Team Members:

- Ahad Alsulami
- Raneem Alomari
- Bedoor Almareni

About 🍲

A recipe generator is a tool or software that uses algorithms to create unique recipes based on specific criteria such as ingredients, dietary restrictions, cooking methods, and cuisine preferences. These generators are designed to help individuals find new and interesting ways to prepare meals and provide inspiration for creating dishes they may not have considered before

Importing libraries

In [1]:
```
import torch
import torch.nn as nn
from transformers import GPT2TokenizerFast, GPT2LMHeadModel
from transformers import Trainer, TrainingArguments
from tqdm.auto import tqdm
import pandas as pd
import numpy as np
import ipywidgets as widgets
```

Model configuration

In [2]:
```
model_name = 'gpt2'# Name of the pre-trained GPT2 model
```

In [3]:
```
model_save_path = './DLProejectGPT' # Path to save the trained model
```

Tokenizer initialization

In [4]:
```
tokenizer = GPT2TokenizerFast.from_pretrained(model_name,
                                              bos_token='<|startoftext|>',# Beginning of sentence token
                                              eos_token='<|endoftext|>',# End of sentence token
                                              unk_token='<|unknown|>', # Unknown token
                                              pad_token='<|pad|>'# Padding token
                                             )
model = GPT2LMHeadModel.from_pretrained(model_name)# Initialize the GPT2 model
model.resize_token_embeddings(len(tokenizer))# Resize the token embeddings to match the tokenizer
```

```
Downloading:   0%|          | 0.00/0.99M [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/446k [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/1.29M [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/665 [00:00<?, ?B/s]

Special tokens have been added in the vocabulary, make sure the associated word embeddings are fine-tuned or traine
d.

Downloading:   0%|          | 0.00/523M [00:00<?, ?B/s]
```

Out[4]:
```
Embedding(50260, 768)
```

GPT2LMHeadModel is a pre-trained language model based on the GPT-2 architecture. It is designed to generate text by predicting the next word in a sequence given the previous words. It is called a "language model" because it models the probability distribution of words in a language. The "LMHead" in the name stands for "Language Model Head", which refers to the fact that the model is trained to predict the next word in a sequence. The "Head" part of the name is because this is the final layer of the model, which produces the output. In this specific code, the GPT2LMHeadModel is used to generate recipes by predicting the next word in the recipe based on the previous words. The Trainer is a class provided by the Hugging Face transformers library that is used to train and evaluate models. It

provides an easy-to-use interface for training and fine-tuning models, including handling data loading, batching, and optimization.In this code, the Trainer is used to fine-tune the GPT2LMHeadModel on a custom recipe dataset. It takes care of training the model for a specified number of epochs, handling the batching of data, and applying the specified optimizer and learning rate scheduler.

In [5]:
```
tokenizer.save_pretrained(model_save_path)# Save the tokenizer to the specified model_save_path
```

Out[5]:
```
('./DLProejectGPT/tokenizer_config.json',
 './DLProejectGPT/special_tokens_map.json',
 './DLProejectGPT/vocab.json',
 './DLProejectGPT/merges.txt',
 './DLProejectGPT/added_tokens.json',
 './DLProejectGPT/tokenizer.json')
```

In [6]:
```
tokenizer.convert_tokens_to_ids(['<|pad|>'])# Convert the empty token to its corresponding token ID
```

Out[6]:
```
[50259]
```

This generate function takes a prompt as input, encodes it using the tokenizer, generates output text based on the prompt using the model, and finally decodes and prints the generated text.

In [7]:
```
def generate(prompt):
    # Encode the prompt using the tokenizer
    inputs = tokenizer.encode_plus(prompt, return_tensors='pt')
    # Generate output text based on the prompt using the model
    output = model.generate(**inputs,max_length=256,do_sample=True,pad_token_id=50259)
    # Decode and print the generated text
    print(tokenizer.decode(output[0]))
```

In [8]:
```
# Get the special tokens map from the tokenizer
tokenizer.special_tokens_map
```

Out[8]:
```
{'bos_token': '<|startoftext|>',
 'eos_token': '<|endoftext|>',
 'unk_token': '<|unknown|>',
 'pad_token': '<|pad|>'}
```

In [9]:
```
tokenizer.convert_tokens_to_ids(['<|startoftext|>'],)# Convert the empty token to its corresponding token ID
```

Out[9]:
```
[50257]
```

Load data

In [10]:
```
# Read the CSV file into a pandas DataFrame called 'clean'
clean = pd.read_csv('/kaggle/input/food-dataset/Food_Recipe_Dataset.csv')
# Shuffle the rows of the DataFrame
clean = clean.sample(frac=1)
# Reset the index of the DataFrame
clean.reset_index(drop=True,inplace=True)
```

In [11]:
```
print(clean['Cuisine'].unique())#unique cuisine found in the dataset
```

```
['Sindhi' 'Mexican' 'Indian' 'Tamil Nadu' 'Chettinad' 'Goan Recipes'
 'North Indian Recipes' 'Karnataka' 'Chinese' 'Continental' 'Rajasthani'
 'Maharashtrian Recipes' 'South Indian Recipes' 'Italian Recipes' 'Udupi'
 'Asian' 'Indo Chinese' 'Bengali Recipes' 'Kerala Recipes' 'Parsi Recipes'
 'French' 'Awadhi' 'European' 'Coorg' 'Assamese' 'Punjabi' 'Kashmiri'
 'Gujarati Recipes\ufeff' 'Andhra' 'Coastal Karnataka' 'Thai'
 'Mediterranean' 'Konkan' 'Japanese' 'Sri Lankan' 'Fusion'
 'South Karnataka' 'Sichuan' 'Himachal' 'Middle Eastern' 'Caribbean'
 'Uttar Pradesh' 'African' 'Jharkhand' 'Nepalese' 'Malabar' 'Mangalorean'
 'Lucknowi' 'Oriya Recipes' 'Vietnamese' 'Mughlai' 'Pakistani'
 'Hyderabadi' 'North Karnataka' 'North East India Recipes' 'Hunan'
 'Malvani' 'Nagaland' 'Malaysian' 'Shandong' 'British' 'Cantonese' 'Greek'
 'Indonesian' 'Bihari' 'Afghan' 'Kongunadu' 'Haryana' 'Lunch' 'Brunch'
 'American' 'Korean' 'Arab' 'Uttarakhand-North Kumaon' 'World Breakfast'
 'Dinner' 'Side Dish' 'Appetizer' 'Snack' 'Dessert' 'Burmese' 'Jewish']
```

In [12]:
```
def print_recipe(idx):
    # Print the ingredients and instructions of the recipe at the specified index.
    print(f"{clean['ingredients'][idx]}\n\n{clean['instructions'][idx]}")
```

the form_string function that takes an ingredient and an instruction as inputs and returns a formatted string combining them.

In [13]:
```
def form_string(ingredient,instruction):
    # Formulate the string combining the ingredients and instructions
    s = f"<|startoftext|>Ingredients:\n{ingredient.strip()}\n\nInstructions:\n{instruction.strip()}<|endoftext|>"
    return s
```

In [14]:
```
# Apply the form_string function to each row in the clean DataFrame
# using 'TranslatedIngredients' and 'TranslatedInstructions' columns as inputs
data = clean.apply(lambda x:form_string(x['TranslatedIngredients'],x['TranslatedInstructions']),axis=1).to_list()
```

https://towardsdatascience.com/guide-to-fine-tuning-text-generation-models-gpt-2-gpt-neo-and-t5-dc5de6b3bc5e (https://towardsdatascience.com/guide-to-fine-tuning-text-generation-models-gpt-2-gpt-neo-and-t5-dc5de6b3bc5e)

splits the data

In [15]:
```
# Set the proportion of data to be used for training
train_size = 0.85
# Calculate the length of the training set based on the specified train_size
train_len = int(train_size * len(data))
# Split the data into training and validation sets
train_data = data[:train_len]# Contains the first train_len elements for training
val_data = data[train_len:] # Contains the remaining elements for validation
```

Defines a RecipeDataset class, which is a PyTorch dataset for working with recipe data. It takes a data list as input during initialization.

The RecipeDataset class has three main methods:

1 - Initializes the dataset by processing the data list. It tokenizes and encodes each item in the data list using the tokenizer. The resulting input_ids and attention_masks are stored in separate lists self.input_ids and self.attn_masks.**

2 - Returns the total number of items in the dataset, which is the length of the data list.

3 - Returns the input_ids and attention_masks for the item at the given index idx.

In [16]:
```
class RecipeDataset:
    def __init__(self,data):
        self.data = data
        self.input_ids = []
        self.attn_masks = []
        # Iterate over the data and process each item
        for item in tqdm(data):
            # Tokenize and encode the item using the tokenizer
            encodings = tokenizer.encode_plus(item,
                                              truncation=True,
                                              padding='max_length',
                                              max_length=1024,
                                              return_tensors='pt'
                                             )
            # Extract and store the input_ids and attention_masks
            self.input_ids.append(torch.squeeze(encodings['input_ids'],0))
            self.attn_masks.append(torch.squeeze(encodings['attention_mask'],0))

    def __len__(self):
        # Return the total number of items in the dataset
        return len(self.data)

    def __getitem__(self,idx):
        # Return the input_ids and attention_masks for the item at the given index
        return self.input_ids[idx], self.attn_masks[idx]
```

collate_fn function collate a batch of data samples in the custom RecipeDataset.

In [17]:
```
def collate_fn(batch):
    # Stack the input_ids, attention_mask, and labels tensors in the batch
    return {
        'input_ids': torch.stack([item[0] for item in batch]),
        'attention_mask': torch.stack([item[1] for item in batch]),
        'labels': torch.stack([item[0] for item in batch])
    }
```

In [18]:
```
# Initialize train_ds with the training data
train_ds = RecipeDataset(train_data)
# Initialize val_ds with the validation data
val_ds = RecipeDataset(val_data)
```

```
  0%|          | 0/5047 [00:00<?, ?it/s]

  0%|          | 0/891 [00:00<?, ?it/s]
```

In [19]:
```
# Initialize args with the training arguments and settings
args = TrainingArguments(
    output_dir=model_save_path,  # Directory to save the trained model
    per_device_train_batch_size=2,  # Batch size for training on each device
    per_device_eval_batch_size=2,  # Batch size for evaluation on each device
    gradient_accumulation_steps=2,  # Number of steps to accumulate gradients before performing optimization
    report_to='none',  # Disable reporting of training progress
    num_train_epochs=3,  # Number of training epochs
    save_strategy='no'  # Disable saving of checkpoints during training
)
```

In [20]:
```
# Initialize the optimizer using the AdamW algorithm
optim = torch.optim.AdamW(model.parameters(), lr=5e-5)
# Initialize the scheduler using the CosineAnnealingWarmRestarts method
scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optim, 20, eta_min=1e-7)
```

In [21]:
```
# Initialize the trainer object for model training
trainer = Trainer(
    model,  # The model to be trained
    args,  # The training arguments and settings
    train_dataset=train_ds,  # The training dataset
    eval_dataset=val_ds,  # The validation dataset
    data_collator=collate_fn,  # The collate function for batching the data
    optimizers=(optim, scheduler)  # The optimizer and scheduler
)
```

In [22]:
```
trainer.train()#starts the training process using the trainer object.
```

```
***** Running training *****
  Num examples = 5047
  Num Epochs = 3
  Instantaneous batch size per device = 2
  Total train batch size (w. parallel, distributed & accumulation) = 8
  Gradient Accumulation steps = 2
  Total optimization steps = 1893
/opt/conda/lib/python3.7/site-packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather along di
mension 0, but all input tensors were scalars; will instead unsqueeze and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

[1893/1893 51:59, Epoch 3/3]

Step	Training Loss
500	1.132700
1000	0.779200
1500	0.745000

```
Training completed. Do not forget to share your model on huggingface.co/models =)
```

Out[22]:
```
TrainOutput(global_step=1893, training_loss=0.8481587340953926, metrics={'train_runtime': 3126.9099, 'train_samples_
per_second': 4.842, 'train_steps_per_second': 0.605, 'total_flos': 7912445313024000.0, 'train_loss': 0.8481587340953
926, 'epoch': 3.0})
```

In [23]:
```
trainer.save_model()#to save the trained model after the training process is completed.
```

```
Saving model checkpoint to ./DLProejectGPT
Configuration saved in ./DLProejectGPT/config.json
Model weights saved in ./DLProejectGPT/pytorch_model.bin
```

In [24]:
```
from transformers import pipeline
```

In [25]:
```
pl = pipeline(task='text-generation',model='/kaggle/working/DLProejectGPT')#initializes a text generation pipeline
```

```
loading configuration file /kaggle/working/DLProejectGPT/config.json
Model config GPT2Config {
  "_name_or_path": "/kaggle/working/DLProejectGPT",
  "activation_function": "gelu_new",
  "architectures": [
    "GPT2LMHeadModel"
  ],
  "attn_pdrop": 0.1,
  "bos_token_id": 50256,
  "embd_pdrop": 0.1,
  "eos_token_id": 50256,
  "initializer_range": 0.02,
  "layer_norm_epsilon": 1e-05,
  "model_type": "gpt2",
  "n_ctx": 1024,
  "n_embd": 768,
  "n_head": 12,
  "n_inner": null,
  "n_layer": 12,
  "n_positions": 1024,
  "reorder_and_upcast_attn": false,
  "resid_pdrop": 0.1,
  "scale_attn_by_inverse_layer_idx": false,
  "scale_attn_weights": true,
  "summary_activation": null,
  "summary_first_dropout": 0.1,
  "summary_proj_to_labels": true,
  "summary_type": "cls_index",
  "summary_use_proj": true,
  "task_specific_params": {
    "text-generation": {
      "do_sample": true,
      "max_length": 50
    }
  },
  "torch_dtype": "float32",
  "transformers_version": "4.20.1",
  "use_cache": true,
  "vocab_size": 50260
}

loading configuration file /kaggle/working/DLProejectGPT/config.json
Model config GPT2Config {
  "_name_or_path": "/kaggle/working/DLProejectGPT",
  "activation_function": "gelu_new",
  "architectures": [
    "GPT2LMHeadModel"
  ],
  "attn_pdrop": 0.1,
  "bos_token_id": 50256,
  "embd_pdrop": 0.1,
  "eos_token_id": 50256,
  "initializer_range": 0.02,
  "layer_norm_epsilon": 1e-05,
  "model_type": "gpt2",
  "n_ctx": 1024,
  "n_embd": 768,
  "n_head": 12,
  "n_inner": null,
  "n_layer": 12,
  "n_positions": 1024,
  "reorder_and_upcast_attn": false,
  "resid_pdrop": 0.1,
  "scale_attn_by_inverse_layer_idx": false,
  "scale_attn_weights": true,
  "summary_activation": null,
  "summary_first_dropout": 0.1,
  "summary_proj_to_labels": true,
  "summary_type": "cls_index",
  "summary_use_proj": true,
  "task_specific_params": {
    "text-generation": {
      "do_sample": true,
      "max_length": 50
    }
  },
  "torch_dtype": "float32",
  "transformers_version": "4.20.1",
  "use_cache": true,
  "vocab_size": 50260
}

loading weights file /kaggle/working/DLProejectGPT/pytorch_model.bin
All model checkpoint weights were used when initializing GPT2LMHeadModel.

All the weights of GPT2LMHeadModel were initialized from the model checkpoint at /kaggle/working/DLProejectGPT.
If your task is similar to the task the model of the checkpoint was trained on, you can already use GPT2LMHeadModel
for predictions without further training.
```

```
loading file /kaggle/working/DLProejectGPT/vocab.json
loading file /kaggle/working/DLProejectGPT/merges.txt
loading file /kaggle/working/DLProejectGPT/tokenizer.json
loading file /kaggle/working/DLProejectGPT/added_tokens.json
loading file /kaggle/working/DLProejectGPT/special_tokens_map.json
loading file /kaggle/working/DLProejectGPT/tokenizer_config.json
```

The create_prompt function takes a string of ingredients and a cuisine name as input and creates a formatted prompt string for generating a recipe.

In [26]:
```python
def create_prompt(cuisine,ingredients):
    # Convert the ingredients to lowercase and remove leading/trailing whitespaces
    ingredients = ','.join([x.strip().lower() for x in ingredients.split(',')])
    # Replace commas with newline characters for better ingredient formatting
    ingredients = ingredients.strip().replace(',', '\n')
    # Create the prompt string with the formatted ingredients and cuisine
    s = f"\n\nCuisine:\n{Cuisine.value}\n\nIngredients:\n{ingredients}\n"
    return s
```

In [27]:
```python
# Cusinie Selection
print("Want to explore new flavors? Choose your cuisine preference!\n")
Cuisine = widgets.Dropdown(options = clean['Cuisine'].unique(),
                                value=None)
display(Cuisine)


# Ingredients Selection
ingredients = [i for i in input("\nAdd your ingredients for a unique recipe!"+
                                "\n(separate them with a comma :)\n").split()]
```

```
Want to explore new flavors? Choose your cuisine preference!


Dropdown(options=('Sindhi', 'Mexican', 'Indian', 'Tamil Nadu', 'Chettinad', 'Goan Recipes', 'North Indian Reci…


Add your ingredients for a unique recipe!
(separate them with a comma :)
 flour,sugar,cinnamon,vanilla
```

In [29]:
```python
for ing in ingredients:
    # Create a prompt using the current ingredient set and the specified cuisine
    prompt = create_prompt(Cuisine.value,ing)
    # Generate a recipe using the pipeline with specified parameters and print the generated recipe
    print(pl(prompt,
         max_new_tokens=512,
         penalty_alpha=0.6,
         top_k=4,
         pad_token_id=50259
        )[0]['generated_text'])
```

```


Cuisine:
Arab

Ingredients:
flour
sugar
cinnamon
vanilla

1 cup raita flour, salt

1 cup rice flour, 1/2 cup water, salt - as required

pinch cinnamon, 1/2 cup sugar, 2 cups water, 1/2 teaspoon turmeric powder

Instructions:
To make the raita rice dough, firstly we will first make the raita.
In a mixer, add the rice flour, salt, turmeric powder, cinnamon, sugar, water and grind to a smooth dough.
Keep it aside.Now heat oil in a pan.
Add cinnamon, sugar, turmeric and cook for 2 minutes.
Once the spices start to sizzle, add the raita flour, rice flour, water and cook for 2 minutes.
Add the remaining water and cook until the raita is cooked well.
After 2 minutes, add the raita dough into the mixer and mix well.
Check the salt and spice levels and adjust according to your taste.
Serve the raita rice along with steamed rice and phulkas for a weekday meal.
You can also serve it with phulkas for a wholesome lunch.
```