and there can probably be a user-level directory as well <code>~/Library/Fonts</code>. \n
2	536	161.0	2008-08-02T18:49:07Z	502	9	You can use ImageMagick's convert utility for this, see some examples in http://studio.imagemagick.org/pipermail/magick-users/2002-May/002636.html\n: \n\n<blockquote>\n<pre><code>Convert taxes.pdf taxes.jpg \n</code></pre>\n \n Will convert a two page PDF file into [2] jpeg files: taxes.jpg.0,\n taxes.jpg.1 \n \n I can also convert these JPEGS to a thumbnail as follows: \n\n<pre><code>convert -size 120x120 taxes.jpg.0 -geometry 120x120 +profile '*' thumbnail.jpg\n</code></pre>\n \n I can even convert the PDF directly to a jpeg thumbnail as follows: \n \n<pre><code>convert -size 120x120 taxes.pdf -geometry 120x120 +profile '*' thumbnail.jpg\n</code></pre>\n \n This will result in a thumbnail.jpg.0 and thumbnail.jpg.1 for the two\n pages. \n</blockquote>\n
3	538	156.0	2008-08-02T18:56:56Z	535	23	One possibility is Hudson. It's written in Java, but there's integration with Python projects: \n\n<blockquote>\n Hudson embraces Python \n</blockquote>\n\n I've never tried it myself, however. \n\n (Update, Sept. 2011: After a trademark dispute Hudson has been renamed to Jenkins.) \n
4	541	157.0	2008-08-02T19:06:40Z	535	20	We run Buildbot - Trac at work, I haven't used it too much since my code base isn't part of the release cycle yet. But we run the tests on different environments (OSX/Linux/Win) and it sends emails --and it's written in python.

After applying:

	Id	OwnerUserId	CreationDate	ParentId	Score	BodyNoHtml
0	497	50.0	2008-08-02T16:56:53Z	469	4	[open, up, a, terminal, (, Applications, -, >, Utilities, -, >, Terminal,), and, type, this, in, :, locate, Insert, ##Font, ##Here, This, will, spit, out, every, file, that, has, the, name, you, want, ., Warning, :, there, may, be, alot, to, wade, through, .]
1	518	153.0	2008-08-02T17:42:28Z	469	2	[I, haven, ', t, been, able, to, find, anything, that, does, this, directly, ., I, think, you, ', ll, have, to, iterate, through, the, various, font, folders, on, the, system, :, /, System, /, Library, /, Fonts, ,, /, Library, /, Fonts, ,, and, there, can, probably, be, a, user, -, level, directory, as, well, ~, /, Library, /, Fonts, .]
2	536	161.0	2008-08-02T18:49:07Z	502	9	[You, can, use, ImageMagick, ', s, convert, utility, for, this, ,, see, some, examples, in, http, :, /, /, studio, ., imagemagick, ., org, /, pipermail, /, magick, -, users, /, 2002, -, May, /, 002, ##63, ##6, ., html, :, Convert, taxes, ., pdf, taxes, ., jpg, Will, convert, a, two, page, PDF, file, into, [, 2,], jpeg, files, :, taxes, ., jpg, ., 0, ,, taxes, ., jpg, ., 1, I, can, also, convert, these, JPEG, ##S, to, a, thumbnail, as, follows, :, convert, -, size, 120, ##x120, taxes, ., jpg, ., 0, -, geometry, 120, ##x120, ...]
3	538	156.0	2008-08-02T18:56:56Z	535	23	[One, possibility, is, Hudson, ., It, ', s, written, in, Java, ,, but, there, ', s, integration, with, Python, projects, :, Hudson, embrace, ##s, Python, I, ', ve, never, tried, it, myself, ,, however, ., (, Update, ,, Sept, ., 2011, :, After, a, trademark, dispute, Hudson, has, been, renamed, to, Jenkins, .,)]
4	541	157.0	2008-08-02T19:06:40Z	535	20	[We, run, Build, ##bot, -, Trac, at, work, ,, I, haven, ', t, used, it, too, much, since, my, code, base, isn, ', t, part, of, the, release, cycle, yet, ., But, we, run, the, tests, on, different, environments, (, OSX, /, Linux, /, Win,), and, it, sends, emails, -, -, and, it, ', s, written, in, python, .]

3.4.6 Dataset Statistics

We do some statistics on data like length of the max, min and mean in title of question (NoStopwordsTitle), body of question (BodyTokened) and body of answer (BodyNoHtml)

```
print("Maximum Length of Answers Body : ",max(ans.BodyNoHtml.apply(len)))
print("Minimum Length of Answers Body:",min(ans.BodyNoHtml.apply(len)))
print("Average Length of Answers Body: ",mean(ans.BodyNoHtml.apply(len)))
```

```
Maximum Length of Answers Body :  18246
Minimum Length of Answers Body: 0
Average Length of Answers Body:  176.67083415927516
```

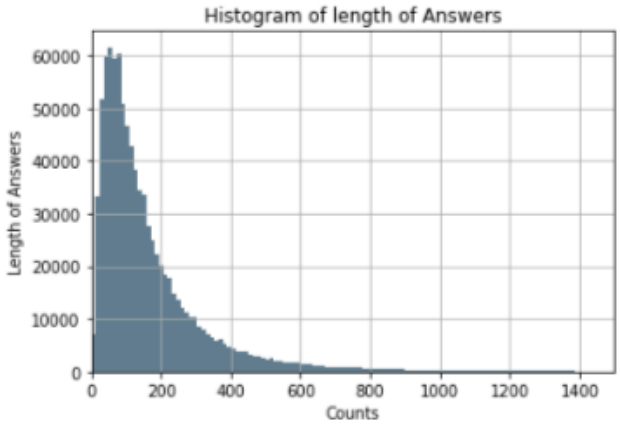
```
print("Maximum Length of Questions Title: ",max(ques.NoStopwordsTitle.apply(len)))
print("Minimum Length of Questions Title: ",min(ques.NoStopwordsTitle.apply(len)))
print("Average Length of Questions Title: ",mean(ques.NoStopwordsTitle.apply(len)))
```

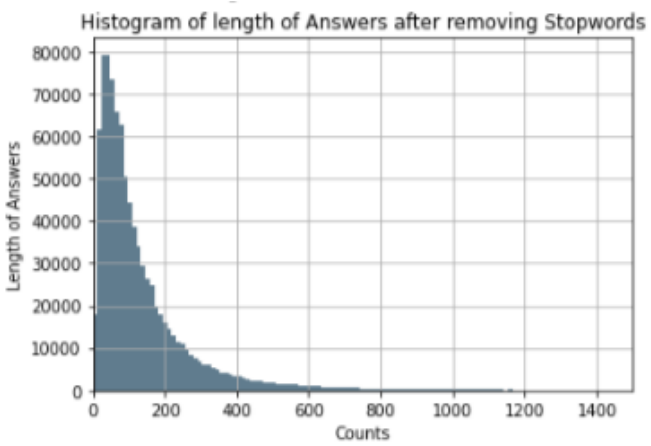
```
Maximum Length of Questions Title:  59
Minimum Length of Questions Title:  0
Average Length of Questions Title:  6.777364412720668
```

```
print("Maxmium Length of Questions Body: ",max(ques.BobyTokened.apply(len)))
print("Minimum Length of Questions Body: ",min(ques.BobyTokened.apply(len)))
print("Average Length of Questions Body: ",mean(ques.BobyTokened.apply(len)))
```

```
Maxmium Length of Questions Body:  21703
Minimum Length of Questions Body:  3
Average Length of Questions Body:  332.91988042508444
```

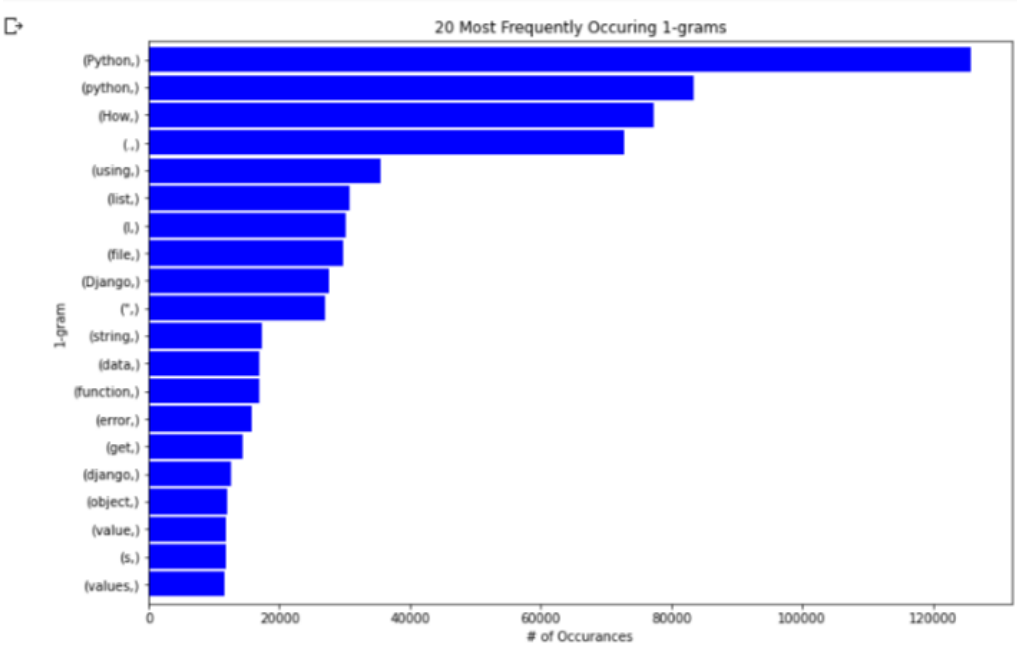
and this histogram represent length of body before and after removing stopwords:



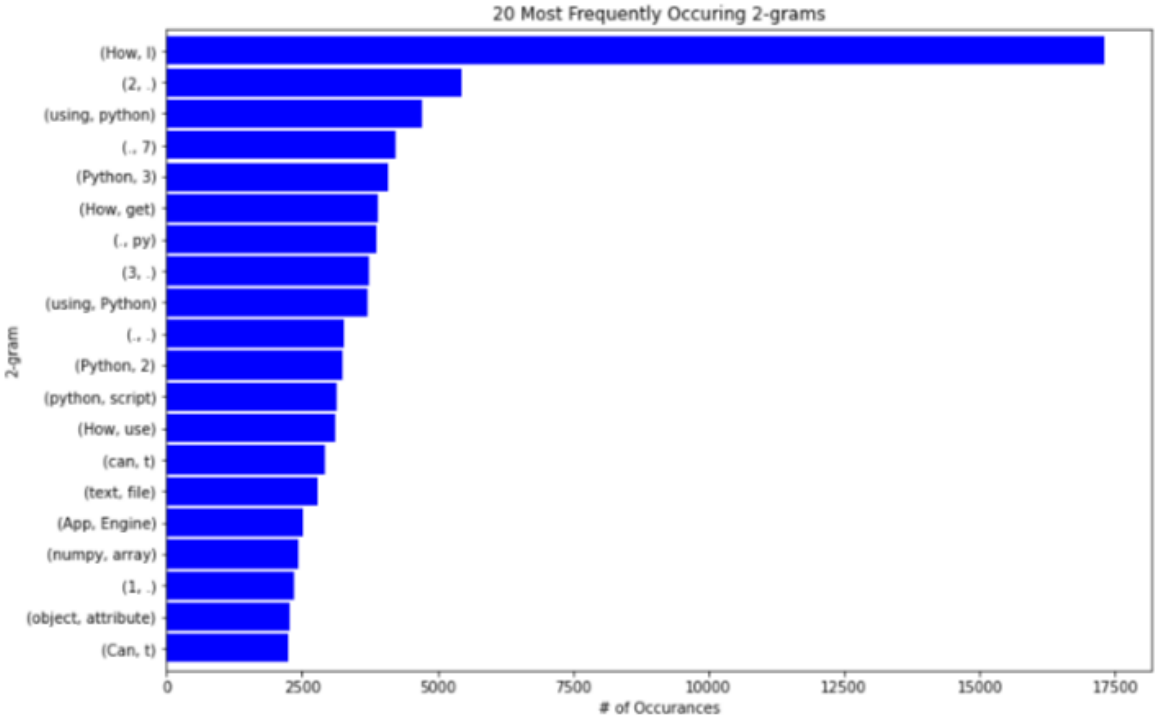


We try to find the most common words that come together by getting the first 4 n-grams.

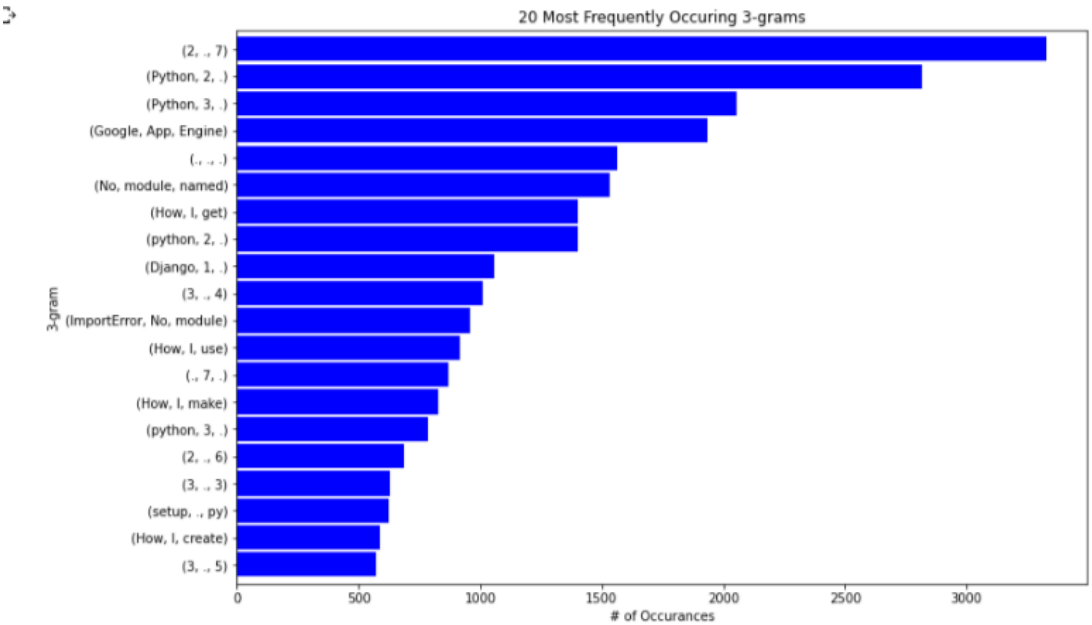
1-grams :



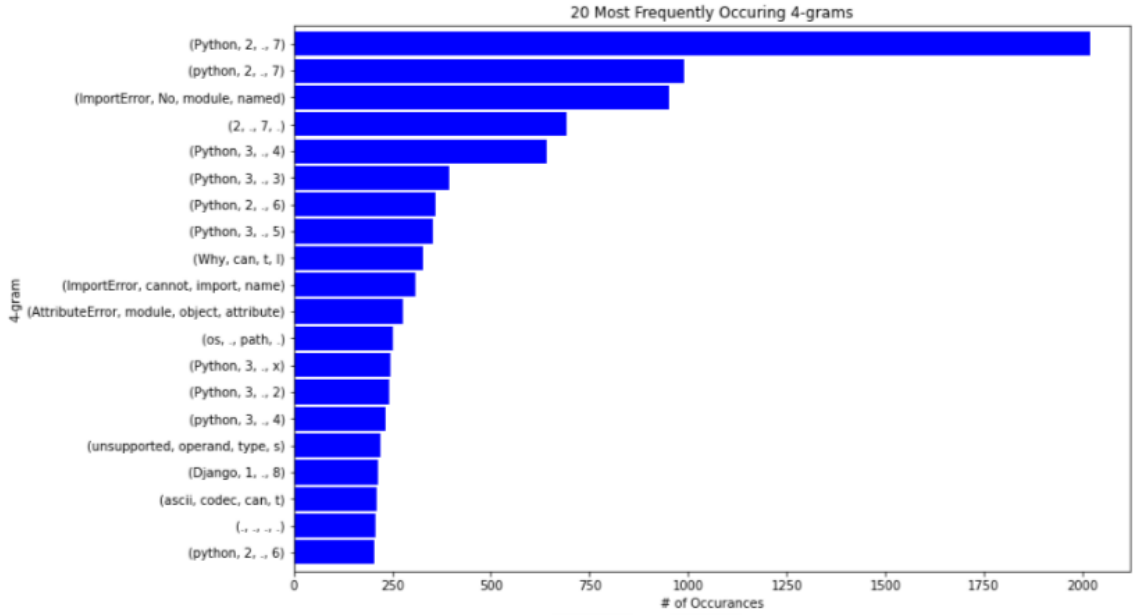
2-grams :



3-grams :



4-grams :


20 Most Frequently Occuring 4-grams
(Python, 2, ., 7)
(python, 2, ., 7)
(ImportError, No, module, named)
(2, ., 7, .)
(Python, 3, ., 4)
(Python, 3, ., 3)
(Python, 2, ., 6)
(Python, 3, ., 5)
(Why, can, t, I)
(ImportError, cannot, import, name)
(AttributeError, module, object, attribute)
(os, ., path, .)
(Python, 3, ., x)
(Python, 3, ., 2)
(python, 3, ., 4)
(unsupported, operand, type, s)
(Django, 1, ., 8)
(ascii, codec, can, t)
(., ., ., .)
(python, 2, ., 6)
4-gram
0
250
500
750
1000
1250
1500
1750
2000
# of Occurances

4 Typos Correction

4.1 Effective Spelling correction.

Testing on typos correction Models.

Benefits:

1- Help us to clear data into English text only with programming concepts and words.

2- Correct the input of the user to get accurate output.

Concept Behind:

Peter Norvig (Naive approach):

This the most famous approach to write a simple good spelling correction by Peter Norvig.

4.1.1 How It Works: Some Probability Theory

Spelling correction isn't easy task because no way to get correct spelling for example: (should lates be corrected to late or latest, lattes , ...?) **,** so we trying to find correction C, out of all possible corrections, so we want maximize the probability of C, given the original corrupted word W :

Correct C = $\text{argmax}_{c \in candidates}$ P(c|w)

Using the knowledge of Bayes' Theorem, we can rewrite this as:

$\text{argmax}_{c \in candidates}$ P(c) P(w|c) / P(w)

Since P(w) is the same for every possible candidate c, we can factor it out, giving:

$\text{argmax}_{c \in candidates}$ P(c) P(w|c)

The four parts of this expression are:

1. **Selection Mechanism**: argmax
   We choose the candidate with the highest combined probability.
2. **Candidate Model**: $c \in candidates$
   This tells us which candidate corrections, *c*, to consider.
3. **Language Model**: P(*c*)
   The probability that *c* appears as a word of English text. For example, occurrences of "the" make up about 7% of English text, so we should have P(*the*) = 0.07.
4. **Error Model**: P(*w*|*c*)
   The probability that *w* would be typed in a text when the author meant *c*. For example, P(*teh*|*the*) is relatively high, but P(*theeexyz*|*the*) would be very low.

One obvious question is: why take a simple expression like P(*c*|*w*) and replace it with a more complex expression involving two models rather than one? The answer is that P(*c*|*w*) is *already* conflating two factors, and it is easier to separate the two out and deal with them explicitly. Consider the misspelled word *w*="thew" and the two-candidate corrections *c*="the" and *c*="thaw". Which has a higher P(*c*|*w*)? Well, "thaw" seems good because the only change is "a" to "e", which is a small change. On the other hand, "the" seems good because "the" is a very common word, and while adding a "w" seems like a larger, less probable change, perhaps the typist's finger slipped off the "e". The point is that to estimate P(*c*|*w*) we must consider both the probability of *c* and the probability of the change from *c* to *w* anyway, so it is cleaner to formally separate the two factors.

4.1.2 How It Works: Some Python

The four parts of the program are:

4.1.2.1 Selection Mechanism

In Python, **max** with a key argument does 'argmax'.

4.1.2.2 Candidate Model

First a new concept: a **simple edit** to a word is a deletion (remove one letter), a transposition (swap two adjacent letters), a replacement (change one letter to another) or an insertion (add a letter). The function edits1 returns a set of all the edited strings (whether words or not) that can be made with one simple edit:

```
def edits1(word):
    "All edits that are one edit away from `word`."
    letters    = 'abcdefghijklmnopqrstuvwxyz'
    splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    deletes    = [L + R[1:]               for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]
    inserts    = [L + c + R               for L, R in splits for c in letters]
    return set(deletes + transposes + replaces + inserts)
```

This can be a big set. For a word of length *n*, there will be *n* deletions, ***n*-1** transpositions, **26*n*** alterations, and **26(*n*+1)** insertions, for a total of **54*n*+25** (of which a few are typically duplicates). For example,

```
>>> len(edits1('somthing'))
442
```

However, if we restrict ourselves to words that are *known*—that is, in the dictionary— then the set is much smaller:

```
def known(words): return set(w for w in words if w in WORDS)
```

```
>>> known(edits1('somthing'))
{'something', 'soothing'}
```

We will also consider corrections that require *two* simple edits. This generates a much bigger set of possibilities, but usually only a few of them are known words:

```
def edits2(word): return (e2 for e1 in edits1(word) for e2 in edits1(e1))
```

Example :

```
>>> len(set(edits2('something'))
90902
```

```
>>> known(edits2('something'))
{'seething', 'smoothing', 'something', 'soothing'}
```

```
>>> known(edits2('somthing'))
{'loathing', 'nothing', 'scathing', 'seething', 'smoothing', 'something', 'soothing', 'sorting'}
```

We say that the results of edits2(w) have an **edit distance** of 2 from w.

4.1.2.3 Language Model

We can estimate the probability of a word, P(word), by counting the number of times each word appears in a text file of about a million words, big.txt. It is a concatenation of public domain book excerpts from Project Gutenberg and lists of most frequent words from Wiktionary and the British National Corpus. The function words break text into words, then the variable WORDS holds a Counter of how often each word appears, and P estimates the probability of each word, based on this Counter:

```
def words(text): return re.findall(r'\w+', text.lower())
WORDS = Counter(words(open('big.txt').read()))
def P(word, N=sum(WORDS.values())): return WORDS[word] / N
```

We can see that there are 32,192 distinct words, which together appear 1,115,504 times, with 'the' being the most common word, appearing 79,808 times (or a probability of about 7%) and other words being less probable:

```
>>> len(WORDS)
32192

>>> sum(WORDS.values())
1115504


>>> WORDS.most_common(10)
[ ('the', 79808),('of', 40024),('and', 38311),('to', 28765),('in', 22020),
 ('a', 21124),('that', 12512),('he', 12401),('was', 11410),('it', 10681),
 ('his', 10034),('is', 9773),('with', 9739),('as', 8064),('i', 7679),
 ('had', 7383),('for', 6938), ('at', 6789),('by', 6735),('on', 6639) ]




>>> max(WORDS, key=P)
'The'

>>> P('the')
0.07154434228832886

>>> P('outrivaled')
8.9645577245801e-07

>>> P('unmentioned')
0.0
```

4.1.2.4 Error Model

flawed error model says that all known words of edit distance 1 are infinitely more probable than known words of edit distance 2, and infinitely less probable than a known word of edit distance 0. So we can make candidates(word) produce the first non-empty list of candidates in order of priority:

- The original word, if it is known; otherwise
- The list of known words at edit distance one away, if there are any; otherwise

- The list of known words at edit distance two away, if there are any; otherwise
- The original word, even though it is not known.

- Then we don't need to multiply by a P(*w*|*c*) factor, because every candidate at the chosen priority will have the same probability (according to our flawed model). That gives us:

```
def correction(word): return max(candidates(word), key=P)
```

```
def candidates(word):



  return known([word]) or known(edits1(word)) or  ...known(edits2(word)) or [word]
```

4.1.2.5 Evaluation

Our testset collected from different sources (Kaggle - autocorrect testset - testset by Norvig)

We will use it to test (accuracy & speed) of each library. Here our custom **test function for our used models.**

```
def spelltest( library , verbose=3):
    import time
    n, bad = 0, 0
    start = time.process_time()

    for target, incorrect_spellings in test_set:
        for incorrect_spelling in incorrect_spellings.split("|"):

            n += 1

            if(s=="org"): #original without edit by Norivg
                    w = correction(incorrect_spelling)

            elif(s=="pcs"): #pycheckerspell
                    w = spell.correction(incorrect_spelling)
            else: #sympellpy
                    suggestions = sym_spell.lookup(incorrect_spelling, Verbosity.CLOSEST,
```

```
                        max_edit_distance=2)

            if len(suggestions)<=0 or contain_digits(incorrect_spelling):
                w = incorrect_spelling

            else:
                w = str(suggestions[0]).split(',')[0]

        #w = spell.correction(incorrect_spelling)
        if w != target:
            bad += 1




    dt = time.process_time() - start
    print('{:.0%} of {} correct ({:.0%} unknown) at {:.0f} words per second '
          .format((n-bad)/ n, n, bad / n, n / dt))

    return bad
```

```
Peter Norvig Naive Evaluation Result:

82% of 10177 correct (18% unknown) at 83 words per second
```

Norvig model Problem:

Is very slow if applied on our medium -large data say applied on 60000 words document.

(60000 / 83) / 60 ~= 12 min .

Symmetric Delete Spelling Correction:

we can apply some modification on the Norvig approach. We will generate terms with an edit distance (deletes only) from each dictionary term and add them

together with the original term to the dictionary. This has to be done only once during a pre-calculation step.

Generate terms with an edit distance (deletes only) from the input term and search them in the dictionary.

For a word of length n, an alphabet size of a, an edit distance of 1, there will be just n deletions, for a total of n terms at search time.

This is **three orders of magnitude less expensive** (36 terms for n=9 and d=2) and **language independent** (the alphabet is not required to generate deletes).

The cost of this approach is the pre-calculation time and storage space of x deletes for every original dictionary entry, which is acceptable in most cases.

The number x of deletes for a single dictionary entry depends on the maximum edit distance: x=n for edit distance=1, x=n*(n-1)/2 for edit distance=2, x=n!/d!/(n-d)! for edit distance=d (combinatorics: k out of n combinations without repetitions, and k=n-d),

E.g. for a maximum edit distance of 2 and an average word length of 5 and 100,000 dictionary entries we need to additionally store 1,500,000 deletes.

The Symmetric Delete spelling correction algorithm **reduces the complexity of edit candidate generation** and dictionary lookup **by using deletes only** instead of deletes +transposes + replaces + inserts. It is six orders of magnitude faster (for edit distance=3) and language independent.

```
def get_deletes_list(w):
    '''The same before but with only deletes no insertion or transition given a word, derive
strings with up to max_edit_distance characters
        deleted'''
    deletes = []
    queue = [w]
    for d in range(max_edit_distance):
        temp_queue = []
```

```
    for word in queue:
        if len(word)>1:
            for c in range(len(word)):  # character index
                word_minus_c = word[:c] + word[c+1:]
                if word_minus_c not in deletes:
                    deletes.append(word_minus_c)
                if word_minus_c not in temp_queue:
                    temp_queue.append(word_minus_c)
    queue = temp_queue

return deletes
```

Note 1: During the precalculation, different words in the dictionary might lead to same delete term: delete(sun,1)==delete(sin,1)==sn.

While we generate only one new dictionary entry (sn), inside we need to store both original terms as spelling correction suggestion (sun,sin)

Note 2: There are four different comparison pair types:

dictionary entry == input entry,

delete(dictionary entry,p1) == input entry

dictionary entry == delete(input entry,p2)

delete(dictionary entry,p1) == delete(input entry,p2)

The last comparison type is required for replaces and transposes only. But we need to check whether the suggested dictionary term is really a replace or an adjacent transpose of the input term to prevent false positives of higher edit distance (bank==bnak and bank==bink, but bank!=kanb and bank!=xban and bank!=baxn).

Note 3: Instead of a dedicated spelling dictionary we are using the search engine index itself. This has several benefits:

It is dynamically updated. Every newly indexed word, whose frequency is over a certain threshold, is automatically used for spelling correction as well.

As we need to search the index anyway the spelling correction comes at almost no extra cost.

When indexing misspelled terms (i.e. not marked as a correct in the index) we do a spelling correction on the fly and index the page for the correct term as well.

Note 4: We have implemented query suggestions/completion in a similar fashion. This is a good way to prevent spelling errors in the first place. Every newly indexed word, whose frequency is over a certain threshold, is stored as a suggestion to all of its prefixes (they are created in the index if they do not yet exist). As we anyway provide an instant search feature the lookup for suggestions comes also at almost no extra cost. Multiple terms are sorted by the number of results stored in the index.

Reasoning

The SymSpell algorithm exploits the fact that the edit distance between two terms is symmetrical:

We can generate all terms with an edit distance <2 from the query term (trying to reverse the query term error) and checking them against all dictionary terms,

We can generate all terms with an edit distance <2 from each dictionary term (trying to create the query term error) and check the query term against them.

We can combine both and meet in the middle, by transforming the correct dictionary terms to erroneous strings, and transforming the erroneous input term to the correct strings.

Because adding a char on the dictionary is equivalent to removing a char from the input string and vice versa, we can on both sides restrict the transformation to deletes only.

We are using variant 3, because the delete-only-transformation is language independent and three orders of magnitude less expensive.

Where does the speed come from?

Pre-calculation, i.e. the generation of possible spelling error variants (deletes only) and storing them at index time is the first precondition.

A fast index access at search time by using a hash table with an average search time complexity of O(1) is the second precondition.

But only the Symmetric Delete Spelling Correction on top of this allows to bring this O(1) speed to spell checking, because it allows a tremendous reduction of the number of spelling error candidates to be pre-calculated (generated and indexed).

Applying pre-calculation to Norvig's approach would not be feasible because pre-calculating all possible delete + transpose + replace + insert candidates of all terms would result in a huge time and space consumption.

Computational Complexity

The SymSpell algorithm is constant time (O(1) time), i.e. independent of the dictionary size (but depending on the average term length and maximum edit distance), because our index is based on a **Hash Table** which has an average search time complexity of O(1).

Comparison to other approaches

BK-Trees have a search time of O(log dictionary size), whereas the SymSpell algorithm is constant time (O(1) time), i.e. independent of the dictionary size.

Tries have a comparable search performance to our approach. But a Trie is a prefix tree, which requires a common prefix. This makes it suitable for autocomplete or search suggestions, but not applicable for spell checking. If your typing error is e.g. in the first letter, than you have no common prefix, hence the Trie will not work for spelling correction.

If you need a very fast auto-complete then try my Pruning Radix Trie

4.1.3 Results

Results using the previous test function and using many python libraries for spell checking.

Peter Norvig Naive Evaluation Result:

82% of 10177 correct (18% unknown) at 83 words per second

Pyspell implementation for Norvig Evaluation Result:

84% of 10177 correct (18% unknown) at 90 words per second

SymPell algorithm Evaluation Result:

80% of 10177 correct (18% unknown) at 16124 words per second

4.2 Further Work

Using BERT to select best candidate.

We trained BERT and tried to use it for predicting the best candidate.

```
from transformers import RobertaConfig, RobertaTokenizerFast,
RobertaForMaskedLM, LineByLineTextDataset,
DataCollatorForLanguageModeling


config = RobertaConfig(
    vocab_size=52_000,
    max_position_embeddings=514,
    Num_attention_heads = 12,
    Num_hidden_layers = 6,
    Type_vocab_size = 1,
)
```

```
dataset = LineByLineTextDataset(
    tokenizer=tokenizer,
    file_path=TRAIN_TEXT_FILE,
    block_size=128,
)
from transformers import
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer, mlm=True, mlm_probability=0.15
)


from transformers import Trainer, TrainingArguments
training_args = TrainingArguments(
    output_dir="~/transformers",
    overwrite_output_dir=True,
    num_train_epochs=3,
    per_gpu_train_batch_size=32,
```

```
    save_steps=10_000,
    save_total_limit=2,
)


trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=dataset,
    prediction_loss_only=False,
)
trainer.train()
```

Using Bert as previous to select most probable n candidate may Boost our algorithm to **90%** accuracy.

Applying final spell correction on data

Clearing our data to get only English words:

1. reveal contractions ex: I’ve ⇒ I have

```
#reveal contractions
import contractions  #to reveal contractions ex: I'll ==> I will
ques['Text'] = ques['Text'].apply(lambda x: contractions.fix(x))
ques['Title'] = ques['Title'].apply(lambda x: contractions.fix(x))
```

2. removing expected noise in our Text.

```
#removing expected noise in our Text.
noise = '!"$%&\'()*+,-./:;?@[\\]^_`{|}~\n<=>'
for i in noise:
```

```
    ques['Text'] = ques['Text'].str.replace(i,' ',regex=True)
    ques['Title'] = ques['Title'].str.replace(i,' ',regex=True)
```

3. Clear extra spaces and new lines,.....

```
def clean(text):


    text = re.sub(r"\'", "'", text)

# match all literal apostrophe pattern then replace them by a single whitespace


    text = re.sub(r"\n", " ", text)

# match all literal Line Feed (New line) pattern then replace them by a single
#whitespace


    text = re.sub(r"\xa0", " ", text)

# match all literal non-breakable space pattern then replace them by a single
#whitespace


    text = re.sub('\s+', ' ', text)

# match all one or more whitespace then replace them by a single whitespace


    text = text.strip(' ')

return text
```

```
#apply clean() on our data frame
ques['Text'] = ques['Text'].apply(lambda x: clean(x))
ques['Title'] = ques['Title'].apply(lambda x: clean(x))
```

Data after Cleaning:

```
ques['Text'][11]  # data after full cleaning
```

>>>

```
'I have got a menu in Python That part was easy I am using to get the selection from the user The problem is that and input require the user to press Enter after they make a selection Is there any way to make the program act immediately upon a keystroke here is what I have got so far It would be great to have something like'
```

```
corpus = [] #corpus of text of ques['text','Title']
for i in range(len(ques)):
    [corpus.append(word.lower())for word in ques['Text'][i].split(' ')]
    [corpus.append(word.lower())for word in ques['Title'][i].split(' ')]
```

Applying correction on the set of our words

```
unique_words = set(corpus)

#using set to reduce runtime of checking typos process

#get unique words either it wrong or right
```

```
print(f"all words in text = {len(corpus)}")
print(f"unique words in text = {len(unique_words)}")
```

>>>

```
all words in text = 63956411
unique words in text = 331706
```

Term Frequency array

We want our program not to change our Keywords of python and language itself like (Tensorflow , pytorch , pandas,.....)

- Coding a dict that count each word and it's frequency.

Ex : implementation : 501250

Implemntaiton : 10

- Generate all possible mistakes for top 100 most common misspelled English words with **Edit Function used before in Peter Norvig model**

```
def edits1(word):
   "All edits that are one edit away from `word`."
   letters    = 'abcdefghijklmnopqrstuvwxyz'
   splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
   deletes    = [L + R[1:]               for L, R in splits if R]
   transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
   replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]
   inserts    = [L + c + R               for L, R in splits for c in letters]
   return set(deletes + transposes + replaces + inserts)


imp = edits1("achieve") #Example on word achieve
```

>>>

```
{'aachieve',  'aahieve', 'abchieve', 'abhieve', 'acahieve', 'acaieve', 'acbhieve', 'acbieve',....}
```

After applying above function on top 100 word most common misspelled English

So after some analysing of data we can see obviously that average count of one possible misspelled word is 20 example word "helo" repeated 15 times.

Results:

That any non-english count above 20 times in our corpus is a python related word like [Tuples, Tensors,.....].

Applying the SymSpell algorithm on our Set of words.

```
# loading typo checking model and it's dictionary

sym_spell = SymSpell(max_dictionary_edit_distance=2, prefix_length=7)
dictionary_path = pkg_resources.resource_filename(
    "symspellpy", "frequency_dictionary_en_82_765.txt")

sym_spell.load_dictionary(dictionary_path, term_index=0, count_index=1)
```

Making dictionary that take input as word and Give a Corrected word

If the input word was correct it will return it the same because it will take it as it's **edit distance is zero** .

```
corrected_words = {} #dictionary to store word and it's correction in unqiue_words.
n, bad = 0, 0  #n is a number of words , bad is a number of wrong words
start = time.process_time()
for word in unique_words:
        n += 1
        suggestions = sym_spell.lookup(word, Verbosity.CLOSEST,
                        max_edit_distance=1)

        if len(suggestions)<=0 or contain_digits(word) or term_frequncy[word]>50:
            w = word

        else:
            w = str(suggestions[0]).split(',')[0]

        if w != word:
            bad +=1

        corrected_words[word] = w

dt = time.process_time() - start
```

```
print('{:.0%} of {} correct ({:.0%} unknown) at {:.0f} words per second '
      .format((n-bad)/ n, n, bad / n, n / dt))
```

Final Result :

1. We corrected around 1% of all the corpus that this words was classified as misspelled.
2. Saving a dictionary that correct our words to use later in our project

```
#saving dictionary to using it in further preprocessing
import pickle
with open('corrected_words.pkl', 'wb') as handle:
   pickle.dump(corrected_words, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

Ex:

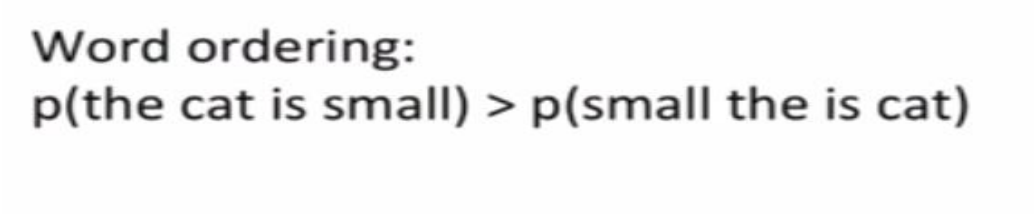
```
Corrected["ello"] >>> "hello"
```

5 Language model

The main job of language model is to make good understanding of corpus, get sense of what sentence mean, learns to predict the probability of a sequence of words. But why do we need to learn the probability of words? Let's understand that with an example.

I am sure you have used Google Translate at some point. We all use it to translate one language to another for varying reasons. This is an example of a popular NLP application called Machine Translation.

In Machine Translation, you take in a bunch of words from a language and convert these words into another language. Now, there can be many potential translations that a system might give you and you will want to compute the probability of each of these translations to understand which one is the most accurate.



In the above example, we know that the probability of the first sentence will be more than the second, right? That is how we arrive at the right translation.

This ability to model the rules of a language as a probability gives great power for NLP related tasks. Language models are used in speech recognition, machine translation, part-of-speech tagging, parsing, Optical Character Recognition, handwriting recognition, information retrieval, and many other daily tasks.

Another example showing the function of Language model and it's importance

In this example the good language model gives high probability to words like (runs, trots, walks) based on understanding the words very well.

5.1 Why we need language models?

→Require understanding of context to make good guesses.

→can learn from massive unlabeled dataset.

→form the foundation of modern NLP systems.

5.2 Importance of language modeling

Language modeling is crucial in modern NLP applications. It is the reason that machines can understand qualitative information. Each language model type, in one way or another, turns qualitative information into quantitative information. This allows people to communicate with machines as they do with each other to a limited extent.

It is used directly in a variety of industries including tech, finance, healthcare, transportation, legal, military and government. Additionally, it is likely most people reading this have interacted with a language model in some way at some point in the day, whether it be through Google search, an autocomplete text function or engaging with a voice assistant.

The roots of language modeling as it exists today can be traced back to 1948. That year, Claude Shannon published a paper titled “A Mathematical Theory of Communication.” In it, he detailed the use of a stochastic model called the Markov chain to create a statistical model for the sequences of letters in English text. This paper had a large impact on the telecommunications industry, laid the groundwork for information theory and language modeling. The Markov model is still used today, and n-grams specifically are tied very closely to the concept.

5.3 Types of Language Models

1. **Statistical Language Models:** These models use traditional statistical techniques like N-grams, Hidden Markov Models (HMM) and certain linguistic rules to learn the probability distribution of words.
2. **Neural Language Models:** These are new players in the NLP town and have surpassed the statistical language models in their effectiveness. They use different kinds of Neural Networks to model language.

5.3.1 Building an N-gram Language Model

What are N-grams (unigram, bigram, trigrams)?: An N-gram is a sequence of N tokens (or words).

Let's understand N-gram with an example. Consider the following sentence:

"I love reading blogs about data science on Analytics Vidhya."

A 1-gram (or unigram) is a one-word sequence. For the above sentence, the unigrams would simply be: "I", "love", "reading", "blogs", "about", "data", "science", "on", "Analytics", "Vidhya".

A 2-gram (or bigram) is a two-word sequence of words, like "I love", "love reading", or "Analytics Vidhya". And a 3-gram (or trigram) is a three-word sequence of words like "I love reading", "about data science" or "on Analytics Vidhya".

Fairly straightforward stuff!

How do N-gram Language Models work?

An N-gram language model predicts the probability of a given N-gram within any sequence of words in the language. If we have a good N-gram model, we can predict p(w | h) – what is the probability of seeing the word w given a history of previous words h – where the history contains n-1 words.

Limitations of N-gram approach to Language Modeling

1. The higher the N, the better is the model usually. But this leads to lots of computation overhead that requires large computation power in terms of RAM.
2. N-grams are a sparse representation of language. This is because we build the model based on the probability of words co-occurring. It will give zero probability to all the words that are not present in the training corpus.

5.3.2 Building a Neural Language Model:

Deep Learning waves have lapped at the shores of computational linguistics for several years now, but 2015 seems like the year when the full force of the tsunami hit the major Natural Language Processing (NLP) conferences." – Dr. Christopher D. Manning.

Deep Learning has been shown to perform really well on many NLP tasks like Text Summarization, Machine Translation, etc. and since these tasks are essentially built upon Language Modeling, there has been a tremendous research effort with great results to use Neural Networks for Language Modeling.

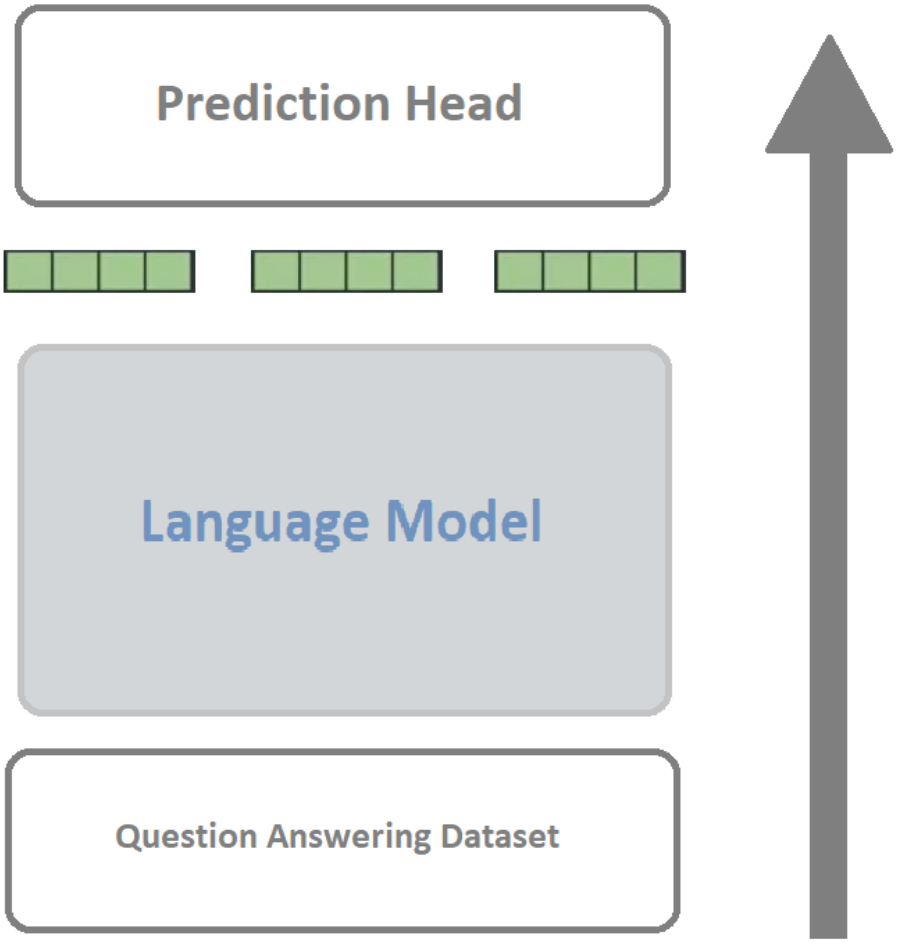
We can essentially build two kinds of language models – character level and word level. And even under each category, we can have many subcategories based on the simple fact of how we are framing the learning problem. We will be taking the most straightforward approach – building a character-level language model.

5.4 Modern NLP workflow using transfer learning

5.4.1 Transfer learning

Using already trained robust models to complete NLP tasks in shorter time and using less resources.

In these recent times, we have become very good at predicting a very accurate outcome with very good training models. But considering most of the machine learning tasks are domain specific, the trained models usually fail to generalize the conditions that it has never seen before. The real world is not like the trained data set, it contains lot of messy data, and the model will make an ill prediction in such condition. The ability to transfer the knowledge of a pre-trained model into a new condition is generally referred to as transfer learning.

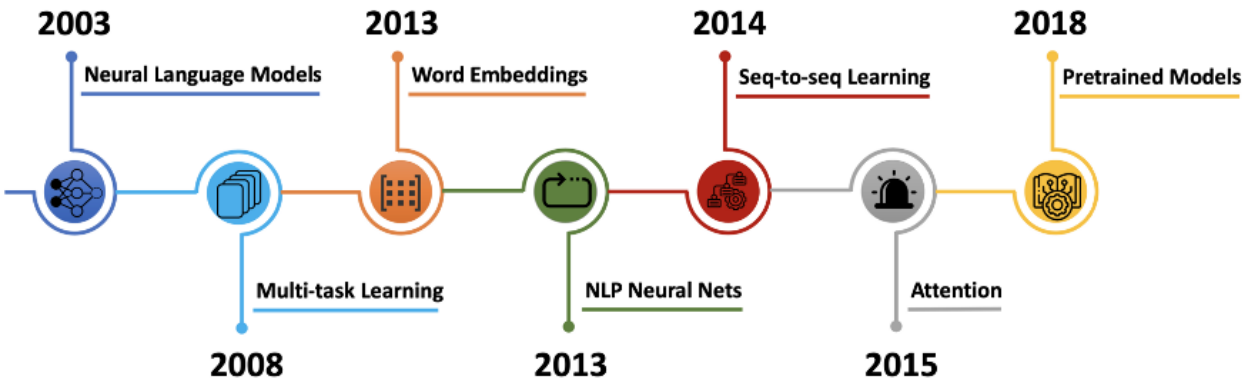


By using transfer learning what we will do here is giving a pretrained Language model unlabeled text corpus which the model can learn from it and after that

understanding the model represent the corpus and tokenize it then sending these tokens to the prediction head to perform any NLP task (here our task is Q&A).

Before discussing any type of language models let’s talk about how we can represent words to pass them to LM (representing the word using numbers ”vectorization”).

5.4.2 Timeline of Language models



Here we will discuss:

1. Word Embeddings.
2. Seq-to-seq Learning.
3. Attention.
4. Pretrained Models

But before that let us discuss the simplest method for representing words.

5.4.2.1 One-Hot Encoding

The simplest method is called one-hot encoding, also known as "1-of-N" encoding which mean that the vector is composed of a single one and a number of zeros.

If a data point belongs to its category, then components of this vector are assigned the value 0 except for its component which is assigned a value of 1. In this way one can keep track of the categories in a numerically meaningful way.

Example: Let us look at the following sentence: "I ate an apple and played the piano." We can begin by indexing each word's position in the given vocabulary set.

I	ate	an	apple	and	played	the	piano
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Position of each word in the vocabulary

The word "I" is at position 1, so its one-hot vector representation would be [1, 0, 0, 0, 0, 0, 0, 0]. Similarly, the word "ate" is at position 2, so its one-hot vector would be [0, 1, 0, 0, 0, 0, 0, 0]. The number of words in the source vocabulary signifies the number of dimensions — here we have eight.

The one-hot embedding matrix for the example text would look like this:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
I	1	0	0	0	0	0	0	0
ate	0	1	0	0	0	0	0	0
an	0	0	1	0	0	0	0	0
apple	0	0	0	1	0	0	0	0
and	0	0	0	0	1	0	0	0
played	0	0	0	0	0	1	0	0
the	0	0	0	0	0	0	1	0
piano	0	0	0	0	0	0	0	1

One-hot vector representation of each word in the vocabulary

Problems with One-Hot Encoding

There are two major issues with this approach for word embedding.

First issue is the curse of dimensionality, which refers to all sorts of problems that arise with data in high dimensions. Even with relatively small eight dimensions, our example text requires exponentially large memory space. Most of the matrix is taken up by zeros, so useful data becomes sparse. Imagine we have a vocabulary of 50,000. (There are roughly a million words in English language.) Each word is represented with 49,999 zeros and a single one, and we need 50,000 squared = 2.5 billion units of memory space. Not computationally efficient.

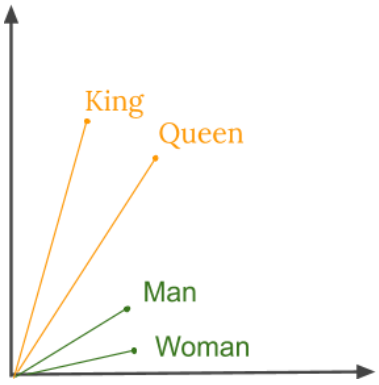
Second issue is that it is hard to extract meanings. Each word is embedded in isolation, and every word contains a single one and N zeros where N is the number of dimensions. The resulting set of vectors do not say much about one another. If our vocabulary had “orange,” “banana,” and “watermelon,” we can see the similarity between those words, such as the fact that they are types of fruit, or that they usually follow some form of the verb “eat.” We can easily form a mental map or cluster where these words exist close to each other. But with one-hot vectors, all words are equal distance apart.

Based on these problems of One-Hot encoding a new method have been raised let us discuss these methods,

5.4.2.2 Word Embeddings

Humans have always excelled at understanding languages. It is easy for humans to understand the relationship between words but for computers, this task may not be simple. For example, we humans understand the words like king and queen, man and woman, tiger and tigress have a certain type of relation between them but how can a computer figure this out?

Word embeddings are basically a form of word representation that bridges the human understanding of language to that of a machine. They have learned representations of text in an n-dimensional space where words that have the same meaning have a similar representation. Meaning that two similar words are represented by almost similar vectors that are very closely placed in a vector space. These are essential for solving most Natural language processing problems.



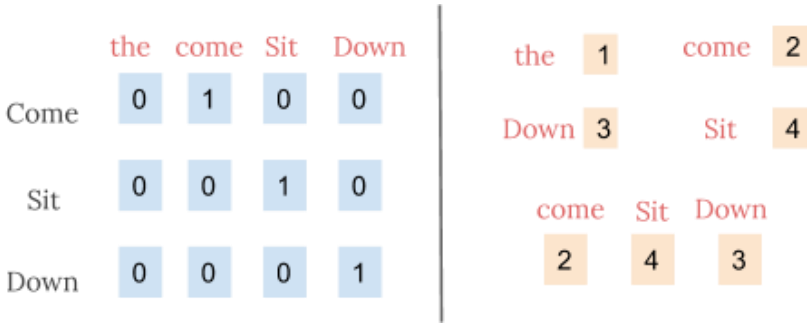
Thus, when using word embeddings, all individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network.

Word2Vec is one of the most popular technique to learn word embeddings using shallow neural network. It was developed by Tomas Mikolov in 2013 at Google.

Why are Word Embeddings used?

As we know the machine learning models cannot process text, so we need to figure out a way to convert these textual data into numerical data. Previously techniques like Bag of Words and TF-IDF have been discussed that can help achieve use this task. Apart from this, we can use two more techniques such as one-hot encoding, or we can use unique numbers to represent words in a vocabulary. The latter approach is more efficient than one-hot encoding as instead of a sparse vector, we now have a dense one. Thus, this approach even works when our vocabulary is large.

In the below example, we assume we have a small vocabulary containing just four words, using the two techniques we represent the sentence ‘Come sit down’.



However, the integer-encoding is arbitrary as it does not capture any relationship between words. It can be challenging for a model to interpret, for example, a linear classifier learns a single weight for each feature. Because there is no relationship between the similarity of any two words and the similarity of their encodings, this feature-weight combination is not meaningful.

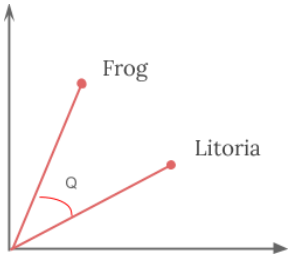
Thus, by using word embeddings, words that are close in meaning are grouped near to one another in vector space. For example, while representing a word such as frog, the nearest neighbor of a frog would be frogs, toads, Litoria. This implies that it is alright for a classifier to not see the word Litoria and only frog during training, and the classifier would not be thrown off when it sees Litoria during testing because the two-word vectors are similar. Also, word embeddings learn relationships. Vector differences between a pair of words can be added to another word vector to find the analogous word. For example, “man” –“woman” + “queen” ≈ “king”.

5.4.2.3 Word2vec

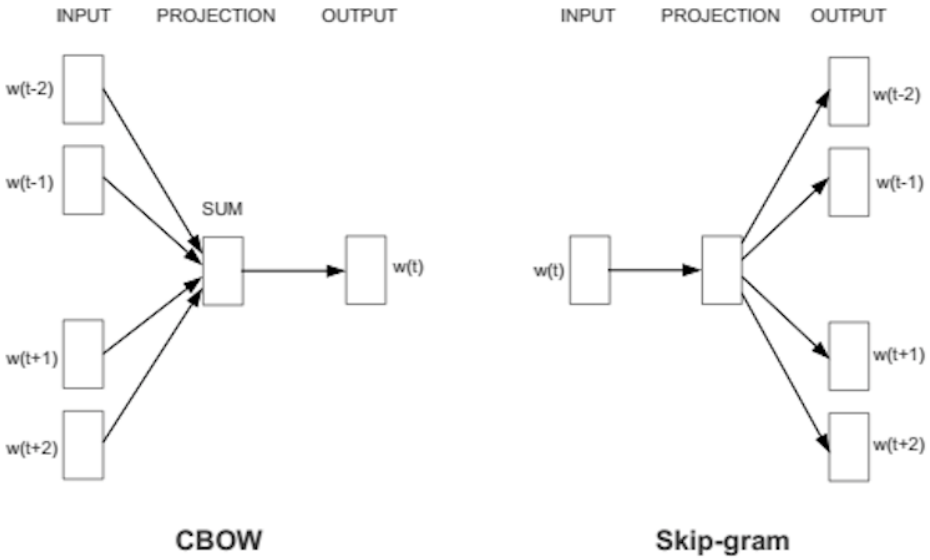
Word2vec is a method to efficiently create word embeddings by using a two-layer neural network. It was developed by Tomas Mikolov, et al. at Google in 2013 as a response to make the neural-network-based training of the embedding more efficient and since then has become the de facto standard for developing pre-trained word embedding.

The input of word2vec is a text corpus and its output is a set of vectors known as feature vectors that represent words in that corpus. While Word2vec is not a deep neural network, it turns text into a numerical form that deep neural networks can understand.

The Word2Vec objective function causes the words that have a similar context to have similar embeddings. Thus, in this vector space, these words are really close. Mathematically, the cosine of the angle (Q) between such vectors should be close to 1, i.e., angle close to 0.



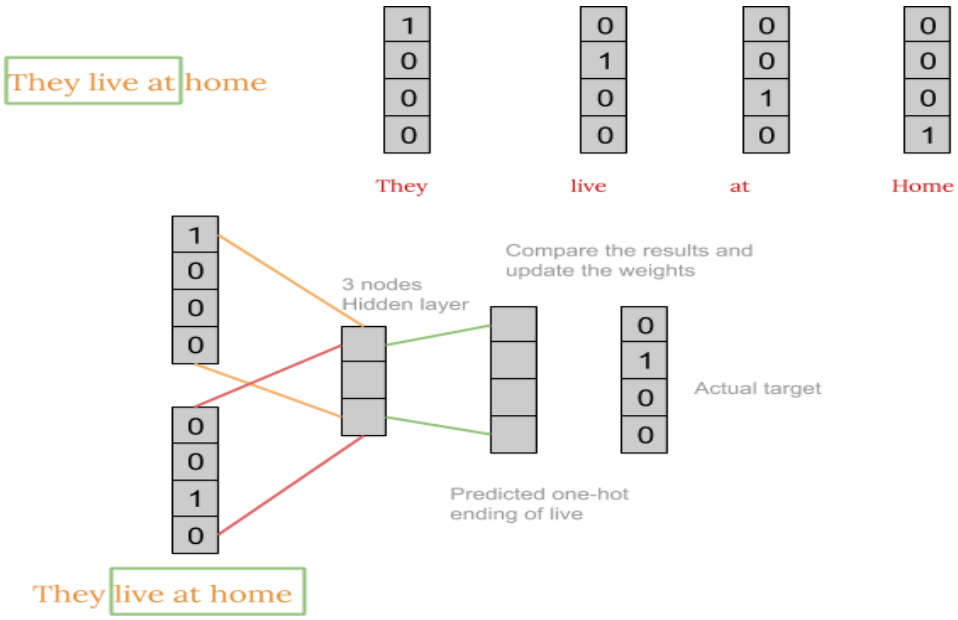
Word2vec is not a single algorithm but a combination of two techniques – CBOW (Continuous bag of words) and Skip-gram model. Both are shallow neural networks which map word(s) to the target variable which is also a word(s). Both techniques learn weights which act as word vector representations.



Continuous Bag-of-Words model (CBOW)

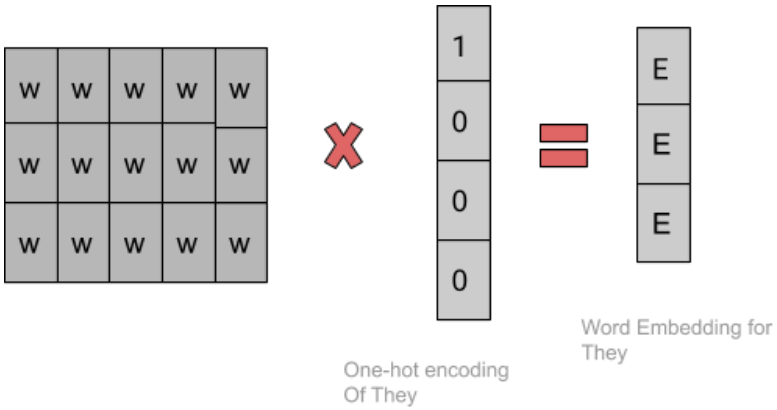
CBOW predicts the probability of a word to occur given the words surrounding it. We can consider a single word or a group of words. But for simplicity, we will take a single context word and try to predict a single target word.

The English language contains almost 1.2 million words, making it impossible to include so many words in our example. So, I ‘ll consider a small example in which we have only four words i.e., live, home, they and at. For simplicity, we will consider that the corpus contains only one sentence, that being, ‘They live at home’.



First, we convert each word into a one-hot encoding form. Also, we will not consider all the words in the sentence but ll only take certain words that are in a window. For example, for a window size equal to three, we only consider three words in a sentence. The middle word is to be predicted and the surrounding two words are fed into the neural network as context. The window is then slid, and the process is repeated.

Finally, after training the network repeatedly by sliding the window a shown above, we get weights which we use to get the embeddings as shown below.

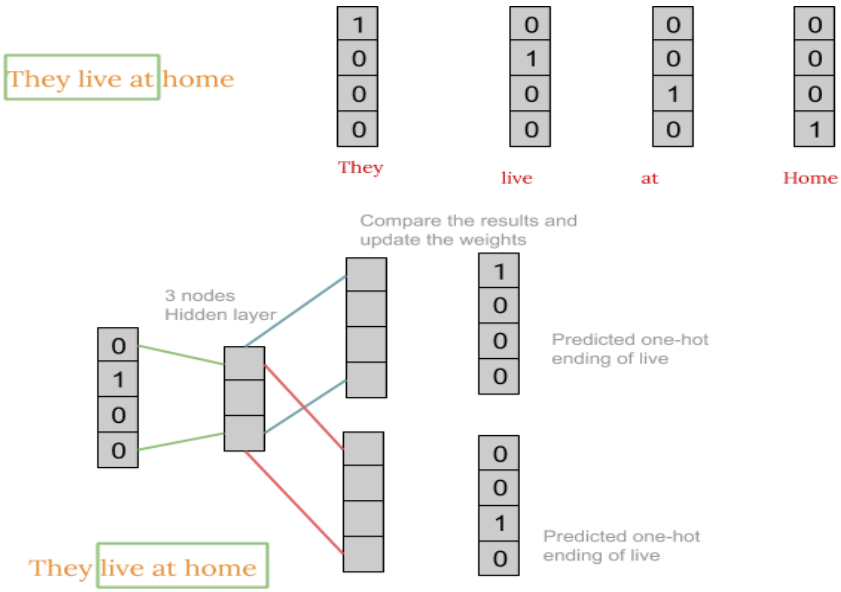


Usually, we take a window size of around 8-10 words and have a vector size of 300.

Skip-gram model

The Skip-gram model architecture usually tries to achieve the reverse of what the CBOW model does. It tries to predict the source context words (surrounding words) given a target word (the center word)

The working of the skip-gram model is quite like the CBOW but there is just a difference in the architecture of its neural network and the way the weight matrix is generated as shown in the figure below:



After obtaining the weight matrix, the steps to get word embedding is same as CBOW.

So now which one of the two algorithms should we use for implementing word2vec? Turns out for large corpus with higher dimensions, it is better to use skip-gram but is slow to train. Whereas CBOW is better for small corpus and is faster to train too.

Word2vec challenges

- Inability to handle unknown or OOV words.
- No shared representations at sub-word levels.
- Scaling to new languages requires new embedding matrices.
- Cannot be used to initialize state-of-the-art architectures.

5.4.2.4 GloVe

GloVe (Global Vectors for Word Representation) is an alternate method to create word embeddings. It is based on matrix factorization techniques on the word-context matrix. A large matrix of co-occurrence information is constructed, and you count each “word” (the rows), and how frequently we see this word in some “context” (the columns) in a large corpus. Usually, we scan our corpus in the following manner: for each term, we look for context terms within some area defined by a window-size before the term and a window-size after the term. Also, we give less weight for more distant words. The number of “contexts” is, of course, large, since it is essentially combinatorial in size. So, then we factorize this matrix to yield a lower-dimensional matrix, where each row now yields a vector representation for each word. In general, this is done by minimizing a “reconstruction loss”. This loss tries to find the lower-dimensional representations which can explain most of the variance in the high-dimensional data.

In practice, we use both GloVe and Word2Vec to convert our text into embeddings and both exhibit comparable performances. Although in real applications we train our model over Wikipedia text with a window size around 5- 10. The number of words in the corpus is around 13 million, hence it takes a huge amount of time and resources to generate these embeddings. To avoid this, we can use the pre-trained word vectors that are already trained, and we can easily use them. Here are the links to download pre-trained Word2Vec or GloVe.

This brings us to the end of this article where we learned about word embedding and some popular techniques to implement them.

5.4.2.5 Sequence-to-sequence Learning

By using RNN basic model we have a big problem, If we have a long sentence this model will give a bad understanding of a meaning, and to solve long dependencies we use seq-to-seq models.

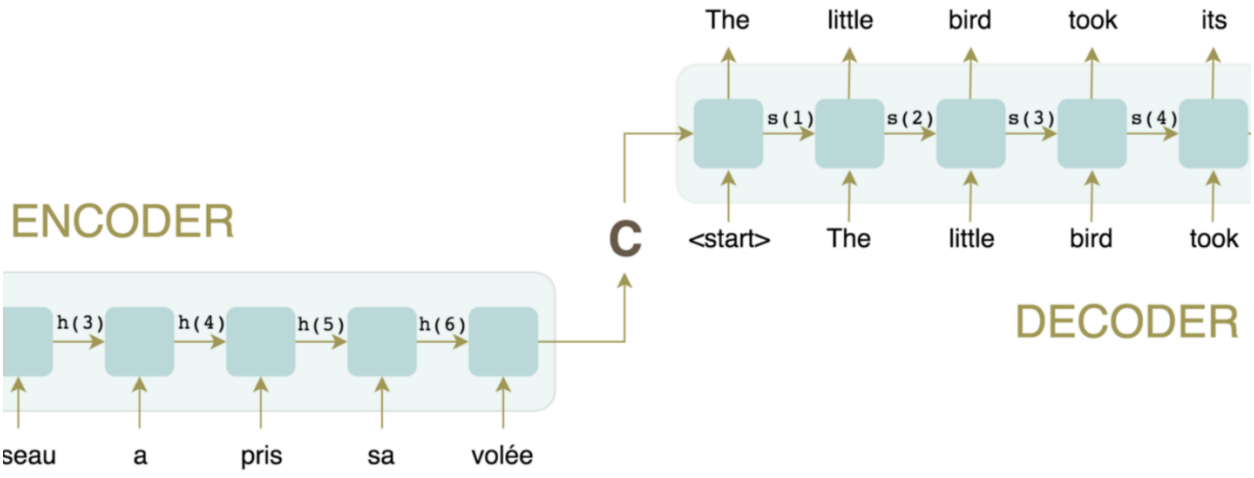
Sequence-to-sequence models are deep learning models that have achieved a lot of success in tasks like machine translation, text summarization, and image captioning. Google Translate started using such a model in production in late 2016.

A sequence-to-sequence model is a model that takes a sequence of words and outputs another sequence of words.

The Model

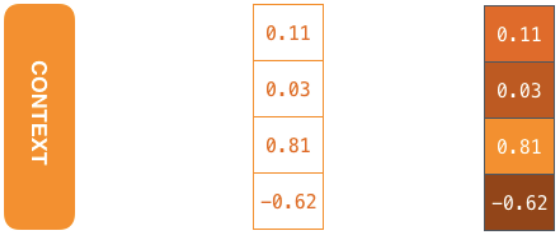
the model is composed of an encoder and a decoder.

The encoder processes each item in the input sequence, it compiles the information it captures into a vector (called the context). After processing the entire input sequence, the encoder sends the context over to the decoder, which begins producing the output sequence item by item.



Encoder-Decoder architecture

The context is a vector The encoder and decoder tend to both be recurrent neural networks.



You can set the size of the context vector when you set up your model. It is basically the number of hidden units in the encoder RNN. These visualizations show a vector of size 4, but in real world applications the context vector would be of a size like 256, 512, or 1024.

By design, a RNN takes two inputs at each time step: an input (in the case of the encoder, one word from the input sentence), and a hidden state.

The next RNN step takes the second input vector and hidden state #1 to create the output of that time step. Ach pulse for the encoder or decoder is that RNN processing its inputs and generating an output for that time step. Since the encoder and decoder are both RNNs, each time step one of the RNNs does some processing, it updates its hidden state based on its inputs and previous inputs it has seen. The decoder also maintains a hidden states that it passes from one time step to the next. We just didn't visualize it in this graphic because we're concerned with the major parts of the model for now.

We have approaches for seq-to-seq, one of them is LSTM.

5.4.2.6 LSTM

Long Short-Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work. They work tremendously well on a large variety of problems and are now widely used.

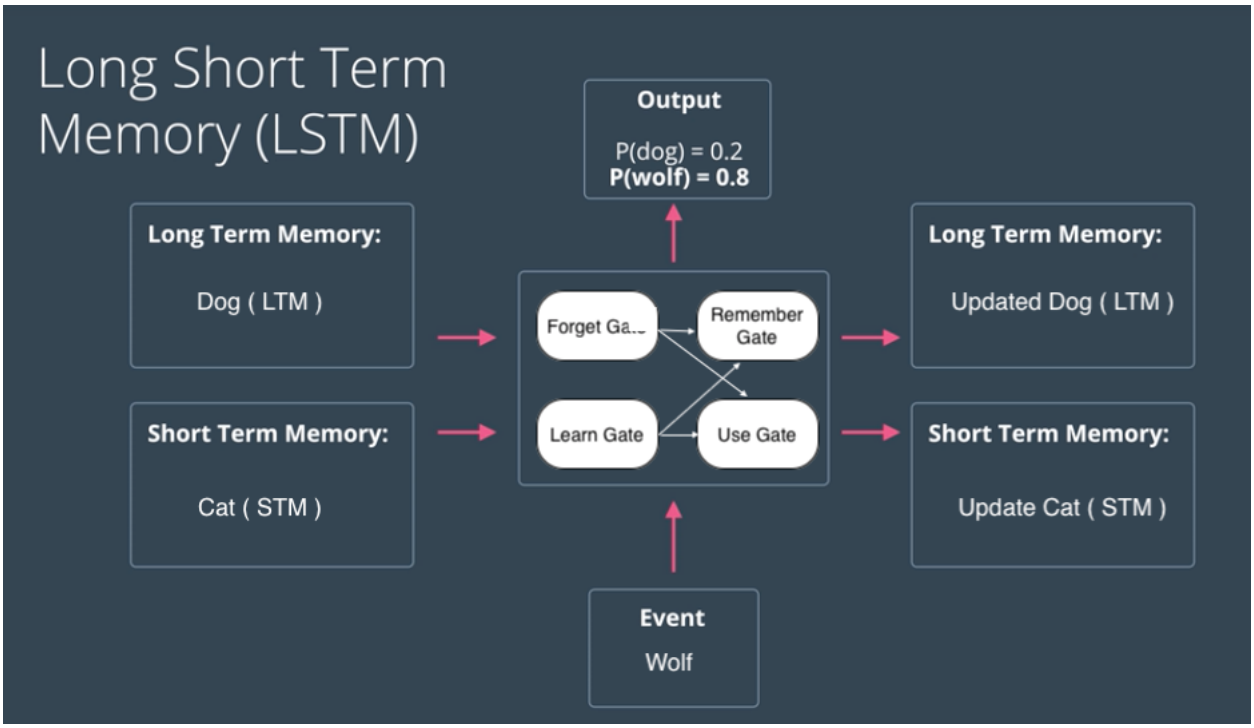
LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

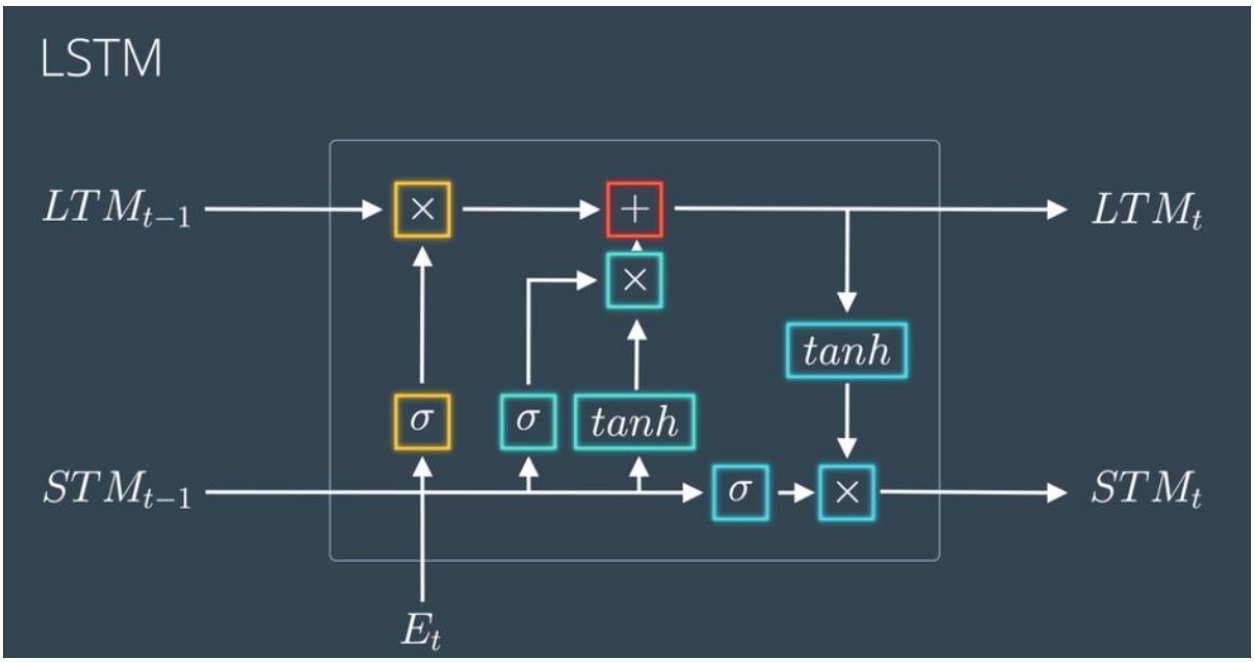
The architecture of LSTM

LSTMs deal with both Long-Term Memory (LTM) and Short-Term Memory (STM) and for making the calculations simple and effective it uses the concept of gates.

1. **Forget Gate:** LTM goes to forget gate and it forgets information that is not useful.
2. **Learn Gate:** Event (current input) and STM are combined together so that necessary information that we have recently learned from STM can be applied to the current input.
3. **Remember Gate:** LTM information that we haven't forget and STM and Event are combined together in Remember gate which works as updated LTM.
4. **Use Gate:** This gate also uses LTM, STM, and Event to predict the output of the current event which works as an updated STM.



The above figure shows the simplified architecture of LSTMs. The actual mathematical architecture of LSTM is represented using the following figure:



Usage of LSTMs

Training LSTMs removes the problem of Vanishing Gradient (weights become too small that under-fits the model), but it still faces the issue of Exploding Gradient (weights become too large that over-fits the model). Training of LSTMs can be easily done using Python frameworks like TensorFlow, Pytorch, Theano, etc. and the catch is the same as RNN, we would need GPU for training deeper LSTM Networks.

Since LSTMs take care of the long-term dependencies its widely used in tasks like Language Generation, Voice Recognition, Image OCR Models, etc. Also, this technique is getting noticed in Object Detection also (mainly scene text detection).

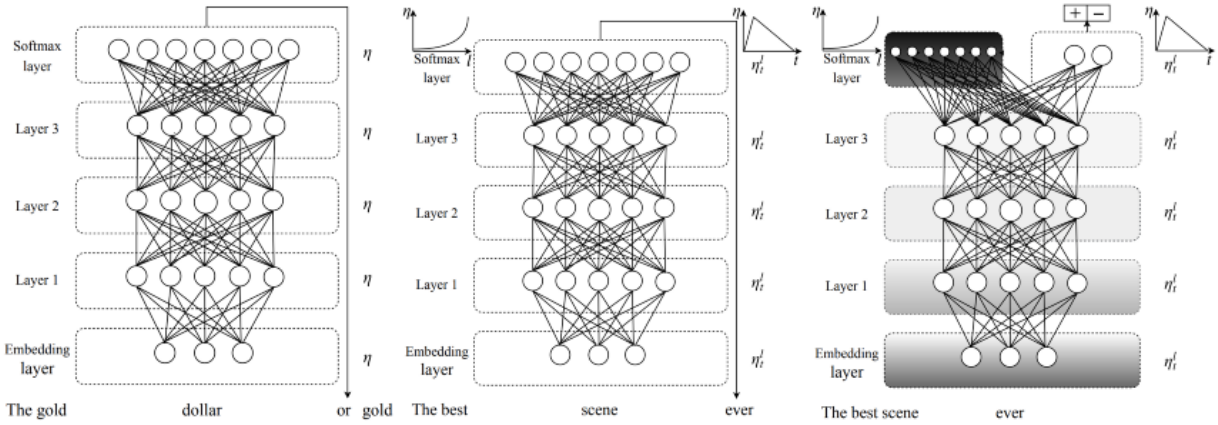
Models based on LSTM.

5.4.2.6.1 ULMFit

(The Universal the Language the Model Fine-tuning).

This ULMFiT is based on AWD-LSTM, which is the best LSTM for language modeling today. When ULMFiT has learned the language model Can be used for various tasks by keeping the lower part and changing only the top layer but implementation might require two more fine-tuning steps. By starting with training, the language model for the work that is used After that, gradually train for real work, such as training to be a classifier.

In this tuning process There are suggestions that should be done. The discriminative fine-tuning is the fine tuning in different layers. Because the models of each layer are representation at different levels and should use slanted triangular learning rates to be able to converge to a good value in the beginning and then gradually adjust to the later and in the final tuning, for example, training to be a classifier, you should adjust the weight one by one. The weight in the unmodified layer will be frozen first. (Meaning keep the original value).



(a) LM pre-training (b) LM fine-tuning (c) Classifier fine-tuning

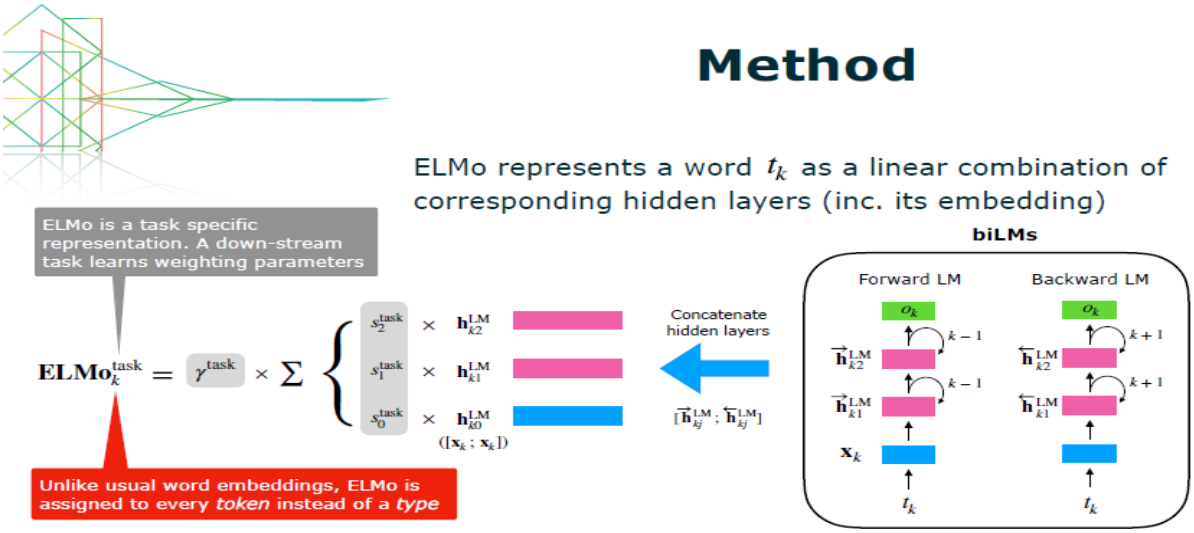
a) ULMFiT training for general language models (b) and then training for a specific language model

© and then actually training for that work (Picture from ULMFiT paper)

5.4.2.6.2 ELMO

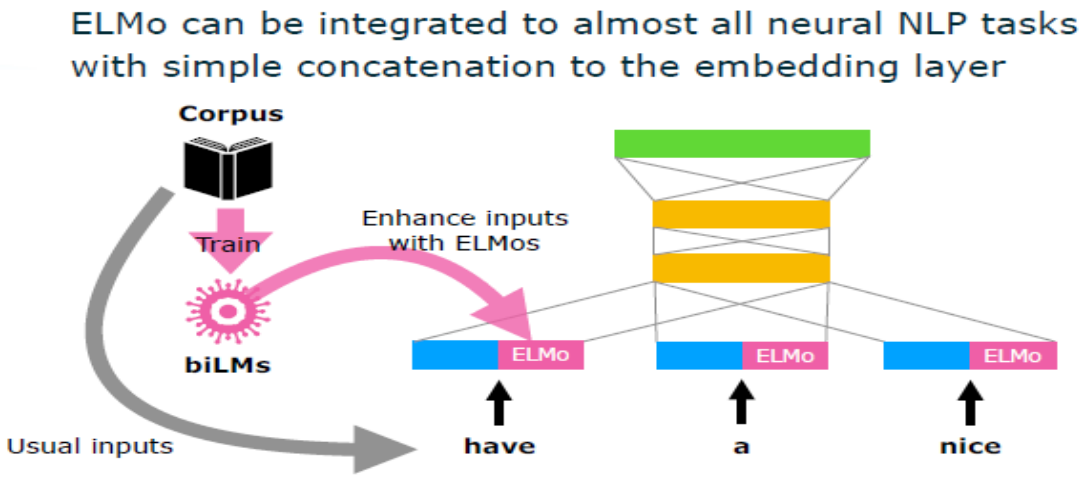
(Embeddings from Language Models)

ELMo's approach is to learn the language model from both the way back and back using LSTM. Time to use Enter our text in this model and remove the hidden state on different layers, giving the weight of how important each layer should be. Which depends on the work we are using. The hidden state in each layer of ELMo is like representation Each level is different.



From right to left, starting from training bidirectional LSTM and implement the hidden state of ELMo.

The use of ELMo is considered contextual embedding, which is better than word embedding. In general, the representation from ELMo also has context in mind. For example, in plain word embedding, the word bank meaning bank and bank means bank is a vector. The same, but because ELMo is the model that receives the whole message to LSTM, it makes the representation That varies according to the incoming message as well. And in the use, add the contextual embedding as the input in the embedding layer as in the picture below.



ELMo's contextual embedding for various tasks.

5.4.2.7 Attention

A neural network is considered to be an effort to mimic human brain actions in a simplified manner. Attention Mechanism is also an attempt to implement the same action of selectively concentrating on a few relevant things, while ignoring others in deep neural networks.

5.4.2.7.1 How Attention Mechanism was Introduced in Deep Learning:

The attention mechanism emerged as an improvement over the encoder decoder-based neural machine translation system in natural language processing (NLP). Later, this mechanism, or its variants, was used in other applications, including computer vision, speech processing, etc.

Before Bahdanau et al proposed the first Attention model in 2015, neural machine translation was based on encoder-decoder RNNs/LSTMs. Both encoder and decoder are stacks of LSTM/RNN units. It works in the two following steps:

1. **The encoder LSTM is used to process the entire input sentence and encode it into a context vector**, which is the last hidden state of the LSTM/RNN. This is expected to be a good summary of the input sentence. All the intermediate states of the encoder are ignored, and the final state id supposed to be the initial hidden state of the decoder.
2. The decoder LSTM or RNN units produce the words in a sentence one after another.

In short, there are two RNNs/LSTMs. One we call the encoder – this reads the input sentence and tries to make sense of it, before summarizing it. It passes the summary (context vector) to the decoder which translates the input sentence by just seeing it.

The main drawback of this approach is evident. If the encoder makes a bad summary, the translation will also be bad. And indeed, it has been observed that the encoder creates a bad summary when it tries to understand longer sentences. It is called the long-range dependency problem of RNN/LSTMs.

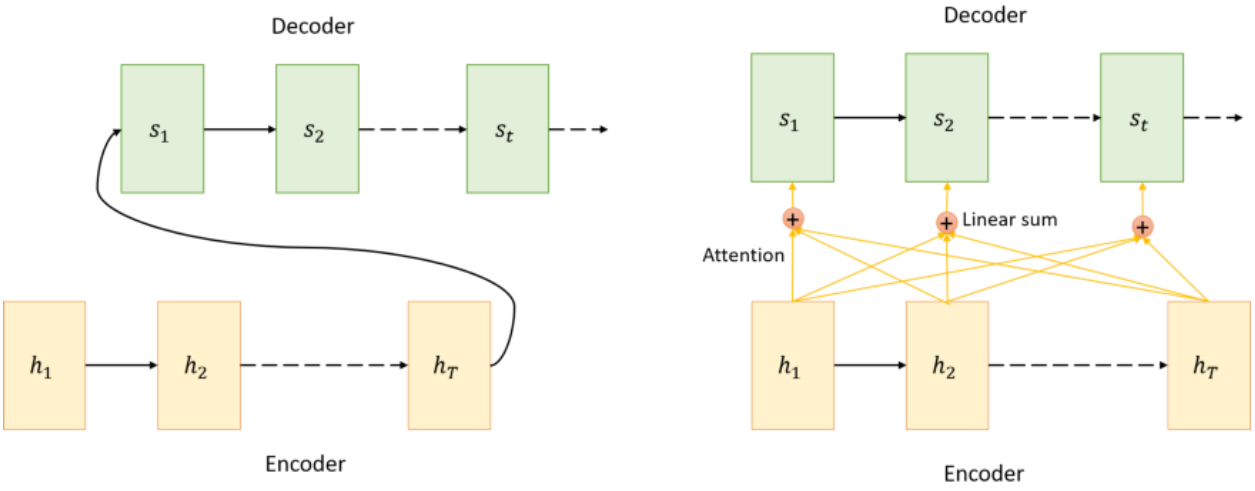
RNNs cannot remember longer sentences and sequences due to the vanishing/exploding gradient problem. It can remember the parts which it has just seen. Even Cho et al (2014), who proposed the encoder-decoder network,

demonstrated that the performance of the encoder-decoder network degrades rapidly as the length of the input sentence increases.

Although an LSTM is supposed to capture the long-range dependency better than the RNN, it tends to become forgetful in specific cases. Another problem is that there is no way to give more importance to some of the input words compared to others while translating the sentence.

5.4.2.7.2 Core Idea behind Attention:

The main assumption in sequence modelling networks such as RNNs, LSTMs and GRUs is that the current state holds information for the whole of input seen so far. Hence the final state of a RNN after reading the whole input sequence should contain complete information about that sequence. This seems to be too strong a condition and too much to ask.



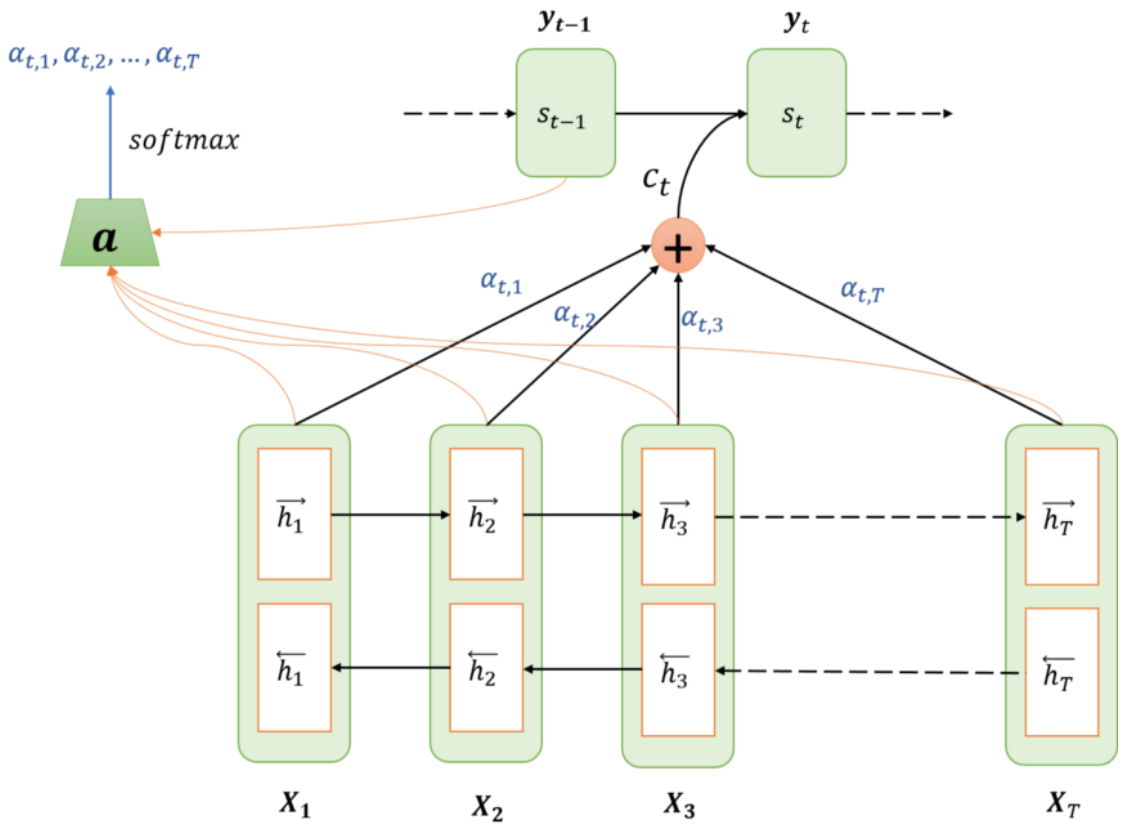
(a) Vanilla Encoder Decoder Architecture **(b) Attention Mechanism**

Attention mechanisms relax this assumption and proposes that we should look at the hidden states corresponding to the whole input sequence in order to make any prediction. But how do we decide which states to look at? Well, try learning that!!

5.4.2.7.3 Step by Step Walk-through

As introduced in the previous section, the task at hand is:

The complete architecture is as follows:



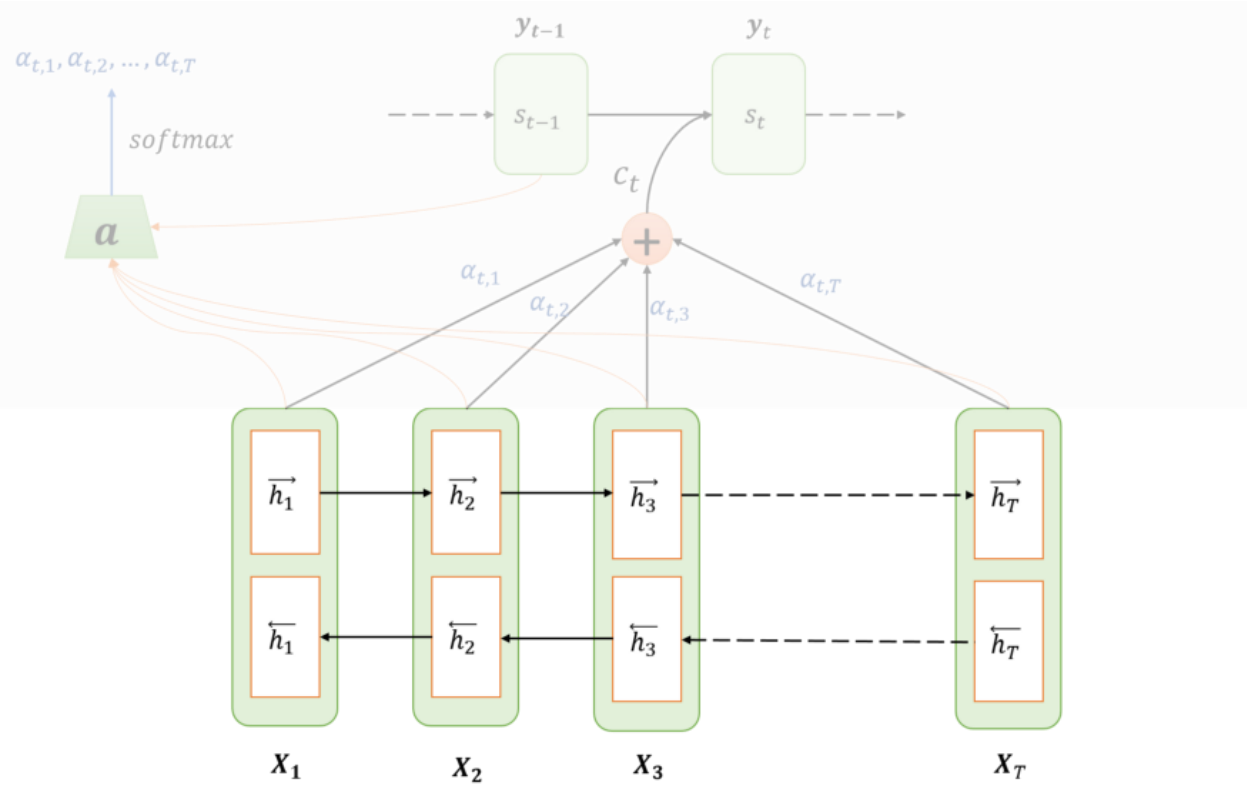
The network is shown in a state when the encoder has computed the hidden states h_j corresponding to each input X_j and the decoder has run for t-1 steps and is now going to produce output for time step t.

Don't get too nervous looking at this seemingly difficult figure. We'll be going through each component one by one. Broadly, the whole process can be divided into four steps:

1. Encoding
2. Computing Attention weights / Alignment
3. Creating context vector
4. Decoding / Translation

Let us have a look!!

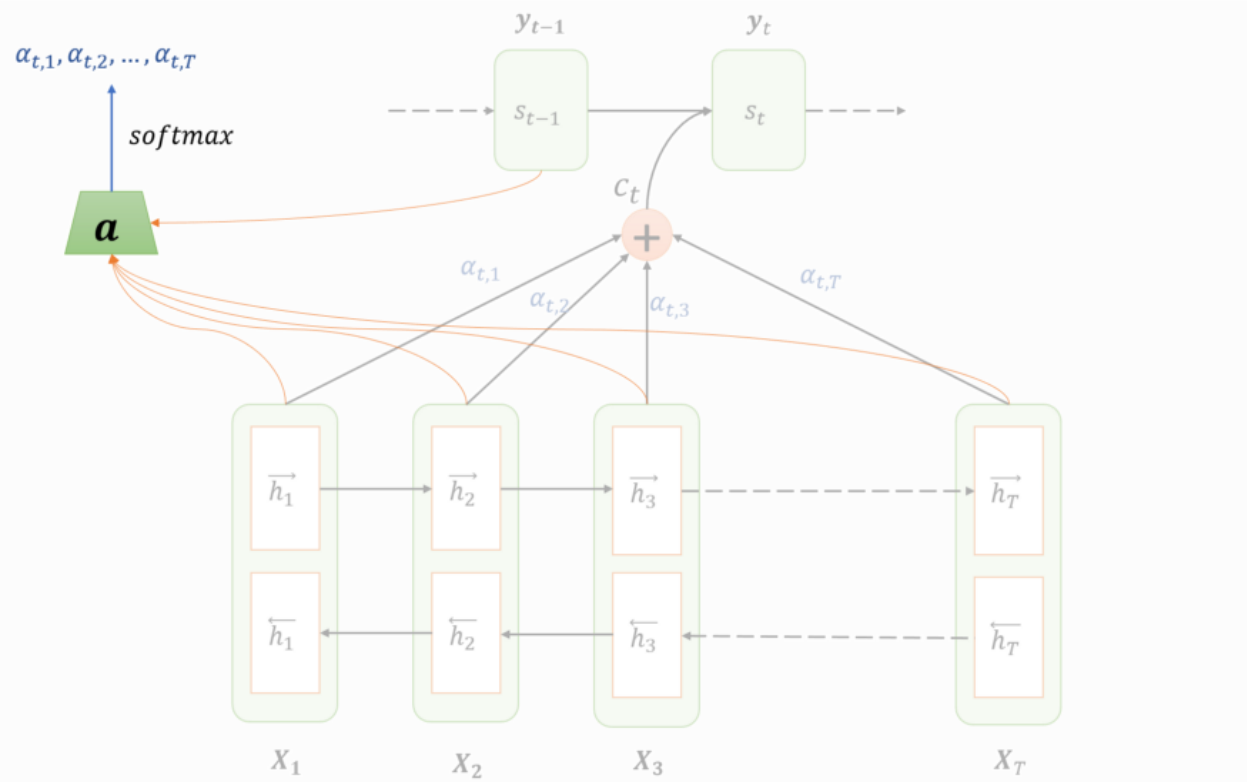
5.4.2.7.3.1 Encoding



$(X_1, X_2, \ldots X_T)$: Input sequence, where T is the length of sequence
$(\vec{h}_1, \vec{h}_2, \ldots \vec{h}_T)$: Hidden states of the forward RNN
$(\overleftarrow{h}_1, \overleftarrow{h}_2, \ldots \overleftarrow{h}_T)$: Hidden states of the backward RNN
$h_j = [\vec{h}_j; \overleftarrow{h}_j], \forall j \in [1, T]$

Hence the hidden state for the j^{th} input h_j is the concatenation of j^{th} hidden states of forward and backward RNNs. We'll be using a weighted linear combination of all of these h_js to make predictions at each step of the decoder. The decoder output length might be same or different than that of encoder.

5.4.2.7.3.2 Computing Attention Weights / Alignment



At each time step t of the decoder the amount of attention to be paid to the hidden encoder unit h_j is denoted by α_{tj} and calculated as a function of both h_j and previous hidden state of decoder s_{t-1}:

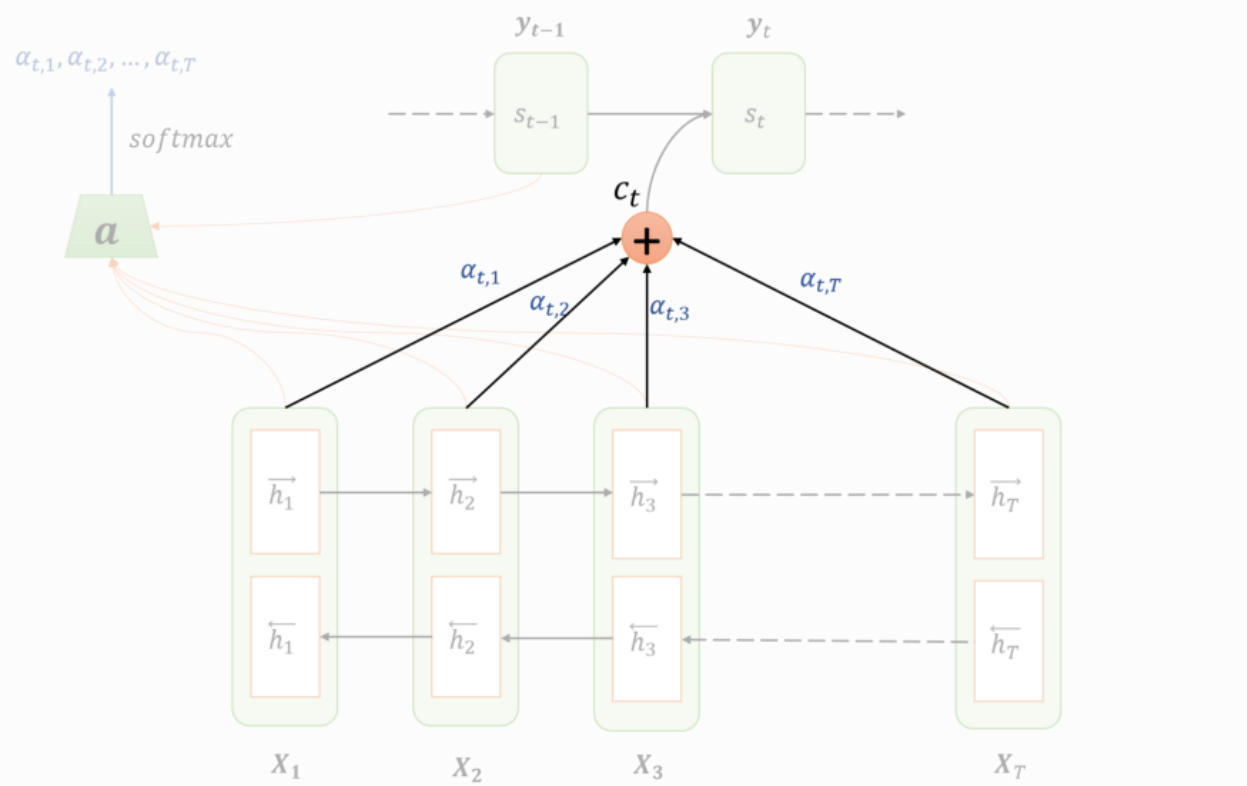
$$e_{tj} = \boldsymbol{a}(h_j, s_{t-1}), \forall j \in [1, T]$$

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{T} \exp(e_{tk})}$$

In the paper a is parametrized as a feedforward neural network that runs for all j at the decoding time step t. Note the $0 \leq \alpha_{tj} \leq 1$ and that all $\sum_j \alpha_{tj} = 1$ because of the softmax on e_{tj}. These α_{tj} can be visualized as the attention paid by decoder at time step t to hidden encoder unit h_j.

5.4.2.7.3.3 Computing context vector

Time to make use of the attention weights we've computed in the preceding step!! Here's another figure to help understand:

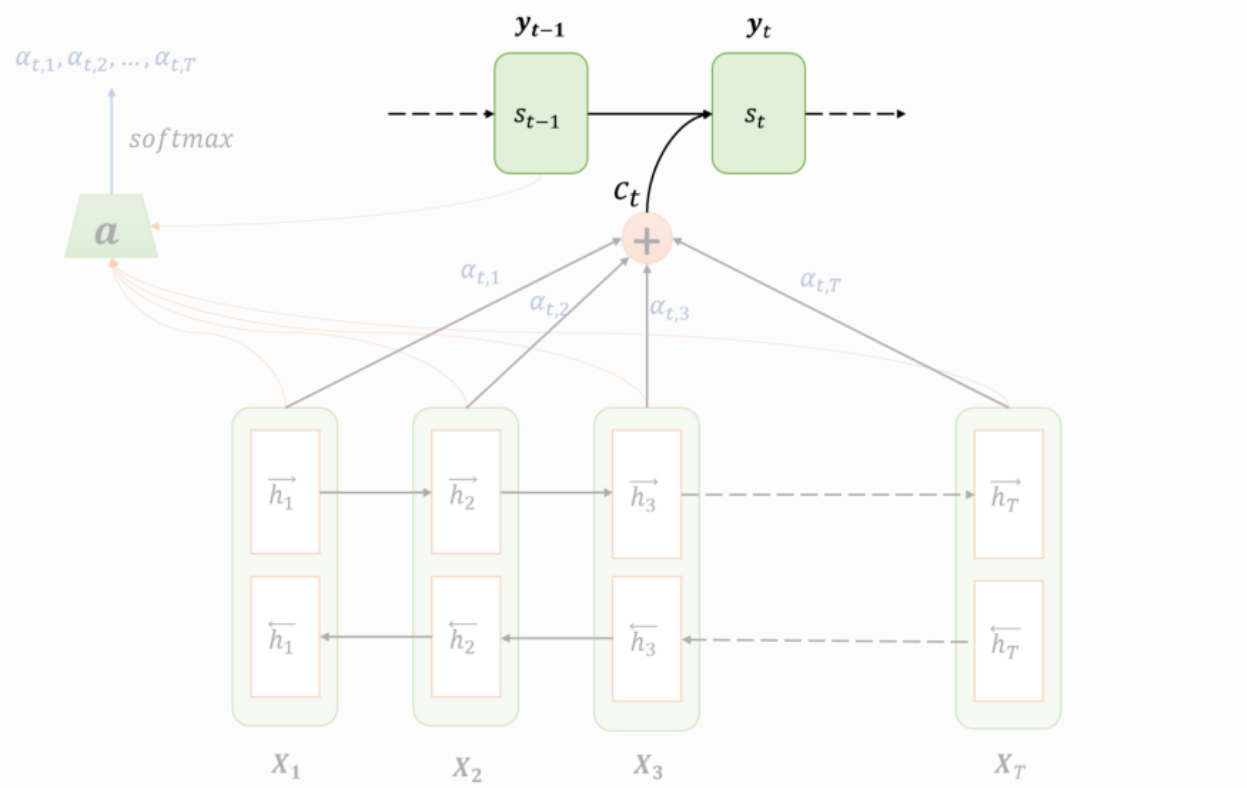


The context vector is simply a linear combination of the hidden weights h_j weighted by the attention values α_{tj} that we computed:

$$c_t = \sum_{j=1}^{T} \alpha_{tj} h_j$$

From the equation we can see that α_{tj} determines how much h_j affects the context c_t. Higher the value, higher the impact of h_j on the context for time t.

5.4.2.7.3.4 Decoding / Translation



We are nearly there! All there remains is to use the context vector c_t we worked so hard to compute, along with the previous hidden state of the decoder s_{t-1} and the previous output $y_{t-1}$and use all of them to compute the new hidden state and output of the decoder: s_t and y_t respectively.

$$s_t = f(s_{t-1}, y_{t-1}, c_t)$$

$$p(y_t \mid y_1, y_2, \ldots y_{t-1}, x) = g(y_{t-1}, s_t, c_i)$$

5.4.2.8 Pretrained Models

5.4.2.8.1 Transformers

a model that uses attention to boost the speed with which these models can be trained. The Transformers outperforms the Google Neural Machine Translation model in specific tasks. The biggest benefit, however, comes from how The Transformer lends itself to parallelization.

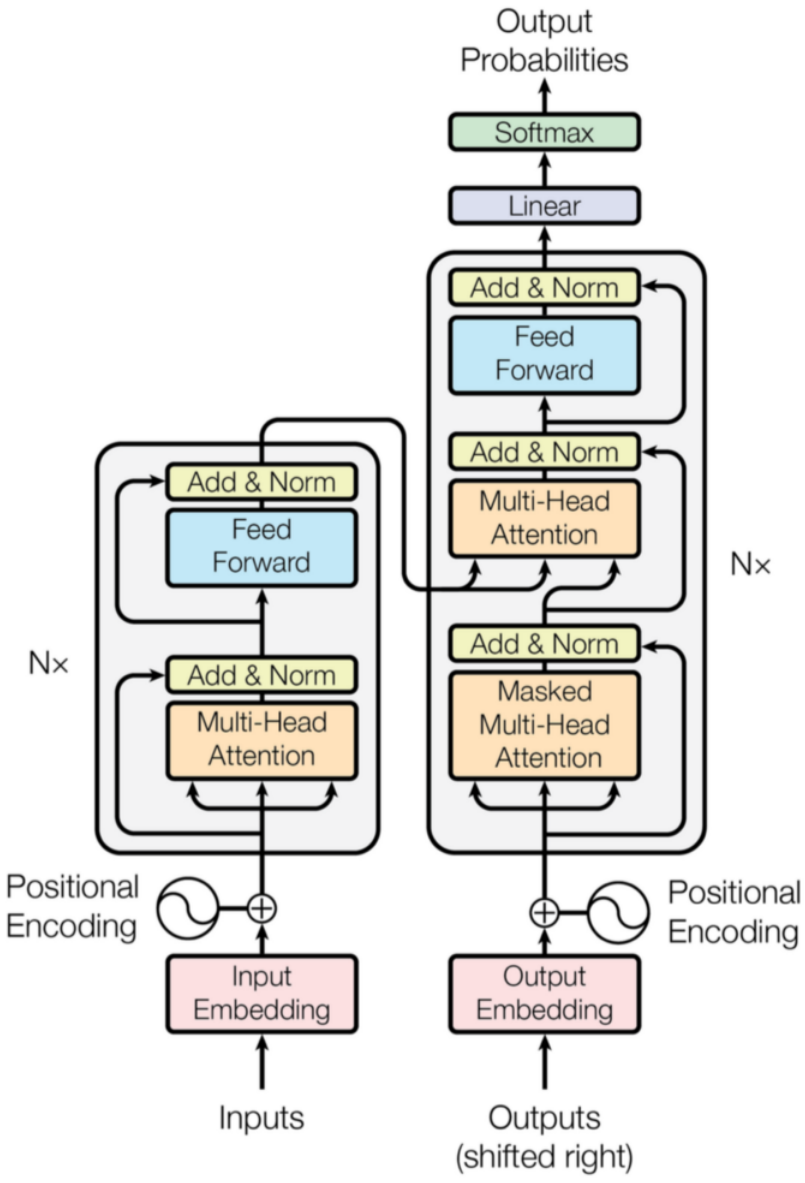


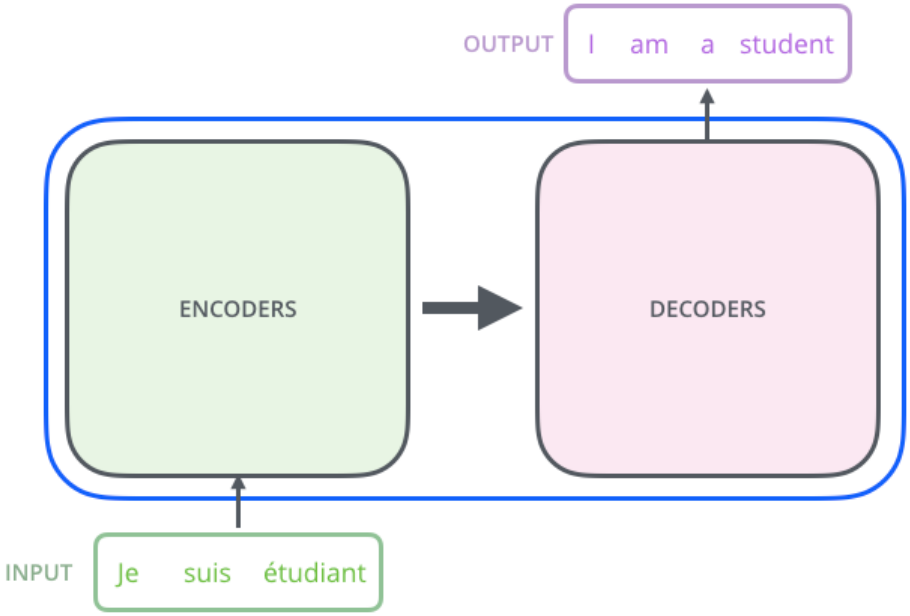
Figure 1: The Transformer - model architecture.

Let’s discuss the structure of transformer:

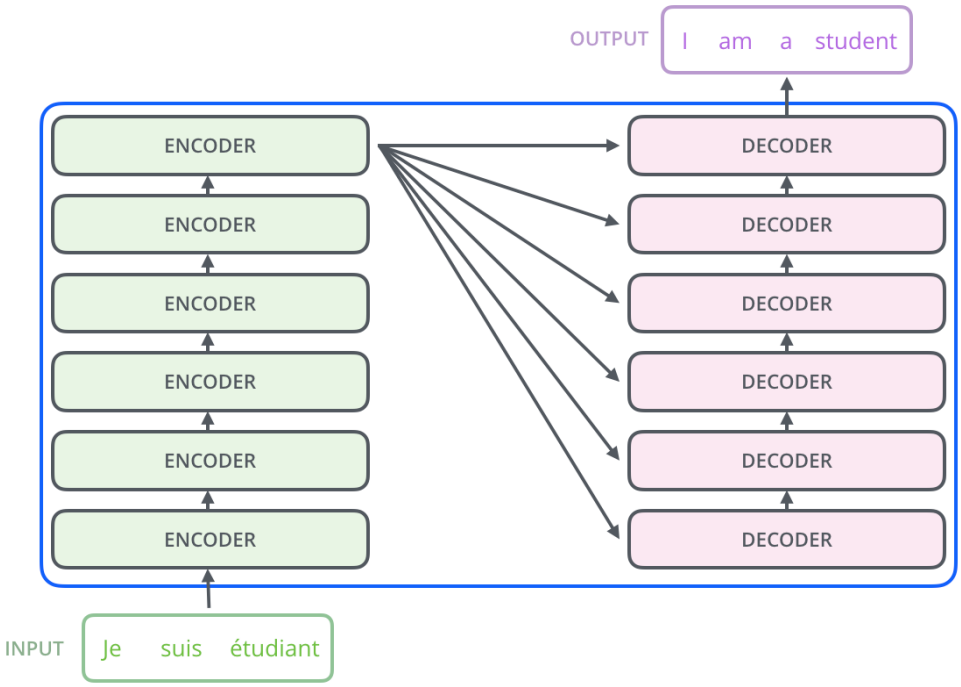
A High-Level Look

Let’s begin by looking at the model as a single black box. In a machine translation application, it would take a sentence in one language, and output its translation in another.

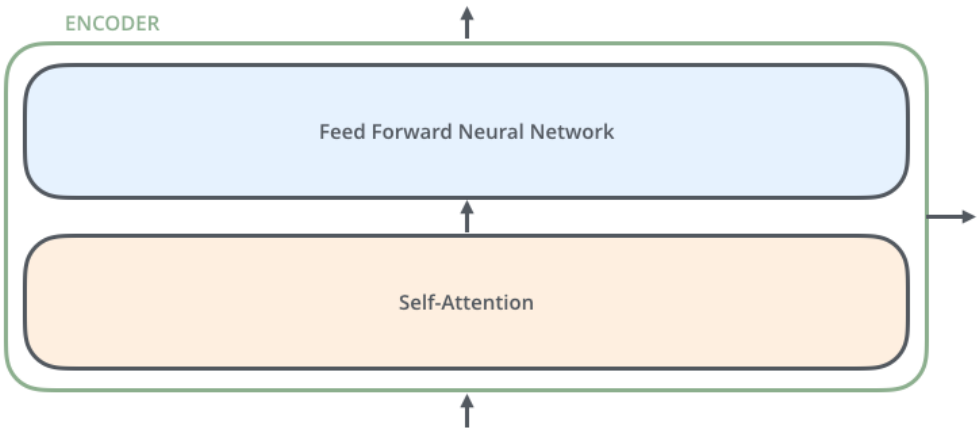
Popping open that Optimus Prime goodness, we see an encoding component, a decoding component, and connections between them.



The encoding component is a stack of encoders (the paper stacks six of them on top of each other – there’s nothing magical about the number six, one can definitely experiment with other arrangements). The decoding component is a stack of decoders of the same number.



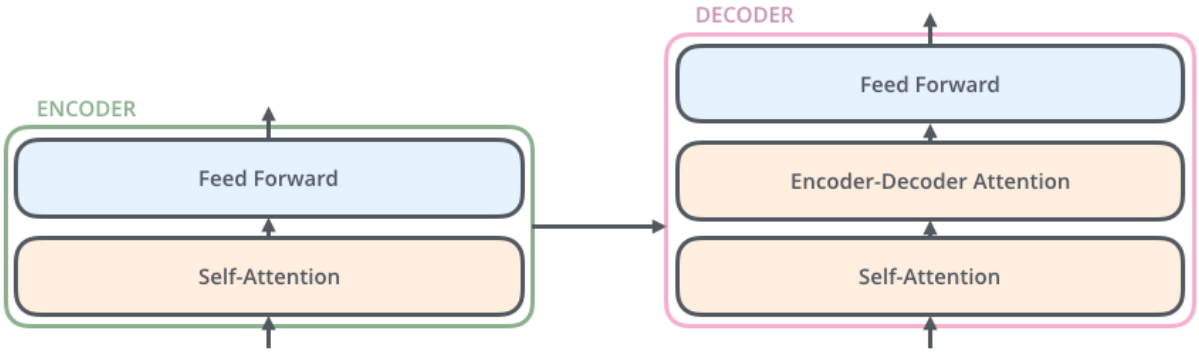
The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:



The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. We will look closer at self-attention later in the post.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

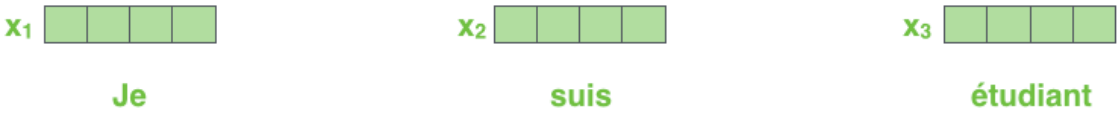
The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in seq2seq models).



Bringing The Tensors Into The Picture

Now that we've seen the major components of the model, let's start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output.

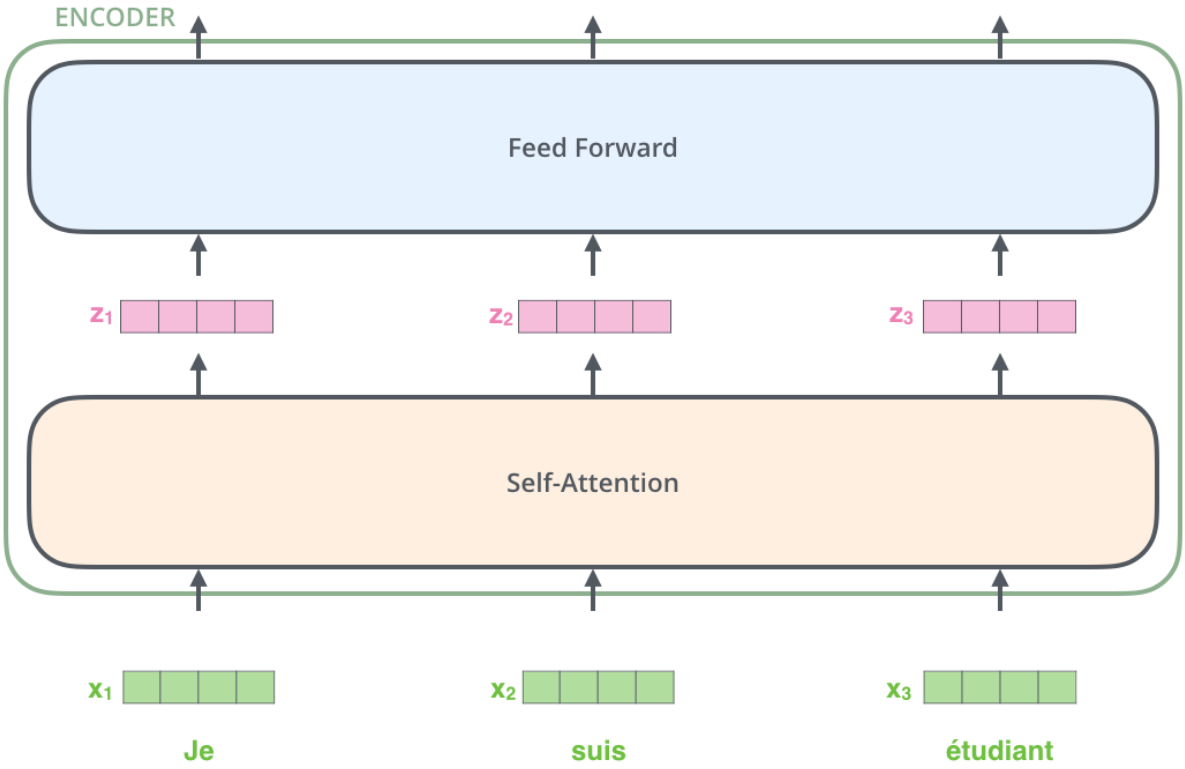
As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm.



Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

The embedding only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512 – In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. The size of this list is hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.

After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.

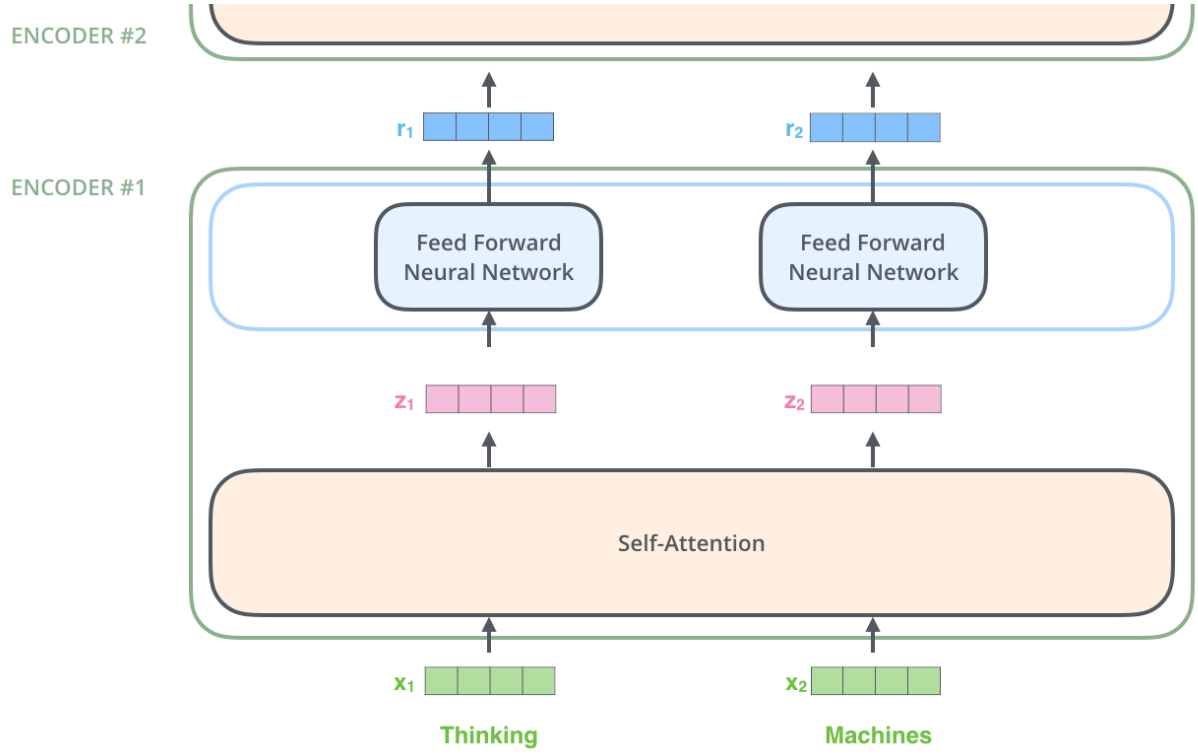


Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer. The feed-forward layer does not have those dependencies; however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.

Next, we will switch up the example to a shorter sentence and we'll look at what happens in each sub-layer of the encoder.

Now we are Encoding!

As we have mentioned already, an encoder receives a list of vectors as input. It processes this list by passing these vectors into a ‘self-attention’ layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.



The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network – the exact same network with each vector flowing through it separately.

Self-Attention at a High Level

Don’t be fooled by me throwing around the word “self-attention” like it’s a concept everyone should be familiar with. I had personally never come across the concept until reading the Attention is All You Need paper. Let us distill how it works.

Say the following sentence is an input sentence we want to translate:

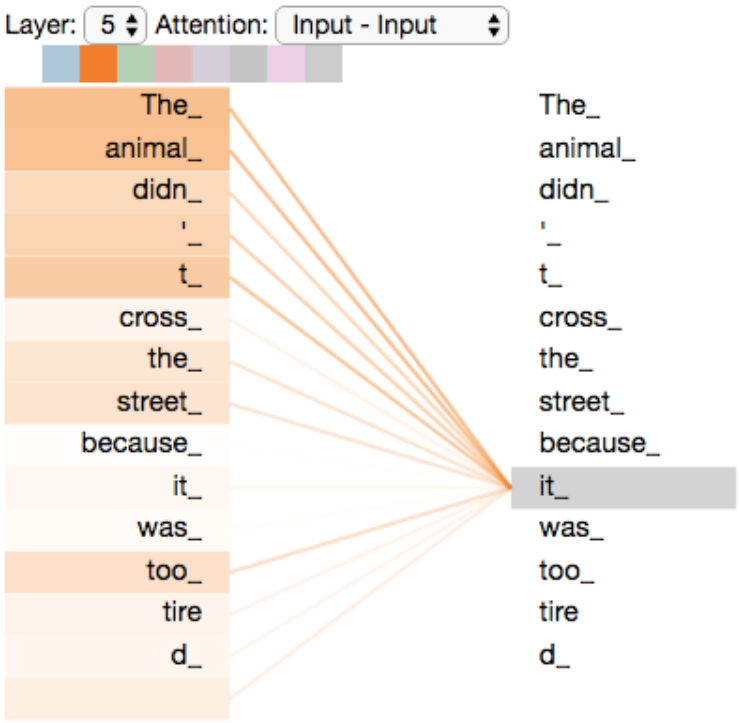
” *The animal didn’t cross the street because it was too tired.*”

What does “it” in this sentence refer to? Is it referring to the street or to the animal? It’s a simple question to a human, but not as simple to an algorithm.

When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.

As the model processes each word (each position in the input sequence), self-attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

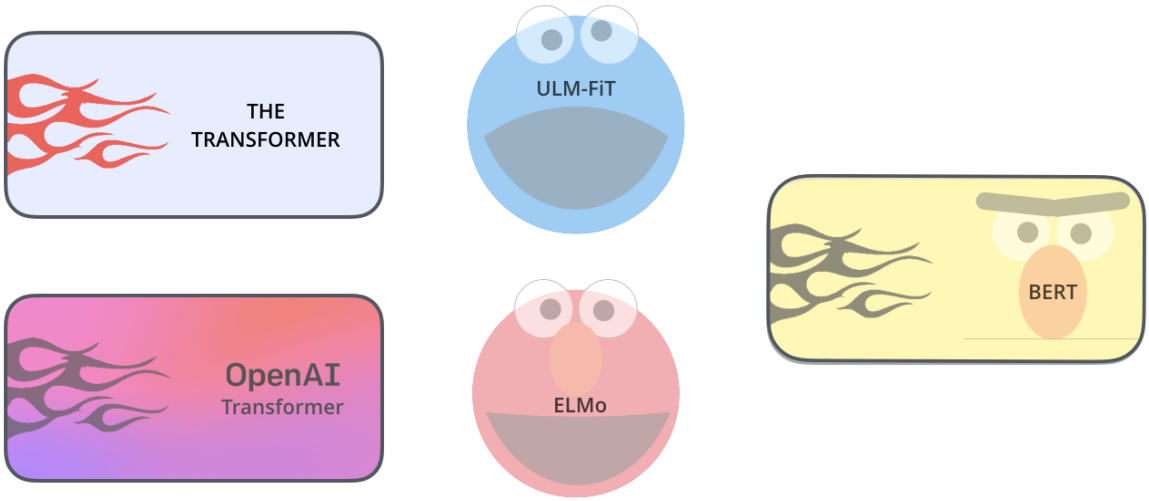
If you are familiar with RNNs, think of how maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it’s processing. Self-attention is the method the Transformer uses to bake the “understanding” of other relevant words into the one we’re currently processing.



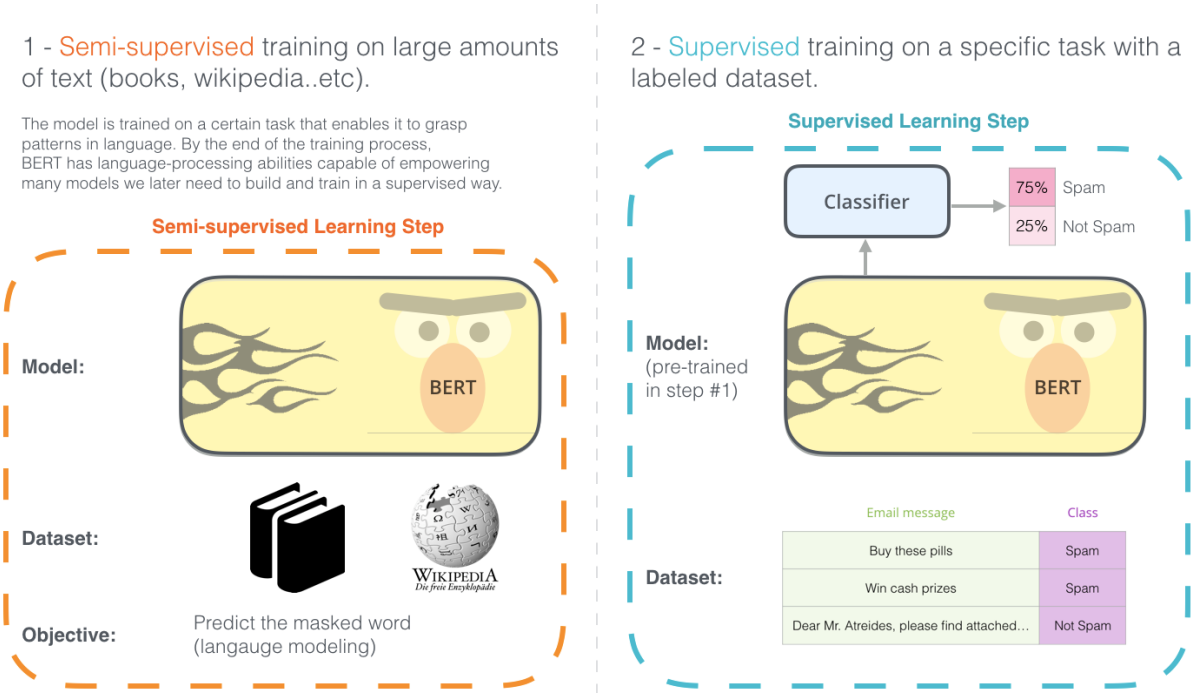
As we are encoding the word “it” in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on “The Animal” and baked a part of its representation into the encoding of “it”.

5.4.2.8.1.1 BERT

The year 2018 has been an inflection point for machine learning models handling text (or more accurately, Natural Language Processing or NLP for short). Our conceptual understanding of how best to represent words and sentences in a way that best captures underlying meanings and relationships is rapidly evolving.



One of the latest milestones in this development is the release of BERT, an event described as marking the beginning of a new era in NLP. BERT is a model that broke several records for how well models can handle language-based tasks. Soon after the release of the paper describing the model, the team also open-sourced the code of the model and made available for download versions of the model that were already pre-trained on massive datasets. This is a momentous development since it enables anyone building a machine learning model involving language processing to use this powerhouse as a readily-available component – saving the time, energy, knowledge, and resources that would have gone to training a language-processing model from scratch.



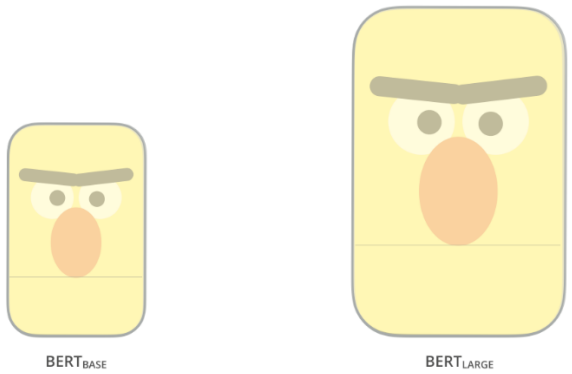
The two steps of how BERT is developed. You can download the model pre-trained in step 1 (trained on un-annotated data), and only worry about fine-tuning it for step 2.

BERT builds on top of a number of clever ideas that have been bubbling up in the NLP community recently – including but not limited to Semi-supervised Sequence Learning (by Andrew Dai and Quoc Le), ELMo (by Matthew Peters and researchers from AI2 and UW CSE), ULMFiT (by fast.ai founder Jeremy Howard and Sebastian Ruder), the OpenAI transformer (by OpenAI researchers Radford, Narasimhan, Salimans, and Sutskever), and the Transformer (Vaswani et al).

There are a number of concepts one needs to be aware of to properly wrap one's head around what BERT is. So, let's start by looking at ways you can use BERT before looking at the concepts involved in the model itself.

Model Architecture

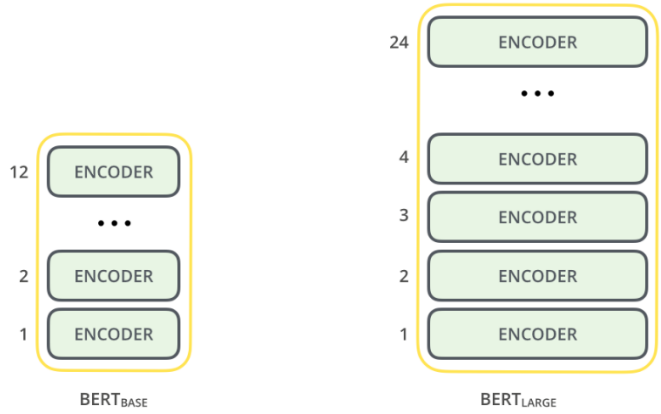
let us take a closer look at how it works.



The paper presents two model sizes for BERT:

- BERT BASE – Comparable in size to the OpenAI Transformer in order to compare performance.
- BERT LARGE – A ridiculously huge model which achieved the state-of-the-art results reported in the paper.

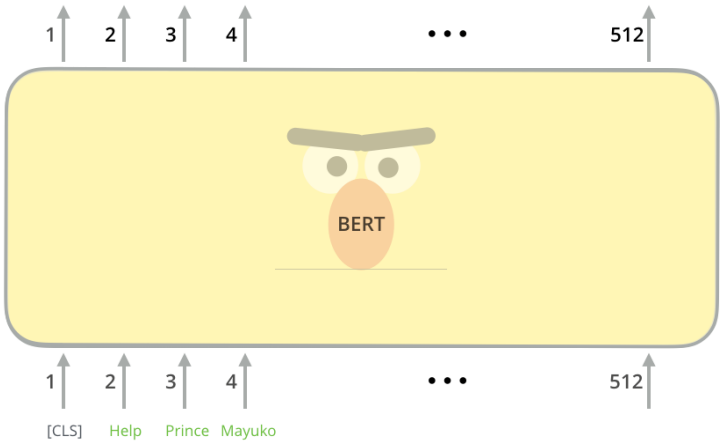
BERT is basically a trained Transformer Encoder stack



Both BERT model sizes have a large number of encoder layers (which the paper calls Transformer Blocks) – twelve for the Base version, and twenty-four for the large version. These also have larger feedforward-networks (768 and 1024 hidden units respectively), and more attention heads (12 and 16 respectively) than the

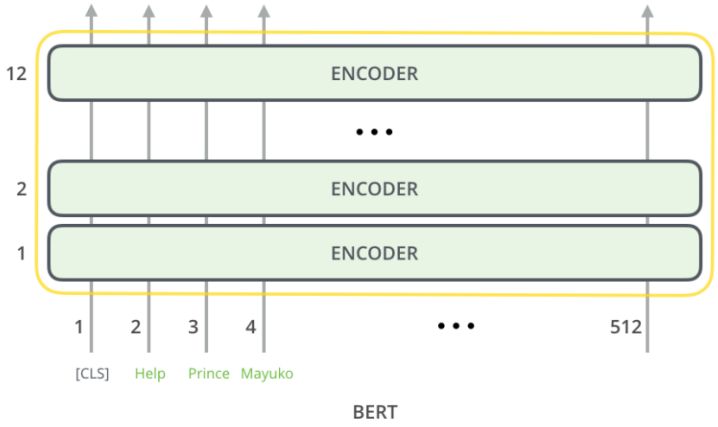
default configuration in the reference implementation of the Transformer in the initial paper (6 encoder layers, 512 hidden units, and 8 attention heads).

Model Inputs



The first input token is supplied with a special [CLS] token for reasons that will become apparent later. CLS here stands for Classification.

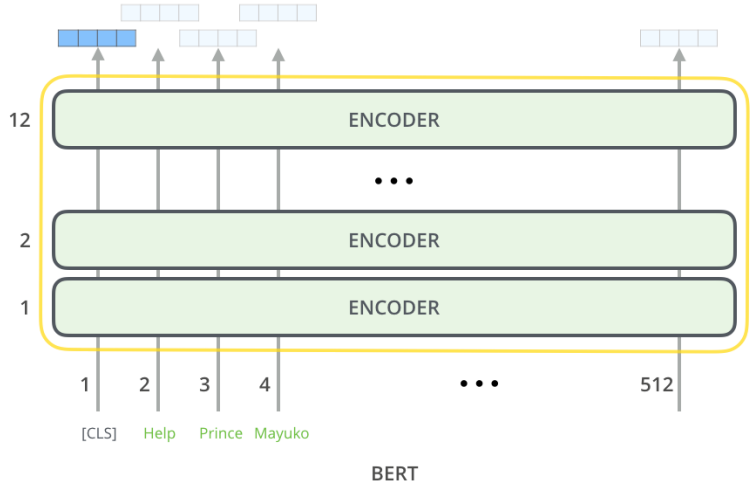
Just like the vanilla encoder of the transformer, BERT takes a sequence of words as input which keep flowing up the stack. Each layer applies self-attention and passes its results through a feed-forward network, and then hands it off to the next encoder.



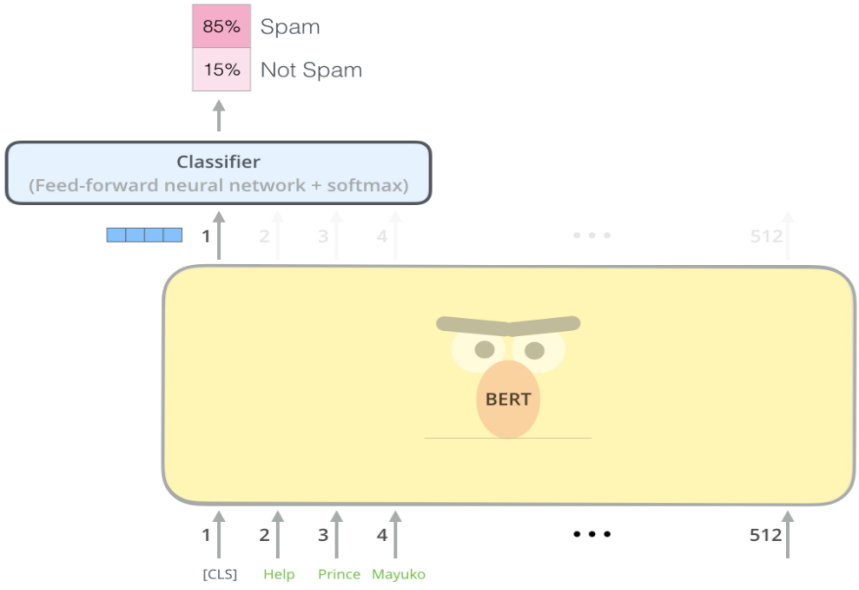
In terms of architecture, this has been identical to the Transformer up until this point (aside from size, which are just configurations we can set). It is at the output that we first start seeing how things diverge.

Model Outputs

Each position outputs a vector of size hidden_size (768 in BERT Base). For the sentence classification example, we've looked at above, we focus on the output of only the first position (that we passed the special [CLS] token to).



That vector can now be used as the input for a classifier of our choosing. The paper achieves great results by just using a single-layer neural network as the classifier.



If you have more labels (for example if you're an email service that tags emails with "spam", "not spam", "social", and "promotion"), you just tweak the classifier network to have more output neurons that then pass through softmax.

When can we use it and how to fine-tune it?

BERT outperformed the state-of-the-art across a wide variety of tasks under general language understanding like natural language inference, sentiment analysis, question answering, paraphrase detection and linguistic acceptability.

Now, how can we fine-tune it for a specific task? BERT can be used for a wide variety of language tasks. If we want to fine-tune the original model based on our own dataset, we can do so by just adding a single layer on top of the core model.

For example, say we are creating a question answering application. In essence question answering is just a prediction task — on receiving a question as input, the goal of the application is to identify the right answer from some corpus. So, given a question and a context paragraph, the model predicts a start and an end token from the paragraph that most likely answers the question. This means that using BERT a model for our application can be trained by learning two extra vectors that mark the beginning and the end of the answer.

- Input Question:

```
Where do water droplets collide with ice
crystals to form precipitation?
```

- Input Paragraph:

```
...  Precipitation forms as smaller droplets
coalesce via collision with other rain drops
or ice crystals within a cloud.  ...
```

- Output Answer:

```
within a cloud
```

Just like sentence pair tasks, the question becomes the first sentence and paragraph the second sentence in the input sequence. However, this time there are two new parameters learned during fine-tuning: a start vector and an end vector.

In the fine-tuning training, most hyper-parameters stay the same as in BERT training; the paper gives specific guidance on the hyper-parameters that require tuning.

Note that in case we want to do fine-tuning, we need to transform our input into the specific format that was used for pre-training the core BERT models, e.g., we would need to add special tokens to mark the beginning ([CLS]) and separation/end of sentences ([SEP]) and segment IDs used to distinguish different sentences — convert the data into features that BERT uses.

BERT: From Decoders to Encoders

The openAI transformer gave us a fine-tunable pre-trained model based on the Transformer. But something went missing in this transition from LSTMs to Transformers. ELMo's language model was bi-directional, but the openAI transformer only trains a forward language model. Could we build a transformer-based model whose language model looks both forward and backwards (in the technical jargon – "is conditioned on both left and right context")?

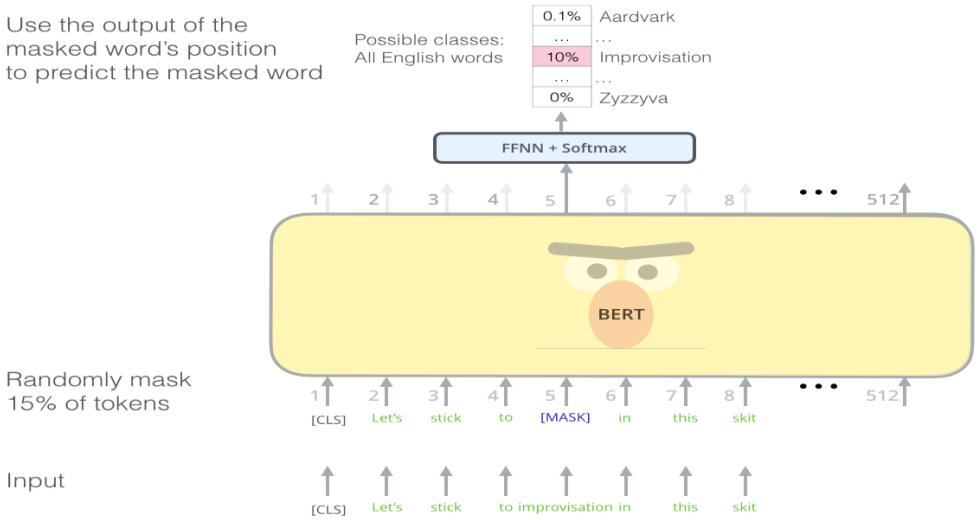
"Hold my beer", said R-rated BERT.

Masked Language Model

"We'll use transformer encoders", said BERT.

"This is madness", replied to Ernie, "Everybody knows bidirectional conditioning would allow each word to indirectly see itself in a multi-layered context."

"We'll use masks", said BERT confidently.



BERT's clever language modeling task masks 15% of words in the input and asks the model to predict the missing word.

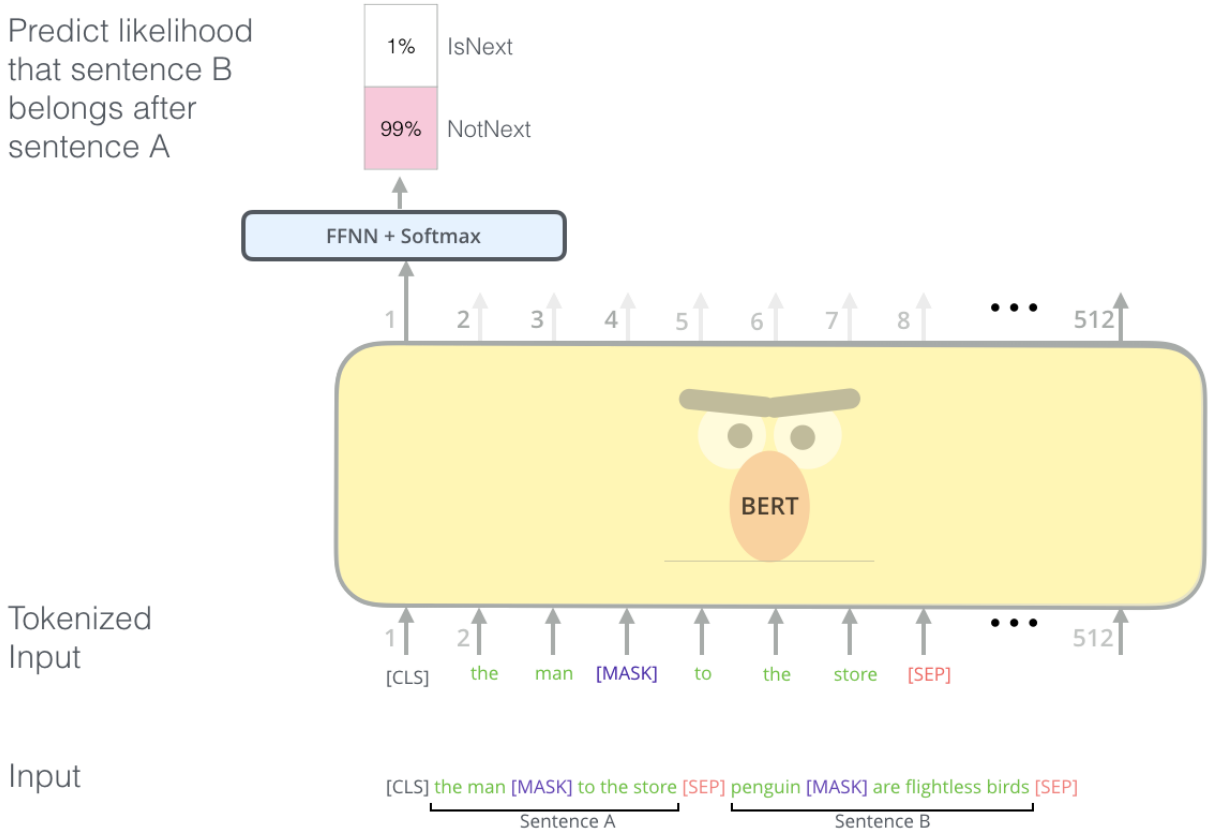
Finding the right task to train a Transformer stack of encoders is a complex hurdle that BERT resolves by adopting a "masked language model" concept from earlier literature (where it's called a Cloze task).

Beyond masking 15% of the input, BERT also mixes things a bit in order to improve how the model later fine-tunes. Sometimes it randomly replaces a word with another word and asks the model to predict the correct word in that position.

Two-sentence Tasks

If you look back up at the input transformations the OpenAI transformer does to handle different tasks, you'll notice that some tasks require the model to say something intelligent about two sentences (e.g. are they simply paraphrased versions of each other? Given a wikipedia entry as input, and a question regarding that entry as another input, can we answer that question?).

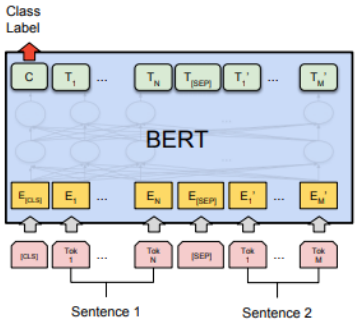
To make BERT better at handling relationships between multiple sentences, the pre-training process includes an additional task: Given two sentences (A and B), is B likely to be the sentence that follows A, or not?



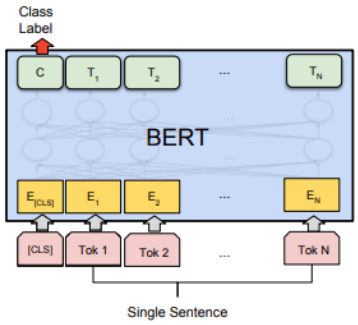
The second task BERT is pre-trained on is a two-sentence classification task. The tokenization is oversimplified in this graphic as BERT actually uses WordPieces as tokens rather than words --- so some words are broken down into smaller chunks.

Task specific-Models

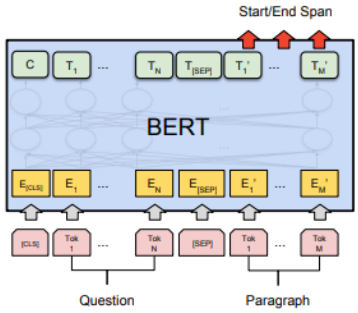
The BERT paper shows a number of ways to use BERT for different tasks.



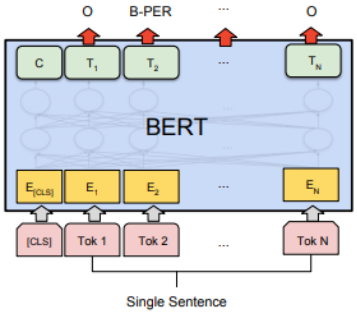
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tasks: SST-2, CoLA



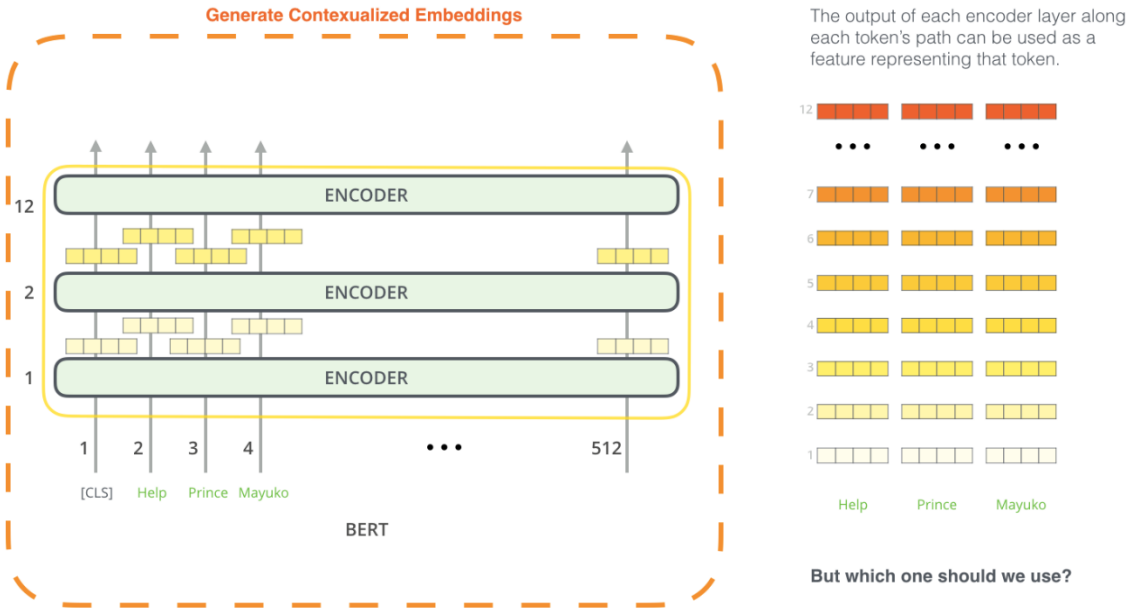
(c) Question Answering Tasks: SQuAD v1.1



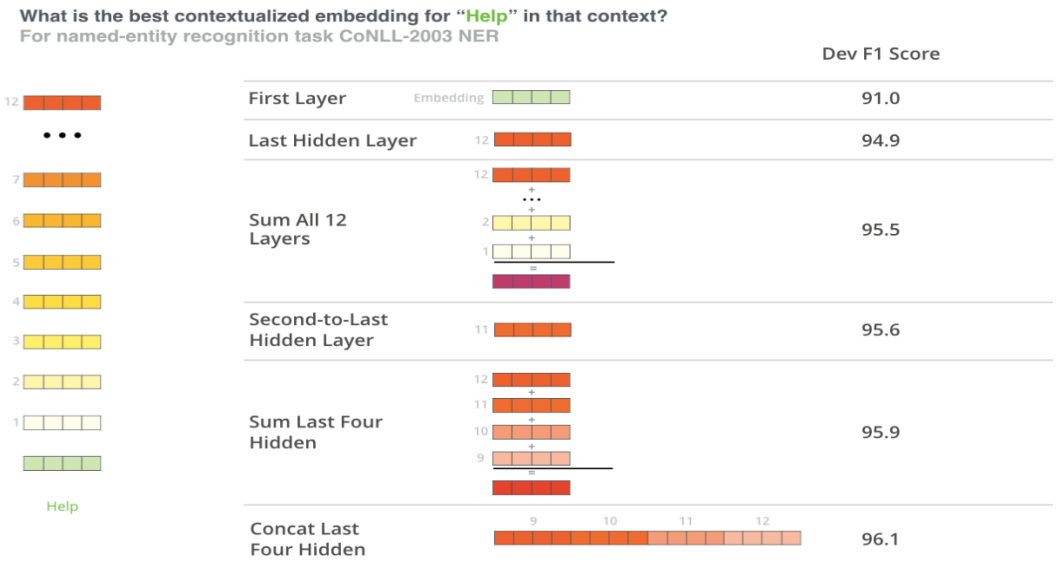
(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

BERT for feature extraction

The fine-tuning approach isn't the only way to use BERT. Just like ELMo, you can use the pre-trained BERT to create contextualized word embeddings. Then you can feed these embeddings to your existing model – a process the paper shows yield results not far behind fine-tuning BERT on a task such as named-entity recognition.



Which vector works best as a contextualized embedding? I would think it depends on the task. The paper examines six choices (Compared to the fine-tuned model which achieved a score of 96.4):



Take BERT out for a spin!

The best way to try out BERT is through the BERT FineTuning with Cloud TPUs notebook hosted on Google Colab. If you've never used Cloud TPUs before, this is also a good starting point to try them as well as the BERT code works on TPUs, CPUs and GPUs as well.

The next step would be to look at the code in the BERT repo:

- The model is constructed in modeling.py (class BertModel) and is pretty much identical to a vanilla Transformer encoder.
- run_classifier.py is an example of the fine-tuning process. It also constructs the classification layer for the supervised model. If you want to construct your own classifier, check out the create_model() method in that file.
- Several pre-trained models are available for download. These span BERT Base and BERT Large, as well as languages such as English, Chinese, and a multi-lingual model covering 102 languages trained on wikipedia.
- BERT doesn't look at words as tokens. Rather, it looks at WordPieces. tokenization.py is the tokenizer that would turns your words into wordPieces appropriate for BERT.

You can also check out the PyTorch implementation of BERT.
The AllenNLP library uses this implementation to allow using BERT embeddings with any model

5.4.2.8.1.2 DistilBERT

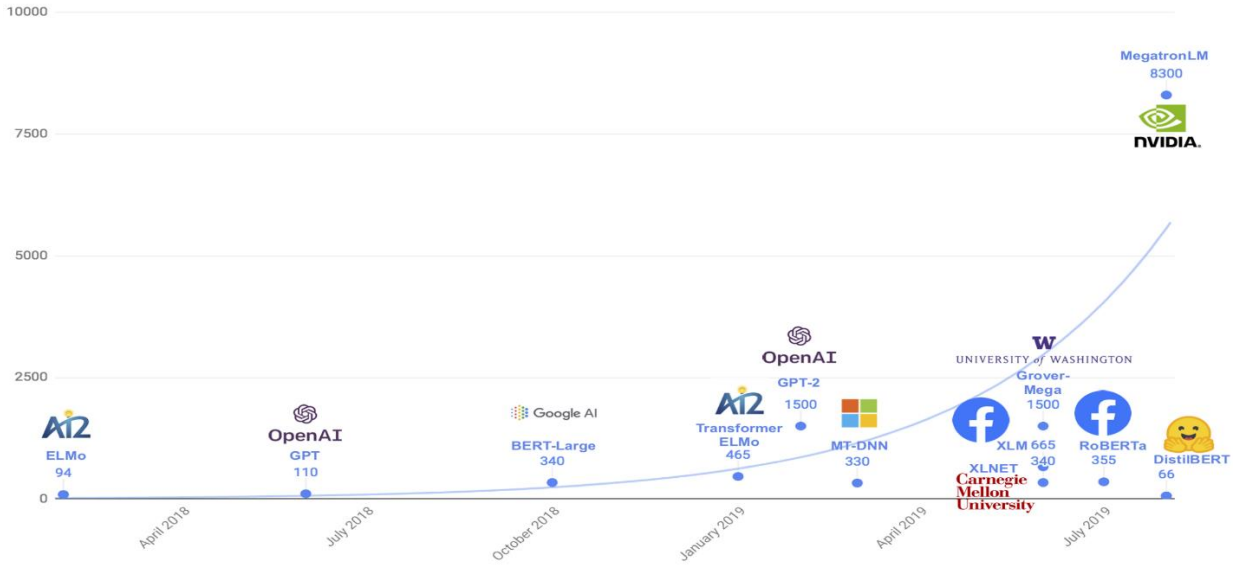
The DistilBERT model was proposed in the blog post Smaller, faster, cheaper, lighter: Introducing DistilBERT, a distilled version of BERT, and the paper DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. DistilBERT is a small, fast, cheap and light Transformer model trained by distilling BERT base. It has 40% less parameters than bert-base-uncased, runs 60% faster while preserving over 95% of BERT's performances as measured on the GLUE language understanding benchmark.

As Transfer Learning from large-scale pre-trained models becomes more prevalent in Natural Language Processing (NLP), operating these large models in on-the-edge and/or under constrained computational training or inference budgets remains challenging. In this work, we propose a method to pre-train a smaller general-purpose language representation model, called DistilBERT, which can then be fine-tuned with good performances on a wide range of tasks like its larger counterparts. While most prior work investigated the use of distillation for building task-specific models, we leverage knowledge distillation during the pre-training phase and show that it is possible to reduce the size of a BERT model by 40%, while retaining 97% of its language understanding capabilities and being 60% faster. To leverage the inductive biases learned by larger models during pre-training, we introduce a triple loss combining language modeling, distillation and cosine-distance losses. Our smaller, faster and lighter model is cheaper to pre-train and we demonstrate its capabilities for on-device computations in a proof-of-concept experiment and a comparative on-device study.

The last two years have seen the rise of Transfer Learning approaches in Natural Language Processing (NLP) with large-scale pre-trained language models becoming a basic tool in many NLP tasks [Devlin et al., 2018, Radford et al., 2019, Liu et al., 2019]. While these models lead to significant improvement, they often have several hundred million parameters and current research1 on pre-trained models indicates that training even larger models still leads to better performances on downstream tasks.

The trend toward bigger models raises several concerns. First is the environmental cost of exponentially scaling these models' computational requirements as mentioned in Schwartz et al. [2019], Strubell et al. [2019]. Second, while operating these models on-device in real-time has the potential to enable novel and

interesting language processing applications, the growing computational and memory requirements of these models may hamper wide adoption.



Student architecture In the present work, the student – DistilBERT – has the same general architecture as BERT. The *token-type embeddings* and the *pooler* are removed while the number of layers is reduced by a factor of 2. Most of the operations used in the Transformer architecture (*linear layer* and *layer normalisation*) are highly optimized in modern linear algebra frameworks and our investigations showed that variations on the last dimension of the tensor (hidden size dimension) have a smaller impact on computation efficiency (for a fixed parameters budget) than variations on other factors like the number of layers. Thus we focus on reducing the number of layers.

Student initialization In addition to the previously described optimization and architectural choices, an important element in our training procedure is to find the right initialization for the sub-network to converge. Taking advantage of the common dimensionality between teacher and student networks, we initialize the student from the teacher by taking one layer out of two.

Distillation We applied best practices for training BERT model recently proposed in Liu et al. (2019). As such, DistilBERT is distilled on very large batches leveraging gradient accumulation (up to 4K examples per batch) using dynamic masking and without the next sentence prediction objective.

Data and compute power We train DistilBERT on the same corpus as the original BERT model: a concatenation of English Wikipedia and Toronto Book Corpus (Zhu et al., 2015). DistilBERT was trained on 8 16GB V100 GPUs for approximately 90 hours. For the sake of comparison, the RoBERTa model (Liu et al., 2019) required 1 day of training on 1024 32GB V100.

EXPERIMENTS

General Language Understanding We assess the language understanding and generalization capabilities of DistilBERT on the General Language Understanding Evaluation (GLUE) benchmark (Wang et al., 2018), a collection of 9 datasets for evaluating natural language understanding systems. We report scores on the development sets for each task by fine-tuning DistilBERT without the use of ensembling or multi-tasking scheme for fine-tuning (which are mostly orthogonal to the present work). We compare the results to the baseline provided by the authors of GLUE: an ELMo (Peters et al. (2018)) encoder followed by two BiLSTMs.

The results on each of the 9 tasks are showed on Table 1 along with the macro-score (average of individual scores). Among the 9 tasks, DistilBERT is always on par or improving over the ELMo baseline (up to 20 points of accuracy on STS-B). DistilBERT also compares surprisingly well to BERT, retaining 97% of the performance with 40% fewer parameters.

Model	Score	CoLA	MNLI	MRPC	QNLI	QQP	RTE	SST-2	STS-B	WNLI
ELMo	68.7	44.1	68.6	76.6	71.1	86.2	53.4	91.5	70.4	56.3
BERT-base	77.6	48.9	84.3	88.6	89.3	89.5	71.3	91.7	91.2	43.7
DistilBERT	76.8	49.1	81.8	90.2	90.2	89.2	62.9	92.7	90.7	44.4

DistilBERT retains 97% of BERT performance. Comparison on the dev sets of the GLUE benchmark. ELMo results as reported by the authors. BERT and DistilBERT results are the medians of 5 runs with different seeds.

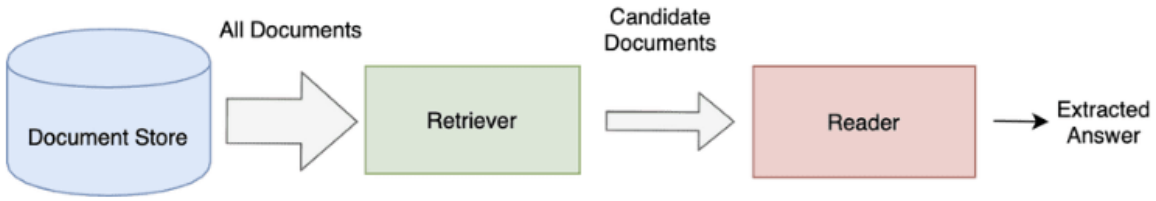
In the next chapter we will discuss DistilBERT in details because we have been used this model in our project.

6 The Retrieval

Goal

Given a Question we want to extract its answer from a huge number of documents.

6.1 System pipeline



Let us talk briefly about each component

6.1.1 Document store

You can think of the Document Store as a "database":

- That stores the answers or passages.
- Provides them to the retriever at query time.

6.1.2 Readers

Are powerful models that do close analysis of documents and perform the core task of question answering (It extracts short answers from each document and gives a score on how relevant this answer is to the question) . However, it is not currently feasible to use the Reader directly on a large collection of documents.

6.1.3 The Retriever

Is a lightweight filter that can quickly go through the full document store and pass on a set of candidate documents that are relevant to the query. When used in combination with a Reader, it is a tool for sifting out the obvious negative cases, saving the Reader from doing more work than it needs to and speeding up the querying process.

The Retriever directly affects the accuracy of our QA system, better retriever results in Better QA system.

6.2 Retriever Types

6.2.1 Sparse Retrievers

At first, search engines (Google, Bing, DuckDuckGo, Yahoo, etc.) were lexical: the search engine looked for literal matches of the query words, without understanding of the query's meaning and only returning links that contained the exact query.

For example, if the user looked for "cis lmu", the homepage of the CIS centre from the LMU university matches the query, since:

- the CIS center's homepage contains these 2 words.
- the URL of the homepage contains these 2 words.
- the page is at the top level of the domain.
- and many other reasons specified by the search engine.

All these criteria are easy to check, and they alone make this page a very good candidate for the top hit of this query. No deeper understanding of what the query actually "meant" or what the homepage is actually "about" were needed.

In the past, we relied on sparse vector retrievers for the task of finding relevant information from our document stores. For this, we used TF-IDF or BM25.

Types of Sparse Retrieval

6.2.1.1 TF-IDF

The TF-IDF algorithm is a popular option for calculating the similarity of two pieces of text.

- **TF** refers to how many words in the query are found in the context.
- **IDF** is the inverse of the fraction of documents containing this word.

These two values are then multiplied to give the TF-IDF score. Now, we may find that the word *"hippocampus"* is shared between the query and context, this would increase the TF-IDF score because:

- **TF** — the word is found in both the query and the context (high score).
- **IDF** — the word *"hippocampus"* is *not* found in many other documents (so the *inverse* of the word frequency is a high number).

Alternatively, if we took the word *"the"*, we would return a low TF-IDF score because:

- **TF** — the word is found in both the query and the context (high score).
- **IDF** — the word *"the"* is found in many other documents (so the *inverse* of the word frequency is a *low* number).

Because **IDF** is a low number due to how common *the* is, the **TF-IDF** score is low too.

So, the TF-IDF score is great for finding sequences that contain the same uncommon words.

6.2.1.2 BM25

BM25 is a variation of TF-IDF. Here, we still calculate TF and IDF, but the TF score is dampened after returning large numbers of matches between the query and contexts.

Additionally, it also considers the document length. The TF-IDF score is normalized so that short documents will score better than long documents given they both have the same number of word matches.

When using sparse retrievers, BM25 is typically favored over TF-IDF.

6.2.2 Dense Retrieval

6.2.2.1 Dense Passage Retrieval (DPR)

For ODQA was introduced in 2020 as an alternative to the traditional TF-IDF and BM25 techniques for passage retrieval.

Pros

The paper that introduced DPR begins by stating that this new approach outperforms current Lucene (the document store) BM25 retrievers by a 9–19% passage retrieval accuracy.

DPR can outperform the traditional sparse retrieval methods for two key reasons:

- Semantically similar words (*"hey", "hello", "hey"*) will not be viewed as a match by *TF*. DPR uses dense vectors encoded with semantic meaning (*so "hey", "hello", and "hey" will closely match*).
- Sparse retrievers are **not** trainable. DPR uses embedding functions that we can train and fine-tune for specific tasks.

Cons

Despite these clear performance benefits, it is not all good news. Yes, we can train our DPR model, but that's also a disadvantage — whereas TF-IDF and BM25 come ready-to-go — DPR does not.

As is usually the case in ML, DPR requires a lot of training data — which in this case is a curated dataset of question and context pairs.

Additionally, DPR requires significantly more compute — during both indexing and retrieval.

6.2.3 Dense vs Sparse

Broadly speaking, retrieval methods can be split into two categories: dense and sparse.

Sparse methods, like TF-IDF and BM25, operate by looking for shared keywords between the document and query. They are:

- simple but effective
- don’t need to be trained
- work on any language

More recently, dense approaches such as Dense Passage Retrieval (DPR) have shown even better performance than their sparse counterparts. These methods embed both document and query into a shared embedding space using deep neural networks and the top candidates are the nearest neighbour documents to the query. They are:

- powerful but computationally more expensive especially during indexing
- trained using labelled datasets
- language specific

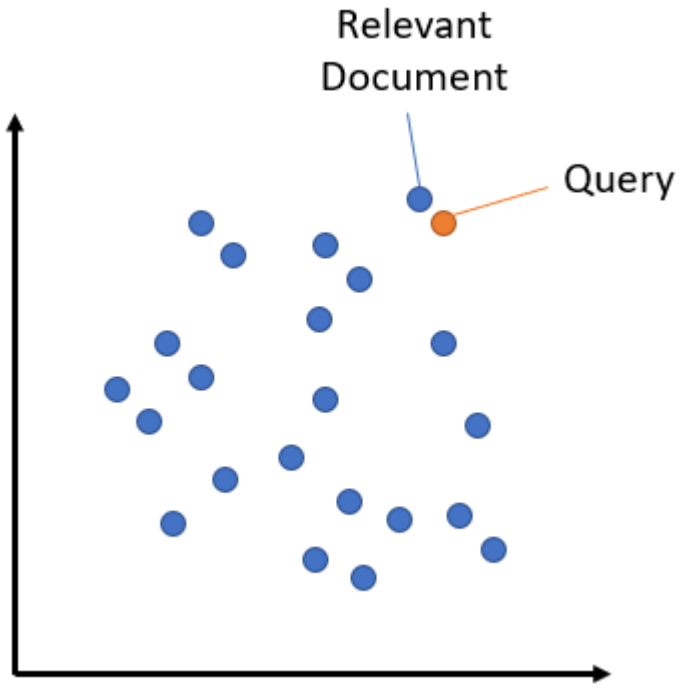
6.3 Semantic Search

Semantic search seeks to improve search accuracy by understanding the content of the search query. In contrast to traditional search engines which only find documents based on lexical matches, semantic search can also find synonyms.

Background

The idea behind semantic search is to embed all entries in your corpus, whether they be sentences, paragraphs, or documents, into a vector space.

At search time, the query is embedded into the same vector space and the closest embeddings from your corpus are found. These entries should have a high semantic overlap with the query.



6.3.1 Symmetric Semantic Search vs. Asymmetric Semantic Search

A critical distinction for your setup is *symmetric* vs. *asymmetric semantic search*:

- For **symmetric semantic search**, your query and the entries in your corpus are of about the same length and have the same amount of content. An example would be searching for similar questions: Your query could for example be *"How to learn Python online?"* and you want to find an entry like *"How to learn Python on the web?"*. For symmetric tasks, you could potentially flip the query and the entries in your corpus.
- For **asymmetric semantic search**, you usually have a **short query** (like a question or some keywords) and you want to find a longer paragraph answering the query. An example would be a query like *"What is Python"* and you want to find the paragraph *"Python is an interpreted, high-level and general-purpose programming language. Python's design philosophy …"*. For asymmetric tasks, flipping the query and the entries in your corpus usually does not make sense.

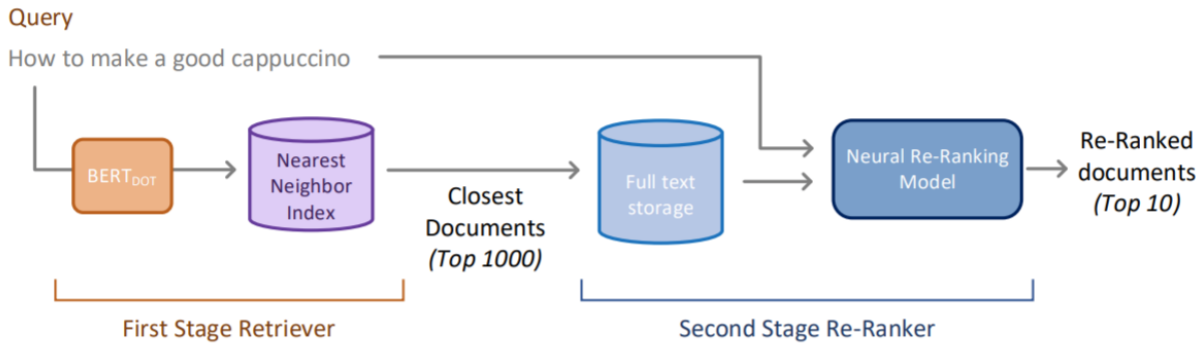
6.4 Dense Retrieval & Re-Ranking

Dense retrieval replaces the traditional first stage

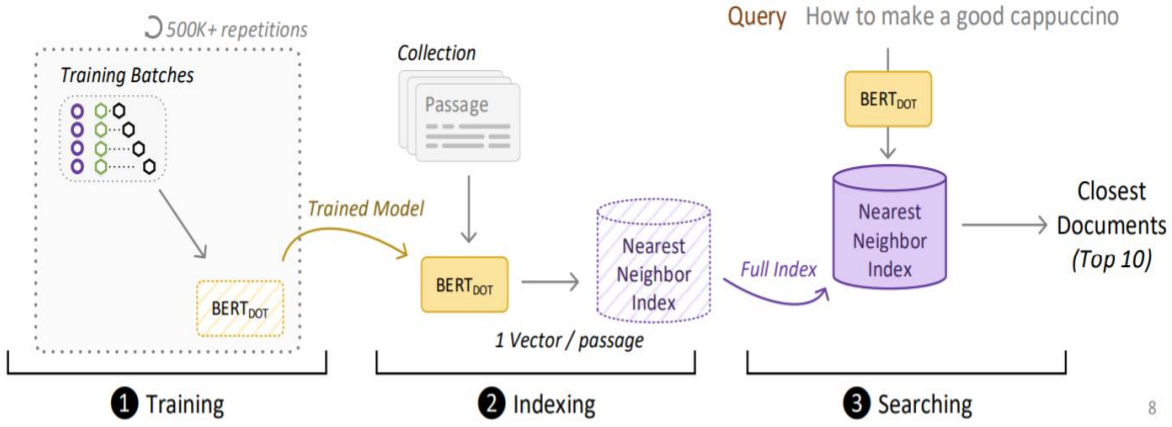
• Using a neural encoder & nearest neighbor vector index

• Can be used as part of a larger pipeline

6.4.1 Dense Retrieval Lifecycle



- 3 major phases in the dense retrieval lifecycle
- Each comes with several complex choices and requires techniques Could skip ❶ if we use a pre-trained model



6.4.2 Dense Passage Retrieval

• Passages and queries are compressed into a single vector

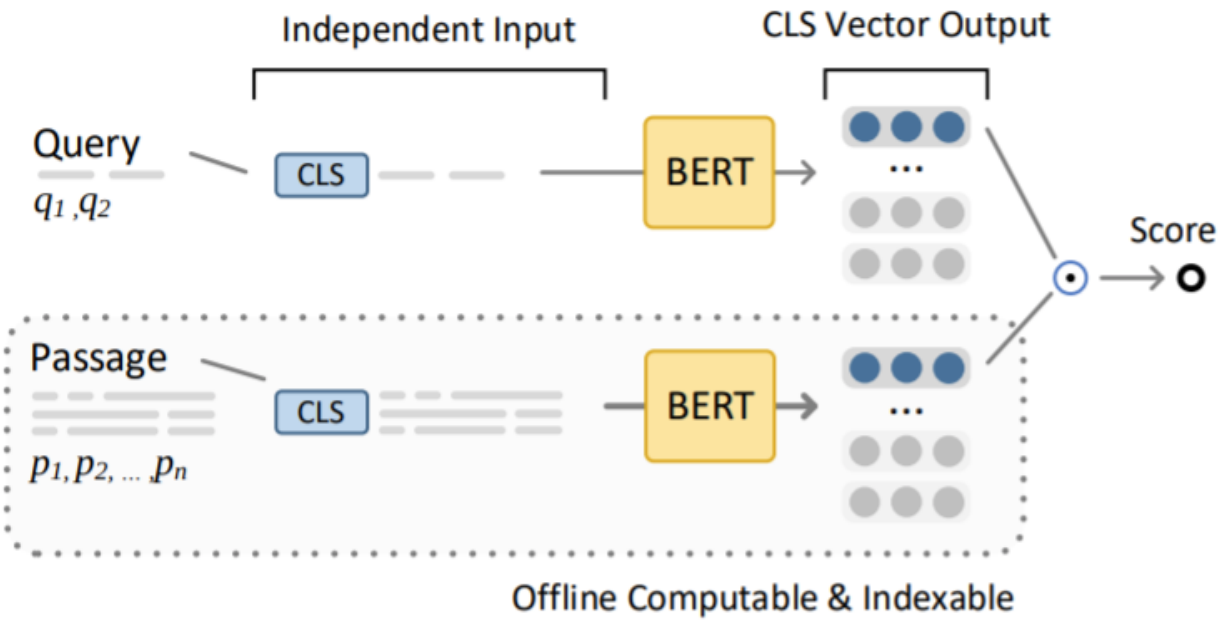
• Passages are completely independent -> moves most computation into the indexing phase

• Only need query encoding at runtime

• Relevance is scored with a dot-product

• Cosine-sim variants also exist

• This allows easy use of an (approximate) nearest neighbor index



The dot product value between the two model outputs EP(p) and EQ(q) measures the similarity between both vectors. A higher dot product correlates to a higher similarity — because the closer two vectors are to each other, the larger the dot product.

6.4.2.1 Training

the two models to output the same vector, we are training the context encoder and question encoder to output very similar vectors for related question-context pairs.

6.4.2.2 Runtime

Once the model (or two models) has been trained, we can begin using them for Q&A indexing and retrieval.

When we first build our document store, we need to encode the data we store in there using the E_P encoder — so during document store initialization (or when adding new documents) — we run every passage of text through the E_P encoder and store the output vectors in our document store.

For real-time Q&A, we only need the EQ encoder. When we ask a question, it will be sent to the EQ encoder which then outputs our EQ vector EQ(q).

Next, the EQ(q) vector is compared against the already indexed $E_P(p)$ vectors in our document store — where we filter for the vectors which return the highest similarity score:

$$sim(q,p) = EQ(q)^T E_P(p)$$

And that is it! Our retriever has identified the most relevant contexts for our question.

These relevant contexts are then passed onto our reader (or generator) model, which will create an answer based on the contexts — our Q&A process is complete!

6.5 Our model Training

6.5.1 Supervised dataset creation

Let us talk about our dataset, we created a Supervised QA dataset using the stack-overflow dataset.

For every question we took the top 3 answers and 3 negative random answers and constructed a 1.6M record in our supervised dataset.

6.5.2 Dataset Split

We Split the dataset into training set 97.5 and validation set 1.5% and Test set 1%.

```
In [255]: df = pd.read_pickle("../big_dataset.pkl")
          df = df.reset_index(drop=True)
          train_dataset,test_dataset = train_test_split(df, test_size=0.01,stratify = df["is_answer"])
          train_dataset,val_dataset = train_test_split(train_dataset, test_size=0.015,stratify = train_dataset["is_answer"])
          print(f"length of train: {len(train_dataset)}, length of validation: {len(val_dataset)}, length of test: {len(test_d
          del df #to reduce memory usage
          pd.to_pickle(train_dataset,"train_ds_big.pkl")
          pd.to_pickle(val_dataset,"val_ds_big.pkl")
          pd.to_pickle(test_dataset,"test_ds_big.pkl")

          length of train: 1626900, length of validation: 24776, length of test: 16684
```

6.5.3 Data loader

We created our data loader to provide us with batches

```
In [14]: class QADataset(Dataset):

             def __init__(self,df):

                 self.ques_title = df["ques_title"].to_numpy()
                 self.ques_body = df["ques_body"].to_numpy()
                 self.ans = df["ans"].to_numpy()
                 self.label = df["is_answer"].to_numpy()


             # support indexing such that dataset[i] can be used to get i-th sample
             def __getitem__(self, index):
                 return {
                     "ques_title":self.ques_title[index],
                     "ques_body":self.ques_body[index],
                     "ans":self.ans[index],
                     "label":self.label[index]
                 }

             # we can call len(dataset) to return the size
             def __len__(self):
                 return len(self.ans)
```

6.5.4 Bert encoder

We used destil-bert as encoder

```
class BertEncoder(nn.Module):
    def __init__(self):
        super(BertEncoder, self).__init__()
        self.bert = AutoModel.from_pretrained(model_name)


    def forward(self, input_ids, attention_mask):
        output = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        hidden_state = output[0]  # (bs, seq_len, dim)
        pooled_output = hidden_state[:, 0]  # (bs, dim)

        return pooled_output
```

6.5.5 Dense passage retrieval model

We define dense passage retrieval model

```
class DPR(nn.Module):
    def __init__(self,question_encoder,answer_encoder):
          super(DPR, self).__init__()
          self.question_encoder = question_encoder
          self.answer_encoder = answer_encoder

    def forward(self, ques_ids, ques_mask, ans_ids, ans_mask):
        ques_encoding = self.question_encoder(input_ids=ques_ids, attention_mask=ques_mask)
        ans_encoding = self.answer_encoder(input_ids=ans_ids, attention_mask=ans_mask)

        return ques_encoding,ans_encoding
```

6.5.6 loss function

We define our loss function we used negative log loss function:-

$$\text{sim}(q, p) = E_Q(q)^\intercal E_P(p). \tag{1}$$

Let $\mathcal{D} = \{\langle q_i, p_i^+, p_{i,1}^-, \cdots, p_{i,n}^- \rangle\}_{i=1}^m$ be the training data that consists of m instances. Each instance contains one question q_i and one relevant (positive) passage p_i^+, along with n irrelevant (negative) passages $p_{i,j}^-$. We optimize the loss function as the negative log likelihood of the positive passage:

$$\begin{aligned} & L(q_i, p_i^+, p_{i,1}^-, \cdots, p_{i,n}^-) \qquad (2) \\ = & -\log \frac{e^{\text{sim}(q_i, p_i^+)}}{e^{\text{sim}(q_i, p_i^+)} + \sum_{j=1}^n e^{\text{sim}(q_i, p_{i,j}^-)}}. \end{aligned}$$

6.5.7 In-batch Negative

We select only the question pairs that have answers to train the model.

We build two batches as input for the network, and we assume that question qi (question i in the first batch) is an answer of ai (answer i in the second batch), but all other answers in the second batch are not answers of qi .

The test set uses the original pairs of questions and the status describing if the answers are answers.

```
In [261]: train_dataset = train_dataset[train_dataset["is_answer"]==1]
          print(f"length of postive examples in train dataset: {len(train_dataset)}")
          val_dataset = val_dataset[val_dataset["is_answer"]==1]
          print(f"length of postive examples in validation dataset: {len(val_dataset)}")
          test_dataset = test_dataset[test_dataset["is_answer"]==1]
          print(f"length of postive examples in validation dataset: {len(test_dataset)}")

          length of postive examples in train dataset: 813450
          length of postive examples in validation dataset: 12388
          length of postive examples in validation dataset: 8342
```

Model Hyperparameters

```
[ ]   batch_size = 16
      adam_eps =  1e-8
      adam_betas =  (0.9, 0.999)
      max_grad_norm =  1.0
      small_batches_step = 100
      big_batches_step = 200
      weight_decay = 0.0
      learning_rate =  1e-5

      # Linear warmup over warmup_steps.
      warmup_steps = 1200 #prev 1000

      # Total number of training epochs to perform.
      num_train_epochs=  4

      #BERT
      ques_max_length=256
      ans_max_length= 256
```

6.5.8 Optimizer and linear scheduler with warm up

```
In [26]: optimizer = AdamW(model.parameters(), lr=learning_rate,weight_decay=weight_decay, eps=adam_eps,betas=adam_betas)

total_steps = len(train_data_loader) * num_train_epochs

scheduler = get_linear_schedule_with_warmup(
  optimizer,
  num_warmup_steps=warmup_steps,
  num_training_steps=total_steps
)
```

6.5.9 Model Evaluation

6.5.9.1 Fake task Accuracy

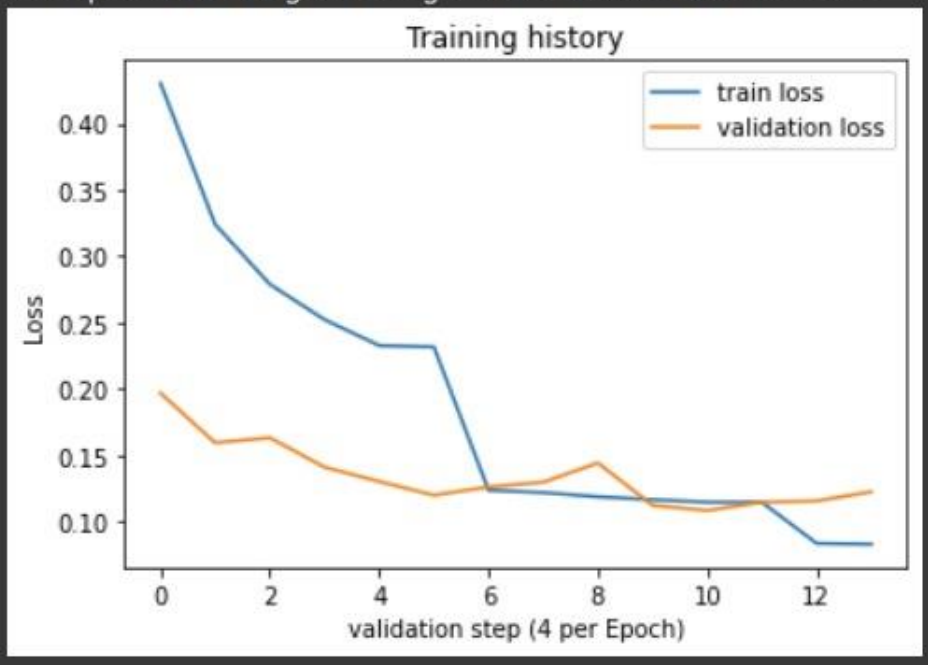
After 12 hours of training:-

```
[26]   plt.plot(history["train_acc"], label='train Accuracy')
       plt.plot(history["val_acc"], label='validation Accuracy')

       plt.title('Training history')
       plt.ylabel('Accuracy')
       plt.xlabel('validation step (4 per Epoch)')
       plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fa3a4c82550>
```

```
[27]   plt.plot(history["train_loss"], label='train loss')
       plt.plot(history["val_loss"], label='validation loss')

       plt.title('Training history')
       plt.ylabel('Loss')
       plt.xlabel('validation step (4 per Epoch)')
       plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fa3a3702f50>
```



We achieved a 97.54% test accuracy.

```
[39]   test_loss,test_acc =eval_model(model,test_data_loader,negative_log_loss)

[45]   print(f"test Accuracy is {test_acc*100}%")

       test Accuracy is 97.54150702426566%

[46]   print(f"test Loss is {test_loss}")

       test Loss is 0.11394203385568819
```

6.5.9.2 Top K retrieval Accuracy

Given 12513 question with their answer from test set and encoding the ques and answer independently.

```
[105]  print("number of questions",len(test_encoding))
       print("number of answers available",len(ans_encoding))

     number of questions 12513
     number of answers available 12513
```

```
def eval_top_k(test_encoding,ans_encoding,k):
    similarity = np.dot(test_encoding,ans_encoding.T)
    pred = np.argpartition(-similarity, k)[:,:k]
    cnt =0
    for i,row in enumerate(pred):
        if i in row:
            cnt+=1
    return cnt/len(test_encoding)*100
```

We achieved 54.84% exact retrieval Accuracy “given question we get top 1-answer from 12513 answers and it is correct answer”
We achieved 73.36% Top-5 retrieval Accuracy “given question we get top 5-answers from 12513 answers and the correct is one of them”
We achieved 79.83% Top-10 retrieval Accuracy “given question we get top 10-answers from 12513 answers and the correct is one of them”
We achieved 94.57% Top-100 retrieval Accuracy “given question we get top 100-answers from 12513 answers and the correct is one of them”

```
[107]  k=1
       print(f"percantage of exact matches: {eval_top_k(test_encoding,ans_encoding,k):.2f}%")

     percantage of exact matches: 53.84%

[108]  k=5
       print(f"percantage of Top-5 retrieved answers matches: {eval_top_k(test_encoding,ans_encoding,k):.2f}%")

     percantage of Top-5 retrieved answers matches: 73.36%

[109]  k=10
       print(f"percantage of Top-10 retrieved answers matches: {eval_top_k(test_encoding,ans_encoding,k):.2f}%")

     percantage of Top-10 retrieved answers matches: 79.83%

[110]  k=100
       print(f"percantage of Top-100 retrieved answers matches: {eval_top_k(test_encoding,ans_encoding,k):.2f}%")

     percantage of Top-100 retrieved answers matches: 94.57%
```

6.6 Facebook AI Similarity Search(FAISS)

Accurate, fast, and memory-efficient similarity search is a hard thing to do — but something that, if done well, lends itself very well to our huge repositories of endless (and exponentially growing) data.

The reason that similarity search is so good is that it enables us to search for images, text, videos, or any other form of data — without getting too specific in our search queries — which is something that we humans are not so great at.

I will use the example of image similarity search. We can take a picture, and search for similar images. This works by first converting every image into a set of automatically generated features — which are represented as a numerical vector.

When we then compare the vectors of two similar images — we will find that they are very similar.

Now, if we took our picture, and searched for other similar images — we don't really want to compare our query vector to every single vector in the database. Imagine trying to do that with a Google Image search — you could be waiting some time.

Instead, we want to find a more efficient approach — and that is exactly what Facebook AI Similarity Search (FAISS) offers.

Vectors

Our story of efficient similarity search begins with vectors — many of them in fact.

If we were to take three vectors:

We can say quite confidently that vectors a and b are closer to each other than c . And this can be visualized with a simple 3-dimensional chart:

In our case, our vectors have many more dimensions — too many to visualize. But we can still use the same distance metrics to calculate the proximity and/or similarity between two highly dimensional vectors. A few of those are:

Euclidean distance (measures magnitude)

Dot product (measures direction and magnitude)

Cosine similarity (measure direction)

FAISS makes use of both Euclidean distance and dot product for comparing vectors.

Given our vectors a , and b . We calculate the Euclidean distance as:

Euclidean distance calculation

Approximate Nearest Neighbors

Given our Euclidean distance metric, we can move onto creating the nearest neighbors (NN) graph using a set of vectors.

NN is an essential component of FAISS, it is how we build the core ‘distance’ property in our index. However, NN-search is computationally heavy due to the curse of dimensionality .

The process consists of calculating the Euclidean distance between two vectors, and then another two, and so on — the nearest neighbors are those with the shortest distance between them.

The most basic approach to NN-search is the brute-force, exhaustive search — where we calculate the distance between all elements.

Now, if we have a few million vectors indexed — or a billion (as is the number that FAISS is built for) — this can become a heavy operation where we sacrifice speed and memory efficiency, for accuracy:

- Brute Force
- Good Accuracy
- Bad Speed
- Bad Memory (eg a lot of memory is required)

So, we can instead perform a non-exhaustive search that does not search all elements in our index and/or transforms the vectors to make them smaller (and therefore faster to compare against).

Because this non-exhaustive (approximate) NN-search uses slightly modified vectors or a restricted search area — it returns an approximate best result, which is not necessarily the absolute best result.

Approximate NN

- Reasonable Accuracy
- Good Speed
- Good Memory

This degradation in accuracy however is very slight and viewed as a fair trade-off for the performance benefits.

6.6.1 How FAISS Makes Search Efficient

First, FAISS uses all the intelligent ANN graph-building logic. But there is more to FAISS.

1. The first of those efficiency savings comes from efficient usage of the GPU, so the search can process calculations in parallel rather than in series — offering a big speed-up.

2. Additionally, FAISS implements three additional steps in the indexing process. A preprocessing step, followed by two quantization operations — the coarse quantizer for inverted file indexing (IVF), and the fine quantizer for vector encoding.

For many of us, that last paragraph is nonsensical gibberish — so let's break each step down and understand what each means.

6.6.2 FAISS phases

- Preprocessing

  We enter this process with the vectors that we would like FAISS to index. The very first step is to transform these vectors into a more friendly/efficient format. FAISS offers several options here.

- PCA
  use principal component analysis to reduce the number of dimensions in our vectors.
- L2norm
  L2-normalize our vectors.
- OPQ
  Rotates our vectors so they can be encoded more efficiently by the fine quantizer — if using product quantization (PQ).
- Pad
  pads input vectors with zeros up to a given target dimension. This operation means that when we do get around to comparing our query vector against already embedded vectors, each comparison will require less computation — making things faster.

- Inverted File Indexing

  The next step is our inverted file (IVF) indexing process. Again, there are multiple options — but each one is aiming to partition data into similar clusters.

This means that when we query FAISS, and our query is converted into a vector — it will be compared against these partition/cluster centroids.

We compare similarity metrics against our query vector and each of these centroids — and once we find the nearest centroid, we then access all of the full vectors within that centroid (and ignore all others).

Immediately, we have significantly reduced the required search area — reducing complexity and speeding up the search.

This is also called the ‘non-exhaustive’ search component — eg the component that allows us to avoid an ‘exhaustive’ (compare everything) search.

Vector Encoding

The final step is a final encoding step for each vector before it is indexed. This encoding process is carried out by our fine quantizer. The goal here is to reduce index memory size and increase search speed.

There are several options:

Flat — Vectors are stored as is, without any encoding.

PQ — Applies product quantization.

SQ — Applies scalar quantization.

It is worth noting that even with the Flat encoding, FAISS is still going to be very fast.

All these steps and improvements combine to create an incredibly fast similarity search engine — which on GPU is still unbeaten.

7 The Reader

learns to solve the reading comprehension task — extract an answer for a given question from a given context document. Here we only discuss approaches for machine comprehension using neural networks.

Reader is the other core component of a modern OpenQA system and the main feature that differentiates QA systems against other IR or IE systems, which is usually implemented as a neural MRC model. It is aimed at inferring the answer in response to the question from a set of ordered documents and is more challenging compared to the original MRC task where only a piece of passage is given in most cases.

Broadly, existing Readers can be categorized into two types: Extractive Reader that predicts an answer span from the retrieved documents, and Generative Reader that generates answers in natural language using sequence-to-sequence (Seq2Seq) models. Most prior OpenQA systems are equipped with an Extractive Reader.

7.1 Types of reader

- Extractive Reader
- Generative Reader

7.1.1 Extractive Reader

Extractive Reader assumes that the correct answer to a given question definitely exists in the context, and usually focuses on learning to predict the start and end position of an answer span from the retrieved documents. The approaches can be divided into two types according to whether the retrieved documents are processed independently or jointly for answer extraction.

7.1.2 Generative Reader

Generative Reader aims to generate answers as natural as possible instead of extracting answer spans, usually relying on Seq2Seq models. For example, S-Net is developed by combining extraction and generation methods to complement each other. It employs an evidence extraction model to predict the boundary of the text span as the evidence to the answer first and then feeds it into a Seq2Seq answer synthesis model to generate the final answer.

8 Conclusion

In this work we presented a comprehensive survey on the latest progress of Open-domain QA (OpenQA) systems. In particular, we first reviewed the development of OpenQA and illustrated a “Retriever-Reader” architecture. Moreover, we reviewed a variety of existing OpenQA systems as well as their different approaches. Finally, we discussed some salient challenges towards OpenQA followed by a summary of various QA benchmarks, hoping to reveal the research gaps so as to push further progress in this field.