2021

NATURAL LANGUAGE PROCESSING (NLP) FOR LANGUAGE IDENTIFICATION

Darius Singh
1720130
B.Sc.(Hons.) Mathematics
darius2121.ds@gmail.com

02

CONTENTS

03

SUMMARY:

In this project, we create Artificial Neural Networks (ANNs) that take audio and text as input and identify the language the input is in.

We implement this using Python and its various open-source libraries that allow us to preprocess our data, create the model, train the model, test the model and deploy the model.

We use the 'Spoken Language Identification' dataset and the 'Language Identification' dataset available on Kaggle.

04

INTRODUCTION:

Natural language refers to how humans communicate with one another. Namely, speech and text.

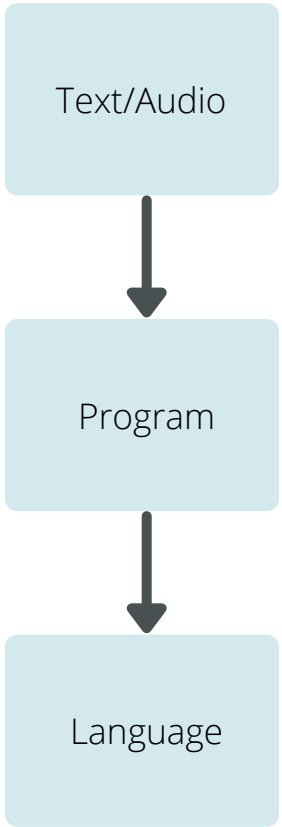
The task of identifying natural language appears frequently in web applications. Users want websites that are relevant to them, and search engines want to help them find them. Content that is difficult to understand is automatically deemed less relevant. Knowing the source language is critical for machine translation, sentiment analysis, and text summarization algorithms.

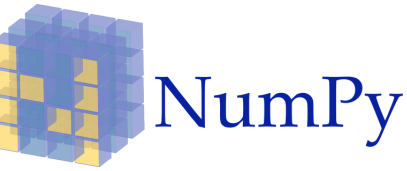
There are several approaches to developing a program that can recognize the language of a certain text document or audio, but machine learning appears to be the most efficient and accurate.

05

THE GOAL OF THE STUDY:

The project's goal is to create a model/program that can take any text or audio as input and return an output that identifies the language it is in.





MATERIALS AND METHODS:

REQUIREMENTS:

- LANGUAGE IDENTIFICATION DATASET
- SPOKEN LANGUAGE IDENTIFICATION DATASET
- NUMPY
- PANDAS
- MATPLOTLIB
- SEABORN
- JSON
- SKLEARN
- NLTK
- WORDCLOUD
- TENSORFLOW
- LIBROSA
- TQDM
- OS
- IPYTHON

TEXT DETECTION MODEL:

IMPORTING DEPENDENCIES

We import all the dependencies required to build the model and configure GPU/CPU to be used with TensorFlow.

Importing Dependencies

In [8]:
```
import numpy as np # linear algerbra
import pandas as pd # data processing
import matplotlib.pyplot as plt # plotting
import seaborn as sns # plotting
import json # data processing(.json)

from sklearn.preprocessing import LabelEncoder # Creates placeholders for categorical variables
from sklearn.feature_extraction.text import CountVectorizer # converts text into vector matrix
from sklearn.model_selection import train_test_split # split data into training and testing sets
from sklearn.naive_bayes import MultinomialNB # ML model for naive bayes
from sklearn.metrics import accuracy_score, confusion_matrix # measure the accuracy of the model
from sklearn.metrics import classification_report # classification report of the model

import re # NLP
import nltk # natural langauge processing
from nltk.tokenize import word_tokenize # tokenizer
from nltk.stem import PorterStemmer # stemmer
from nltk.corpus import stopwords # stopwords

from wordcloud import WordCloud # create word cloud images of text

import tensorflow as tf # create neural networks
from tensorflow.keras import Sequential # create squential NN model
from tensorflow.keras.layers import Dense # implements the operation: output = activation(dot(input, kernel) + bias)
from tensorflow.keras.utils import plot_model # plot model architecture
from tensorflow.keras.callbacks import EarlyStopping # early stopping of training
from tensorflow.keras.models import load_model # load saved model

from sklearn.model_selection import GridSearchCV # hyperparameter optimization
from sklearn.model_selection import RandomizedSearchCV # hyperparameter optimization
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier # linking keras model to sklearn

%matplotlib inline
# With this backend, the output of plotting commands is displayed inline within frontends
# like the Jupyter notebook, directly below the code cell that produced it. The resulting
# plots will then also be stored in the notebook document
```

We configure CPU/GPU for computation in tensorflow.

In [9]:
```
# ensure CPU is enabled for tensorflow
# skip this step if you have never configured GPU for tensorflow

import os

os.environ['CUDA_VISIBLE_DEVICES'] = '-1' # 0 for GPU
                                          # -1 for CPU
if tf.test.gpu_device_name():
    print('GPU found')
else:
    print("No GPU found")
```

```
No GPU found
```

08

LOADING THE DATA

We upload the 'Language Identification Dataset' available on Kaggle. The data is in .csv format and is extracted from WiLI-2018, the Wikipedia language identification benchmark dataset which contains 235000 paragraphs of 235 languages.

We use a dataset that contains 22 selective languages from the original dataset which includes the following languages:

- English
- Arabic
- French
- Hindi
- Urdu
- Portuguese
- Persian
- Pushto
- Spanish
- Korean
- Tamil
- Turkish
- Estonian
- Russian
- Romanian
- Chinese
- Swedish
- Latin
- Indonesian
- Dutch
- Japanese
- Thai

In [11]:
```
# read dataset which in .csv format

data = pd.read_csv('dataset.csv', encoding='utf-8').copy() # creates a dataframe of a copy of the dataset
                                                           # utf-8 encoding use to be able to read text in other langauge

print(data.shape)  # shape of the dataset
data.head()        # first 5 rows of the dataset
```

```
(22000, 2)
```

Out[11]:

	Text	language
0	kiement gottwaldi surnukeha palsameeriti ning ...	Estonian
1	sebes joseph pereira thomas på eng the jesuit...	Swedish
2	ถนนเจริญกรุง อักษรโรมัน thanon charoen krung ւ...	Thai
3	விசாகப்பட்டினம் தமிழ்ச்சங்கத்தை இந்துப் பத்திர...	Tamil
4	de spons behoort tot het geslacht haliclona en...	Dutch

In [12]:
```
data.tail() # last 5 rows of the dataset
```

Out[12]:

	Text	language
21995	hors du terrain les années et sont des année...	French
21996	ใน พศ หลักจากที่เสด็จประพาสแหลมมลายู ชวา อินเ...	Thai
21997	con motivo de la celebración del septuagésimoq...	Spanish
21998	年月，當時還只有歲的她在美國出道，以mai-k名義推出首張英文《baby i like》，由...	Chinese
21999	aprilie sonda spațială messenger a nasa și-a ...	Romanian

In [13]:
```
# categories in our target variable

data['language'].unique()
```

Out[13]:
```
array(['Estonian', 'Swedish', 'Thai', 'Tamil', 'Dutch', 'Japanese',
       'Turkish', 'Latin', 'Urdu', 'Indonesian', 'Portugese', 'French',
       'Chinese', 'Korean', 'Hindi', 'Spanish', 'Pushto', 'Persian',
       'Romanian', 'Russian', 'English', 'Arabic'], dtype=object)
```

We see the 22 different categories that we will be classifiying the dataset into.

PREPROCESSING THE DATA: NULL AND DUPLICATE VALUES

We check for null values in our dataset and drop any duplicate samples in our dataset.

Preprocessing the data

In [14]:
```
# checking for null values

data.isnull().sum()
```

Out[14]:
```
Text        0
language    0
dtype: int64
```

In [15]:
```
# checking the datatype of the features

data.dtypes
```

Out[15]:
```
Text        object
language    object
dtype: object
```

In [16]:
```
# number of of samples per language (category)

data['language'].value_counts()
```

Out[16]:
```
Turkish       1000
Pushto        1000
Latin         1000
Hindi         1000
Spanish       1000
Russian       1000
Swedish       1000
Indonesian    1000
English       1000
Persian       1000
Arabic        1000
Portugese     1000
Urdu          1000
Thai          1000
Korean        1000
French        1000
Tamil         1000
Romanian      1000
Chinese       1000
Estonian      1000
Japanese      1000
Dutch         1000
Name: language, dtype: int64
```

The dataset contains 1000 samples for each language, hence are dataset has a total of 22000 rows.

In [17]:
```
# dropping duplicate samples

data = data.drop_duplicates(subset='Text')
data = data.reset_index(drop=True)
```

In [18]:
```
# rechecking the number of samples per language

data['language'].value_counts()
```

Out[18]:
```
Turkish       1000
Japanese      1000
Chinese       1000
Romanian      1000
English       1000
Korean        1000
Thai          1000
Urdu          1000
Persian       1000
Estonian       999
Russian        999
Arabic         998
Portugese      997
Spanish        996
Dutch          996
Pushto         993
Swedish        992
French         990
Hindi          990
Tamil          981
Indonesian     975
Latin          953
Name: language, dtype: int64
```

We see that some of the languages had text that had been repeated and hence we dropped those samples.

PREPROCESSING THE DATA: REMOVING STOPWORDS

Stop words are words that we want to ignore, so we filter them out of our text when we're processing it. Very common words like 'in', 'is', and 'an' are often used as stop words since they don't add a lot of meaning to a text in and of themselves.

We create a list of stopwords for each language and create a list of alphanumeric characters and combine them to create a list containing all stopwords.

We obtain stopwords through 3 formats :

- Through NLTK
- In a .txt dataset
- In a .json dataset

We apply different methods for extracting stopwords from each of the 3 formats.

We create a list of stopwords for each language and create a list of alphanumeric characters and combine them to create one list of list containing all stopwords.

We obtain stopwords through 3 formats :

- Through NLTK
- In a .txt dataset
- In a .json dataset We apply different methods for extracting stopwords from each of the 3 formats.

In [21]:

```
# creating lists of language wise stopwords
# for langauges supported by NLTK, stopwords can be easily be obtained through corpus
# for language not supported by NLTK, stopwords are obtained through .txt or .json datasets

arabic_stopwords = stopwords.words('arabic')

chinese_stopwords = pd.read_csv('stopwords-chinese.txt', sep='\n', index_col=False, squeeze=True).tolist()

dutch_stopwords = stopwords.words('dutch')

english_stopwords = stopwords.words('english')

estonian_stopwords = pd.read_csv('estonian-stopwords.txt', sep='\n', index_col=False, squeeze=True).tolist()

french_stopwords = stopwords.words('french')

hindi_stopwords = pd.read_csv('hindi-stopwords.txt', sep='\n', index_col=False, squeeze=True).tolist()

indonesian_stopwords = stopwords.words('indonesian')

japanese_stopwords = pd.read_csv('japanese-stopwords.txt', sep='\n', index_col=False, squeeze=True).tolist()

korean_stopwords = pd.read_csv('korean-stopwords.txt', sep='\n', index_col=False, squeeze=True).tolist()

latin_stopwords = pd.read_csv('latin-stopwords.txt', sep='\n', index_col=False, squeeze=True).tolist()

persian_stopwords = pd.read_csv('persian-stopwords.txt', sep='\n', index_col=False, squeeze=True,quoting=3 , error_bad_lines = False).tolist()

portuguese_stopwords = stopwords.words('portuguese')

file = open('pashto-stopwords.json', 'r')
pushto_stopwords_dict = json.load(file)
pushto_stopwords = []
for i in range(len(pushto_stopwords_dict['word'])):
    pushto_stopwords.append(pushto_stopwords_dict['word'][str(i)])
file.close()

romanian_stopwords = stopwords.words('romanian')

russian_stopwords = stopwords.words('russian')

spanish_stopwords = stopwords.words('spanish')

swedish_stopwords = stopwords.words('swedish')

tamil_stopwords = pd.read_csv('tamil-stopwords.txt', sep='\n', index_col=False, squeeze=True).tolist()

thai_stopwords = pd.read_csv('thai-stopwords.txt', sep='\n', index_col=False, squeeze=True).tolist()

turkish_stopwords = stopwords.words('turkish')

urdu_stopwords = pd.read_csv('urdu-stopwords.txt', sep='\n', index_col=False, squeeze=True).tolist()
```

11

PREPROCESSING THE DATA: REMOVING STOPWORDS, TOKENIZING AND STEMMING

Stemming is a text processing task in which we reduce words to their root, which is the core part of a word. For example, the words “helping” and “helper” share the root “help.”

By **Tokenizing**, we can conveniently split up the text by word or by sentence. This will allow us to work with smaller pieces of text that are still relatively coherent and meaningful even outside of the context of the rest of the text. It’s our first step in turning unstructured data into structured data, which is easier to analyze.

We create a function that takes text as input and preprocesses it to give clean text as output.
The function clean_text():

- tokenizes the text into a list of words
- lowers all words into lowercase
- removes stopwords from the list of lowered words
- stems the words
- returns them in form of a string separated by " "

In [22]:
```
# combining stopwords of all langauges

stopwords = arabic_stopwords + chinese_stopwords + dutch_stopwords + english_stopwords + estonian_stopwords + french_stopwords + hindi_stopwords + indonesian_stopwords + japanes
```

In [23]:
```
# adding nonalphanumeric char to stopwords

nonalphanumeric = ['\'', '.', ',', '\"', ':', ';', '!', '@', '#', '$', '%', '^', '&',
                   '*', '(', ')', '-', '_', '+', '=', '[', ']', '{', '}', '\\', '?',
                   '/','>', '<', '|', ' ']

stopwords = stopwords + nonalphanumeric
```

In [24]:
```
# total stop words

len(stopwords)
```

Out[24]: 16023

Our stopwords list contains a total of 16023 stopwords that will be reomoved form the dataset.

We create a function that takes text as input and preprocesses it to give clean text as output.

The function clean_text():

- tokenizes the text into a list of words
- lowers all words into lowercase
- removes stopwords from the list of lowered words
- stems the words
- returns them in form of a string seperated by " "

In [25]:
```
def clean_text(text):
    """
    takes text as input and returns cleaned text after tokenization,
    stopwords removal and stemming
    """
    tokens = word_tokenize(text) # creates text into list of words
    words = [word.lower() for word in tokens if word not in stopwords] # creates a list with words which are not stopwords
    words = [PorterStemmer().stem(word) for word in words] # stems(remove suffixes and prefixes)  words
    return " ".join(words) # joins the list of cleaned words into a sentence string
```

In [26]:
```
# applying clean_text function to all rows in 'Text' column

data['clean_text'] = data['Text'].apply(clean_text)
```

PREPROCESSING THE DATA: LABEL ENCODING THE TARGETS

The computer cannot understand text data, it can only understand numbers. Hence we provide placeholder values to each language category using the label encoder.

The **Label Encoder** encodes target labels with values between 0 and n_classes-1.

Computers cannot understand text data, they only understand numbers. Hence we provide placeholder numbers to each language category using the label encoder.

In [27]:
```
# using LabelEncoder to get placeholder number values for categorical variabel 'language'

le = LabelEncoder()
data['language_encoded'] = le.fit_transform(data['language'])
data.head()
```

Out[27]:

	Text	language	clean_text	language_encoded
0	klement gottwaldi surnukeha palsameeriti ning ...	Estonian	klement gottwaldi surnukeha palsameer paigutat...	4
1	sebes joseph pereira thomas på eng the jesuit...	Swedish	sebe joseph pereira thoma eng jesuit sino-russ...	17
2	ถนนเจริญกรุง อักษรโรมัน thanon charoen krung ...	Thai	ถนนเจริญกรุง อักษรโรมัน thanon charoen krung ...	19
3	விசாகப்பட்டினம் தமிழ்ச்சங்கத்தை இந்துப் பத்திர...	Tamil	விசாகப்பட்டினம் தமிழ்ச்சங்கத்தை இந்துப் பத்திர...	18
4	de spons behoort tot het geslacht haliclona en...	Dutch	spon behoort geslacht haliclona behoort famili...	2

In [28]:
```
# list of languages encoded with thier respective indices representing their placeholder numbers

lang_list = [i for i in range(22)]
lang_list = le.inverse_transform(lang_list)
lang_list = lang_list.tolist()
lang_list
```

Out[28]:
```
['Arabic',
 'Chinese',
 'Dutch',
 'English',
 'Estonian',
 'French',
 'Hindi',
 'Indonesian',
 'Japanese',
 'Korean',
 'Latin',
 'Persian',
 'Portugese',
 'Pushto',
 'Romanian',
 'Russian',
 'Spanish',
 'Swedish',
 'Tamil',
 'Thai',
 'Turkish',
 'Urdu']
```

lang_list contains a list of all encoded languages with thier indices as thier placeholder values.

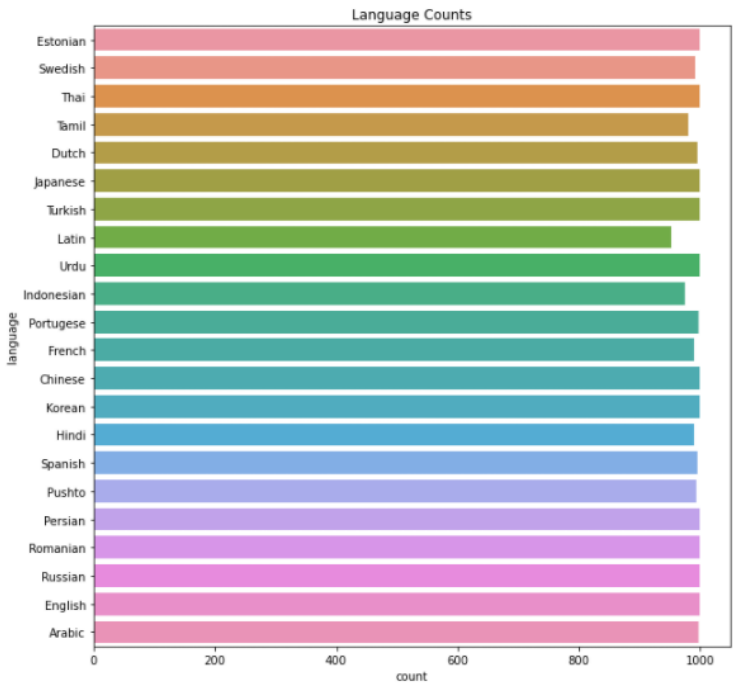
EXPLORING THE DATA: LANGUAGE FREQUENCY PLOT

We create a language-wise frequency distribution for the number of samples for each language.

The frequency plot of each language shows us the number of samples we have for each language. All languages except Estonian, Russian, Arabic, Portuguese, Dutch, Spanish, Pushto, Swedish, Hindi, French, Tamil, Indonesian , and Latin have 1000 samples.

Exploring the data

In [22]:
```
# plotting a language-wise freqeuncy distribtion for number of samples in each language

plt.figure(figsize=(10,10))
plt.title('Language Counts')
ax = sns.countplot(y=data['language'], data=data)
plt.show()
```



The frequency plot of each language shows us the amount of samples we have for each language. All languages except Estonian, Russian, Arabic, Portugese, Dutch, Spanish, Pushto, Swedish, Hindi, French, Tamil, Indonesian and Latin have 1000 samples.

EXPLORING THE DATA: WORD CLOUD

We create word clouds for each language to explore further into our data.

We do so by creating a function that takes in language and font_path(path of the folder with font in it for languages in different scripts) and creates a word cloud of that language using text from our dataset.

For languages in scripts other than **Latin script**(English is in Latin script) or **Arabic script**(Urdu is in Arabic script), download a font for the language in .ttc or .ttf format, paste it in C:\Windows\Fonts and mention the path in font_path.

This is where someone with domain knowledge for each language can step in help further improve the dataset.

Word Clouds are graphical representations of word frequency that give greater prominence to words that appear more frequently in a source text.

We create a function that takes in language and font_path(path of folder with font in it for languages in different scripts) and creates a wordcloud of that language using text from our dataset.

For languages in scripts other than Latin script(english is in latin script) or Arabic script(urdu is in arabic script), download a font for the language in .ttc or .ttf format, paste it in C:\Windows\Fonts and mention the path in font_path.

In [23]:
```
def word_cloud(language, font_path='arial'):
    """
    takes in langauge and font_path(folder where font is saved) as input and returns wordcloud
    """
    data_language = data[data['language']==language]
    wordcloud = WordCloud(font_path=font_path, background_color="white").generate(' '.join(data_language['clean_text'].tolist()))
    fig = plt.figure(figsize=(16,8),)
    fig.suptitle(f'Word Cloud {language}', fontsize=40)
    plt.axis('off')
    plt.imshow(wordcloud)
```

In [24]:
```
word_cloud('Arabic')
```

Word Cloud Arabic

EXPLORING THE DATA: WORD CLOUD

While creating the word cloud for the Chinese language we notice that there are few English words in the samples too. Hence we remove the English words from our Chinese samples.

We do so by creating a function that takes text as input and returns text without any words in Latin script.

In [25]:
```
word_cloud('Chinese', font_path=r'C:\Windows\Fonts\msyh.ttc')
```

We see that there are english words present in the chinsese text and hence we will remove them.

We create a function that takes text as input and removes all words in latin script from it.

In [29]:
```
def remove_english(text):
    """
    function that takes text as input and returns text without english words
    """
    pat = "[a-zA-Z]+"
    text = re.sub(pat, "", text)
    return text
```

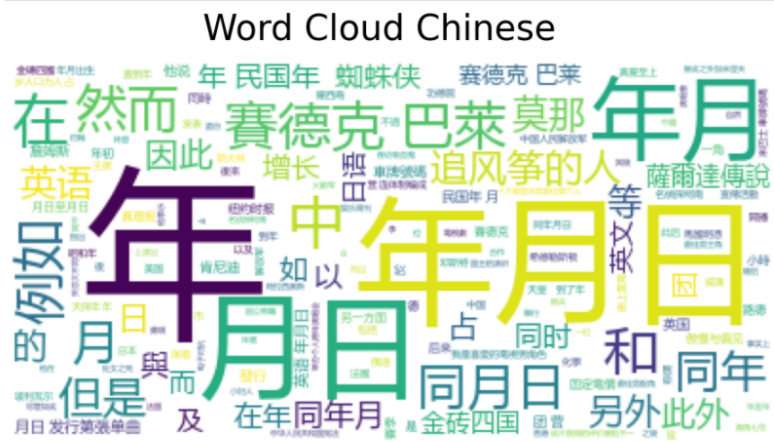
In [30]:
```
data_Chinese = data[data['language']=='Chinese'] # Chinese data in dataset

clean_text = data.loc[data.language=='Chinese']['clean_text']
clean_text = clean_text.apply(remove_english) # removing english words

data_Chinese.loc[:,'clean_text'] = clean_text
```

In [31]:
```
# removing old chinese text and appending new cleaned chinese text

data.drop(data[data['language']=='Chinese'].index, inplace=True, axis=0)
data = data.append(data_Chinese)
```

We re-shuffle the dataset to have train and test samples representative of the entire dataset.

In [32]:
```
# shuffling dataframe and resetting index

data =data.sample(frac=1).reset_index(drop=True)
```

In [30]:
```
word_cloud('Chinese', font_path=r'C:\Windows\Fonts\msyh.ttc')
```



EXPLORING THE DATA: WORD CLOUD

Similarly, we create word clouds for all the languages in our dataset.

In [31]:
```
word_cloud('Dutch')
```

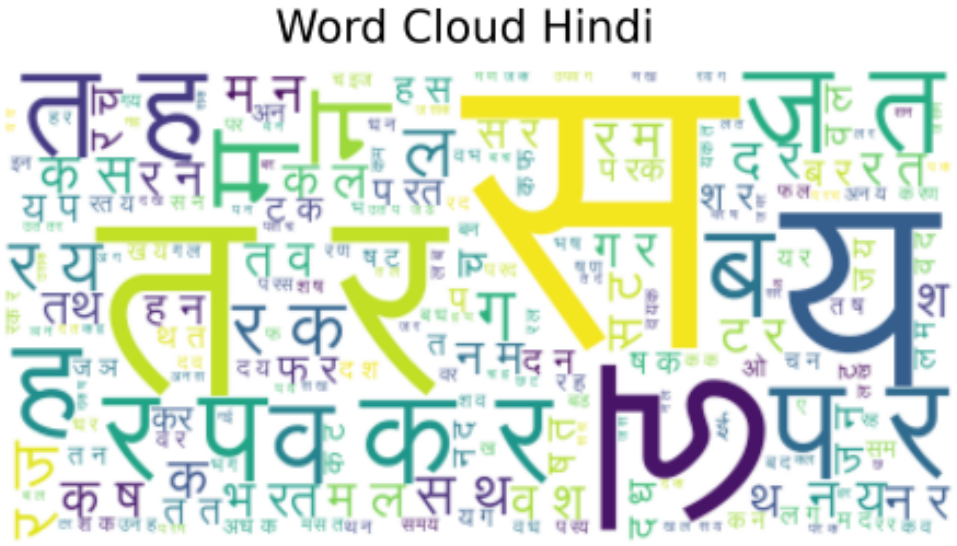
In [32]:
```
word_cloud('English')
```

EXPLORING THE DATA: WORD CLOUD

In [33]:
```
word_cloud('Estonian')
```

In [34]:
```
word_cloud('French')
```

In [35]:
```
word_cloud('Hindi', r'C:\Users\dariu\AppData\Local\Microsoft\Windows\Fonts\Lohit-Devanagari.ttf')
```



EXPLORING THE DATA: WORD CLOUD

In [36]:
```
word_cloud('Indonesian')
```

Word Cloud Indonesian

In [37]:
```
word_cloud('Japanese', r'C:\Windows\Fonts\msgothic.ttc')
```

Word Cloud Japanese

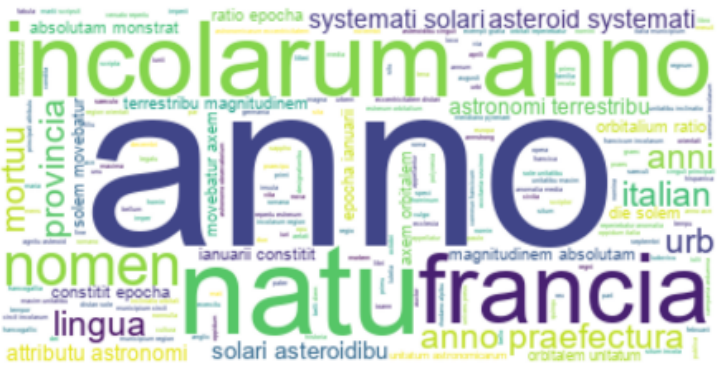
In [38]:
```
word_cloud('Korean', r'C:\Users\dariu\AppData\Local\Microsoft\Windows\Fonts\CookieRun Regular.ttf')
```

Word Cloud Korean

EXPLORING THE DATA: WORD CLOUD

```
In [39]: word_cloud('Latin')
```

Word Cloud Latin




```
In [40]: word_cloud('Persian')
```

Word Cloud Persian

```
In [41]: word_cloud('Portugese')
```

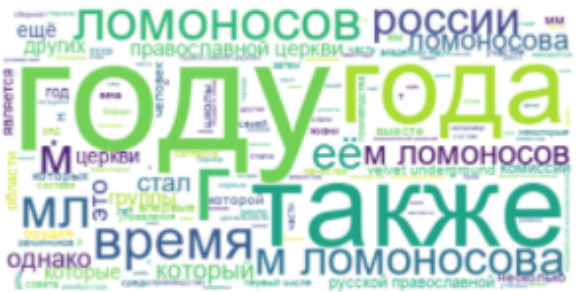
Word Cloud Portugese

EXPLORING THE DATA: WORD CLOUD

Word Cloud Pushto

Word Cloud Romanian

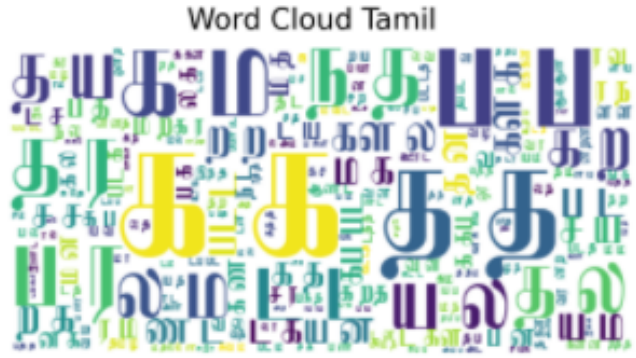
Word Cloud Russian



Word Cloud Spanish

Word Cloud Swedish

EXPLORING THE DATA: WORD CLOUD



SPLITTING THE DATA INTO INPUTS AND TARGETS:

We split the dataset into X (independent) variable or inputs and y (dependent) variable or targets.
We use Count Vectorizer to transform the input text into a sparse matrix based on the frequency(count) of each word that occurs in the entire text.

CountVectorizer (also known as **Bag of Words**) creates a matrix in which each unique word is represented by a column of the matrix, and each text sample from the document is a row in the matrix. The value of each cell is nothing but the count of the word in that particular text sample.

We convert the datatype from int64 to uint8 which consumes 8 times fewer bytes in comparison.
A **UINT8** is an 8-bit unsigned integer (range: 0 through 255 decimal).

We split the dataset into training and testing datasets, using 20% of the dataset for testing and the rest for training.

Splitting into inputs and targets

We split the dataset into X (independent) variable or inputs and y (dependent) variable or targets.

We use Count Vectorizer to transform the input text into a sparse matrix on the basis of the frequency(count) of each word that occurs in the entire text.

CountVectorizer (also known as Bag of Words) creates a matrix in which each unique word is represented by a column of the matrix, and each text sample from the document is a row in the matrix. The value of each cell is nothing but the count of the word in that particular text sample.

In [33]:
```
# defining input variable
# vectorizing input varible 'clean_text' into a matrix

x = data['clean_text']

cv = CountVectorizer() # ngram_range=(1,2)
x = cv.fit_transform(x)

# changing the datatype of the number into uint8 to consume less memory
x = x.astype('uint8') # uint8 and float32
```

We convert the datatype from int64 to uint8 which consumes 8 times less bytes in comparison.

A UINT8 is an 8-bit unsigned integer (range: 0 through 255 decimal).

In [34]:
```
# defining target variable

y = data['language_encoded']
```

We split the dataset into training and testing dataset, using 20% of the dataset for testing and rest for training.

In [35]:
```
# splitting data into training and testing datasets

x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.2)
```

CREATING A BASELINE NAIVE BAYES MODEL:

We create a baseline model to compare our Multilayer Perceptron Neural Network against.

MultinomialNB is suitable for classification with discrete features and hence works well with our given dataset.

We obtain an accuracy of **95.74%** with our Multinomial Naive Bayes model.

Creating a Naive Bayes model

We create a baseline model to compare our Multilayer Perceptron Neural Network against.

MultinomialNB is suitable for classification with discrete features and hence works well with our given dataset.

In [36]:
```
# fitting the Multinomial Naive Bayes model

NB_model = MultinomialNB()
NB_model.fit(x_train, y_train)
```

Out[36]:
```
MultinomialNB()
```

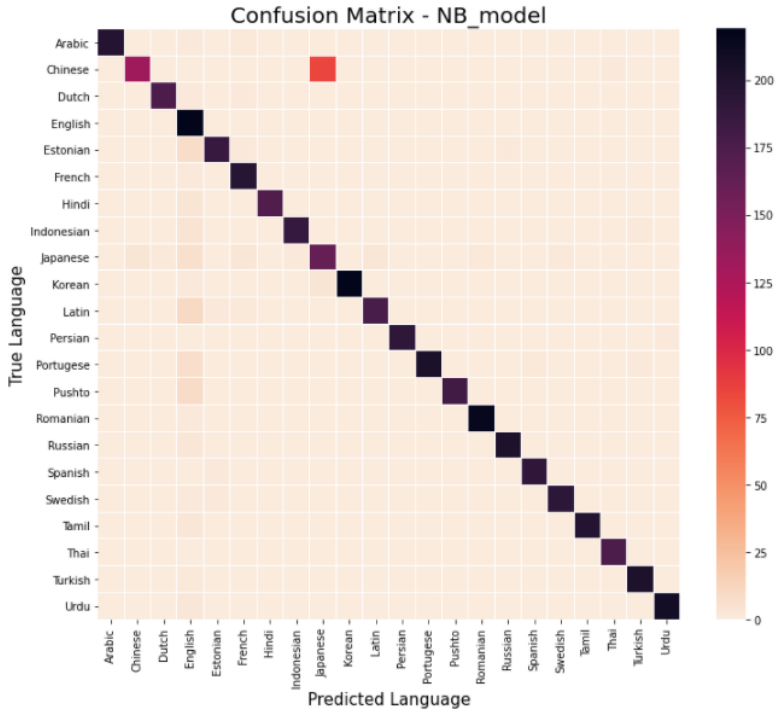
In [37]:
```
# predicting using the naive bayes model

y_pred = NB_model.predict(x_test)
```

In [38]:
```
# accuracy of the naive bayes model

accuracy_score(y_test, y_pred)
```

Out[38]:
```
0.9574565416285453
```

In [39]:
```
# creating confusion matrix heatmap

cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(12,10))
plt.title('Confusion Matrix - NB_model', Fontsize=20)
sns.heatmap(cm, xticklabels=lang_list, yticklabels=lang_list, cmap='rocket_r', linecolor='white', linewidth=.005)
plt.xlabel('Predicted Language', fontsize=15)
plt.ylabel('True Language', fontsize=15)
plt.show()
```

```
<ipython-input-39-2c6d0fa9351d>:6: MatplotlibDeprecationWarning: Case-insensitive properties were deprecated in 3.3 and support will be removed two minor releases later
  plt.title('Confusion Matrix - NB_model', Fontsize=20)
```



CREATING A BASELINE NAIVE BAYES MODEL: CLASSIFICATION REPORT

Looking at the **confusion matrix** we see that the model predicted Chinese as Japanese an unusual number of times, this could be due to the similarity between the **Hanzi Script (Chinese)** and the **Kanji Script (Japanese)**.

Precision attempts to answer what proportion of positive identifications were actually correct?

Recall attempts to answer what proportion of actual positives were identified correctly?

[40]:
```
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	1.00	0.99	0.99	201
1	0.97	0.61	0.75	218
2	0.99	0.99	0.99	177
3	0.77	1.00	0.87	219
4	0.98	0.95	0.97	197
5	0.96	0.99	0.98	197
6	1.00	0.98	0.99	177
7	0.99	0.97	0.98	194
8	0.65	0.88	0.75	182
9	1.00	0.98	0.99	224
10	0.98	0.93	0.95	191
11	1.00	0.99	1.00	193
12	1.00	0.95	0.98	213
13	1.00	0.95	0.98	190
14	0.99	1.00	0.99	216
15	1.00	0.99	0.99	205
16	0.99	0.99	0.99	192
17	0.99	0.99	0.99	195
18	1.00	0.99	0.99	201
19	0.99	1.00	1.00	175
20	0.99	0.99	0.99	204
21	1.00	0.99	0.99	211
accuracy			0.96	4372
macro avg	0.97	0.96	0.96	4372
weighted avg	0.97	0.96	0.96	4372

CREATING A MULTILAYER PERCEPTRON (MLP) MODEL:
SETTING HYPERPARAMETERS

An **MLP** is a class of multilayer feedforward artificial neural networks (ANN).

We choose ANNs due to their ability to represent various functions to a high degree of accuracy.

Hyperparameters are the variables that determine the network structure and the variables which determine how the network is trained. Hyperparameters are set before training(before optimizing the weights and bias).

We set the hyperparameters for our MLP model.

Creating a Multilayer Perceptron model

Tensorflow does not take sparse matrices as input and hence we have to convert our sparse matrix into an array. This requires alot of memory and hence uit8 data type helps.

In [41]:
```
# converting csr matrix into np.ndarray supported by tensorflow

x_train = x_train.toarray()
x_test = x_test.toarray()
```

In [42]:
```
# shapes of the various datasets

print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)
```

```
(17487, 261254) (4372, 261254) (17487,) (4372,)
```

It is convention to define hyperparameters separately with capital letters.

In [43]:
```
# input size hyperparameter

INPUT_SIZE = x_train.shape[1]
INPUT_SIZE
```

Out[43]:
```
261254
```

In [44]:
```
# outputsize hyperparatmeter

OUTPUT_SIZE = len(data['language_encoded'].unique())
OUTPUT_SIZE
```

Out[44]:
```
22
```

EPOCHS indicates the number of passes of the entire training dataset the machine learning algorithm has completed.

We divide the dataset into batches of BATCH_SIZE for ease of computation.

In [45]:
```
# epochs and batch_size hyperparameters

EPOCHS = 10
BATCH_SIZE = 128
```

CREATING A MULTILAYER PERCEPTRON (MLP) MODEL:
BUILDING, COMPILING AND FITTING THE MODEL

We create a baseline model which we will be optimizing later.

We create a **Sequential mode**l with activation function as **Rectified Linear Unit(ReLU)** and initializer as **he_normal** as it works well with ReLU.

Our Neural Network is **5** layers deep and **261254** layers wide.

It contains 3 hidden layers:

- Hidden Layer 1: **100 nodes**
- Hidden Layer 2: **80 nodes**
- Hidden Layer 3: **50 nodes**

The output layer uses the **SoftMax** activation function which returns an array of length 22 containing the probability of an element's index being the output.

We use the **Adaptive Moment Estimation (Adam** algorithm for optimization as it is good with sparse data. We use **Sparse Categorical Crossentropy** as the loss/objective function as it performs well for multi-classification problems.

CREATING A MULTILAYER PERCEPTRON (MLP) MODEL: BUILDING, COMPILING AND FITTING THE MODEL

We create a Sequential model with activation function as Rectified Linear Unit(ReLU) and intializer as HE Normal as it works well with ReLU.

Our Neural Network is 5 layers deep and 261254 layers wide.

It contains 3 hidden layers:

- Hidden Layer 1 : 100 nodes
- Hidden Layer 2 : 80 nodes
- Hidden Layer 3 : 50 nodes

The output layer uses the Soft Max activation function which returns an array of lenght 22 contianing probablity of an element's index being the output.

In [144…

```
# creating the MLP model

model = Sequential([
    Dense(100, activation='relu', kernel_initializer='he_normal', input_shape=(INPUT_SIZE,)),
    Dense(80, activation='relu', kernel_initializer='he_normal'),
    Dense(50, activation='relu', kernel_initializer='he_normal'),
    Dense(OUTPUT_SIZE, activation='softmax')
])
```

We use the Adaptive Moment Estimation (Adam) algorithm for optimization as it is good with sparse data. We use Sparse Categorical Crossentropy as the loss/objective function as it performs well for muliclassification problems.

In [145…

```
# compiling the MLP model

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

We split 30% of the training data into validation data to check if the model is overfitting.

In [146…

```
# fitting the MLP model

hist = model.fit(x_train, y_train, epochs=EPOCHS, batch_size=BATCH_SIZE, validation_split=0.3, verbose=2)
```

```
Epoch 1/10
96/96 - 20s - loss: 1.3444 - accuracy: 0.7517 - val_loss: 0.2489 - val_accuracy: 0.9586
Epoch 2/10
96/96 - 18s - loss: 0.0337 - accuracy: 0.9975 - val_loss: 0.1732 - val_accuracy: 0.9623
Epoch 3/10
96/96 - 18s - loss: 0.0026 - accuracy: 0.9998 - val_loss: 0.1685 - val_accuracy: 0.9621
Epoch 4/10
96/96 - 18s - loss: 0.0017 - accuracy: 0.9998 - val_loss: 0.1689 - val_accuracy: 0.9628
Epoch 5/10
96/96 - 18s - loss: 0.0014 - accuracy: 0.9998 - val_loss: 0.1700 - val_accuracy: 0.9617
Epoch 6/10
96/96 - 18s - loss: 0.0011 - accuracy: 0.9998 - val_loss: 0.1714 - val_accuracy: 0.9613
Epoch 7/10
96/96 - 18s - loss: 9.8561e-04 - accuracy: 0.9998 - val_loss: 0.1734 - val_accuracy: 0.9626
Epoch 8/10
96/96 - 17s - loss: 8.9579e-04 - accuracy: 0.9998 - val_loss: 0.1833 - val_accuracy: 0.9617
Epoch 9/10
96/96 - 17s - loss: 7.9612e-04 - accuracy: 0.9998 - val_loss: 0.1900 - val_accuracy: 0.9596
Epoch 10/10
96/96 - 17s - loss: 7.1880e-04 - accuracy: 0.9998 - val_loss: 0.1881 - val_accuracy: 0.9575
```

In [147…

```
# summary of the MLP model

model.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_36 (Dense)             (None, 100)               26125500
_________________________________________________________________
dense_37 (Dense)             (None, 80)                8080
_________________________________________________________________
dense_38 (Dense)             (None, 50)                4050
_________________________________________________________________
dense_39 (Dense)             (None, 22)                1122
=================================================================
Total params: 26,138,752
Trainable params: 26,138,752
Non-trainable params: 0
_________________________________________________________________
```

CREATING A MULTILAYER PERCEPTRON (MLP) MODEL:
MODEL ARCHITECTURE

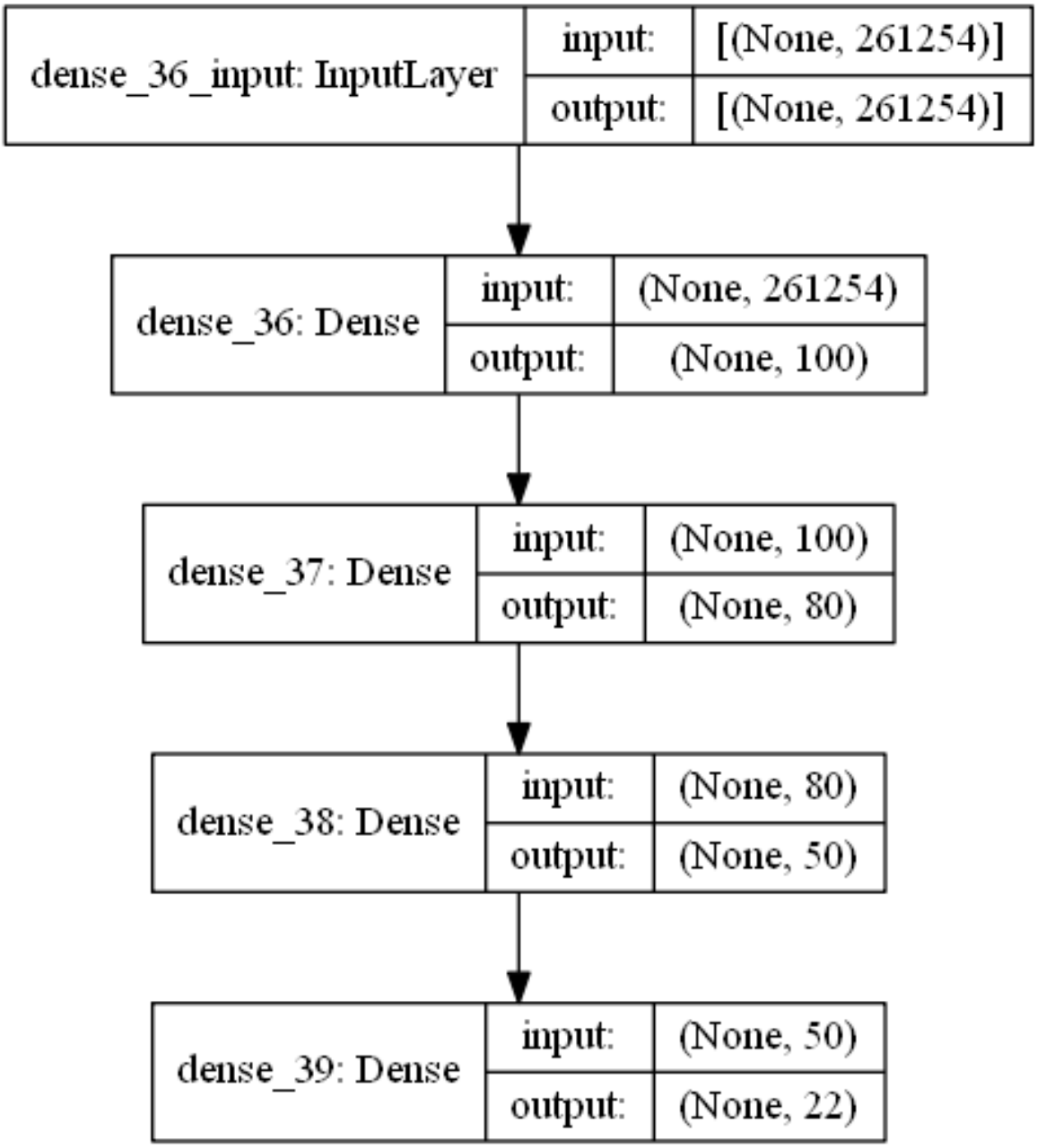
Our model is a 5 layer Neural Network with the following architecture.

Our model is a 5 layer Neural Network with the following architecture.

In [148…

```
# architetcure of the MLP model

plot_model(model, show_shapes=True)
```

Out[148…



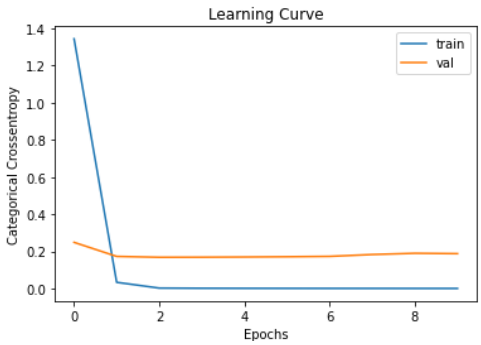
CREATING A MULTILAYER PERCEPTRON (MLP) MODEL:
LEARNING CURVES AND ACCURACY

By plotting the learning curves we observe that the model begins overfitting after 2 epochs and this could lead to a decrease in the model's accuracy.

We obtain an accuracy of **95.4%**. This is the accuracy that we have to beat after optimization.

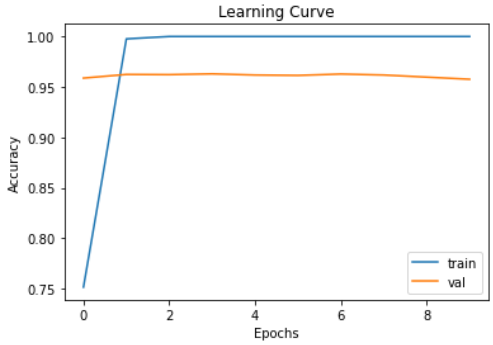
In [149…

```
# creating loss vs epochs plot

plt.title('Learning Curve')
plt.xlabel('Epochs')
plt.ylabel('Categorical Crossentropy')
plt.plot(hist.history['loss'], label='train')
plt.plot(hist.history['val_loss'], label='val')
plt.legend()
plt.show()
```



In [150…

```
# creating accuracy vs epochs plot

plt.title('Learning Curve')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.plot(hist.history['accuracy'], label='train')
plt.plot(hist.history['val_accuracy'], label='val')
plt.legend()
plt.show()
```



We see that are model is overfitting which can lead our accuracy to get lower for new data.

In [151…

```
# evaluating the loss and accuracy of the model

loss, accuracy = model.evaluate(x_test, y_test, verbose=2)
print('Accuracy %.3f'%accuracy)
```

```
137/137 - 3s - loss: 0.1845 - accuracy: 0.9540
Accuracy 0.954
```

We obtain an accuracy of 95.4% this is the accuracy that we have to beat after optimization.

HYPERPARAMETER TUNING: GRID SEARCH

We define a function that creates a model, this is required by the KerasClassifier. We use the KerasClassifier to link our Keras model to Sklearn so that we can perform **Grid Search** and **Randomized Search** on its hyperparameters.

We define the hyperparameter grid and implement **Grid Search** while conducting **3 fold cross-validation**. The Grid Search Algorithm goes through **16 (4*4)** possible combinations of hyperparameters to find the combination that provides the highest accuracy.

We define a function that creates a model, this is required by the KerasClassifier. We use the KerasClassifier to link our keras model to sklearn so that we can perform Grid Search and Randomized Search on it's hyperparameters.

In [78]:
```
def create_model(optimizer='adam', init_mode='he_normal', activation='relu'):
    model = Sequential()
    model.add(Dense(100, activation=activation, kernel_initializer=init_mode, input_shape=(INPUT_SIZE,)))
    model.add(Dense(80, activation=activation, kernel_initializer=init_mode))
    model.add(Dense(50, activation=activation, kernel_initializer=init_mode))
    model.add(Dense(OUTPUT_SIZE, activation='softmax'))

    model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model
```

In [79]:
```
model = KerasClassifier(build_fn=create_model, verbose=2)
```

We define the hyperparameter grid and implement Grid Search while conducting 3 fold cross validation. The Grid Search Algorithm goes through 16 (4*4) possible combinations of hyperparameters to find the combination that provides the highest accuracy.

In [80]:
```
epochs = [2, 4, 6, 8]
batch_size = [32, 64, 128, 256]
param_grid = dict(epochs=epochs, batch_size=batch_size)
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=3, cv=3)
grid_result = grid_search.fit(x_train, y_train)
```

In [81]:
```
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
Best: 0.951376 using {'batch_size': 256, 'epochs': 6}
```

In [82]:
```
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) %r" % (mean, stdev, param))
```

```
0.940750 (0.002427) {'batch_size': 32, 'epochs': 2}
0.928750 (0.001095) {'batch_size': 32, 'epochs': 4}
0.923499 (0.006444) {'batch_size': 32, 'epochs': 6}
0.918749 (0.006929) {'batch_size': 32, 'epochs': 8}
0.938500 (0.005537) {'batch_size': 64, 'epochs': 2}
0.935375 (0.000359) {'batch_size': 64, 'epochs': 4}
0.930374 (0.005533) {'batch_size': 64, 'epochs': 6}
0.918623 (0.009160) {'batch_size': 64, 'epochs': 8}
0.938376 (0.004903) {'batch_size': 128, 'epochs': 2}
0.947751 (0.003579) {'batch_size': 128, 'epochs': 4}
0.946874 (0.004122) {'batch_size': 128, 'epochs': 6}
0.934624 (0.006902) {'batch_size': 128, 'epochs': 8}
0.875748 (0.010311) {'batch_size': 256, 'epochs': 2}
0.950626 (0.004433) {'batch_size': 256, 'epochs': 4}
0.951376 (0.009078) {'batch_size': 256, 'epochs': 6}
0.950124 (0.005922) {'batch_size': 256, 'epochs': 8}
```

We see that we get the highest accuracy using a batch size of 256 and 6 epochs.

HYPERPARAMETER TUNING: RANDOMIZED SEARCH

We define the hyperparameter distribution and implement **Randomized Search** while conducting 3 fold cross-validation. The Randomized Search Algorithm goes through **60** possible combinations of hyperparameters out of a total of **448 (8*8*7)** combinations to find the combination that provides the highest accuracy.

According to the following law, we can say that we have the best 5% of hyperparameters with 95% confidence.
With 60 iterations 95% of the time, the best 5% sets of parameters can be found regardless of grid size.

We define the hyperparameter distribution and implement Randomized Search while conducting 3 fold cross validation. The Randomized Search Algorithm goes through 60 possible combinations of hyperparameters out of a total of 448 (8*8*7) combinations to find the combination that provides the highest accuracy.

According to the following law we can say that we have the best 5% of hyperparameters with 95% confidence.

With 60 iterations 95% of the time, best 5% sets of parameters can be found regardless of grid size.

In [83]:
```
model = KerasClassifier(build_fn=create_model, verbose=2, epochs=8, batch_size=256)
```

In [84]:
```
init_mode = ['uniform', 'lecun_uniform', 'normal', 'zero', 'glorot_normal', 'glorot_uniform', 'he_normal', 'he_uniform']
activation = ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear']
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']

param_dist = dict(init_mode=init_mode, activation=activation, optimizer=optimizer)
random_search = RandomizedSearchCV(estimator=model, param_distributions=param_dist, n_jobs=3, n_iter=60, cv=3)
random_result = random_search.fit(x_train, y_train)
```

```
Epoch 1/8
32/32 - 4s - loss: 2.2809 - accuracy: 0.8430
Epoch 2/8
32/32 - 3s - loss: 0.8234 - accuracy: 0.9930
Epoch 3/8
32/32 - 3s - loss: 0.2737 - accuracy: 0.9990
Epoch 4/8
32/32 - 3s - loss: 0.1185 - accuracy: 1.0000
Epoch 5/8
32/32 - 4s - loss: 0.0688 - accuracy: 1.0000
Epoch 6/8
32/32 - 3s - loss: 0.0464 - accuracy: 1.0000
Epoch 7/8
32/32 - 3s - loss: 0.0339 - accuracy: 1.0000
Epoch 8/8
32/32 - 3s - loss: 0.0261 - accuracy: 1.0000
```

In [85]:
```
print("Best: %f using %s" % (random_result.best_score_, random_result.best_params_) )
```

```
Best: 0.958875 using {'optimizer': 'Adam', 'init_mode': 'glorot_uniform', 'activation': 'softsign'}
```

HYPERPARAMETER TUNING: RESULT

We see that we get the highest accuracy while using Adam as optimizer, Glorot Uniform as the kernel_intializer and Softsign as the activation function.

Hence we conclude that the optimal hyperparameters for our model are:

- BATCH_SIZE = 256
- EPOCHS = 6
- optimizer = Adam
- kenel_initialzier = glorot_uniform
- activation = softsign

CREATING THE OPTIMIZED MODEL: SETTING HYPERPARAMETERS AND CONFIGURING EARLY STOPPING

We create an optimized model using the hyperparameters found in Grid Search and Randomized Search.

Optimal hyperparameters for our model are:

- BATCH_SIZE = 256
- EPOCHS = 6
- optimizer = Adam
- kenel_initialzier = glorot_uniform
- activation = softsign

We configure early stopping to avoid overfitting.

We create an optimized model using the hyperparameters found in Grid Search and Randomized Search.

In [62]:
```
# input size hyperparameter

INPUT_SIZE = x_train.shape[1]
INPUT_SIZE

# outputsize hyperparatmeter

OUTPUT_SIZE = len(data['language_encoded'].unique())
OUTPUT_SIZE
```

Out[62]: 22

In [63]:
```
BATCH_SIZE = 256
EPOCHS = 8
```

We configure early stopping to avoid overfitting.

In [64]:
```
# configuring early stopping

es = EarlyStopping(monitor='accuracy', patience=1)
```

CREATING THE OPTIMIZED MODEL: CREATING, COMPILING AND FITTING THE MODEL

We create a **Sequential model** with activation function as **soft sign** and initializer as **glorot_uniform**.

Our Neural Network is **5** layers deep and **26125**4 layers wide.
It contains 3 hidden layers:

- Hidden Layer 1: **100 nodes**
- Hidden Layer 2: **80 nodes**
- Hidden Layer 3: **50 nodes**

The output layer uses the **SoftMax** activation function which returns an array of length **22** containing the probability of an element's index being the output.

We use the **Adaptive Moment Estimation (Adam)** algorithm for optimization as it is good with sparse data. We use **Sparse Categorical Crossentropy** as the loss/objective function as it performs well for multi-classification problems.

We split 30% of the training data into **validation data** to check if the model is overfitting.

CREATING THE OPTIMIZED MODEL: CREATING, COMPILING AND FITTING THE MODEL

We create a Sequential model with activation function as softsign and intializer as glorot_uniform. Our Neural Network is 5 layers deep and 261254 layers wide.

It contains 3 hidden layers:

- Hidden Layer 1 : 100 nodes
- Hidden Layer 2 : 80 nodes
- Hidden Layer 3 : 50 nodes

The output layer uses the Soft Max activation function which returns an array of lenght 22 contianing probablity of an element's index being the output.

In [65]:
```
# creating the MLP model

model = Sequential([
    Dense(100, activation='softsign', kernel_initializer='glorot_uniform', input_shape=(INPUT_SIZE,)),
    Dense(80, activation='softsign', kernel_initializer='glorot_uniform'),
    Dense(50, activation='softsign', kernel_initializer='glorot_uniform'),
    Dense(OUTPUT_SIZE, activation='softmax')
])
```

We use the Adaptive Moment Estimation (Adam) algorithm for optimization as it is good with sparse data. We use Sparse Categorical Crossentropy as the loss/objective function as it performs well for muliclassification problems.

In [66]:
```
# compiling the MLP model

model.compile(optimizer='Adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

We split 30% of the training data into validation data to check if the model is overfitting.

In [67]:
```
# fitting the model with earlystopping callback to avoid overfitting

hist = model.fit(x_train, y_train, epochs=EPOCHS, batch_size=BATCH_SIZE, validation_split=0.3, callbacks=[es], verbose=2)
```

```
Epoch 1/8
48/48 - 23s - loss: 1.8727 - accuracy: 0.8836 - val_loss: 0.9309 - val_accuracy: 0.9522
Epoch 2/8
48/48 - 13s - loss: 0.4052 - accuracy: 0.9956 - val_loss: 0.3557 - val_accuracy: 0.9665
Epoch 3/8
48/48 - 13s - loss: 0.1117 - accuracy: 0.9995 - val_loss: 0.2301 - val_accuracy: 0.9680
Epoch 4/8
48/48 - 13s - loss: 0.0534 - accuracy: 0.9998 - val_loss: 0.1871 - val_accuracy: 0.9684
Epoch 5/8
48/48 - 13s - loss: 0.0332 - accuracy: 0.9998 - val_loss: 0.1660 - val_accuracy: 0.9684
Epoch 6/8
48/48 - 13s - loss: 0.0233 - accuracy: 0.9998 - val_loss: 0.1542 - val_accuracy: 0.9676
```

In [68]:
```
# summary of the MLP model

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 100)               26125500
_________________________________________________________________
dense_5 (Dense)              (None, 80)                8080
_________________________________________________________________
dense_6 (Dense)              (None, 50)                4050
_________________________________________________________________
dense_7 (Dense)              (None, 22)                1122
=================================================================
Total params: 26,138,752
Trainable params: 26,138,752
Non-trainable params: 0
_________________________________________________________________
```

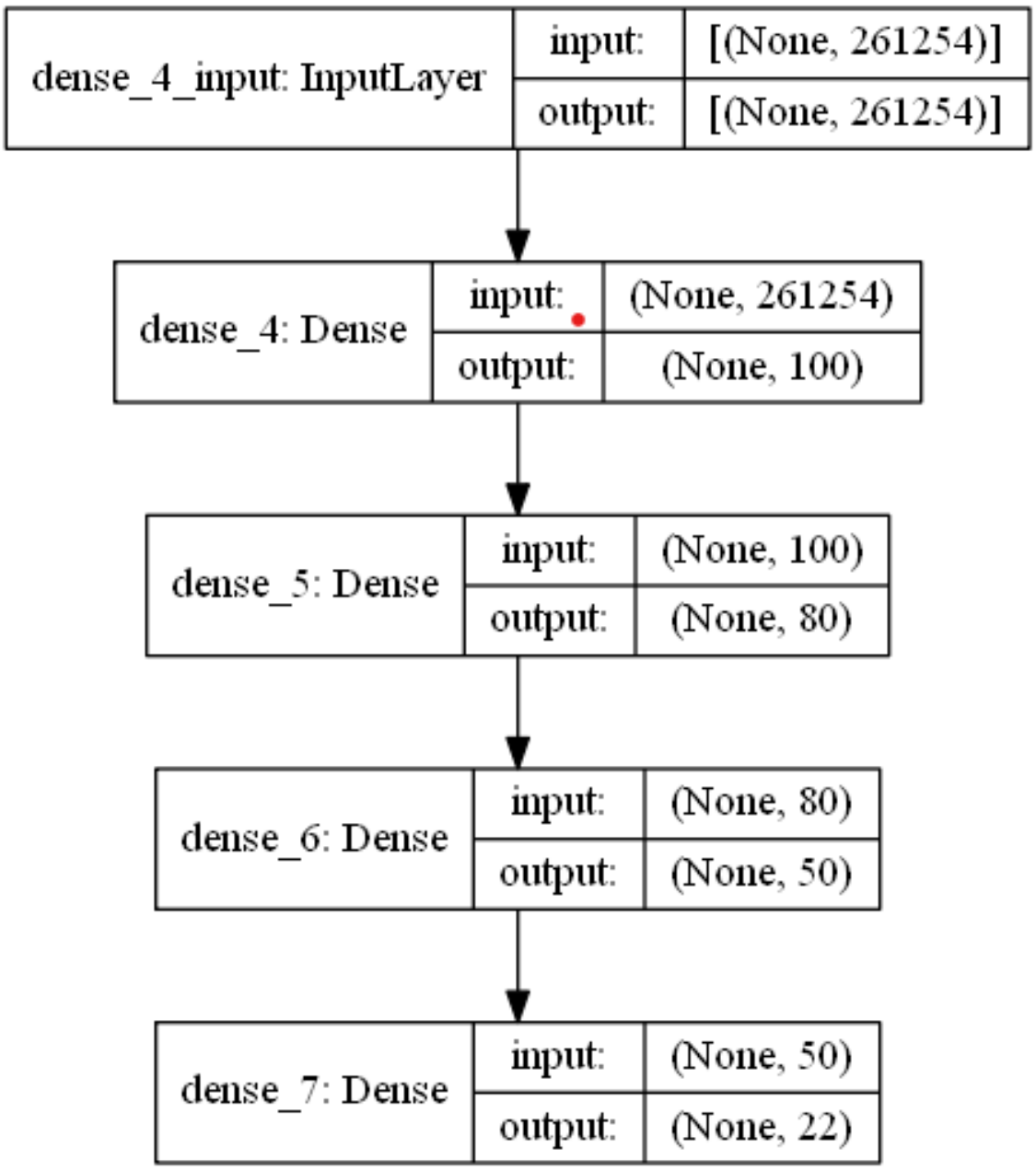
CREATING THE OPTIMIZED MODEL: MODEL ARCHITECTURE

Our model is a 5 layer Neural Network with the following architecture.

Our model is a 5 layer Neural Network with the following architecture.

In [69]:
```
# architetcure of the MLP model

plot_model(model, show_shapes=True)
```

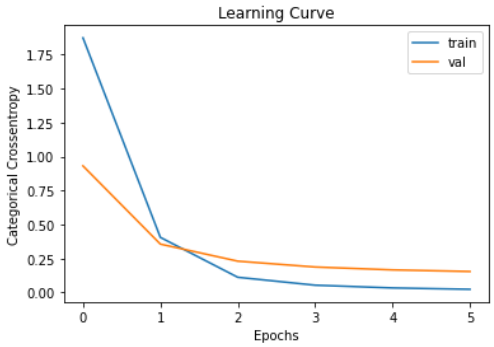
Out[69]:



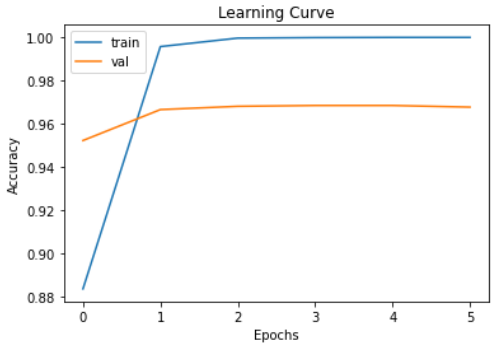
CREATING THE OPTIMIZED MODEL: LEARNING CURVES AND ACCURACY

Our model MLP obtains an accuracy of **96.8%** which beats our MultinomialNB model (95.74%).

In [70]:
```
# creating loss vs epochs plot

plt.title('Learning Curve')
plt.xlabel('Epochs')
plt.ylabel('Categorical Crossentropy')
plt.plot(hist.history['loss'], label='train')
plt.plot(hist.history['val_loss'], label='val')
plt.legend()
plt.show()
```



In [71]:
```
# creating accuracy vs epochs plot

plt.title('Learning Curve')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.plot(hist.history['accuracy'], label='train')
plt.plot(hist.history['val_accuracy'], label='val')
plt.legend()
plt.show()
```



[72]:
```
# evaluating the loss and accuracy of the model

loss, accuracy = model.evaluate(x_test, y_test, verbose=2)
print('Accuracy %.3f'%accuracy)
```

```
137/137 - 3s - loss: 0.1590 - accuracy: 0.9680
Accuracy 0.968
```

Our model MLP obtains an accuracy of 96.8% which beats are MultinomialNB model (96.8%).

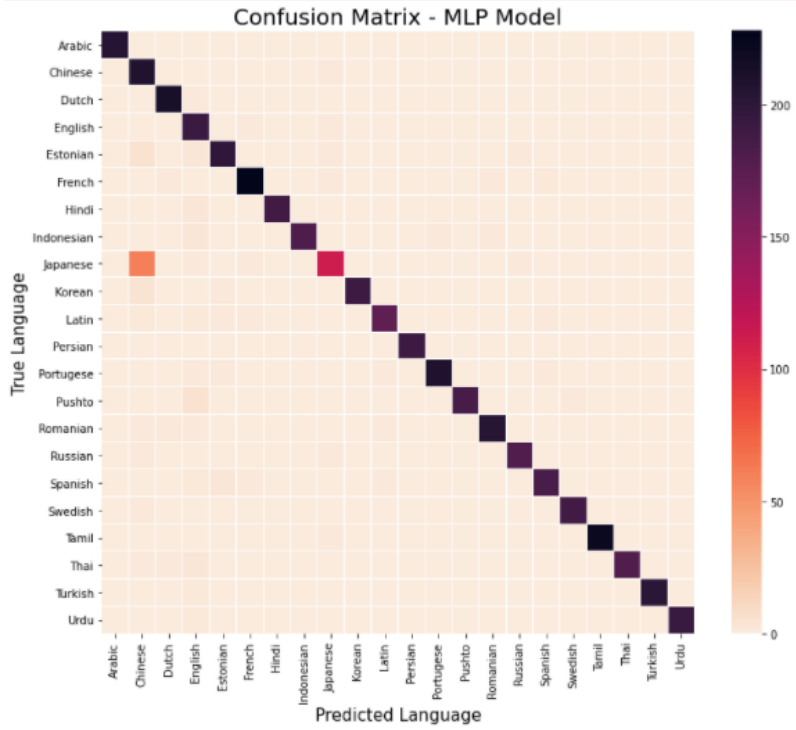
CREATING THE OPTIMIZED MODEL: CONFUSION MATRIX

Looking at the **confusion matrix** we see that the model predicted Japanese as Chinese an unusual number of times, this could be due to the similarity between the **Hanzi Script (Chinese)** and the **Kanji Script (Japanese)**.

In [73]:

```
y_pred_prob = model.predict(x_test) # returns an array containing probability for each category being output
y_pred = []
for i in y_pred_prob:
    out = np.argmax(i) # taking the highest probability category as output
    y_pred.append(out)
y_pred = np.array(y_pred)

cm = confusion_matrix(y_test, y_pred) # confusion matrix

# heat map of confusion matrix
plt.figure(figsize=(12,10))
plt.title('Confusion Matrix - MLP Model', fontsize=20)
sns.heatmap(cm, xticklabels=lang_list, yticklabels=lang_list, cmap='rocket_r', linecolor='white', linewidth=.005)
plt.xlabel('Predicted Language', fontsize=15)
plt.ylabel('True Language', fontsize=15)
plt.show()
```

```
<ipython-input-73-2471fde572b2>:12: MatplotlibDeprecationWarning: Case-insensitive properties were deprecated in 3.3 and support will be removed two minor releases later
  plt.title('Confusion Matrix - MLP Model', fontsize=20)
```



Looking at the confusion matrix we see that the model predicted Japanese as Chinese an unsual number of times, this could be due to the similarity between the Hanzi Script (Chinese) and the Kanji Script (Japanese).

CREATING THE OPTIMIZED MODEL: CLASSIFICATION REPORT

Precision attempts to answer what proportion of positive identifications were actually correct?

Recall attempts to answer what proportion of actual positives were identified correctly?

In [74]:
```
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	205
1	0.73	1.00	0.84	209
2	0.99	1.00	0.99	214
3	0.87	0.99	0.92	195
4	0.97	0.95	0.96	210
5	0.98	0.98	0.98	233
6	1.00	0.98	0.99	191
7	1.00	0.97	0.99	186
8	0.95	0.64	0.76	177
9	1.00	0.97	0.98	198
10	0.98	0.96	0.97	178
11	1.00	1.00	1.00	192
12	1.00	0.98	0.99	213
13	1.00	0.96	0.98	192
14	0.99	0.98	0.99	208
15	0.99	0.99	0.99	182
16	0.98	0.96	0.97	191
17	0.99	0.99	0.99	189
18	1.00	1.00	1.00	223
19	1.00	0.97	0.99	185
20	1.00	0.99	1.00	205
21	1.00	0.99	0.99	196
accuracy			0.97	4372
macro avg	0.97	0.97	0.97	4372
weighted avg	0.97	0.97	0.97	4372

SAVING AND PREDICTING USING THE MODEL:

We save the model for deployment in the future.

We load the model and deploy it to predict the language of the unknown text.

Saving the model

We save the model for deployment in the future.

In [75]:
```
# saving the model

model.save('language_identifcation_model.h5')
```

Predicting using the model

We load the model and deploy it to predict the language of unknown text.

In [76]:
```
# loading the model

model = load_model('language_identifcation_model.h5')
```

In [77]:
```
# using the model for prediction

sent = """आप कितना सोचते हो
अगर आप ठिठुरती रातों को गिनें
अरे क्या आप मिल सकते हैं (अरे, क्या आप मिल सकते हैं?)
क्या तुम मिलोगे (क्या तुम मिलोगे?)
सर्दियों का अंत बताओ
एक कोमल वसंत के दिन तक
मैं चाहता हूं कि तुम तब तक रहो जब तक फूल खिल न जाएं
ज्यों का त्यों"""


sent = cv.transform([sent])
ans = model.predict(sent)
ans = np.argmax(ans)
le.inverse_transform([ans])
```

Out[77]:
```
array(['Hindi'], dtype=object)
```

AUDIO DETECTION MODEL:

IMPORTING DEPENDENCIES:

We import all the dependencies required to build the model and configure GPU/CPU to be used with TensorFlow.

Importing dependencies

In [1]:
```
import numpy as np # linear algebra
import pandas as pd # dataset handling
import librosa # audio data
import librosa.display # audio data plots
import IPython.display as ipd # playing audio
import matplotlib.pyplot as plt # plotting
import os # interacting with operating system
import tensorflow as tf # tensorflow
from tqdm import tqdm # progress bar

from sklearn.preprocessing import LabelEncoder # add placeholder values for categorical variables
from sklearn.metrics import classification_report, confusion_matrix # accuracy metrics for classification model
from sklearn.model_selection import GridSearchCV # grid search hyperparameters

from tensorflow.keras import Sequential # squential neural network
from tensorflow.keras.layers import Dense # feed forward layer
from tensorflow.keras.layers import Conv2D # convolutional layer
from tensorflow.keras.layers import MaxPool2D # pooling layer
from tensorflow.keras.layers import Flatten # flatten 2d/3d array into 1d array
from tensorflow.keras.layers import Dropout # reduce overfitting
from tensorflow.keras.callbacks import EarlyStopping # early stopping to reduce overfitting
from tensorflow.keras.optimizers import Adam # Adam optimizer
from tensorflow.keras.utils import plot_model # plot model architecture
from tensorflow.keras.layers import BatchNormalization # standardize the input for each mini batch
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier # link tensorflow model with sklearn
```

In [2]:
```
# configuring gpu for tensorflow
# skip this if you have not setup gpu for tensorflow

os.environ['CUDA_VISIBLE_DEVICES'] = '0' # 0 for GPU
                                         # -1 for CPU
if tf.test.gpu_device_name():
    print('GPU found')
else:
    print("No GPU found")
```

```
GPU found
```

EXPLORING THE VOICE DATA: GERMAN VOICE

We set up the **German** audio path and file name and extract the audio in array form along with its sampling rate.

By default, Librosa converts the audio into **22kHz** sampling rate and scales the data and converts the audio into a **mono channel.**

We play the audio and plot a wave plot for it.

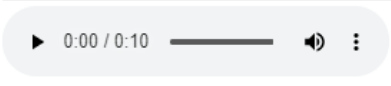
The **sampling rate** refers to the number of samples of audio recorded every second.

German voice

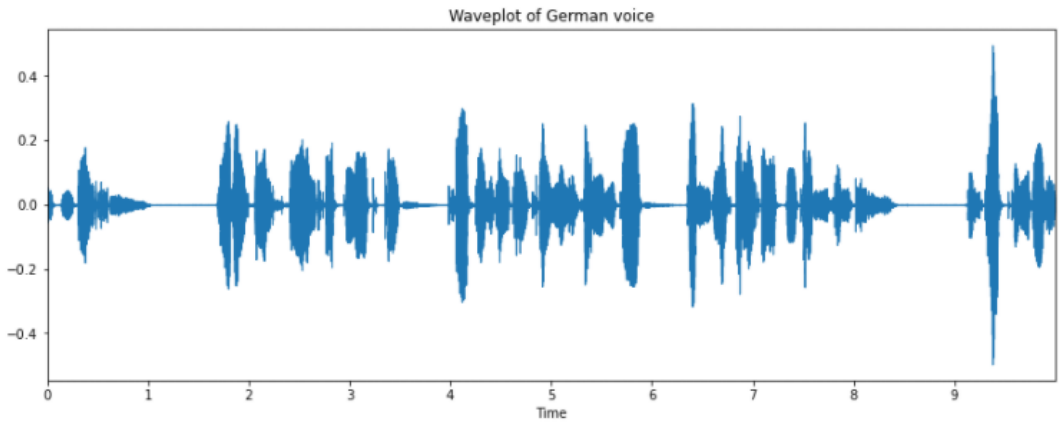
In [3]:
```
# settting train path and file name
# and extracting the audio (German) in array form along with its samplerate
# by default, librosa converts the audio into 22 kHz sample rate, scales the data and converts the audio into mono channel

train_path = r'C:\Users\dariu\Jupyter Notebook\Natural Language Processing\Audio NLP Files\train\train\\'
filename_de = 'de_f_5d2e7f30d69f2d1d86fd05f3bbe120c2.fragment1.flac'
x_de, sr_de = librosa.load(train_path+filename_de)

print(x_de.shape)
print(x_de)
print(sr_de)
```

```
(220500,)
[ 0.01663208 -0.01715088  0.00863647 ...  0.00540161  0.00170898
  0.00082397]
22050
```

In [4]:
```
# playing the audio

ipd.Audio(data=x_de, rate=sr_de)
```

Out[4]: 

In [5]:
```
# waveplot of the audio

plt.figure(figsize=(14,5))
plt.title('Waveplot of German voice')
librosa.display.waveplot(x_de, sr_de)
plt.show()
```



EXPLORING THE VOICE DATA: GERMAN VOICE MFCCS

We extract Mel Frequency Cepstral Coefficients (MFCCs) for our **German** audio and plot an MFCC spectrogram for it.

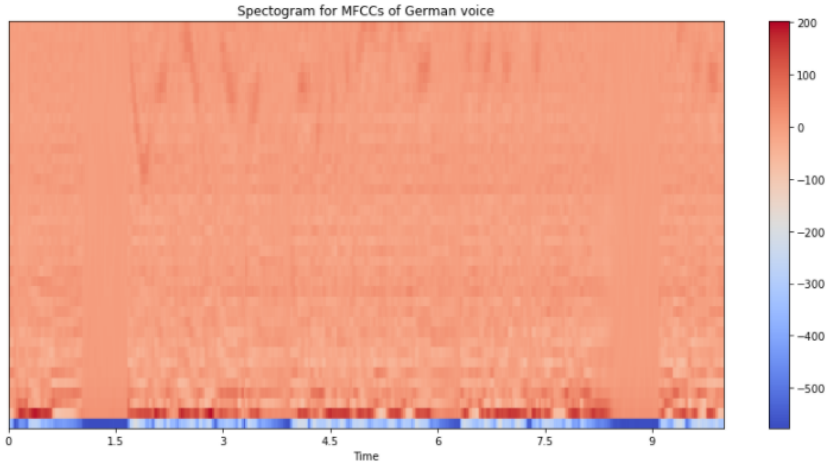
A **spectrogram** is a visual way of representing the signal strength, or “loudness”, of a signal over time at various frequencies present in a particular waveform.

Mel Frequency Cepstral Coefficient (MFCC) features are rooted in the recognized discrepancy of the human ear's critical bandwidths with frequency filters spaced linearly at low frequencies and logarithmically at high frequencies have been used to retain the phonetically vital properties of the speech signal.

In [6]:
```
# extracting Mel Frequency Cepstral Coefficients (MFCCs) from the audio

mfccs_de = librosa.feature.mfcc(x_de, sr_de, n_mfcc=40)
print(mfccs_de.shape)
print(mfccs_de)
```

```
(40, 431)
[[-3.38363495e+02 -3.36173248e+02 -3.73291779e+02 ... -3.13378082e+02
  -3.13951508e+02 -3.15486542e+02]
 [-7.77327652e+01 -7.15431290e+01 -6.02841072e+01 ... -8.36011200e+01
  -8.38067017e+01 -6.68007355e+01]
 [ 8.11036682e+00  2.18050003e-01 -2.80894017e+00 ...  1.56011267e+01
   1.58308525e+01  1.85882187e+01]
 ...
 [-3.90985584e+00 -3.47313213e+00 -4.10862684e+00 ...  8.50367832e+00
   5.25859785e+00  2.59182262e+00]
 [-4.78532028e+00 -1.22923422e+00 -1.52569771e-01 ... -3.23789048e+00
  -5.06710148e+00 -1.20688403e+00]
 [-6.11020565e+00 -2.39128399e+00 -1.41881600e-01 ... -1.40471554e+00
  -9.30427313e-01 -4.74035740e-03]]
```

In [7]:
```
# plotting the MFCCs spectogram

plt.figure(figsize=(15,7))
plt.title('Spectogram for MFCCs of German voice')
librosa.display.specshow(mfccs_de, sr=sr_de, x_axis='time')
plt.colorbar()
plt.show()
```



EXPLORING THE VOICE DATA: ENGLISH VOICE

We set up the **English** audio path and file name and extract the audio in array form along with its sampling rate.

By default, Librosa converts the audio into **22kHz** sampling rate and scales the data and converts the audio into a **mono channel.**

We play the audio and plot a wave plot for it.

The sampling rate refers to the number of samples of audio recorded every second.

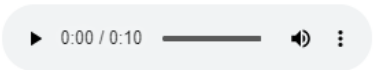
English voice

In [8]:
```
# settting train path and file name
# and extracting the audio (English) in array form along with its samplerate
# by default, librosa converts the audio into 22 kHz sample rate, scales the data and converts the audio into mono channel

train_path = r'C:\Users\dariu\Jupyter Notebook\Natural Language Processing\Audio NLP Files\train\train\\'
filename_en = 'en_m_65733942d40b2431cf22fe21495693f7.fragment5.noise8.flac'
x_en, sr_en = librosa.load(train_path+filename_en)

print(x_en.shape)
print(x_en)
print(sr_en)
```

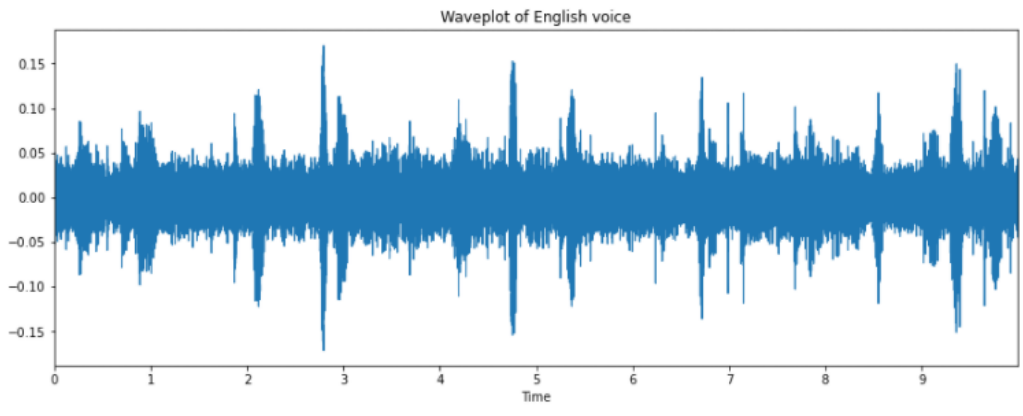
```
(220500,)
[ 0.01806641  0.03265381  0.03326416 ... -0.02374268 -0.02389526
 -0.02331543]
22050
```

In [9]:
```
# playing the audio

ipd.Audio(data=x_en, rate=sr_en)
```

Out[9]:



In [10]:
```
# waveplot of the audio

plt.figure(figsize=(14,5))
plt.title('Waveplot of English voice')
librosa.display.waveplot(x_en, sr_en)
plt.show()
```



EXPLORING THE VOICE DATA: ENGLISH VOICE MFCCS

We extract Mel Frequency Cepstral Coefficients (MFCCs) for our **English** audio and plot an MFCC spectrogram for it.

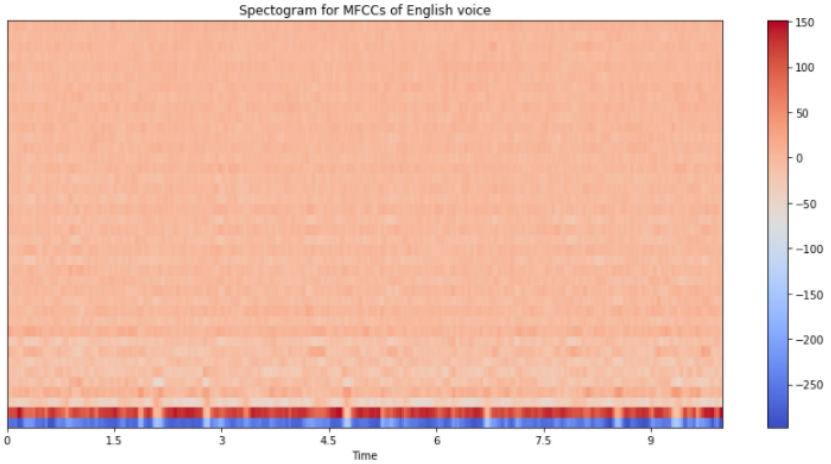
A **spectrogram** is a visual way of representing the signal strength, or “loudness”, of a signal over time at various frequencies present in a particular waveform.

Mel Frequency Cepstral Coefficient (MFCC) features are rooted in the recognized discrepancy of the human ear's critical bandwidths with frequency filters spaced linearly at low frequencies and logarithmically at high frequencies have been used to retain the phonetically vital properties of the speech signal.

In [11]:
```
# extracting Mel Frequency Cepstral Coefficients (MFCCs) from the audio

mfccs_en = librosa.feature.mfcc(x_en, sr_en, n_mfcc=40)
print(mfccs_en.shape)
print(mfccs_en)
```

```
(40, 431)
[[-238.03094    -245.11372    -256.28253    ... -241.09186
  -288.05023    -285.54575   ]
 [  93.12717     103.34192     112.61098    ...   91.479355
   133.23985     151.01459   ]
 [ -17.982334    -21.30424     -28.723568   ...  -26.01315
   -48.428688    -44.89864   ]
 ...
 [   4.256668      4.7556925     0.76691943 ...    1.4248196
    -0.47866392    0.8116276 ]
 [   1.8517773     1.595865     -3.7998354  ...   -3.4249628
    -2.8405504    -2.109693  ]
 [  14.043443     10.8478155     1.7877384  ...    4.8450513
     1.5311663    -1.4085169 ]]
```

In [12]:
```
# plotting the MFCCs spectogram

plt.figure(figsize=(15,7))
plt.title('Spectogram for MFCCs of English voice')
librosa.display.specshow(mfccs_en, sr=sr_en, x_axis='time')
plt.colorbar()
plt.show()
```



EXPLORING THE VOICE DATA: SPANISH VOICE

We set up the **Spanish** audio path and file name and extract the audio in array form along with its sampling rate.

By default, Librosa converts the audio into **22kHz** sampling rate and scales the data and converts the audio into a **mono channel.**

We play the audio and plot a wave plot for it.

The sampling rate refers to the number of samples of audio recorded every second.

Spanish voice

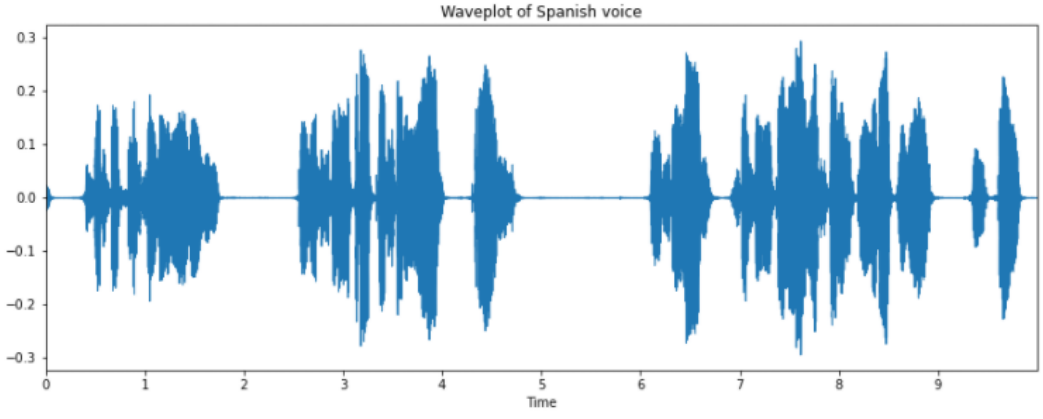
In [13]:
```
# settting train path and file name
# and extracting the audio (Spanish) in array form along with its samplerate
# by default, librosa converts the audio into 22 kHz sample rate, scales the data and converts the audio into mono channel

train_path = r'C:\Users\dariu\Jupyter Notebook\Natural Language Processing\Audio NLP Files\train\train\\'
filename_es = 'es_m_f16e9da2ba7126292cb4e46ce16995bc.fragment25.pitch7.flac'
x_es, sr_es = librosa.load(train_path+filename_es)

print(x_es.shape)
print(x_es)
print(sr_es)
```

```
(220500,)
[1.8280029e-02 2.3925781e-02 2.0141602e-02 ... 9.1552734e-05 0.0000000e+00
 3.0517578e-05]
22050
```

In [14]:
```
# playing the audio

ipd.Audio(data=x_es, rate=sr_es)
```

Out[14]:

In [15]:
```
# waveplot of the audio

plt.figure(figsize=(14,5))
plt.title('Waveplot of Spanish voice')
librosa.display.waveplot(x_es, sr_es)
plt.show()
```



EXPLORING THE VOICE DATA: SPANISH VOICE MFCCS

We extract Mel Frequency Cepstral Coefficients (MFCCs) for our **Spanish** audio and plot an MFCC spectrogram for it.

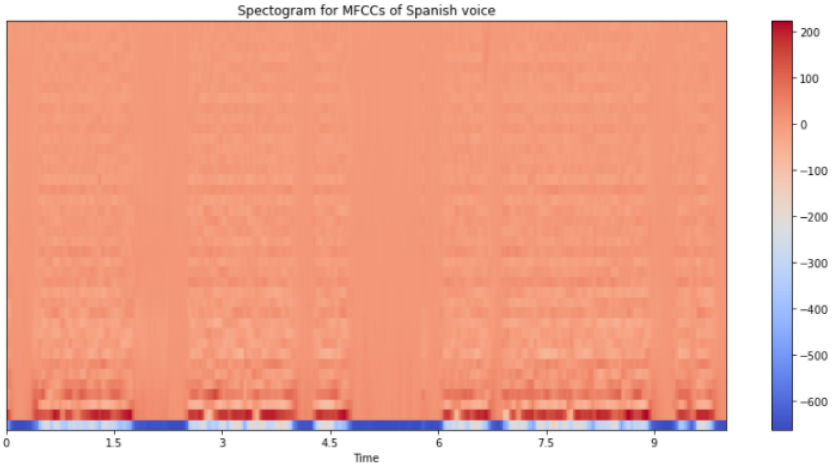
A **spectrogram** is a visual way of representing the signal strength, or “loudness”, of a signal over time at various frequencies present in a particular waveform.

Mel Frequency Cepstral Coefficient (MFCC) features are rooted in the recognized discrepancy of the human ear's critical bandwidths with frequency filters spaced linearly at low frequencies and logarithmically at high frequencies have been used to retain the phonetically vital properties of the speech signal.

In [16]:
```
# extracting Mel Frequency Cepstral Coefficients (MFCCs) from the audio

mfccs_es = librosa.feature.mfcc(x_es, sr_es, n_mfcc=40)
print(mfccs_es.shape)
print(mfccs_es)
```

```
(40, 431)
[[-4.4253195e+02 -4.7378436e+02 -5.5647052e+02 ... -6.5107355e+02
  -6.5425201e+02 -6.5586389e+02]
 [ 1.6602249e+02  1.6130263e+02  1.1064873e+02 ...  2.5271605e+01
   2.1006039e+01  1.8747192e+01]
 [ 2.5196711e+01  2.2606220e+01  3.4949577e+01 ...  2.3809818e+01
   2.0180216e+01  1.7980892e+01]
 ...
 [-1.0159103e+01 -6.6296959e+00 -7.8373492e-01 ... -8.0345583e-01
  -8.6552238e-01 -1.2443852e+00]
 [-4.0571909e+00 -6.2247705e+00 -6.0992184e+00 ... -5.8509809e-01
  -9.9615812e-01 -1.8054054e+00]
 [ 6.6129980e+00  2.9288621e+00 -3.5558860e+00 ... -4.8249924e-01
  -1.1723199e+00 -2.2749543e+00]]
```

In [17]:
```
# plotting the MFCCs spectogram

plt.figure(figsize=(15,7))
plt.title('Spectogram for MFCCs of Spanish voice')
librosa.display.specshow(mfccs_es, sr=sr_es, x_axis='time')
plt.colorbar()
plt.show()
```



LOADING THE AUDIO DATA AND PREPROCESSING: THE DATASET

We use the 'Spoken Language Identification' dataset from Kaggle.
The data contains male and female audio files in the languages **German**, **Spanish** and **English**.

LibriVox recordings were used to prepare the dataset.
Particular attention was paid to a big variety of unique speakers.
Big variance forces the model to concentrate more on language properties than a specific voice.
Samples were equally balanced between languages, genders and speakers in order not to favour any subgroup.
Finally, the dataset is divided into train and test sets.

Speakers present in the test set, are not present in the train set.
This helps estimate a **generalization error.**
The core of the train set is based on **420 minutes (2520 samples)** of original recordings.
After applying several audio transformations (pitch, speed and noise) the train set was extended to **12180 minutes (73080 samples)**.
The test set contains **90 minutes (540 samples)** of original recordings. No data augmentation has been applied.
Original recordings contain **90** unique speakers.
The number of unique speakers was increased by adjusting pitch (8 different levels) and speed (8 different levels).
After applying audio transformations there are **1530** unique speakers.

LOADING THE AUDIO DATA AND PREPROCESSING: DATA STRUCTURE

The dataset is divided into **2** directories:

- **train (73080 samples)**
- **test (540 samples)**

Each sample is a **FLAC** audio file with:

- sample rate: **22050**
- bit depth: **16**
- channels: **1**
- duration: **10 seconds (sharp)**

The filename of the sample has the following syntax:
(language)_(gender)_(recording ID).fragment(index)[.(transformation)(index)].flac

...and variables:

- **language**: en, de, or es
- **gender**: m or f
- **recording ID**: a hash of the URL
- **fragment index**: 1-30
- **transformation**: speed, pitch or noise
- **transformation inde**x:
  - if **speed**: 1-8
  - if **pitch**: 1-8
  - if **noise**: 1-12

LOADING THE AUDIO DATA AND PREPROCESSING: CREATING META DATASET

We create a dictionary containing language code and language as key-value pairs.

- '**de**': 'German'
- '**es**': 'Spanish'
- '**en**': 'English'

We create a meta dataset (data frame) containing the filename, language code and the language.

Loading the audio data and creating meta dataset

In [18]:
```
# creating a language code-language dictonary with the 3 langauges the dataset contians

lang_code = {'de':'German', 'es':'Spanish', 'en':'English'}
```

In [19]:
```
# iterating through the folder containing training audio creating lists containing
# the audio filename, the language code of the audio and the language of the audio

filename_list = []
language_code_list = []
language_list = []
for filename in os.listdir(train_path):
    lang = filename[:2]
    file = filename
    filename_list.append(file)
    language_code_list.append(lang)
    language_list.append(lang_code[lang])
```

In [20]:
```
# checking if all files have been iterated through

print(len(filename_list), len(language_list), len(language_code_list))
```

```
73080 73080 73080
```

In [21]:
```
# creating a metadataset containing filename, language code and language for training data

meta_data = pd.DataFrame({'filename':filename_list, 'lang_code':language_code_list, 'language':language_list})
meta_data = meta_data.sample(frac=1).reset_index(drop=True)
```

In [22]:
```
# meta dataset

print(meta_data.shape)
meta_data.head()
```

```
(73080, 3)
```

Out[22]:

	filename	lang_code	language
0	en_f_fff98d05640df27726dcd6183642ffaa.fragment...	en	English
1	en_m_d2f13c7f37ecec67df0f46408c224bc1.fragment...	en	English
2	en_m_6a5535d7512259e7bef76ff997750873.fragment...	en	English
3	en_m_3a8f30bef6d7b6a1332629491eabcbf0.fragment...	en	English
4	es_f_e27d922d06d483e4df1305235a6cd87b.fragment...	es	Spanish

5 1

LOADING THE AUDIO DATA AND PREPROCESSING:

We create a function that takes in the filename and file path of an audio file as input and returns an array containing MFCCs features.

We iterate through all the files in the train data folder extracting their MFCCs features.

We create a feature data dataset (data frame) that contains MFCC features, language and filename.

```
def feature_extractor(filename,path):
    """
    function that takes audio filename and its path as input and returns an array containing mfccs
    """
    audio, sample_rate = librosa.load(path+filename)
    mfccs_features = librosa.feature.mfcc(audio, sample_rate, n_mfcc=40)
    mfccs_features = mfccs_features.T
    return mfccs_features
```

```
# iterating through all the files in training folder and extracting lists that contain mfccs features, language of audio
# and name of the audio file

feature_list = []
language_list = []
filename_list = []
for index, row in tqdm(meta_data.iterrows()):
    file = row['filename']
    feature = feature_extractor(file,train_path)
    lang = row['language']
    feature_list.append(feature)
    language_list.append(lang)
    filename_list.append(file)
```

```
73080it [28:52, 42.18it/s]
```

```
# creating a dataset with filename, mfccs features and language

feature_data = pd.DataFrame({'filename':filename_list, 'mfccs_feature':feature_list, 'language':language_list})
print(feature_data.shape)
feature_data.head()
```

```
(73080, 3)
```

	filename	mfccs_feature	language
0	es_f_53b555eab2b3baada380f7d3ede20b20.fragment...	[[-336.688, 89.03323, -13.615833, 50.66044, 11...	Spanish
1	de_m_3aa01c55ba7ca85cad2977c848eb826f.fragment...	[[-452.59262, 118.58557, 41.230923, 44.446392,...	German
2	de_m_2cfa348b2b6852e38637a4225158678d.fragment...	[[-174.84755, 173.71323, -61.01853, 18.509579,...	German
3	en_m_011f3a2d0aa2880305c08b76873c3e10.fragment...	[[-249.14648, 153.65164, -56.78586, 3.911124, ...	English
4	de_m_fc6bd6bb9d66a89bb8d8a8a7efa23e6b.fragment...	[[-458.494, 146.9469, 56.362698, 60.76719, 62....	German

SPLITTING DATA INTO INPUTS AND TARGETS:

We create an **input (independent)** variable array containing MFCCs features.

We create a **target (dependent)** variable array containing placeholder values for each category

We reshape the input array to be compatible with our **CNN** model.

Splitting the data into inputs and targets

```
# creating input (dependent) variable containing mfccs features

X = np.array(feature_list)

print(X.shape)
# (73080, 431, 40)
```

```
(73080, 431, 40)
```

```
# reshaping the input data into 3D array compatiable with CNNs

X = X.reshape((X.shape[0], X.shape[1], X.shape[2], 1))
print(X.shape)
```

```
(73080, 431, 40, 1)
```

```
# creating target (dependent) variable with placeholder values for various categories

le = LabelEncoder()
y = le.fit_transform(feature_data['language'])
y = np.array(y.tolist())
```

```
# shape of X and y

print(X.shape, y.shape)
```

```
(73080, 431, 40, 1) (73080,)
```

CREATING A BASELINE CONVOLUTIONAL NEURAL NETWORK (CNN) MODEL: SETTING HYPERPARAMETERS, BUILDING AND COMPILING THE MODEL

We create a baseline **Sequential CNN model** with 2 **Convolutional layers** and **Pooling layers** which are later flattened to a dense layer. each layer has **ReLU** as activation function and **he_normal** as kernel initializer. the output layer has **softmax** activation as we have a multiclassification problem.

We compile the model with **Adaptive Moment Estimation (Adam)** as an optimizer and **Sparse Categorical Crossentropy** as a loss.

Deep learning **Convolutional Neural Network (CNN) models** to train and test, each input image will pass it through a series of **convolution layer**s with **filters (Kernals)**, **Pooling**, **fully connected layers (FC)** and apply **Softmax** function to classify an object with probabilistic values between 0 and 1.

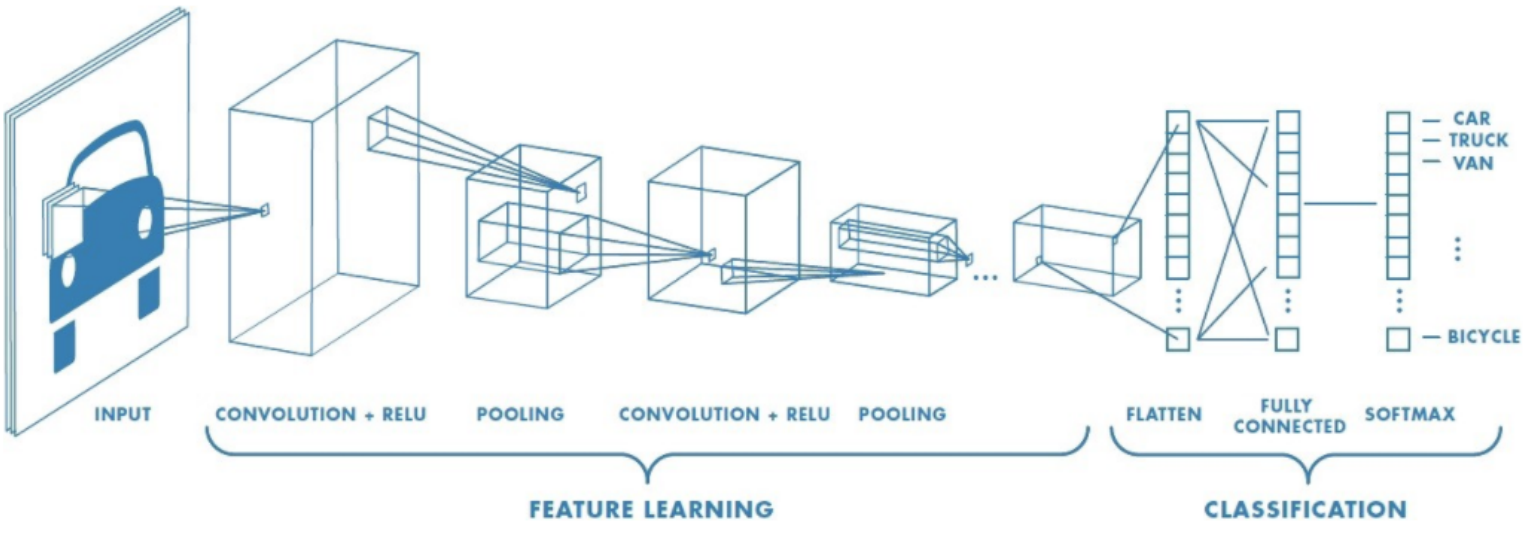


Figure 2 : Neural network with many convolutional layers

CREATING A BASELINE CONVOLUTIONAL NEURAL NETWORK (CNN) MODEL: SETTING HYPERPARAMETERS, BUILDING, COMPILING AND FITTING THE MODEL

Creating a baseline Convolutional Neural Network (CNN) model

```
# input and output shape hyperparameters

INPUT_SHAPE = X.shape[1:]
OUTPUT_SHAPE = 3
print(INPUT_SHAPE, OUTPUT_SIZE)
```

```
(431, 40, 1) 3
```

```
# creating a sequential CNN model with 2 Convolutional and Pooling layers which are later flattened and fed to dense layer
# each layer has ReLU has activation function and he_normal as kernel intializer
# ouput layer has softmax activation as we have multiclassification problem

model = Sequential()
model.add(Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', input_shape=INPUT_SHAPE))
model.add(BatchNormalization())
model.add(MaxPool2D((2,2)))
model.add(Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal'))
model.add(BatchNormalization())
model.add(MaxPool2D((2,2)))
model.add(Flatten())
model.add(Dropout(0.1))
model.add(Dense(100, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(OUTPUT_SHAPE, activation='softmax'))
```

```
# compiling the model with adam as optimizer and sparse_categorical_crossentropy as loss

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
# fitting the model with 6 epochs, a batch size of 32 and 30% of training data as validation data

hist = model.fit(X, y, epochs=6, batch_size=32, validation_split=0.3 ,verbose=2)
```

```
Epoch 1/6
1599/1599 - 62s - loss: 1.0187 - accuracy: 0.4257 - val_loss: 0.9810 - val_accuracy: 0.4480
Epoch 2/6
1599/1599 - 63s - loss: 0.8258 - accuracy: 0.5627 - val_loss: 0.7025 - val_accuracy: 0.6143
Epoch 3/6
1599/1599 - 64s - loss: 0.6263 - accuracy: 0.6533 - val_loss: 0.5241 - val_accuracy: 0.7567
Epoch 4/6
1599/1599 - 64s - loss: 0.2944 - accuracy: 0.8777 - val_loss: 0.1789 - val_accuracy: 0.9325
Epoch 5/6
1599/1599 - 63s - loss: 0.0946 - accuracy: 0.9682 - val_loss: 0.1354 - val_accuracy: 0.9558
Epoch 6/6
1599/1599 - 63s - loss: 0.0653 - accuracy: 0.9794 - val_loss: 0.2542 - val_accuracy: 0.9400
```

CREATING A BASELINE CONVOLUTIONAL NEURAL NETWORK (CNN) MODEL: SUMMARY OF THE MODEL

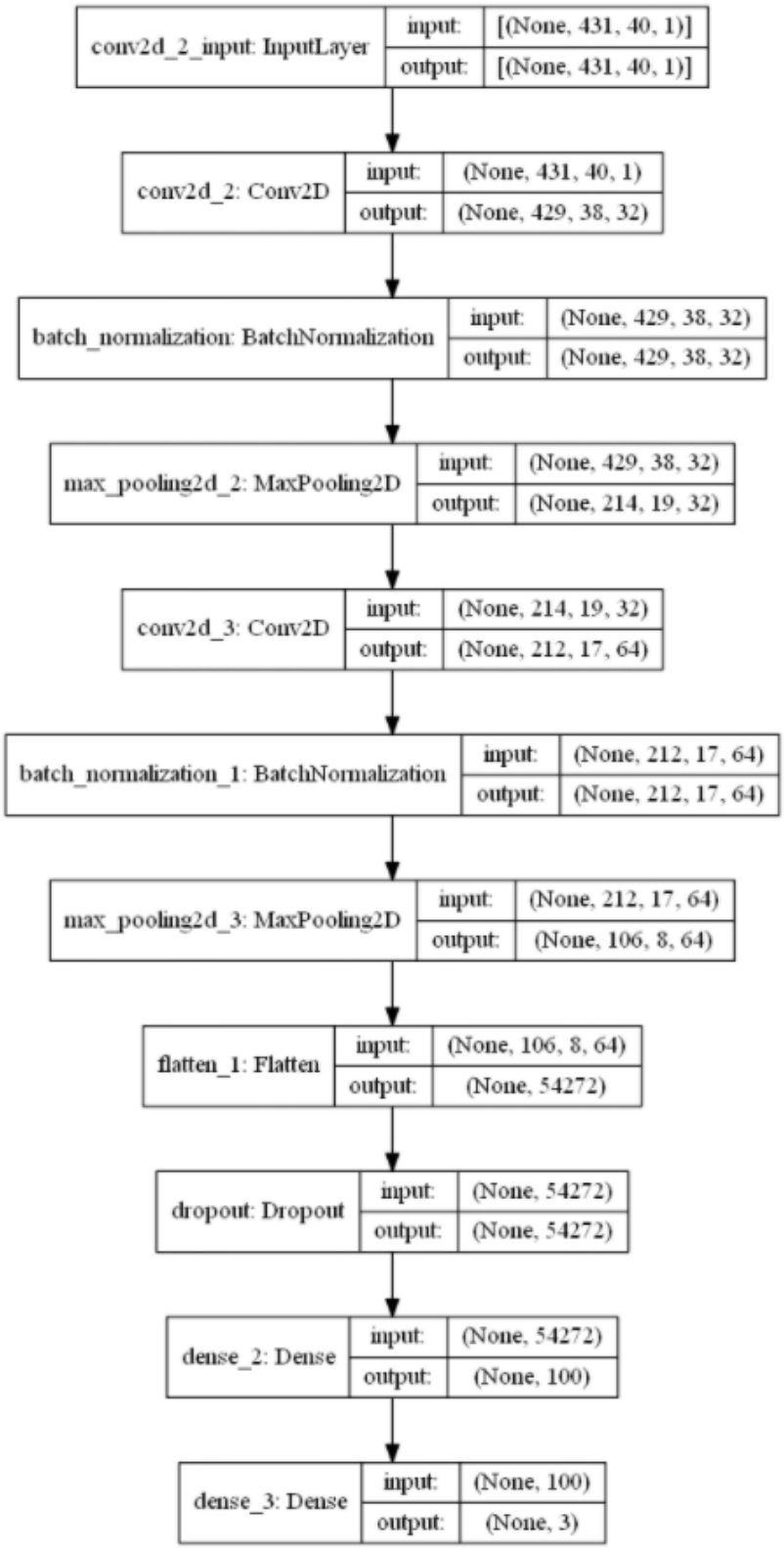
```
# summary of the model

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 429, 38, 32)       320
_________________________________________________________________
batch_normalization (BatchNo (None, 429, 38, 32)       128
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 214, 19, 32)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 212, 17, 64)       18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 212, 17, 64)       256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 106, 8, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 54272)             0
_________________________________________________________________
dropout (Dropout)            (None, 54272)             0
_________________________________________________________________
dense_2 (Dense)              (None, 100)               5427300
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 303
=================================================================
Total params: 5,446,803
Trainable params: 5,446,611
Non-trainable params: 192
_________________________________________________________________
```

CREATING A BASELINE CONVOLUTIONAL NEURAL NETWORK (CNN) MODEL: ARCHITECTURE PLOT OF THE MODEL

In [71]:
```
# architecture plot of the model

plot_model(model, show_shapes=True)
```

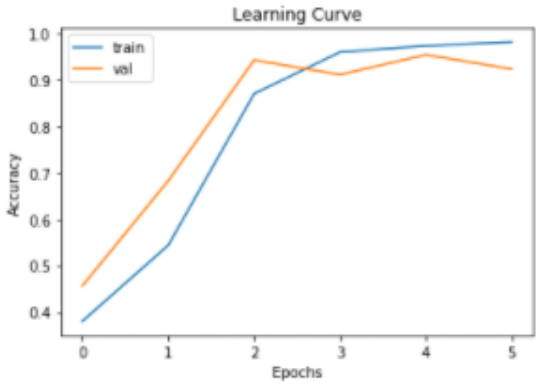
Out[71]:



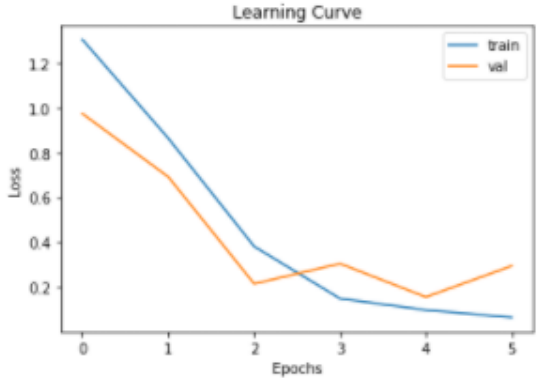
CREATING A BASELINE CONVOLUTIONAL NEURAL NETWORK (CNN) MODEL: LEARNING CURVES

By plotting the learning curves we observe that the model begins overfitting after **3 epochs** and this could lead to a decrease in the model's accuracy.

In [44]:
```
# learning curve plot of training accuracy vs validation accuracy

plt.title('Learning Curve')
plt.plot(hist.history['accuracy'], label='train')
plt.plot(hist.history['val_accuracy'], label='val')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



In [43]:
```
# learning curve plot of training accuracy vs validation accuracy

plt.title('Learning Curve')
plt.plot(hist.history['loss'], label='train')
plt.plot(hist.history['val_loss'], label='val')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



OPTIMIZING THE MODEL: GRID SEARCH

We define a function that creates a model, this is required by the KerasClassifier. We use the KerasClassifier to link our Keras model to Sklearn so that we can perform **Grid Search** on its hyperparameters.

We optimize the model by changing the learning rate hyperparameter.

In [47]:
```
# creating a function that creates and returns the model, required by KerasClassifier

def create_model(learning_rate=0.001):
    """creates, compiles and returns CNN model"""
    model = Sequential()
    model.add(Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', input_shape=INPUT_SHAPE))
    model.add(BatchNormalization())
    model.add(MaxPool2D((2,2)))
    model.add(Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal'))
    model.add(BatchNormalization())
    model.add(MaxPool2D((2,2)))
    model.add(Flatten())
    model.add(Dropout(0.1))
    model.add(Dense(100, activation='relu', kernel_initializer='he_normal'))
    model.add(Dense(OUTPUT_SHAPE, activation='softmax'))

    optimizer = Adam(learning_rate=learning_rate)
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    return model
```

In [48]:
```
# configuring the model for grid search with sklearn

model = KerasClassifier(build_fn=create_model, epochs=4, batch_size=32, verbose=2)
```

In [49]:
```
# setting learning rate hyperparameter values we want to iterate through and GridSearching

learning_rate = [0.0005, 0.001, 0.01]

param_grid = dict(learning_rate=learning_rate)
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid_search.fit(X, y)
```

```
Epoch 1/4
157/157 - 50s - loss: 2.9877 - accuracy: 0.3434
Epoch 2/4
157/157 - 48s - loss: 1.0989 - accuracy: 0.3364
Epoch 3/4
157/157 - 48s - loss: 1.0978 - accuracy: 0.3374
Epoch 4/4
157/157 - 48s - loss: 1.0956 - accuracy: 0.3358
```

In [50]:
```
# best learning rate for our model

print(f"Best : {grid_result.best_score_} using {grid_result.best_params_}")
```

```
Best : 0.6371981302897135 using {'learning_rate': 0.01}
```

In [52]:
```
# result of our gridsearch

means =grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) %r" % (mean, stdev, param))
```

```
0.501025 (0.109356) {'learning_rate': 0.0005}
0.371016 (0.057700) {'learning_rate': 0.001}
0.637198 (0.034436) {'learning_rate': 0.01}
```

CREATING THE OPTIMIZED MODEL: BUILDING, COMPILING AND FITTING THE MODEL

We create the optimized model with epochs = **4** and learning rate = **0.001**

In [47]:
```
# setting input size and output size hyperparamters

INPUT_SHAPE = X.shape[1:]
OUTPUT_SHAPE = 3
print(INPUT_SHAPE, OUTPUT_SHAPE)
```

```
(431, 40, 1) 3
```

In [48]:
```
# creating a sequential CNN model with 2 Convolutional and Pooling layers which are later flattened and fed to dense layer
# each layer has ReLU has activation function and he_normal as kernel intializer
# ouput layer has softmax activation as we have multiclassification problem

model = Sequential()
model.add(Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', input_shape=INPUT_SIZE))
model.add(BatchNormalization())
model.add(MaxPool2D((2,2)))
model.add(Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal'))
model.add(BatchNormalization())
model.add(MaxPool2D((2,2)))
model.add(Flatten())
model.add(Dropout(0.1))
model.add(Dense(100, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(OUTPUT_SIZE, activation='softmax'))
```

In [49]:
```
# compiling the model with optimized learning rate hyperparameter value

optimizer = Adam(learning_rate=0.001)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

In [50]:
```
# fitting the model

hist = model.fit(X, y, epochs=4, batch_size=32, validation_split=0.3 ,verbose=2)
```

```
Epoch 1/4
1599/1599 - 35s - loss: 0.8048 - accuracy: 0.7782 - val_loss: 0.2472 - val_accuracy: 0.9110
Epoch 2/4
1599/1599 - 26s - loss: 0.1537 - accuracy: 0.9449 - val_loss: 0.1776 - val_accuracy: 0.9385
Epoch 3/4
1599/1599 - 26s - loss: 0.0841 - accuracy: 0.9712 - val_loss: 0.2032 - val_accuracy: 0.9422
Epoch 4/4
1599/1599 - 26s - loss: 0.0629 - accuracy: 0.9800 - val_loss: 0.1931 - val_accuracy: 0.9456
```

CREATING THE OPTIMIZED MODEL: MODEL SUMMARY

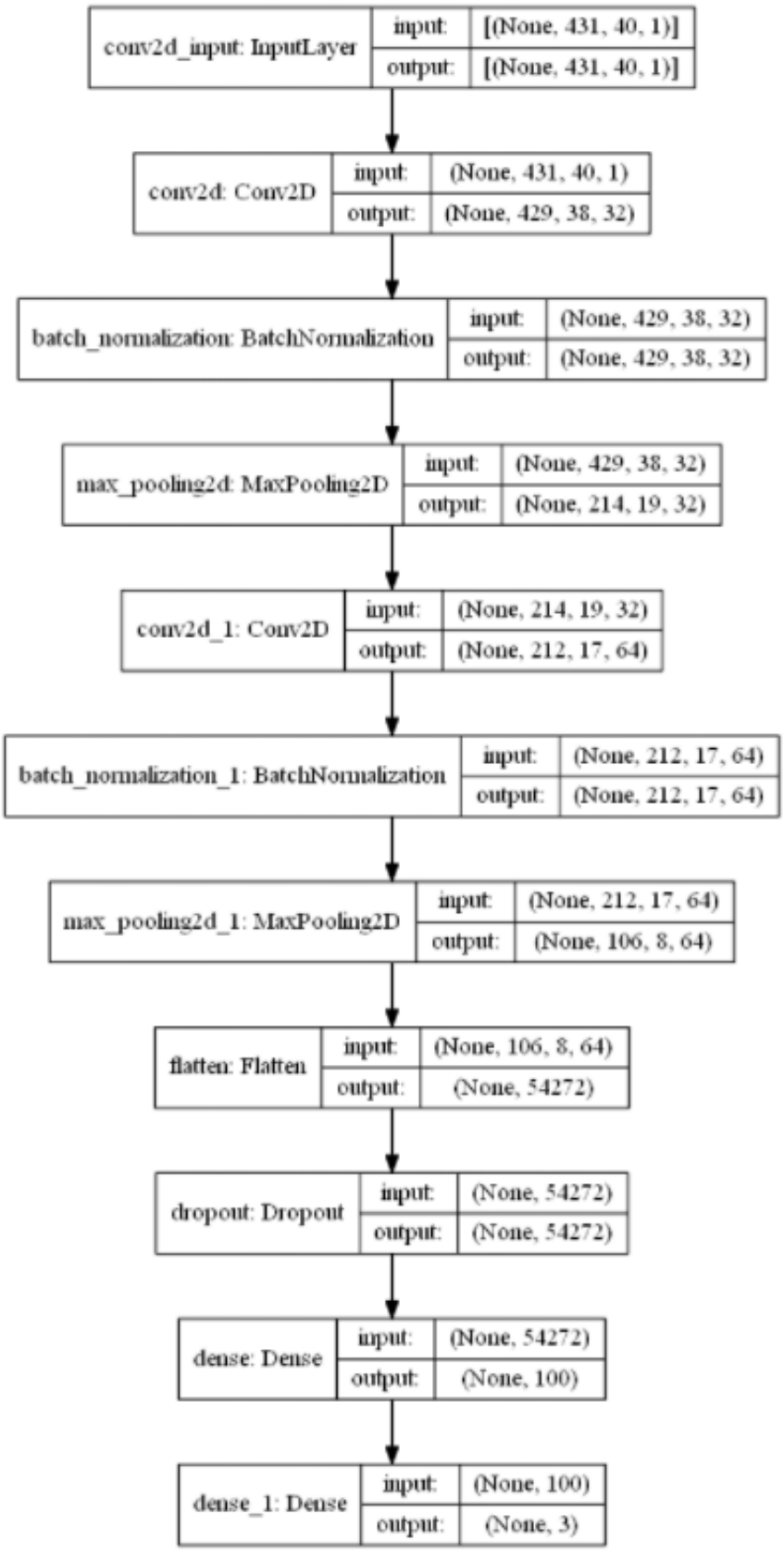
```
# summary of the model

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 429, 38, 32)       320
_________________________________________________________________
batch_normalization (BatchNo (None, 429, 38, 32)       128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 214, 19, 32)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 212, 17, 64)       18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 212, 17, 64)       256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 106, 8, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 54272)             0
_________________________________________________________________
dropout (Dropout)            (None, 54272)             0
_________________________________________________________________
dense (Dense)                (None, 100)               5427300
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 303
=================================================================
Total params: 5,446,803
Trainable params: 5,446,611
Non-trainable params: 192
_________________________________________________________________
```

CREATING THE OPTIMIZED MODEL: MODEL LEARNING CURVES

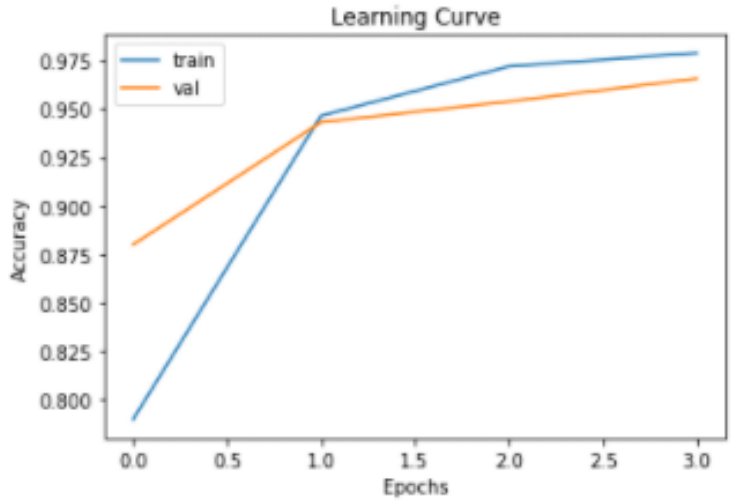
In [40]:
```
# architecture plot of the model

plot_model(model, show_shapes=True)
```

Out[40]:

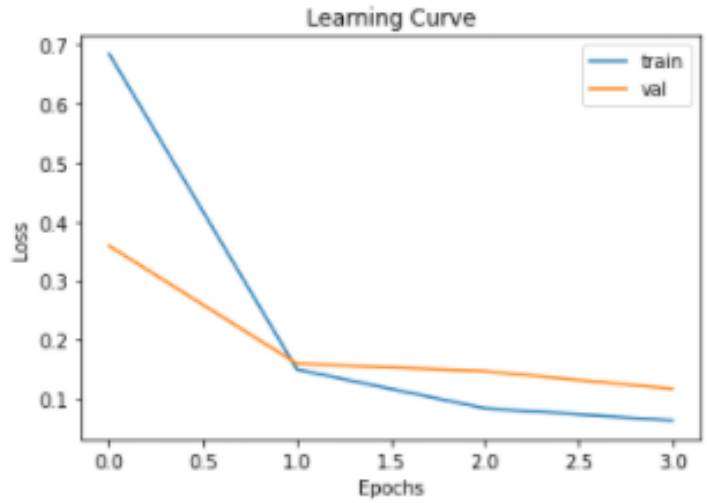


CREATING THE OPTIMIZED MODEL: LEARNING CURVES

In [41]:
```
# learning curve plot of training accuracy vs validation accuracy

plt.title('Learning Curve')
plt.plot(hist.history['accuracy'], label='train')
plt.plot(hist.history['val_accuracy'], label='val')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



In [42]:
```
# learning curve plot of training accuracy vs validation accuracy

plt.title('Learning Curve')
plt.plot(hist.history['loss'], label='train')
plt.plot(hist.history['val_loss'], label='val')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



TESTING THE MODEL: LOADING AND PREPROCESSING TEST DATA

Testing data

In [43]:
```
# iterating through the folder containing testing audio creating lists containing
# the audio filename, the language code of the audio and the language of the audio

test_path = r'C:\Users\darlu\Jupyter Notebook\Natural Language Processing\Audio NLP Files\test\test\\'
filename_list = []
language_code_list = []
language_list = []
for filename in os.listdir(test_path):
    lang = filename[:2]
    file = filename
    filename_list.append(file)
    language_code_list.append(lang)
    language_list.append(lang_code[lang])
```

In [44]:
```
# creating a dataset containg test meta data

test_data = pd.DataFrame({'filename':filename_list, 'lang_code':language_code_list, 'language':language_list})
```

In [45]:
```
# shuffling dataset

test_data = test_data.sample(frac=1).reset_index(drop=True)
```

In [46]:
```
print(test_data.shape)
test_data.head()
```

```
(540, 3)
```

Out[46]:

	filename	lang_code	language
0	de_m_923561cb71cc437382d0294cda2dd0aa.fragment...	de	German
1	es_m_08111ce8d8a1ebc6cd2c2fe82a3c98f0.fragment...	es	Spanish
2	es_f_50298ab71aaba8508ebeef49d853df11.fragment...	es	Spanish
3	de_m_923561cb71cc437382d0294cda2dd0aa.fragment...	de	German
4	de_f_83f5b79c78cfaa1a4bbd1c40f54b188e.fragment...	de	German

In [47]:
```
# iterating through all the files in testing folder and extracting lists that contain mfccs features, language of audio
# and name of the audio file

feature_list1 = []
language_list1 = []
filename_list1 = []
for index, row in tqdm(test_data.iterrows()):
    file = row['filename']
    feature = feature_extractor(file,test_path)
    lang = row['language']
    feature_list1.append(feature)
    language_list1.append(lang)
    filename_list1.append(file)
```

```
540it [00:08, 61.47it/s]
```

In [48]:
```
# creating a dataset with filename, mfccs features and language

test_feature_data = pd.DataFrame({'filename':filename_list1, 'mfccs_feature':feature_list1, 'language':language_list1})
print(test_feature_data.shape)
test_feature_data.head()
```

```
(540, 3)
```

Out[48]:

	filename	mfccs_feature	language
0	de_m_923561cb71cc437382d0294cda2dd0aa.fragment...	[[-143.80557, 112.18857, -22.420055, 33.581825...	German
1	es_m_08111ce8d8a1ebc6cd2c2fe82a3c98f0.fragment...	[[-385.5093, 169.23886, 18.599333, 35.59578, 3...	Spanish
2	es_f_50298ab71aaba8508ebeef49d853df11.fragment...	[[-380.50027, 34.887474, 57.489117, 44.12831, ...	Spanish
3	de_m_923561cb71cc437382d0294cda2dd0aa.fragment...	[[-175.05588, 70.56954, -2.6202817, 36.999048,...	German
4	de_f_83f5b79c78cfaa1a4bbd1c40f54b188e.fragment...	[[-563.02045, 2.915547, 2.7394772, 2.4591236, ...	German

In [49]:
```
X_test = np.array(feature_list1)
print(X_test.shape)
```

```
(540, 431, 40)
```

In [50]:
```
# reshaping X data to work with model

X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], X_test.shape[2], 1))
print(X_test.shape)
```

```
(540, 431, 40, 1)
```

In [51]:
```
# label encoding the categorical variables

y_test = le.transform(test_feature_data['language'])
y_test = np.array(y_test.tolist())
print(y_test.shape)
```

```
(540,)
```

TESTING THE MODEL: ACCURACY AND CONFUSION MATRIX

We obtain an accuracy of **92.41%**

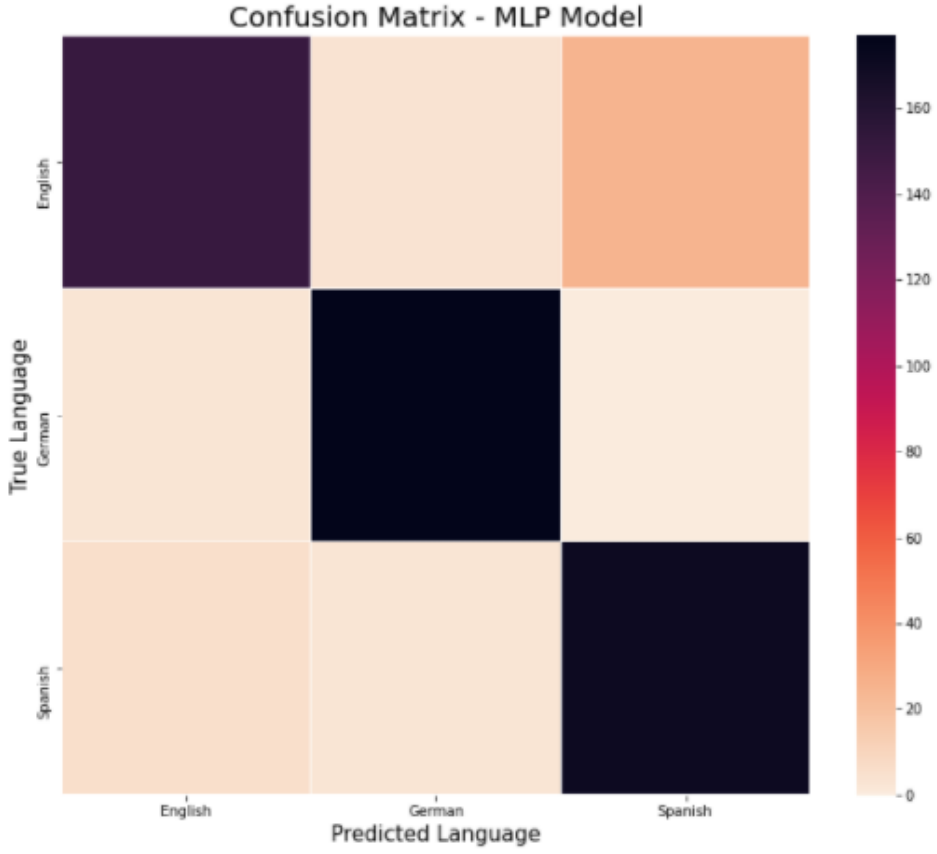
We see that **English** has been misclassified as **Spanish**. This could be probably due to similarities in words in spoken English and Spanish.

In [52]:
```
# evaluating the model

loss, accuracy = model.evaluate(X_test, y_test, verbose=2)
```

```
17/17 - 0s - loss: 0.2454 - accuracy: 0.9241
```

In [63]:
```
lang_list = ['English', 'German', 'Spanish']
y_pred_prob = model.predict(X_test) # returns an array containing probability for each category being output
y_pred = []
for i in y_pred_prob:
    out = np.argmax(i) # taking the highest probability category as output
    y_pred.append(out)
y_pred = np.array(y_pred)

cm = confusion_matrix(y_test, y_pred) # confusion matrix

# heat map of confusion matrix
plt.figure(figsize=(12,10))
plt.title('Confusion Matrix - MLP Model', fontsize=20)
sns.heatmap(cm, xticklabels=lang_list, yticklabels=lang_list, cmap='rocket_r', linecolor='white', linewidth=.005)
plt.xlabel('Predicted Language', fontsize=15)
plt.ylabel('True Language', fontsize=15)
plt.show()
```

```
<ipython-input-63-f8f9c65c4600>:13: MatplotlibDeprecationWarning: Case-insensitive properties were deprecated in 3.3 and suppor
t will be removed two minor releases later
  plt.title('Confusion Matrix - MLP Model', fontsize=20)
```



TESTING THE MODEL: CLASSIFICATION REPORT

Precision attempts to answer what proportion of positive identifications were actually correct?

Recall attempts to answer what proportion of actual positives were identified correctly?

```
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.94	0.84	0.89	180
1	0.96	0.98	0.97	180
2	0.87	0.95	0.91	180
accuracy			0.92	540
macro avg	0.93	0.92	0.92	540
weighted avg	0.93	0.92	0.92	540

SAVING AND PREDICTING USING THE MODEL:

We save the model for deployment in the future.

We load the model and deploy it to predict the language of the unknown audio.

In [66]:
```
# saving the model

model.save('audio_lang_model.h5')
```

In [69]:
```
# loading the model

model = load_model('audio_lang_model.h5')
```

In [70]:
```
file = 'en_m_b74b2bf2af570393cae91f4ed89cece7.fragment88.flac'
path = r'C:\Users\dariu\Jupyter Notebook\Natural Language Processing\Audio NLP Files\test\test\\'

inp = feature_extractor(file,path)
inp = inp.reshape(1, inp.shape[0], inp.shape[1], 1)

out = model.predict(inp)

output = np.argmax(out)
output = le.inverse_transform([output])
output
```

Out[70]:
```
array(['English'], dtype=object)
```

RESULTS:

Text Detection Model:

Our **Multilayer Perceptron (MLP) Model** for text language detection was able to classify text with an accuracy of **96.80%**

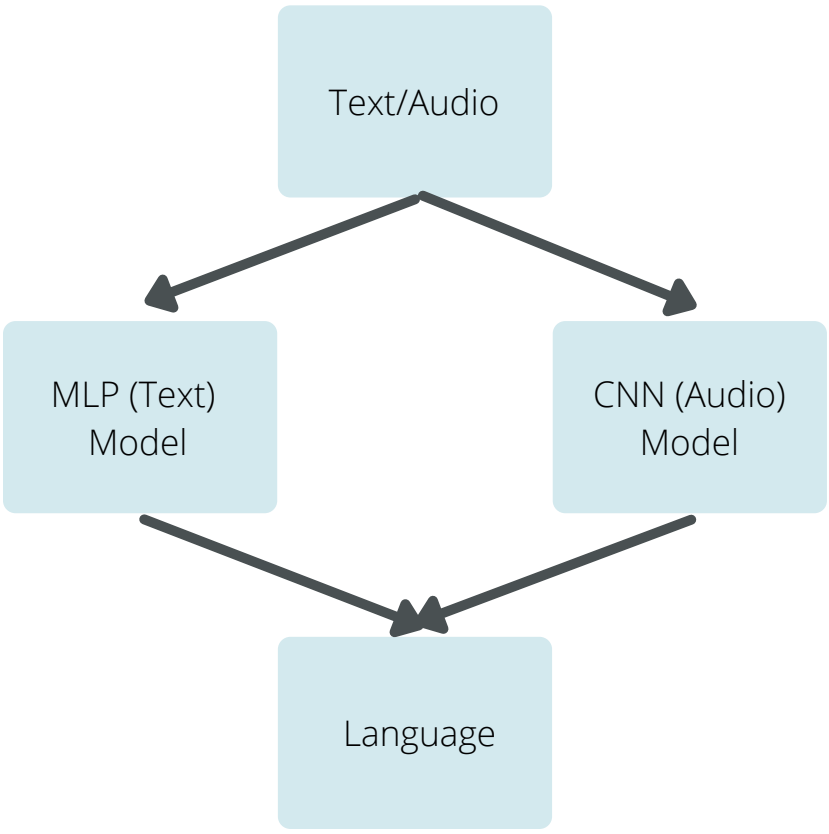
Audio Detection Model:

Our **Convolution Neural Network (CNN) Model** for audio language detection was able to classify audio with an accuracy of **92.41%**

CONCLUSION:

The goal of the project was to create a model/program that can take any text or audio and return an output that identifies the language it is in.

We developed 2 models that take in audio/text as input and return their language with a competent degree of accuracy.



REFERENCES:

- What is Natural Language Processing?
- Your Guide to Natural Language Processing (NLP)
- Natural Language Processing With Python's NLTK Package
- NLTK Book
- Natural Language Processing with Python
- AI - Natural Language Processing
- Natural Language Processing (NLP) Simplified: A Step-by-step Guide
- The WiLI benchmark dataset for written language identification
- How to solve 90% of NLP problems: a step-by-step guide
- 10+ Examples for Using CountVectorizer
- How To Improve Deep Learning Performance
- How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras
- TensorFlow 2 Tutorial: Get Started in Deep Learning With tf.keras
- Audio Data Analysis Using Deep Learning with Python
- Simple explanation of convolutional neural network
- EDA-Audio Classification Project Using Deep Learning

SUPPLEMENTARY INFORMATION

- Deep Learning for Natural Language Processing
- How to Control the Stability of Training Neural Networks With the Batch Size
- TF-IDF from scratch in python on a real-world dataset
- Mel-Frequency Cepstral Coefficients Explained Easily
- Neural Networks and Deep Learning
- Neural networks class - Université de Sherbrooke
- Deep Learning - An MIT Press book