NLP Problem Statement

Design a ML model to identify the language of a sentence using the following dataset.

Dataset:

Tatoeba *- Contains all the sentences in the selected language, approximately having 9.9 million sentences in 409 supported languages. Each sentence is associated with a unique id and an ISO 639-3 language code. Fields and structure -* ***Sentence id [tab] Lang [tab] Text***
Resource Page: https://tatoeba.org/en/downloads
Download Link: https://downloads.tatoeba.org/exports/sentences.tar.bz2 (~155 MB compressed)

Approach

The problem of language detection is formulated as a character sequence classification problem. In order to build an effective language detector, following methods / algorithms have been tried.

1. Open-source python package – langid.py
2. Baseline model with character tokenizer, TF-IDF transformer for feature extraction and Naïve bayes algorithm for classification.
3. Bi-directional LSTM with character embedding in pytorch – no transfer learning
4. Language Modelling on character tokens (using UMLFIT) and classification using AWD-LSTM using Fastai
5. "Integrated gradients" has been used to derive local explanation for each inference to show attribution of each character towards prediction of the model.

Finally, a demo of language detection app (with and without explanation) is hosted on Heroku.

The work and specifically the Bi-directional LSTM model and the performance metrics used to compare the models have taken inspiration from the paper mentioned at [1].

Please see details for each step in the sections below.

Tech stack

Below table mentions basic information of the technology stack that is used to develop the models. This does not intend to cover the python package level details.

Programming language	Python 3.7
Development platform	Google Colab
Deep learning packages	Pytorch, Fastai
Deployment platform	Heroku

Data exploration

In this step, the data have been investigated to understand the structure of file, missing values, representation of different languages, etc. Key information from this step is summarised in below

Total Number of records	~9.9 million
Number of records where label is missing	15
Number of records where text is missing	0
Number of unique labels	402
Number of records in training after train – test split	~7.9 million

Summary statistics on number of records per language	`mean 1.977424e+04`
	`std 9.872701e+04`
	`min 1.000000e+00`
	`25% 2.200000e+01`
	`50% 7.650000e+01`
	`75% 1.239500e+03`
	`max 1.223235e+06`

Number of records for top 100 languages (by number of representation) in training	~7.86 million
Most represented language in training	English
Languages having more than or equal to 500 records in training	133

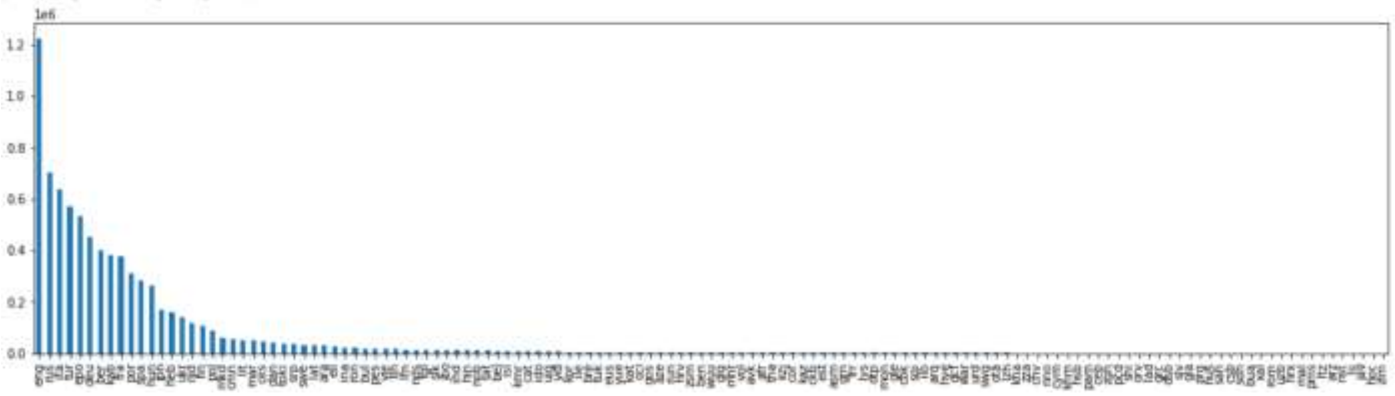
Number of records for top 133 languages (by number of representation) in training	~7.89 million

Summary statistics for number of characters in each sentence	`mean 3.532629e+01`
	`std 5.577729e+01`
	`min 1.000000e+00`
	`25% 2.300000e+01`
	`50% 3.100000e+01`
	`75% 4.100000e+01`
	`max 5.642300e+04`

If sentences are cut after 256 characters how many sentences will be truncated (from training data set)	9k (0.1%)

Please note following assumptions / decisions-

- The data initially is split into train and test (80%-20%). And then the train data is again split into train and validation (80-20% respectively)
- It is decided that the training process will only include languages that have more than or equal to 500 records in training (including validation) data. Therefore, the models are built on 133 languages
- Distribution of number of records for selected 133 languages below. After few experimentations, it is decided that training process will only use 10k of observations for each of the languages where they have more that 10k observations (i.e. down sampled). For languages having less than 10k observations, they will be kept as it is. (i.e. no up sampling)



Open-source python package for language detection

- There are open-source python packages available for language detection. However, they have limited support. I used "langid" package to understand performance of such a package for provided data.
- It is found that langid package supports 97 languages in total and there are 88 languages that are common with our data.
- Below is the performance metrics for this method based on the test data for the supported languages

Accuracy	Macro F1	Weighted F1
89%	0.57	0.91

However, this metrics is not used in final performance comparisons as it can only support a subset of languages selected for model building.

Baseline model – Naïve Bayes

This is used as the baseline model. Below list provides the main features of the model

- A sklearn pipeline is used with following components
  - character tokenizer
  - TF-IDF transformer for feature
  - Multinomial Naïve Bays
- Grid search with k-fold cross validation is used to select best parameters for the model
- Comparison of this base line model with other deep learning models are provided in the Model comparison section below

Pytorch model without transfer learning

This uses the model architecture as prescribed in the paper mentioned in [1]. Taken reference from [2] for coding the bi-lstm in pytorch

Below list provides the main features of the model.

- It uses datasets library for data pre-processing, i.e. character tokenisation, numericalization
- One embedding layer, one Bidirectioanl LSTM layer and one Linear layer is used as model architecture
- Dropouts are used after Bi-LSTM and Linear layer
- Cross entropy loss is used as loss function and Adam is used as optimizer
- 20 epochs were used for training.
- Comparison of this model with other models are provided in the Model comparison section below

Fastai models

Fastai model used the ULMFIT language modelling approach. However, it does not use any pre-trained language model, instead it creates character sequence based language model (i.e. the task of next character prediction) from scratch.

Below list provides the main features of the model.

- Fastai data block and data loader is used for character tokenisation, numericalization.
- It uses weight dropped LSTM (2 layers) for language modelling and classification. This technique was introduced in paper [3]
- I did not use the pre-trained wights as those were not relevant (pre-trained weights are relevant for word sequences) for character sequence. The encoder wight based on training the language model used to initialise the embedding layer of the classification model.
- It uses half precision training to allow reduced memory size and faster training. Fastai provides the ability as a callback.
- Optimum learning rates have been decided based on call-back provided by fastai (proposed in paper [4])
- One cycle policy is used for training where learning rate is increased in half of iterations (up to optimum LR provided) and then decreased (momentum goes the other way) in SGD optimiser. This is also inspired by paper [4] and implemented by fastai
- Used discriminative layer training for classification model training where different layer groups are trained using different learning rates. The layers toward the head will be trained using higher learning rates as they need more trained for the given task.

Please note, I tried building bi-directional AWD-LSTM for the classification model. However, did not include those results as there was not enough performance boost.

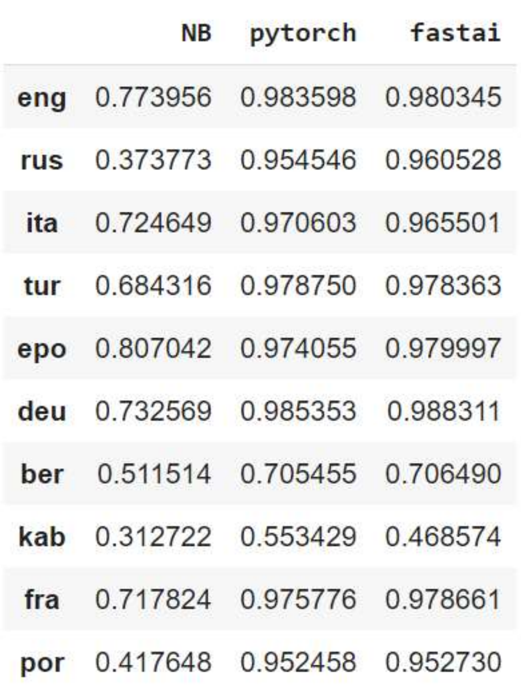
Model comparison

The models have been compared based on accuracy, macro f1 score, weighted f1 score, inference time and size on disk. The paper [1] has used accuracy, macro f1 score, weighted f1 score to provide different insights. Macro-F1 averages the per language results and considers languages equally; therefore, macro F1 score indicates language coverage. Weighted-F1 takes into account the popularity of the different languages in the data and thus indicates model performance on the data set. In multi-class classification, micro-F1 equals accuracy and accuracy is also presented in below table.

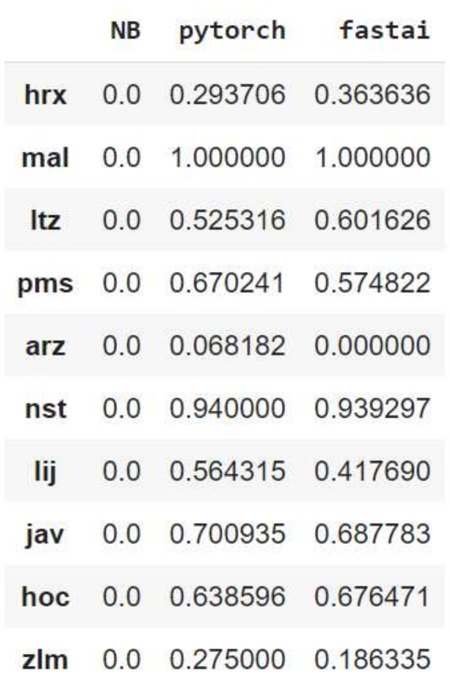
Model	Accuracy	Macro F1	Weighted F1	Inference time (on CPU)	Size on disk
Multinomial Naïve Bayes	52.5%	0.142	0.545	773 micro seconds	2 MB
Bi-LSTM in Pytorch	91.8%	0.759	0.922	223 milli seconds	4 MB
AWD-LSTM in Fastai	91.9%	0.759	0.92	188 milli seconds	155 MB

Additionally, the models have been compared based on F1 scores for individual languages for top 10 and bottom 10 languages with respect to number of records for each language.

Top 10 languages



	NB	pytorch	fastai
eng	0.773956	0.983598	0.980345
rus	0.373773	0.954546	0.960528
ita	0.724649	0.970603	0.965501
tur	0.684316	0.978750	0.978363
epo	0.807042	0.974055	0.979997
deu	0.732569	0.985353	0.988311
ber	0.511514	0.705455	0.706490
kab	0.312722	0.553429	0.468574
fra	0.717824	0.975776	0.978661
por	0.417648	0.952458	0.952730



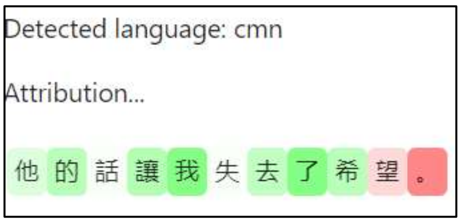
	NB	pytorch	fastai
hrx	0.0	0.293706	0.363636
mal	0.0	1.000000	1.000000
ltz	0.0	0.525316	0.601626
pms	0.0	0.670241	0.574822
arz	0.0	0.068182	0.000000
nst	0.0	0.940000	0.939297
lij	0.0	0.564315	0.417690
jav	0.0	0.700935	0.687783
hoc	0.0	0.638596	0.676471
zlm	0.0	0.275000	0.186335

Bottom 10 languages

Explainability

Local explainability is derived using Integrated gradients. Integrated Gradients compute gradient of the prediction output with respect to features of the input. It needs a baseline token against which the attributions are calculated. In this case the token for unknown character is used as baseline to calculate the gradients.

A typical explanation will look like following. Green colour indicates positive attribution and red colour indicates negative attribution. Darker the colour, higher the strength.



Detected language: cmn

Attribution...

他 的 話 讓 我 失 去 了 希 望 。

Deployment

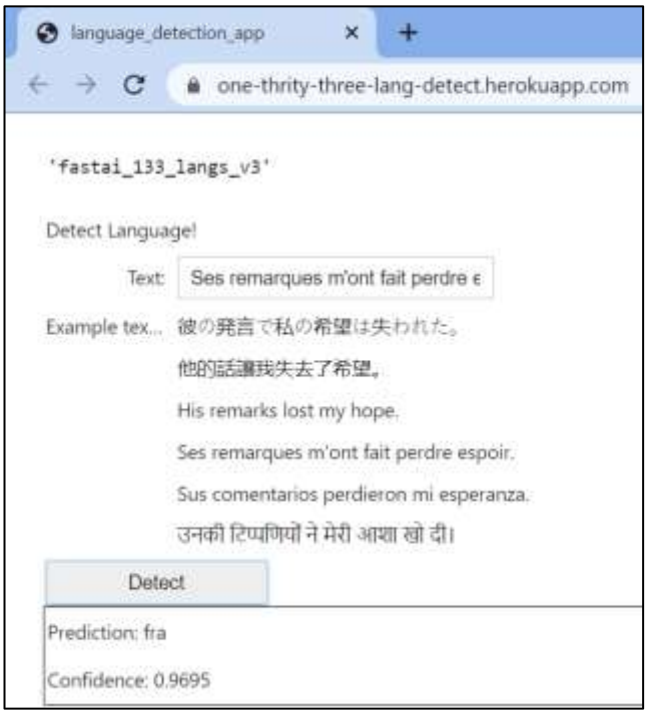
This section covers the apps that are deployed for demo purposes only and the deployment strategy that can be considered for actual implementation

Online demo

For demonstration, couple of apps are deployed on Heroku platform using jupyter notebook, ipython widgets and voila (reference [6]).

App name	Description	Comments
Language detection	App for language detection with confidence score	Loading of the page will take little longer for the first time as it will download the model from Google drive.
Language detection with explanation	App for showing language detection attribution.	This is built as a separate app as installation of all required packages is causing the app the crash intermittently due to memory restrictions in Heroku (500MB for free tier). If the app crashes, it might help to reload the page

App screenshots below.


language_detection_app
one-thrity-three-lang-detect.herokuapp.com

'fastai_133_langs_v3'

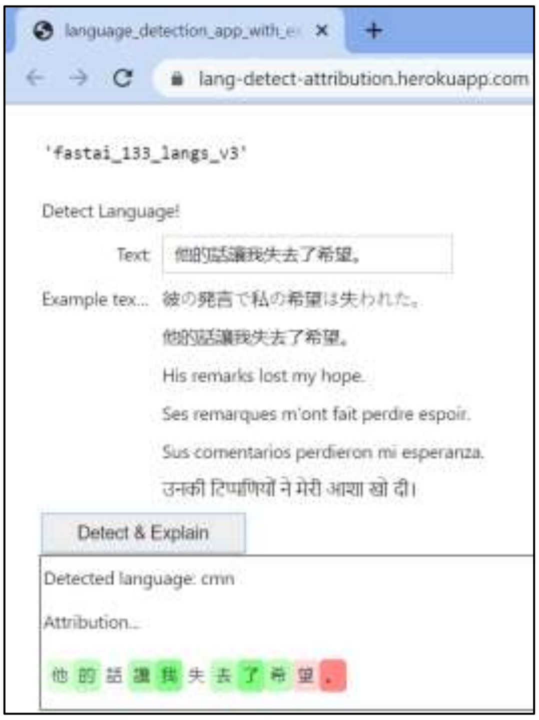
Detect Language!

Text: Ses remarques m'ont fait perdre e

Example tex... 彼の発言で私の希望は失われた。
他的話讓我失去了希望。
His remarks lost my hope.
Ses remarques m'ont fait perdre espoir.
Sus comentarios perdieron mi esperanza.
उनकी टिप्पणियों ने मेरी आशा खो दी।

Prediction: fra

Confidence: 0.9695


language_detection_app_with_e
lang-detect-attribution.herokuapp.com

'fastai_133_langs_v3'

Detect Language!

Text: 他的話讓我失去了希望。

Example tex... 彼の発言で私の希望は失われた。
他的話讓我失去了希望。
His remarks lost my hope.
Ses remarques m'ont fait perdre espoir.
Sus comentarios perdieron mi esperanza.
उनकी टिप्पणियों ने मेरी आशा खो दी।

Detected language: cmn

Attribution...

他 的 話 讓 我 失 去 了 希 望 。

Implementation strategy

This section discusses the model deployment strategy based on following considerations.

- The model prediction will be used as part of a bigger application, therefore the deployment should provide an end-point to retrieve detected language (and the confidence score) based on the text provided.
- Deployment of prediction pipeline is considered. Deployment strategies would be different for training pipeline.
- The model will need to be served in real time (i.e. synchronous) and detection will need to happen on individual sentences. Batch inference has not been considered.
- The model can be served form a server. Deployment in edge devices is not considered.

Considering the above assumptions, I would like to propose following deployment strategy

1. Build a REST service wrapper on the prediction function using FastAPI or Flask
2. Build a docker container that includes the model and all dependencies and exposes the rest service as the end point.
3. Publish the docker container in the container registry of the selected cloud platform. Selection of cloud platform should consider ability to auto scale (horizontal scaling) and pricing of each instance.
4. Computing instance needed for serving the model does not need GPU. It needs a reasonable CPU and needs at least 4GB of RAM (I have seen the RAM size to grow up to 2 GB in the Heroku instance for language detection app).
5. Deploy the container on the cloud instance. There should be a load balancer on top of the deployed instances and the applications that need integration will connect to the rest service via the load balancer.

I would also like the following to be built / considered for monitoring and management of the prediction pipeline post deployment. However, implementation of these would need understanding of the context in which language detection will be used.

1. Feedback loop to understand trend in detection error per language
2. Drift detection – data and concept drift. If possible, performance drift.
3. Bias detection – focused towards latent bias due to per-trained weights (from language model).
4. Retraining pipeline – to be triggered based on frequency and / or detected performance drift

Submitted files

Github repo: https://github.com/sayanbanerjee32/tatoeba_language_detection

All jupyter notebooks and corresponding HTML files are present in this repo.

Sl. No.	File name	Description
1	Data_exploration.ipynb	This includes data exploration code
2	langid_baseline.ipynb	This is the code for using langid.py – the open-source python package on the test data.
3	ML_baseline_133_langs.ipynb	Sklearn pipeline with multinomial naïve bayes model
4	pytorch_133_langs.ipynb	Bidirectional LSTM using pytorch
5	fastai_133_langs.ipynb	AWD-LSTM using fastai
6	analysis_of_models.ipynb	Comparison of performance metrics and inference time for all 3 models
7	local_explanation_fastai_133_langs.ipynb	Local explanation with Integrated gradients
8	language_detection_app.ipynb	App for language detection (only)
9	language_detection_app_with_explanation.ipynb	App for showing attribution of each character towards prediction from model

Conclusion and scope of further improvements

We can see the performance of both the deep learning models are significantly better than the baseline Multinomial Naïve Bayes model (and better than the performance of lang id, but comparison is not apt).

The model performance with respect to accuracy, F1 scores and inference timing, the bi-LSTM and AWD-LSTM are almost similar. Due to ease of development, debugging and building inference pipeline, I have chosen the Fastai model for creating the language detection and explanation apps on Heroku.

However, if the need is to deploy the model on edge devices (or needs a very small memory footprint), it is recommended to use the Bi-LSTM model in pytorch.

I would like to try following approaches to see if any of them can give better performance with respect to accuracy / F1 scores and inference time and memory foot print.

1. Use of sub-word tokenization (e.g., SentencePiece, Byte-Pair encoding) instead of character tokenization. In that case, we shall train a sub-word tokenizer form scratch so that it can deal with all the languages together. Current pre-trained sub-word tokenizers are trained for single languages.
2. Would like to investigate performance of a transformer-based language model. Could not find any pre-trained models for language identification. Therefore, it would need to be trained from scratch.
3. Seeing the memory requirement of the pytorch model, would like to invest more time fine tuning the model using the techniques that fastai offers (fastai supports custom pytorch model as the base model).
4. Would like to increase coverage of languages. Instead of using 500 records as threshold for selecting a language, would like to see how overall performance of the model varies if I set the threshold as 100, 200 and 300 records.

References

1. A reproduction of Apple's bi-directional LSTM models for language identification in short strings - https://arxiv.org/pdf/2102.06282v1.pdf
2. Pytorch code reference - https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/2_lstm.ipynb
3. AWD-LSTM - https://arxiv.org/abs/1708.02182
4. Cyclical Learning Rates for Training Neural Networks - https://arxiv.org/pdf/1506.01186.pdf
5. Integrated gradients for fastai v1 - https://github.com/MichaMucha/awdlstm-integrated-gradients/blob/master/Interactive.ipynb
6. Heroku using voila - https://towardsdatascience.com/creating-interactive-jupyter-notebooks-and-deployment-on-heroku-using-voila-aa1c115981ca