**VOICE ASSISTANT USING PARLER MODEL**

**Table of Contents**

[**Introduction 3**](#_heading=)

[Parler-TTS 3](#_heading=h.9mvz548jar2j)

[Kaggle notebook 3](#_heading=h.cb9cfwzyw2h)

[HuggingFace Hub 6](#_heading=h.q4patcaravaa)

[**Data Creation 7**](#_heading=)

[Annotate the dataset 8](#_heading=h.it9vwu5q8ntl)

[Map annotations to text bins 8](#_heading=h.bj9vsptnp76)

[Create natural language descriptions from those text bins 9](#_heading=h.p8pu6daxekfx)

[**Training the model 10**](#_heading=)

[Training Process 10](#_heading=h.uw1f8iy2v46c)

[Explanation of Steps for Training: 13](#_heading=h.5uflix148cp9)

[**Load the Trained Model and Use it 14**](#_heading=h.vxphl3r5mraa)

# Introduction

Reference:

<https://huggingface.co/parler-tts>

<https://github.com/huggingface/parler-tts>

## Parler-TTS

Parler-TTS is a text-to-speech (TTS) model that can generate high-quality, natural sounding speech in the style of a given speaker (gender, pitch, speaking style, etc). This model is different than the VITS model by “user description”

Normally we use three platform to train the model and for inference process :

1) kaggle (gpu 30 gb )

2) colab (15 gb )

3) ai server (gpu 16 gb)

for Parler-TTS we need large gpu power for training so we prefer to use kaggle for model training

link of kaggle notebook : [click !](https://drive.google.com/drive/folders/1Abf49Aqh5BLLxPrLEhnJjK7PZvHZqO_k?usp=drive_link)

## Kaggle notebook

Kaggle Notebooks are a powerful tool for data scientists and machine learning practitioners, providing a convenient and efficient way to write, execute, and share code in a collaborative cloud-based environment.

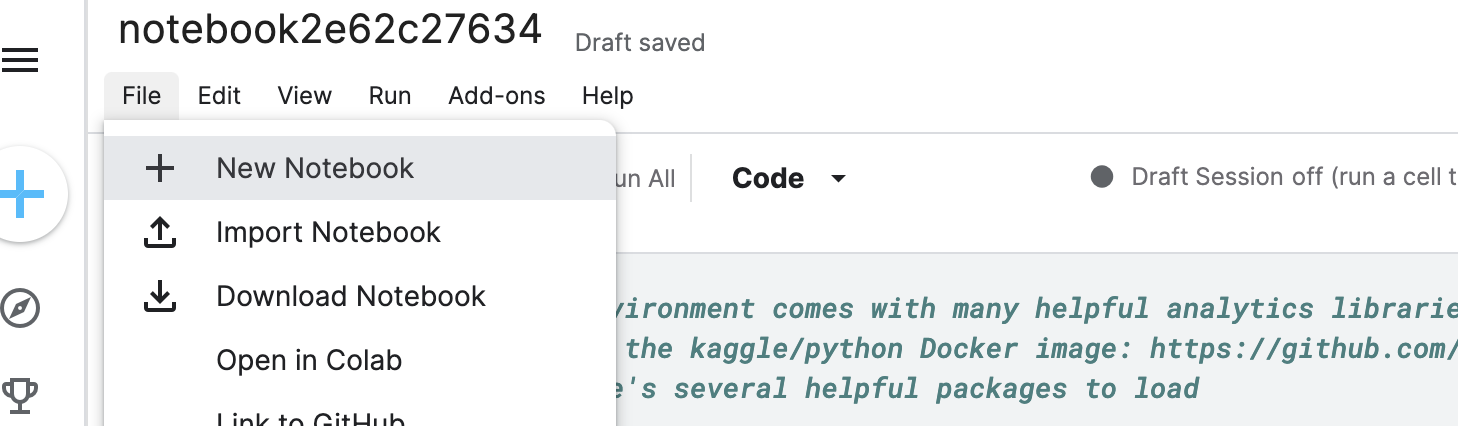
**Code location:**

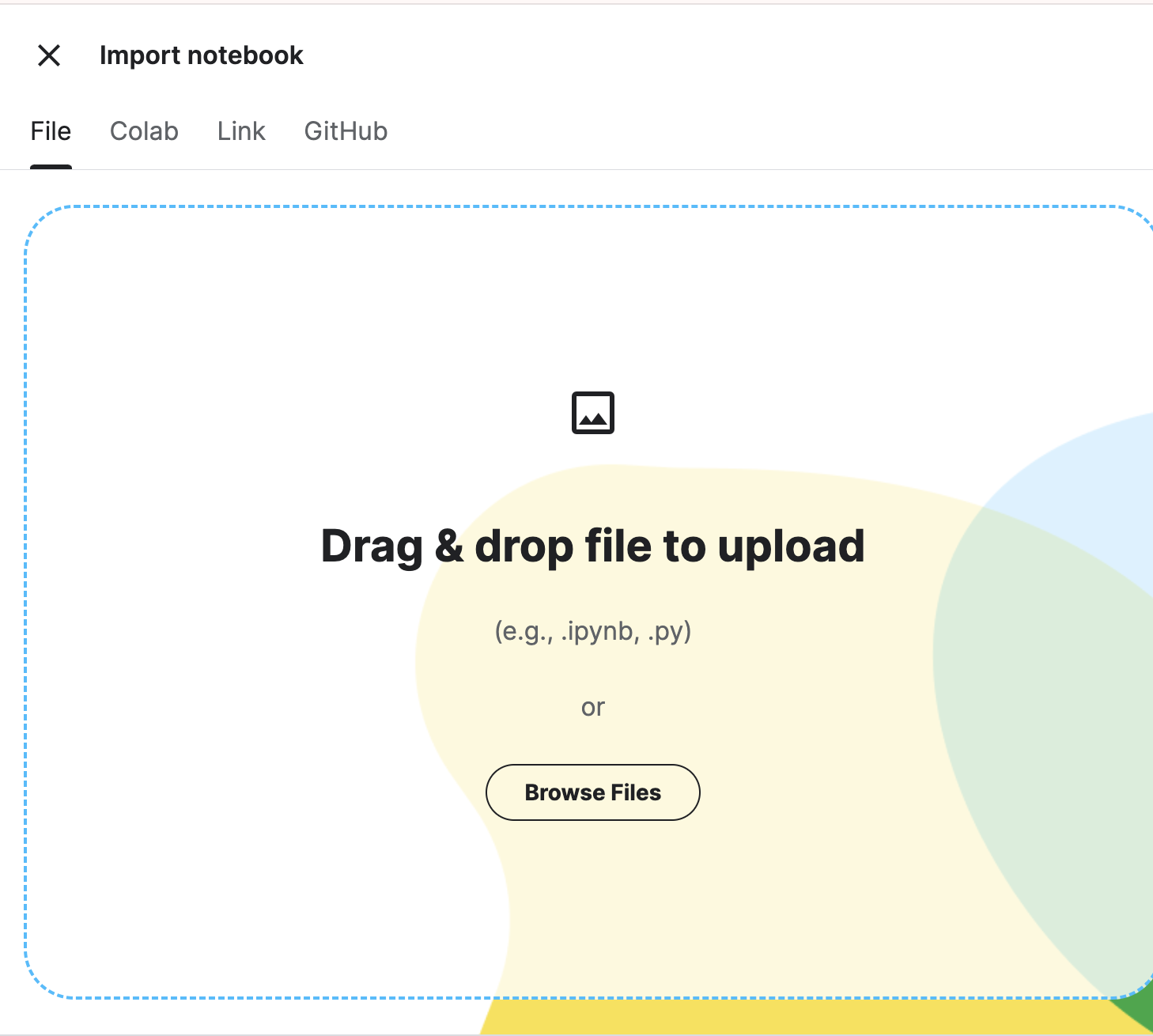
Due to low resource in the server we are unable to train it in server, so we have configure the notebook to be used in kaggle.

Drive Link: [ParlerTTS](https://drive.google.com/drive/folders/1Abf49Aqh5BLLxPrLEhnJjK7PZvHZqO_k)

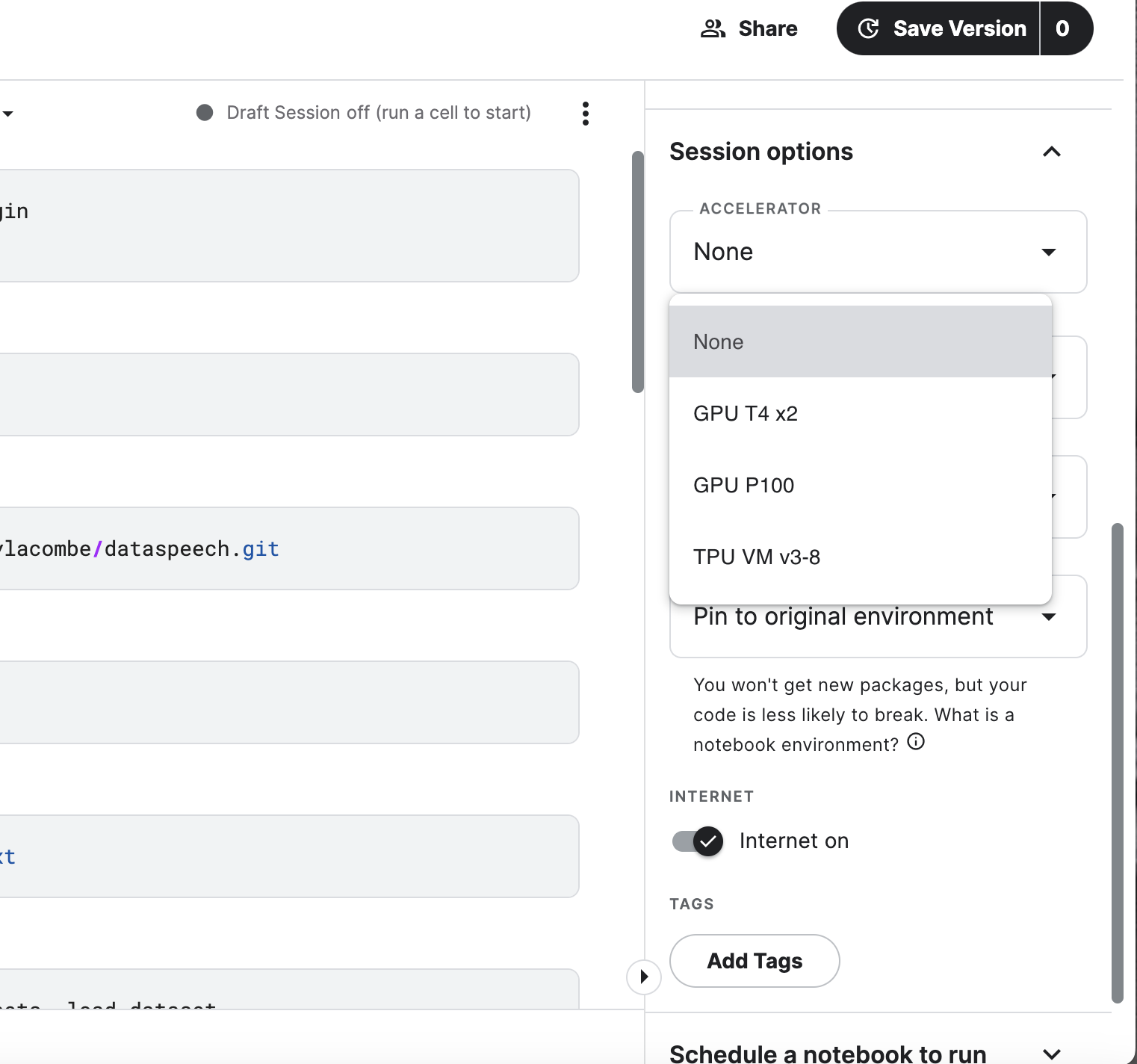
**Load the code in kaggle:**

Step1: Load the dataset processing code available in the Drive Link above in kaggle by importing the file.

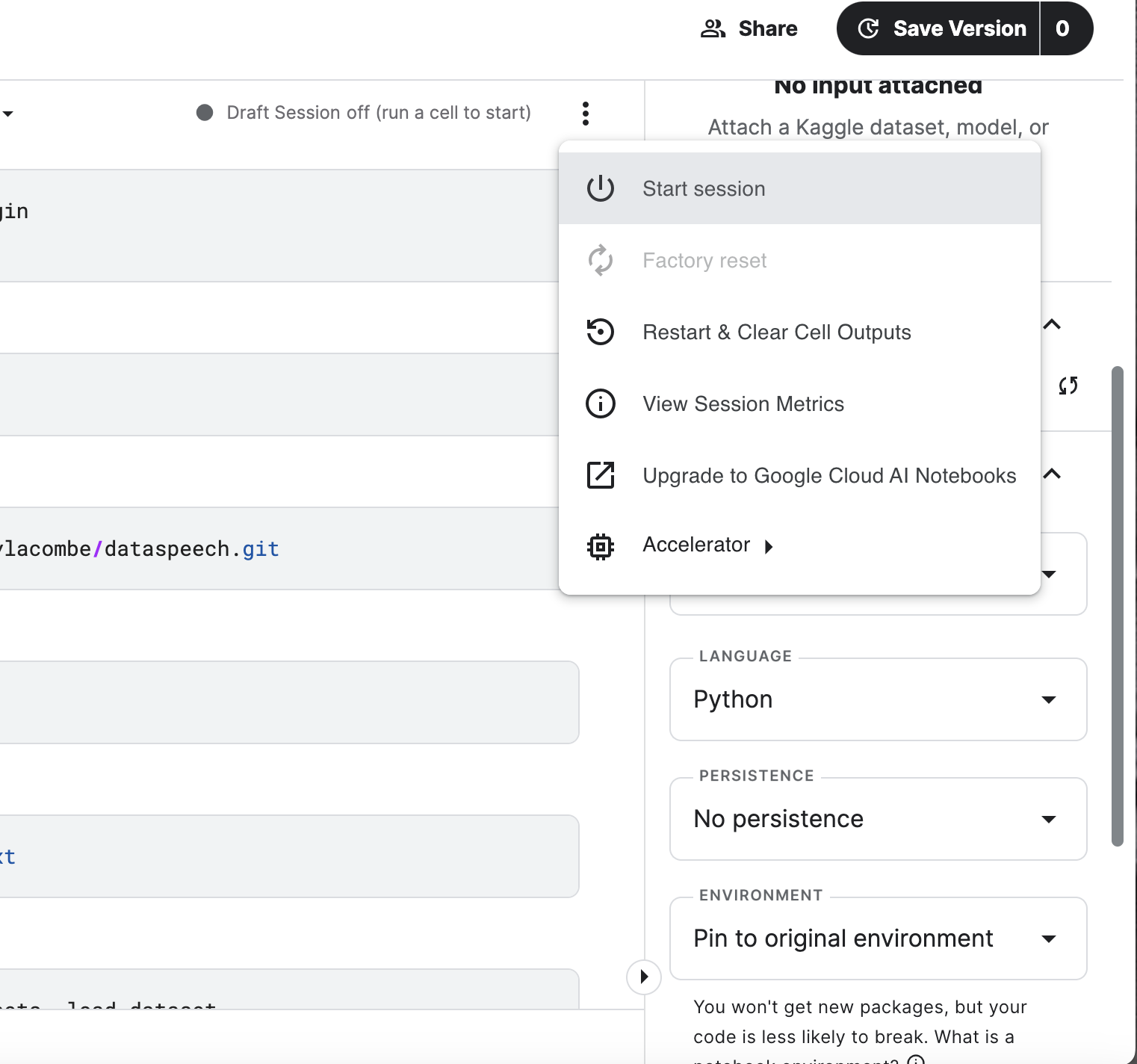




Step2: Assign GPU T4 X 2 from the Accelerator options and also check the internet option as on in the Session Options which you can find in the right side of the notebook

.

Step3 : Then on the upper right corner in the three dots select the Start Session to connect your notebook to the GPU. Once this step is completed, you are ready to run the code.



## HuggingFace Hub

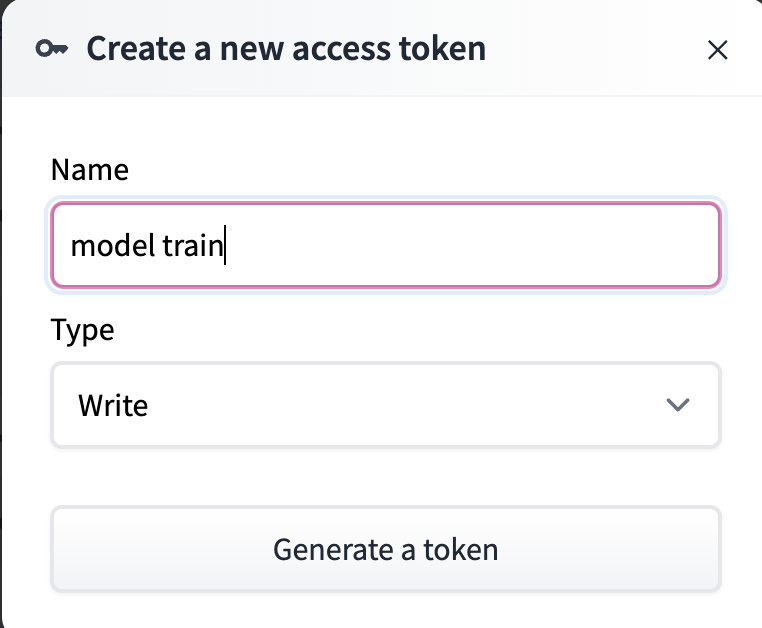
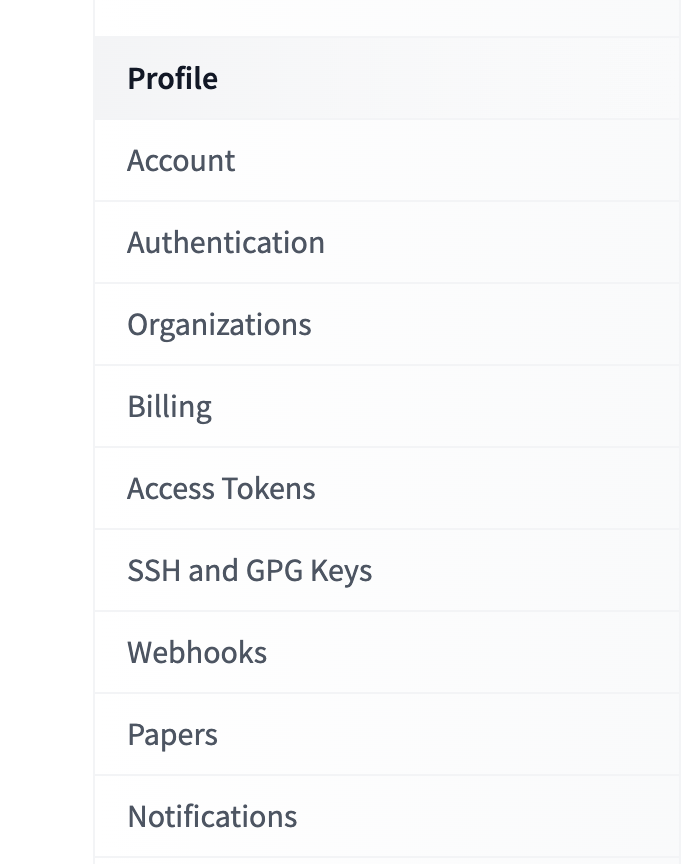
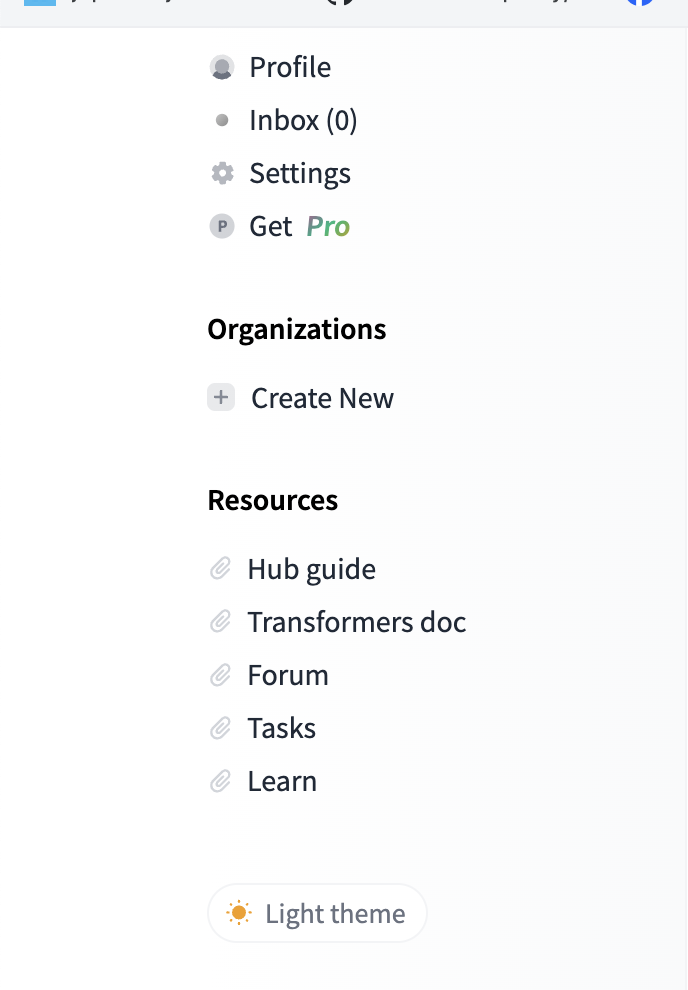
HuggingFace is an open platform for Machine learning which provides access to a wide range of pre-trained open-source models, datasets and other resources.

We use this platform to store our project datasets, trained models and the models under experimentation.

Before running the script, ensure you have the secret token of hugging face.

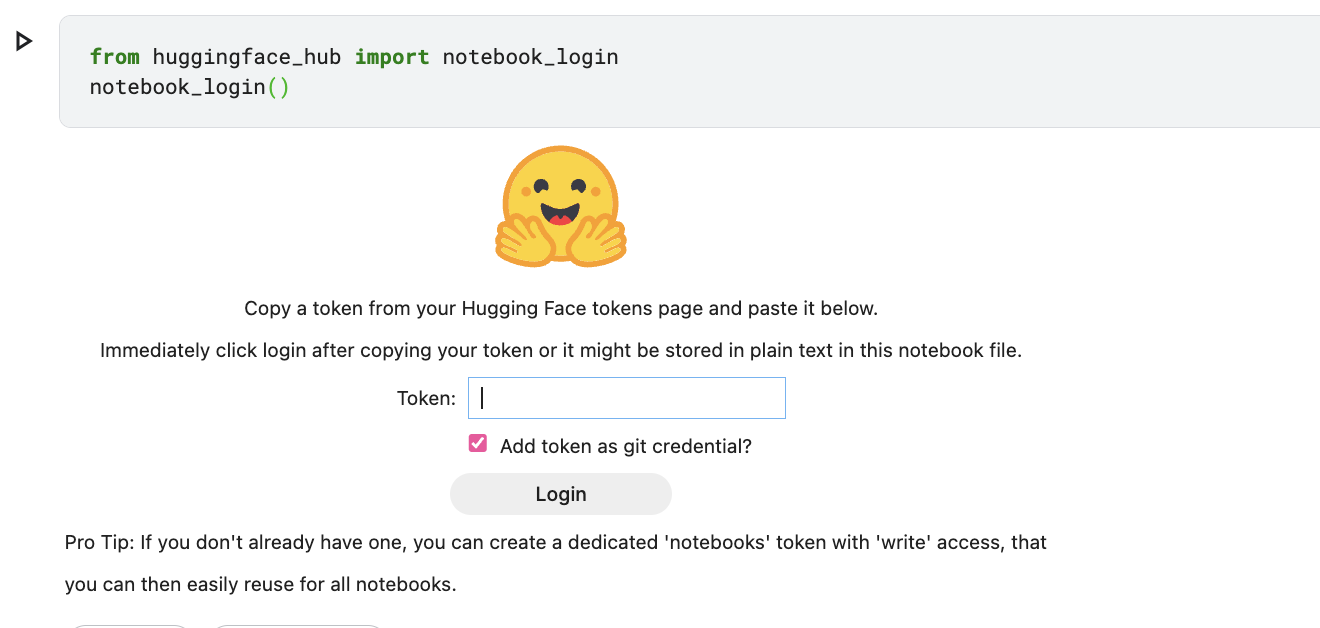
The secret token can be created by : Settings -> Access Tokens -> New Token

Here you can generate a new token with Type - Write



When we are using Kaggle notebook then to login from Kaggle we need to run the following scripts:

* !pip install huggingface\_hub
* from huggingface\_hub import notebook\_login
* notebook\_login()



# Data Creation

Code Link: [parllerDatacreation.ipynb](https://drive.google.com/file/d/1dMiFpIik5TxPhqfVC1PFzMNLsJ8eBCo0/view?usp=sharing)

Once the login process is completed as above, and the notebook is uploaded in kaggele, we ensure all the other required libraries are installed.

1. Install the datasets library, which is used for handling and processing datasets in the Hugging Face ecosystem.

!pip install datasets

1. Clone the dataspeech repository from GitHub, which contains scripts and resources for the project.

!git clone https://github.com/ylacombe/dataspeech.git

1. Change the current working directory to the dataspeech repository directory.

%cd dataspeech

1. Install the dependencies listed in the requirements.txt file within the dataspeech repository.

!pip install -r requirements.txt

1. Load the dataset named Modified\_audio\_data\_API from the user Procit004 using the datasets library which is a collection of raw sounds.

from datasets import list\_datasets, load\_dataset

dataset = load\_dataset("Procit004/Modified\_audio\_data\_API")

Once all the requirements are installed and the raw data is loaded we can now use the following script :

#### Annotate the dataset

!python main.py "Procit004/Modified\_audio\_data\_API" \ -> Use your raw dataset

--configuration "default" \

--text\_column\_name "text" \

--audio\_column\_name "audio" \

--cpu\_num\_workers 8 \

--num\_workers\_per\_gpu\_for\_pitch 2 \

--rename\_column \

--repo\_id "Procit004/OwnData\_english\_2" → Name the dataset as you like

The above main.py script adds utterance\_pitch, phonemes, Signal-to-noise ratio (snr), speaking\_rate, c50 (clarity index or clarity measure) to the dataset. The resulting dataset then can be pushed to the HuggingFace hub under your desired handle. Ours was saved to “**Procit004/OwnData\_english\_2**”.

#### Map annotations to text bins

!python ./scripts/metadata\_to\_text.py \

"Procit004/OwnData\_english\_2" \

--repo\_id "Procit004/OwnData\_english\_2" \

--configuration "default" \

--cpu\_num\_workers 10\

--path\_to\_bin\_edges "./examples/tags\_to\_annotations/v01\_bin\_edges.json" \

--avoid\_pitch\_computation

The above script metadata\_to\_text.py adds text bins such as noise, reverbation and speech monotony to the samples. The resulting dataset was then saved to “**Procit004/OwnData\_english\_2**”.

#### Create natural language descriptions from those text bins

!python ./scripts/run\_prompt\_creation.py \

--speaker\_name "ram" \

--is\_single\_speaker \

--dataset\_name "Procit004/OwnData\_english\_2" \

--output\_dir "./tmp\_sita" \

--dataset\_config\_name "default" \

--model\_name\_or\_path "google/gemma-2b-it" \

--per\_device\_eval\_batch\_size 10 \

--attn\_implementation "sdpa" \

--dataloader\_num\_workers 3 \

--push\_to\_hub \

--hub\_dataset\_id "OwnData\_english\_model\_v2" \

--preprocessing\_num\_workers 3

The above script run\_prompt\_creation.py adds text\_descriptions to the “**OwnData\_english\_2**” dataset and saves it to hugging face as “**OwnData\_english\_model\_v2**”

Once the dataset is created, we move on to training our Parler Model with our dataset.

# Training the model

Code Link: [parllerModelTraining.ipynb](https://drive.google.com/file/d/19lIksAc_wTCs4StQnHHf5g_23C1dF4w2/view?usp=sharing)

1. Log into the Hugging Face Hub, so that you can access and manage models.

from huggingface\_hub import login

login()

1. Clone the Parler TTS repository from GitHub which installs the necessary dependencies for training.

!git clone https://github.com/huggingface/parler-tts.git

%cd parler-tts

!pip install --quiet -e .[train]

1. Upgrade the protobuf library and install a specific version of Weights and Biases (W&B) for experiment tracking.

!pip install --upgrade protobuf wandb==0.16.6

1. Log into W&B, allowing you to track and visualize your training experiments.

import wandb

import os

os.environ['WANDB\_API\_KEY'] = 'your\_wandb\_api\_key'

wandb.login()

Once all the requirements for the model are installed we can now train the model by running the following:

#### Training Process

The code below defines and runs the training command using the accelerate launcher from Hugging Face. The training parameters and dataset details are all specified in the code.

import subprocess

# Define the command you want to run

command = [

"accelerate", "launch", "./training/run\_parler\_tts\_training.py",

"--model\_name\_or\_path", "procit001/parler-tts-mini\_fbdata\_v1.2",

"--feature\_extractor\_name", "parler-tts/dac\_44khZ\_8kbps",

"--description\_tokenizer\_name", "procit001/parler-tts-mini\_fbdata\_v1.2",

"--prompt\_tokenizer\_name", "procit001/parler-tts-mini\_fbdata\_v1.2",

"--report\_to", "wandb",

"--overwrite\_output\_dir", "true",

"--train\_dataset\_name", "Procit004/Modified\_audio\_data\_API",

"--train\_metadata\_dataset\_name", "Procit004/OwnData\_english\_model\_v2",

"--train\_dataset\_config\_name", "default",

"--train\_split\_name", "train",

"--eval\_dataset\_name", "Procit004/Modified\_audio\_data\_API",

"--eval\_metadata\_dataset\_name", "Procit004/OwnData\_english\_model\_v2",

"--eval\_dataset\_config\_name", "default",

"--eval\_split\_name", "train",

"--max\_eval\_samples", "8",

"--per\_device\_eval\_batch\_size", "8",

"--target\_audio\_column\_name", "audio",

"--description\_column\_name", "text\_description",

"--prompt\_column\_name", "text",

"--max\_duration\_in\_seconds", "20",

"--min\_duration\_in\_seconds", "2.0",

"--max\_text\_length", "400",

"--preprocessing\_num\_workers", "2",

"--do\_train", "true",

"--num\_train\_epochs", "40",

"--gradient\_accumulation\_steps", "18",

"--gradient\_checkpointing", "true",

"--per\_device\_train\_batch\_size", "2",

"--learning\_rate", "0.00008",

"--adam\_beta1", "0.9",

"--adam\_beta2", "0.99",

"--weight\_decay", "0.01",

"--lr\_scheduler\_type", "constant\_with\_warmup",

"--warmup\_steps", "50",

"--logging\_steps", "2",

"--freeze\_text\_encoder", "true",

"--audio\_encoder\_per\_device\_batch\_size", "4",

"--dtype", "float16",

"--seed", "456",

"--output\_dir", "./output\_dir\_training/",

"--temporary\_save\_to\_disk", "./audio\_code\_tmp/",

"--save\_to\_disk", "./tmp\_dataset\_audio/",

"--dataloader\_num\_workers", "2",

"--do\_eval",

"--predict\_with\_generate",

"--include\_inputs\_for\_metrics",

"--group\_by\_length", "true"

]

# Run the command

process = subprocess.Popen(command, stdin=subprocess.PIPE, text=True)

process.communicate(input="3\n") # Automatically choose option 3

# Wait for the process to complete

process.wait()

By using the subprocess module, the code bypasses interactive widget issues in Kaggle environments by simulating user input. This ensures that the training process runs smoothly without requiring manual intervention.

#### Explanation of Steps for Training

1. **Import the subprocess module**:
   * The subprocess module is essential for spawning new processes, connecting to their input/output/error pipes, and obtaining their return codes.
2. **Define the Training Command**:
   * A list named command is created to hold the training command and its associated arguments. This command includes all necessary parameters for training the TTS model, such as model paths, dataset details, training configurations, and hyperparameters.
3. **Parameters and Arguments**:
   * accelerate launch ./training/run\_parler\_tts\_training.py: Specifies the script to be run using the accelerate launcher.
   * --model\_name\_or\_path, --feature\_extractor\_name, --description\_tokenizer\_name, --prompt\_tokenizer\_name: Define the paths to the model, feature extractor, and tokenizers.
   * --report\_to "wandb": Indicates that training metrics should be reported to Weights & Biases.
   * --overwrite\_output\_dir "true": Ensures that the output directory is overwritten if it already exists.
   * --train\_dataset\_name, --eval\_dataset\_name: Specify the names of the training and evaluation datasets.
   * --train\_metadata\_dataset\_name, --eval\_metadata\_dataset\_name: Provide the names of the metadata datasets.
   * --max\_eval\_samples "8": Limits the number of evaluation samples to 8.
   * --per\_device\_eval\_batch\_size "8", --per\_device\_train\_batch\_size "2": Define the batch sizes for evaluation and training.
   * Other parameters: Include settings for duration limits, text length, preprocessing workers, epochs, learning rate, Adam optimizer settings, weight decay, learning rate scheduler, warmup steps, logging steps, and output directories.
4. **Run the Training Command**:
   * subprocess.Popen is used to start the training process. The command is passed as a list, and stdin is set to subprocess.PIPE to enable sending input to the process.
5. **Automate User Input**:
   * process.communicate(input="3\n"): This line sends the "3" key followed by a newline character to the process, automating the selection of the third option. This is crucial in Kaggle environments where widgets may be blocked, preventing manual input.
6. **Wait for Completion**:
   * process.wait(): This line waits for the training process to complete before proceeding. It ensures that the script does not exit prematurely, allowing the training to finish.

# Load the Trained Model and Use it

The code below loads the trained model and tokenizer onto the appropriate device (GPU if available).

from parler\_tts import ParlerTTSForConditionalGeneration

from transformers import AutoTokenizer

import torch

device = "cuda:0" if torch.cuda.is\_available() else "cpu"

print("device",device)

model = ParlerTTSForConditionalGeneration.from\_pretrained("output\_dir\_training/", torch\_dtype=torch.float16).to(device)

tokenizer = AutoTokenizer.from\_pretrained("output\_dir\_training/")

Generate speech and test the sound using the format below.

prompt = "Hey this is a test sample. Isn’t the weather lovely?"

description = "The speech sample is very clear and has a very fast pace, but it is very confined in its sound"

input\_ids = tokenizer(description, return\_tensors="pt").input\_ids.to(device)

prompt\_input\_ids = tokenizer(prompt, return\_tensors="pt").input\_ids.to(device)

generation = model.generate(input\_ids=input\_ids, prompt\_input\_ids=prompt\_input\_ids)

audio\_arr = generation.cpu().numpy().squeeze()

If satisfied with the output voice, save the model to your Hugging Face.

model.push\_to\_hub("parler-tts-OwnVoice\_v2")

tokenizer.push\_to\_hub("parler-tts-OwnVoice\_v2")