# NLP Project Report

*by* Maruthamuthu Sowmeya

**Submission date:** 02-May-2023 11:54AM (UTC-0400)
**Submission ID:** 2082099808
**File name:** ata_analysis_on_environmental_disaster_monitoring_using_NLP.docx (80.08K)
**Word count:** 10227
**Character count:** 54272

2023

# PROJECT TITLE: Twitter data analysis on environmental disaster monitoring using NLP & ML

"All of the writing in this document is my original work, except for the code excerpts and any quoted material for which I have provided a citation."

MARUTHAMUTHU KANAGARATHINAM SOWMEYA

MKSOWMEY@SYR.EDU

# Twitter data analysis on environmental disaster monitoring using NLP & ML

## Introduction

We have a dataset that consists of tweet messages of diverse subject matters and the goal is to find if the text is talking about an environmental disaster or not which is a classification problem and series of run through of various models starting from simple machine learning model to deep learning ones including feed forward, CNN, RNN using Long short term memory and bi-directional, GRU's, TensorFlow hub pretrained feature extractor. That dataset consists of unique tweet ID, keyword that explains the tweet which has a lot of missing data, location from where the tweets are posted, Tweet itself and finally the target variable which is binary with 0/1 that returns whether the tweet is about environmental disaster or not. We have total of 7613 data points in train data and 3263 in test dataset. To predict the target variable we have primarily used text pre-processing like tokenization etc., then learned representation of word embeddings are created using those tokens and finally text vectorization is performed by giving numeric weights to the individual tokens. I have forked this project from Github repository of "DHARINI PARGUNAN"

Various models have been implemented to predict the tweet analysis of environmental disaster including

- Naive Bayes with TF-IDF Encoder (baseline)
- Feed-Forward Neural Network (dense Model)
- LSTM (RNN)
- GRU (RNN)
- Bidirectional -LSTM (RNN)
- 1D Convolutional Neural Network
- Tensorflow Hub Pretrained Feature Extractor
- Tensorflow Hub Pretrained Feature Extractor (10% of Data)

## Technology Considerations:

- Cloud Platform:
    - Google Colab

    - python 3
- Libraries & Frameworks:
    - Tensorflow
    - TextVectorization
    - iterTools
    - Numpy
    - Confusion Matrix
    - Zipfile
    - OS
    - DateTime
    - Pandas
- Data Storage:
    - Disk – 107.7 GB
- Performance:
    - System RAM – 12.7 GB
    - Version control – To keep track of changes made in Git hub to revert the changes.

## Bibliography:

https://github.com/dharinipargunan/NLP-Diaster-or-Not

## Citation:

Name of the original author: **DHARINI PARGUNAN**
Original author's GitHub username: dharinipargunan
Citation: https://github.com/dharinipargunan/NLP-Diaster-or-Not

Forked numbered, run notebook in my GitHub account URL:
https://github.com/mksowmeya/NLP-Diaster-or-Not/blob/main/NLP_disaster_numbered_file.ipynb

## Explanation of Code blocks

### Task 1:

Load the important libraries required for predicting whether the tweeted contents are about environmental disaster or not. The packages installed are tensorflow, itertools, pre-processing package that contains text vectorization, plot, numpy, sklearn metrics that contains confusion matrix, zip file, os, datetime, pandas.

**Task 2:**

The dataset is downloaded from Kaggle, that consists of data about the several tweets and it's respective associative entities.

**Task 3:**

def unzip_data(filename) unzips from target data folder and unzip data that takes a filename as a single argument and extract them.

zip_ref = zipfile.ZipFile(filename, "r") – this creates a object named zip_ref by giving the filename to the constructor and the r specifies that document is accessed in only read only mode.

Zip_ref.extractall() – extractall() method is called and unzips all files into present working directory.

zip_ref.close() – finally object zip_ref is closed so that system resources are free to use for other purpose.

**Task 4:**

The extracted data is stored in a variable named unzip_data.

**Task 5:**

This Python code utilizes the pandas library to read data from two CSV files, "train.csv" and "test.csv". The data is then stored in two separate data frame objects, train_df and test_df, respectively. These data frames can be utilized to analyse and manipulate the data within the CSV files. The read_csv() function is employed to read in the data from the CSV files and create the data frame objects. The resulting data frames contain the information from the CSV files, with each record represented by a row and each feature by a column.

**Task 6:**

The train dataset is explored using head() function where we attributes named id which contains unique tweet id's, keywords that consists of main content word of the text given in tweet, location that explains from which place the tweet is posted, text attribute consists of actual tweet contents posted by variety of people and finally a target variable by assigning a 0 or 1 to whether the tweet contents are about environmental disaster or not.

**Task 7:**

The code is shuffling the rows of a Pandas Data Frame object named train_df by using the sample method. The frac parameter is set to 1, which means that the sample should include all the rows of the Data Frame object. The random_state parameter is set to 42, which sets the random seed for reproducibility. The shuffled DataFrame is assigned to a new variable called train_df_shuffled. The resulting train_df_shuffled DataFrame will have the same columns and rows as train_df, but the rows will be randomly reordered. The head() method is used to display the first five rows of the shuffled Data Frame. Also, this breaks any pattern that's present in the dataset or also avoids noise in the dataframe.

**Task 8:**

The code is counting the number of occurrences of each unique value in the target column of a Pandas Data Frame object named train_df. The value_counts() method is used to generate a Series object that lists the counts of each unique value in the target column. The preliminary outcome of the chunk produces the frequency distribution of the target attribute in the dataset. This is useful for understanding how the target variable is distributed across the dataset, which is important in various machine learning tasks such as classification, as it can affect the choice of algorithms, evaluation metrics, and the interpretation of the results. We are able to see 4342 observations has 0 to target variable and 3271 observations has 1 as target variable that explains we have 3271 tweets that are about environmental disaster to train the model and remaining 4342 are about random tweets.

**Task 9:**

In Python, the len() function is used to determine the length of a list, tuple, or any sequence. The code len(train_df) and len(test_df) assumes that two variables train_df and test_df exist, and returns the number of rows (or observations) in each dataframe respectively. This code is likely used to check the size of the train and test datasets before and after splitting a dataset into training and testing sets. This can help ensure that the data has been split correctly, and that the training and testing sets are of the desired size. And we are able to see that train data has 7613 observations and 3263 observations by test data which is a 70:30 split for train and test data respectively.

**Task 10:**

The train_test_split() function split a data into validation & training sets for machine learning purposes. The train_df_shuffled data frame is assumed to have two columns, "text" and "target", which represent the features and labels of the training dataset, respectively. The to_numpy() method is used to convert these pandas data frames into numpy arrays. The function then splits the data into training & validation sets with a test size of 10% and returns four arrays representing the features and labels of the training and validation sets. These arrays are typically meant to train and evaluate ML models.

**Task 11:**

So, there are two sets of data: a training data and a validation one. The size of the training data is 6851, (i.e) there are 6851 sentences in training data, & the size of the validation data is 762, (i.e) there are 762 sentences in the validation data. Additionally, the no. of labels in the training data is also 6851, and the number of labels in the validation set is 762. This suggests that each sentence has a corresponding label indicating its class or category. Overall, the code and its results suggest that there is a supervised machine learning task at hand, where the aim is likely to train model to predict the labels of new sentences based on their features.

**Task 12:**

This code calculates the total no. of words in training sentences data. It does so by iterating over each sentence in the training set, splitting the sentence into words using whitespace as the separator, and then summing the total words in each phrase. The output of this code is 102087, which depicts the total number of words present in the training set. It provides insight into the size and complexity of the training dataset. Knowing the total number of words can be useful in several ways, such as determining the vocabulary size required to represent the training data or estimating the amount of computational resources needed to process the data. Additionally, it can help identify any imbalances or biases in the data and inform decisions about how to pre-process or augment the data performance  has to be improved of the machine learning model.

**Task 13:**

The train_sentences dataset's average word count per sentence is determined by this code. It does this by iterating through each sentence in the dataset, splitting the sentence into a list of words based on whitespace characters,

calculating the length of each sentence in terms of the number of words, and adding up the lengths of all the sentences. The total is then divided by the sum total of sentences in the given data to give an average number of words per sentence. Finally, the result is rounded to the nearest whole number. This code is useful for analysing the characteristics of text data and can help in tasks such as text classification or analysing the sentiments.

**Task 14:**

The code creates a TextVectorization layer for preparing text data in a machine learning model. It limits the vocabulary size to 10,000 unique words and converts the text to lowercase and removes punctuation. The text is split into individual words based on whitespace, and n-grams are not used. The output is represented as integer indexes of tokens in the vocabulary, with each sequence having a length of 15 tokens. If a sequence is shorter than 15, it is padded with zeros, and if it is longer, it is truncated.

**Task 15:**

The code adapts the TextVectorization layer, text_vectorizer, to the train_sentences data. This process involves analysing the text to determine word frequencies, building a vocabulary of the most common words based on the max_tokens parameter, and updating the layer's internal state to reflect the vocabulary and other settings. The **adapt**() method automatically applies these transformations to the training data. By adapting the layer to the training data, the model can better learn patterns and relationships in the data, and it will be ready to pre-process new text data that may contain different words or frequencies. Overall, this step is important for optimizing the text pre-processing layer's performance in the machine learning model.

**Task 16:**

The get_vocabulary() method is used to retrieve the list of words that have been assigned integer indexes based on frequency and other settings defined in the text_vectorizer layer. This vocabulary contains the words that will be used in the analysis and is a result of the preprocessing step performed by the text_vectorizer layer.
The second line of code calculates the length of the vocabulary list using the len() function. This gives us an output of 10000 words included in the vocabulary, which corresponds to the max_tokens parameter defined in the text_vectorizer layer.

Overall, these two lines of code provide us with valuable information about the vocabulary generated by the TextVectorization layer, which can help us evaluate the quality of the preprocessing step and ensure that the model is adequately prepared for the analysis of the text data.

**Task 17:**

The code snippet imports the "layers" module from TensorFlow's Keras API and creates an embedding layer named "embedding". This layer is configured to process input integer sequences with a maximum integer value of 9999, convert each input integer into a dense vector of length 128, and process sequences with a maximum length of 15 integers.
The layer of embedding is commonly used as the 1st layer in neural networks that handle text processing tasks like sentiment analysis and language translation. It transforms sequences of integers into dense vector representations, which are learned during neural network training. This enables the network to process sequences of varying lengths and capture the meaning of words and phrases in the input text.

**Task 18:**

The TfidfVectorizer class is imported from Scikit-learn's feature_extraction.text module, which will be used to transform the text data into a numerical format.
from sklearn.naive_bayes import MultinomialNB - Imports the MultinomialNB class from Scikit-learn's naive_bayes module, which is a simple and effective algorithm for text classification.
from sklearn.pipeline import Pipeline - Imports the Pipeline class from Scikit-learn's pipeline module, which will be used to build a pipeline for our text classification model.

**Task 19:**

It creates a pipeline object called "model_0" for text classification. The pipeline has two stages:
"tfidf": This stage converts text data to numerical data using the TF-IDF algorithm. It creates a matrix of numerical features to represent the input text data.
"clf": This stage trains a Naive Bayes classifier using the numerical features from the previous stage to predict the label or category of each document.

In summary, this pipeline can be used for text classification tasks where the aim is to predict the category of each document. The TfidfVectorizer stage transforms text data to numerical data and the MultinomialNB classifier predicts the labels based on this numerical representation.

**Task 20:**

The code trains a machine learning model called model_0 on a dataset of input sentences (train_sentences) and their corresponding target labels (train_labels).
During the training process, model_0 adjusts its internal parameters to learn patterns or relationships between the input sentences and target labels in order to minimize a chosen loss function. This is typically done using an optimization algorithm such as stochastic gradient descent.
Once trained, model_0 should be able to accurately predict target labels for new input sentences that it hasn't seen before.

**Task 21:**

The machine learning model called model_0 is trained to make predictions on a validation dataset of sentences called val_sentences.
The model consists of predict() method that takes the validation sentences in place of input and outputs the target labels for each sentence as final prediction. These predicted labels are stored in a variable called baseline_preds.
It's important to verify that the accuracy of the predictions are based on the quality of the trained model & the similarity with the validation data to the training data. It's common to estimate the working of the training data model upon a separate test data to get a better idea of how well it can generalize to new, unseen data.

**Task 22:**

It defines a Python function called calculate_results that takes two arguments: y_true and y_pred, which are assumed to be the true labels and predicted labels, respectively, for a set of data points. The function uses three evaluation metrics from the scikit-learn library, namely accuracy_score, precision_recall_fscore_support.

accuracy_score() calculates the proportion of data points for which the predicted label matches the true label, which is called the accuracy of the model's predictions.
precision_recall_fscore_support() calculates several criterion, like F1 score precision & recall for each class in the data. These metrics can provide insight into how well the model identifies positive and negative instances, as well as balancing precision and recall.
The calculate_results function calculates and returns a dictionary of evaluation metrics based on the input y_true and y_pred. Specifically, it calculates the model_accuracy metric, which is the accuracy of the model's predictions expressed as a percentage, and the model_precision, model_recall, and model_f1 metrics, which are the average weights of the F1 gain, precision & recall for all classes. The function then returns a dictionary named model_results that contains these evaluation metrics under the keys "accuracy", "precision", "recall", and "f1". This dictionary is used as performance evaluation metric of our machine learning model.

**Task 23:**

The provided code evaluates a binary classification model on the validation set using the true labels and predicted labels generated by a baseline model.
The calculate_results function is utilized to calculate the performance of model measures F1 score, precision & recall, Accuracy were based on the predicted and true labels. The resulting performance scores are stored in a dictionary object named baseline_results.
The output shows that the baseline model achieved an accuracy score of 79.26%, indicating that it correctly predicted the label for almost 80% of the validation set instances. The precision result of 81.11% indicates that the model accurately identified 81.11% of the +ve instances out of the total positive predictions. The recall gain of 79.27% presents that the model perfectly identified 79.27% of the +ve instances in the validation set. The F1 score of 78.62% provides a single score that combines precision and recall. Overall, the baseline model performs reasonably well, but there may be room for further analysis and improvements.

**Task 24:**

This code build model with functional API and one with vectorized text layer. It creates a neural network model that takes in text data and classifies it into two categories. The text data is first converted into a sequence of numerical values

using a vocabulary, then mapped to dense vector representations. A global max pooling operation is then applied to the vector output representation layer to get a fixed length of vector for each text input. Finally, an activation function of sigmoid is used in final layering to output a binary classification result for each text input. The name of the model is "model_1_dense", and each layer is added to the model sequentially.

**Task 25:**

This code displays a summary of the neural network model created earlier using the model_2 object. The model's name is "model_1_dense". The summary lists the type of each layer, its output shape, and the no. of criterion in the layer.
The model consists of four layers: a TextVectorization layer, an embedding , a GlobalMaxPooling1D and a Dense sheet.
The TextVectorization layer has no trainable parameters since it only maps text input to numerical representations based on a pre-defined vocabulary.
The Embedding layer has 1,280,000 trainable parameters because it maps each token ID to a dense vector of length 128.
The GlobalMaxPooling1D layer has no trainable parameters since it applies a fixed operation to the Embedding layer's output.
The Dense layer has 129 trainable parameters since it maps the fixed-length vector representation to a single output unit.
Overall, the model has a total of 1,280,129 parameters, all of which are trainable.

**Task 26:**

It compiles the neural network model created earlier (model_2) using the compile method with the following arguments:
loss: the loss function to be used during training. Here, binary_crossentropy is used since the model is performing binary classification.
optimizer: the optimization algorithm to be used during training. Here, Adam is used as the optimizer.
metrics: the evaluation metrics to be used during training and evaluation. Here, we are using "accuracy" as the evaluation metric.
This step prepares for training of the model by enumerating the loss function, optimizer, and metrics for evaluation to be used.

**Task 27:**

This code trains the neural network model (model_2) on the training data (train_sentences and train_labels) for 5 epochs, while also validating the model's performance on the validation data (val_sentences and val_labels) during training. The training progress and validation metrics are printed after each epoch.
The fit method trains the model for the specified number of epochs using the training data points and updates the parameters to reduces the loss function. During training, the model conduct on the validation data is also evaluated & the training history is stored in the model_2_history object.
In this specific training run, we see that the model's loss and accuracy improve with each epoch, both for the validation & training data. After the last epoch, the model training accuracy is 93.15% and validation accuracy is 78.08%. This indicates that the model is overfitted with the training dataset, as the validation accuracy is much lesser than the training accuracy.
Overall, this code trains a neural network model for text classification, using a combination of embedding and dense layers, and evaluates the model's performance on a validation dataset during training to prevent overfitting. The results of the training are printed after each epoch, allowing us to monitor the model's performance over time.

**Task 28:**

In this code snippet, the trained model_2 is applied to make predictions on the validation dataset val_sentences. The predictions are represented as probabilities of the input sentences belonging to the positive class (spam), and these probabilities are stored in model_2_pred_prob.
To convert these probabilities into binary predictions, the round function is used to round the probabilities to the nearest integer. The squeeze function is then used to remove any unnecessary dimensions in the tensor, resulting in the binary predictions being stored in model_2_preds.
Finally, the code prints a summary of the prediction results which shows that the predictions were made on 24 instances in the validation dataset, and that the processing time for each instance was 0.002 seconds.

**Task 29:**

This code computes metrics performance for the model_2 on the validation dataset. The metrics include, F1 score, precision, recall & Accuracy , and they are used for evaluating model's performance in identifying spam messages.

The evaluation results show that the model_2 classified 78.08% of the validation dataset instances correctly. When the model predicted a message as spam, it was correct 78.6% of the time. The model accurately identified 78.0% of the actual spam messages. The f1 gain, which is the average of precision & recall, is 0.777.
In summary, the results indicate that the model_2 performs reasonably well in identifying spam messages. However, there is still room for improvement, particularly in terms of recall.

**Task 30:**

The code retrieves the vocabulary (i.e., the set of unique words) that was created by the text_vectorizer object. The variable words_in_vocab is a list of strings that contains all the words in the vocabulary, ordered by frequency (i.e., the most frequent words appear first).
The code then prints the first 20 words in the vocabulary using Python's list slicing notation. As we can see, the most frequent word is an empty string (' '), which corresponds to the padding token that was added to make all the sentences the same length. The second most frequent word is [UNK], which corresponds to out-of-vocabulary (OOV) tokens, i.e., words that were not present in the training data.

**Task 31:**

It looks like the code is extracting the weights of the embedding layer from a neural network model called model_2. The weights are stored in a list and are accessed using the get_weights() method of the embedding layer.
The weights themselves are a single 2D numpy array of shape (vocab_size, embedding_dim). Each row of the array depicts the embedding of vector for a words present in the vocabulary of the model. The values in the array are the actual weights that the model has learned during training to represent words as dense vectors in the embedding space.

**Task 32:**

The code is using a neural network model that includes an layer of embedding, where every word belonging to the input vocabulary is mapped to a dense vector representation (also known as an embedding).

embed_weights[0] refers to the weights (i.e., the actual embedding vectors) of this layer. The shape of (10000, 128) indicates that there are 10,000 words in the vocabulary, and each word is represented by a dense vector of size 128. In other words, this model's embedding layer is using a 128-dimensional vector to represent each word in the vocabulary. The total amount of words in the vocabulary is represented by the first measure (10000), and the size of each embedding vector is represented by the second measure (128).

**Task 33:**

This code is building RNN model using LSTM. It creates a sequential model for a (RNN) using (LSTM) cells.
The input shape is a single string, and it is passed through a text vectorizer layer before being fed into an embedding layer. The embedding layer converts the words into a dense vector representation that can be understood by the LSTM cells.
The LSTM layer has 64 units, which means that it will be able to recall information from the before 64 time steps. The layer LSTM is followed by a single output from dense layer and further activation function of sigmoid is defined, (i.e) is suitable for binary classification problems.
This model can be trained on text data for tasks such as sentiment analysis or language modelling, where the aim is to predict the consequential word in a sequence.

**Task 34:**

The summary of model_3 is executed in this code. The model has three layers: a text vectorization layer, an embedding layer, an LSTM layer, and a dense output stack.
The first layer is a TextVectorization layer that converts input text into numerical vectors. The shape of this layer is (None, 15), which means that it takes input sequences of up to 15 tokens.
The second one is an Embedding layer that maps input numerical vectors to dense embedding vectors of size 128. The resulting shape of this layer is (None, 15, 128), which means that it produces a 2D tensor of shape (batch_size, sequence_length, embedding_dimension).
The third layer is an LSTM layer that processes the input sequence of embedding vectors and outputs a single vector of size 64. The final shape of this layer is (None, 64), which means that it produces a 2D tensor of shape (batch_size, lstm_units).

The fourth is a Dense layer which is the final one that produces a single output value. The shape of this layer is (None, 1), which means that it produces a 2D tensor of shape (batch_size, 1).
The model has a total of 1,329,473 parameters, which are all trainable.

**Task 35:**

The code compiles a deep learning model named "model_3". It use the binary cross entropy with loss function, which is usually applied in binary classification problems to measure the difference between predicted and actual class probabilities. The optimizer used is Adam, which is an adaptive optimization algorithm popular for its efficiency in convergence. Lastly, the accuracy metric is used to detect performance of the model, measuring the proportion of correct predictions out of all predictions made.

**Task 36:**

This code trains the compiled "model_3" on the learning dataset and validates its results on a separate validation data. The training is done for five epochs. The training dataset is provided as "train_sentences" and "train_labels", while the validation dataset is provided as "val_sentences" and "val_labels".
During the training process, the model's loss and accuracy metrics are monitored and printed for each epoch. The validation loss and accuracy are also computed and printed for each epoch.
The training process takes 5 epochs to complete. In each epoch, it's trained on 215 batches of data. The training process updates the model's parameters to minimize the binary cross entropy function as given as loss. The validation acc. and loss measures indicate how well the model is generalizing to the validation dataset.
After training, the final model achieved a validation accuracy of 0.7638 on the validation dataset, which means that it correctly predicted the sentiment of 76.38% of the validation samples.

**Task 37:**

This code uses the "model_3" to forecast the sentiment of the validation sentences stored in "val_sentences".
The "model_3.predict" function returns the predicted probability of the positive sentiment for each sentence in the validation set, which is stored in the "model_3_pred_prob" variable.

The "tf.round" function rounds the predicted probabilities to the nearest integer, and the "tf.squeeze" function removes any dimensions with a size of 1 from the output. This creates a 1D array of predicted sentiment labels, which is stored in "model_3_preds".
Finally, the code prints a summary of the prediction results which shows that the predictions were made on 24 instances in the validation dataset, and that the processing time for each instance was 0.003 seconds.

**Task 38:**

The evaluation of performance of "model_3" on the validation dataset is taken into consideration. The evaluation metrics computed are F1, precision, recall & Accuracy which are the measures of the model ability to predict the correct sentiment label for the given text.
The results of the evaluation are stored in a dictionary called "model_3_results". The dictionary contains the computed evaluation metric scores, which indicate how well the model performed on the validation dataset. For example, the accuracy of the model is 76.38%, which means it predicted the correct sentiment label for 76.38% of the validation dataset. Similarly, the precision gain of the RNN using LSTM is 0.766, which indicates that out of which all the predictions made by the given model that were labelled +ve, 76.6% were correct. The recall outcome of the above model is 0.764, (i.e) the model perfectly identified 76.4% of the +ve sentiment labels in the validation set. Finally, the F1 result of the model is 0.761, (i.e) a balanced measure of the model's recall & precision.
These evaluation metric scores can be used to assess the performance of various models or to fine-tune the hyperparameters to improve its performance on sentiment analysis tasks.

**Task 39:**

This code creates a sequential model called "model_4" for text classification with recurrent neural networks using GRU. It takes in strings as input and applies a series of layers to convert them into a binary classification label. The layers include a text vectorization layer, an embedding layer, two stacked GRU layers, and a single neuron layer which is a dense layer and the activation function as sigmoid. The GRU layers can capture sequential dependencies in the data, and the dense layer that gives predicted categorized label. The model can be trained and optimized to improve its performance and accuracy in sentiment classification.

**Task 40:**

It's the summary of the "model_4" sequential model for text classification. It includes a text vectorization layer, an embedding layer with 128 units, two stacked GRU layers with 64 units each, and a dense layer with a single neuron. The text vectorization layer has no trainable parameters. The model has a total of 1,342,273 trainable parameters and can be optimized by fine tuning the parameters during training. The summary provides information on the output shape and the number of trainable and non-trainable parameters for each layer in the model.

**Task 41:**

This is the "model_4" compilation with Accuracy as the evaluation metric, Adam as the optimizer, and cross-entropy as binary for the function loss. Usually for 2 category classification problems cross entropy is used as binary with loss func., and the optimization algo as Adam that uses adaptive learning rates to update model weights during training. The accuracy metric evaluates the execution of the model during training & validation.

**Task 42:**

The code is training model_4 on the training data and evaluating it on the validation data for five epochs. During each epoch, the model calculates the accuracy & loss on the training & validation data. The results show that the model is overfitting, as the training set accuracy is more than that on the validation. Specifically, the model achieves a 92.59% Accuracy on the training set and 76.90% on the validation in first epoch. In the subsequent epochs, the accuracy on the validation data decreases while the loss increases. Overall, model_4 appears to be performing worse than the previous models.

**Task 43:**

This code is using the trained model model_4 to make predictions on the validation sentences val_sentences. model_4.predict() returns the predicted probabilities for each sentence, and then tf.squeeze(tf.round()) rounds those probabilities and squeezes them to obtain predicted labels. The output shows that the predictions have been made successfully for all 24 validation sentences with each processing instance duration of about 1.003 seconds

**Task 44:**

The chunk of code evaluates model_4 on the val. set. model_4_pred_prob is used to generate the predicted probabilities for each example in the validation set. model_4_preds rounds these predicted probabilities to the nearest integer (0 or 1) to generate the final predictions for each example in the validation set. model_4_results calculates the performance metrics (f1-score, precision, recall and accuracy) of model_4 on the validation set using the calculate_results function. The results show that model_4 has an accuracy of 75.98%, precision of 76.05%, recall of 75.98%, and f1-score of 75.81% on the validation set. This indicates that model_4 is performing slightly worse than model_3.

**Task 45:**

We are building a model for RNN using Bidirectional LSTM using keras API in Tensorflow named model_5_bidirectional consisting of several layers: an input layer with a single string input, a text vectorization layer to convert the input text to numerical vectors, an embedding layer to transform the vectors into dense representations, a bidirectional LSTM layer with 64 hidden units to capture sequence details from the directions, and an output layer as dense layer with sigmoid activation function to predict binary classification. The model is designed to learn from text data and make binary predictions.

**Task 46:**

The code shows a summary of the neural network model called "model_5_bidirectional". The summary provides a table of information about the layers in the model, including their types, output shapes, and number of trainable parameters. The model has four layers: a text vectorization layer, an embedding layer, a bidirectional LSTM layer & a dense output layer. The text vectorization layer has no trainable parameters because it is only used to transform the input text into numerical vectors. The embedding layer has 1,280,000 trainable parameters because it maps each token in the input text to a dense representation of size 128. The bidirectional LSTM layer has 98,816 trainable parameters because it has 64 hidden units and processes the input sequences in both forward and backward directions. The dense output layer has 129 trainable parameters because it has one neuron that outputs a binary prediction. The model has a total of 1,378,945 trainable parameters and no non-trainable parameters.

**Task 47:**

The code compiles the neural network model called "model_5_bidirectional" using the Keras API in TensorFlow. The compile() function configures the learning process by specifying the loss function, optimizer, and metrics to evaluate the execution of the model. In this case, the model is trained for binary classification, so the binary_crossentropy function is used to give the difference between the predicted and actual labels. Adam's optimizer is taken to improve the model's weights during training. Finally, the accuracy measures judge the evaluation of model based on the ratio of accurately predicted labels. After compilation, the model is ready for training using the fit() function.

**Task 48:**

This code trains a deep learning model model_5 on train_sentences and train_labels data for 5 epochs using the fit() function. The validation data is provided through the validation_data parameter and consists of val_sentences and val_labels.
During training, the model computes the training loss and accuracy, as well as the validation loss and accuracy. The training progress is displayed for each epoch, including the time taken to complete each epoch.
The accuracy parameter represents the acc. of the model upon training set, which is 0.9799 after the final epoch. This means that the model correctly predicted the sentiment of 97.99% of the training data.
The val_accuracy parameter represents the acc. of the model on the validation data, which is 0.7651 after the final epoch. This means that the model correctly predicted the sentiment of 76.51% of the validation data.
It's worth noting that the val. accuracy is less than the training accuracy, that is a common issue in deep learning models. This indicates that the model could be overfitted to the training data and not working well to new data.

**Task 49:**

This code calls a function calculate_results with two arguments: the val_labels (validation labels) and model_5_preds (model predictions on validation data). The function calculate_results takes these two arguments and computes four evaluation metrics: F1-score, precision, recall, & accuracy. The evaluation

measures are returned as a dictionary, with keys for each metric and their respective values.
The returned results for model_5_results show that the model accuracy on the validation data is 76.51%, which means that the model accurately predicted the sentiment of the movie reviews for 76.51% of the validation set. The precision is 0.77, meaning that out of all the positive predictions made by the model, 77% were correct. Recall is 0.77, meaning that 77% of all the positive reviews in the validation set were correctly identified by the model. The harmonic mean of precision and recall is 0.76 and provides a comprehensive evaluation of the model's performance.

**Task 50:**

This code calls a function calculate_results with two arguments: the val_labels (validation labels) and model_5_preds (model predictions on validation data). The function calculate_results takes these two arguments and computes four evaluation metrics: F1-score, precision, recall, & accuracy. The evaluation metrics are returned as a dictionary, with keys for each metric and their respective values.
The returned results for model_5_results show that the model's accuracy on the validation set is 76.51%, which means that the model perfectly predicted the sentiment of the movie reviews for 76.51% of the validation set. The precision is 0.77, meaning that out of all the positive predictions made by the model, 77% were correct. The recall is 0.77, indicating that the model correctly identified 77% of all the positive reviews in the validation set. The F1-score is 0.76, which is the harmonic mean of precision and recall, and provides a combined measure of the model's performance.

**Task 51:**

We are building a model for convolutional neural networks that defines a sequential pattern named "model_6_CNN". A string of text is taken as input, represented as a 1D tensor. The first layer in the model is a text vectorization layer which tokenizes and preprocesses the input text data. Then, an embedding layer is added which converts the tokenized text into dense vectors of fixed size.
The model then adds a 1D convolutional layer with 128 filters, a kernel size of 5, 1 stride, and uses the activation function of rectified linear unit. This layer performs a sliding window over the input text and extracts local features.

Next, a global-max pooling layer is added, finds the max. number of individual filter and reduces the feature maps to a fixed length. This layer is taken upon two fully connected layers, one with 64 units and ReLU activation, and the other with a single sigmoid output unit. The final layer gives a probability between 0 & 1, indicating the likelihood of the text given as input belonging to a particular class.

**Task 52:**

The code defines a convolutional neural network (CNN) model named "model_6_CNN" using the Keras API in TensorFlow. The summary function displays the architecture of the model.
The model takes a string as input and applies text vectorization to convert it to a sequence of token indices. The text vectorization layer is not trainable, meaning the weights are not updated during training.
The model then applies an embedding that transforms the token indices to dense vectors of fixed size (128 in this case).
Next, the model applies a 1D convolutional layer with 128 filters, a filter size of 5, a 1 stride, and Rectified Linear Unit activation function. This layer learns features from the input sequences and reduces their length.
After the convolutional layer, the model applies a global max pooling layer that reduces to convolutional layer to a single vector.
Then, the model applies two dense layers, with 64 and 1 units, respectively. The first dense output has a Rectified linear unit activation function, and the final dense output uses a sigmoid activation func.
The model has a total of 1,370,369 parameters, all of which are trainable.

**Task 53:**

This code compiles the model_6 by specifying the loss function to use during training, the optimizer to use, and the evaluation metrics to monitor. Specifically, loss = 'binary_crossentropy' specifies binary cross-entropy loss function as the metric to optimize the model for binary classification. The optimizer = tf.keras.optimizers.Adam() specifies the Adam optimizer which is an efficient gradient descent optimization algorithm. Finally, the metrics = ["accuracy"] specifies the evaluation metric to monitor, which in this case is accuracy, i.e., the ratio of perfectly predicted binary labels.

**Task 54:**

The code is training model_6 on training datapoints (train_sentences and train_labels) for 5 epochs and validating it on the validation data (val_sentences and val_labels). The training progress and validation performance are printed out for each epoch.
The results show the loss and accuracy metrics for each epoch of the training process. The loss metric measures the error of the model's predictions relative to the actual labels. The accuracy metric measures the percentage of correctly classified samples in the training and validation data.
The accuracy achieved on the training data (accuracy) ranges from 0.9803 in the 1st epoch to 0.9845 in last epoch. The val_accuracy ranges from 0.7625 in the first epoch to 0.7559 in the last epoch.
Overall, the training data accuracy is high, but the val. accuracy is not as high and tends to decrease after the second epoch, indicating its'a an overfitting for the training data.

**Task 55:**

This code predicts the probabilities of the validation sentences using the trained model model_6.
Then, it rounds the predicted probabilities to either 0 or 1 by applying the tf.round() function and squeezing the resulting tensor to remove any dimensions of size 1 using the tf.squeeze() function. This creates the predicted labels for the validation set.
The output shows that the prediction process was performed on 24 batches of data, each batch consisting of several sentences. The processing time for each batch is 3.004 seconds.

**Task 56:**

The code uses the calculate_results() function to calculate the model's performance on the validation set. The val_labels contains the actual labels for the validation set, and model_6_preds contains the predicted labels generated by the model_6.
The calculate_results() function returns a dictionary containing the f1 score, precision, recall, and accuracy for the model. The accuracy of model_6 on the validation set is 51.97%, which is quite low compared to the other models. The precision is 38.08%, which means that out of all the examples predicted as positive, only 38.08% were actually positive. The recall is 51.97%, which means that out of all the positive examples, the model correctly identified only

51.97% of them. The f1 score is 38.43%, which is the harmonic mean of precision and recall.

**Task 57:**

This code imports the TensorFlow Hub library and loads the Universal Sentence Encoder model from version 4 available on the TensorFlow Hub website.
The code then uses the loaded model to generate embeddings (vector representations) for two example sentences by calling the embed() function with a list of strings as input.

**Task 58:**

The resulting embeddings variable will contain a 2D array of shape (2, 512) where each row corresponds to the embedding for one of the input sentences, and each column contains a float value representing a feature of the embedding vector. The code is calling the shape attribute on the first element of a variable named embeddings.
Assuming that embeddings is a list or array of embeddings, embeddings[0] accesses the first embedding in the list, and shape returns the dimensions of that embedding.
In this case, the output of the code is (512,), which means that the first embedding is a one-dimensional vector with 512 elements. This suggests that the embeddings were likely generated using a language model or other deep learning model that outputs dense representations of text, where each element in the vector represents a different feature or aspect of the text.

**Task 59:**

Here we are using Transfer Learning for NLP, specially using tensorflow Hub's universal sentence encoder. The given code initializes a Keras layer that uses the Universal Sentence Encoder (USE) from TensorFlow Hub to encode variable-length text sentences into a fixed-length vector representation. The layer is set to be non-trainable, and the input data type is set to string. This layer can be used in a deep learning model for various natural language processing tasks and in our case we are using to detect the environmental disaster from text which is based on text classification.

**Task 60:**

The given code initializes a Keras Sequential model named model_7 with three layers. The first layer uses the Universal Sentence Encoder (USE) to encode variable-length text sentences into vector representation of fixed length. The second layer is a neural network layer of fully connected with 128 hidden units and ReLU activation function. The third layer is another fully connected neural network layer with a single output unit and sigmoid activation function. These layers are stacked sequentially. The name parameter is used to assign a name to the model.

**Task 61:**

The given code initializes a Keras model that uses the Universal Sentence Encoder (USE) to encode variable-length text sentences into a fixed length representation if vectors. The model consists of three layers - the USE layer, a neural network fully connected with 128 latent units and ReLU activation function. The model is designed for 2-way classification using a activation function of sigmoid in its final output layer. It is assembled with 2 fold cross-entropy as the loss-function, along with the Adam's optimizer, and accuracy is used as the evaluation metric during training and testing. These specified parameters enable the model to be trained and evaluated appropriately for its intended purpose.

**Task 62:**

The given code prints a summary of the model_7 Keras model. The output shows the layers in the model, the output shape of each layer, and the number of trainable and non-trainable parameters.
The model has three layers:
The first layer is the USE layer with an output shape of (None, 512), which means that it outputs a tensor of rank 2 (i.e., a matrix) with an unknown batch size and 512 columns.
The second layer is a fully connected neural network layer with 128 hidden units and an output shape of (None, 128).
The third layer is another fully connected neural network layer with a single output unit for binary classification and an output shape of (None, 1).
The model has a total of 256,863,617 parameters, out of which only 65,793 are trainable and the rest are non-trainable. The USE layer has a massive number of non-trainable parameters (256,797,824) because it is pre trained on a huge corpus of text data.

The summary provides useful information about the structure and size of the model, which can help in debugging and optimizing the model for better performance.

**Task 63:**

The code is training a neural network model (model_7) to classify the sentiment of textual data into positive or negative categories. The input data consists of train_sentences and train_labels, and the validation data is val_sentences and val_labels. The trained model is run for 15 epochs.
The training history of the model is printed, showing the validation & training loss & accuracy for each epoch. The loss parameter refers to the loss function used to train the model, and the accuracy parameter refers to the accuracy of the model in predicting the correct label. Similarly, val_loss and val_accuracy represent the loss & accuracy of the model on the val. set.
The results show that the model got a 89.55% accuracy on the training datapoints and 82.02% on the validation set after 15 epochs. The val. accuracy is little lower than the accuracy of training data, which is expected due to overfitting of the model to the training set.
Overall, the model performed well in classifying the sentiment of textual data, achieving an accuracy of over 80% on the validation set. However, further optimization of the model may be required to reduce overfitting and improve generalization to new data.

**Task 64:**

The code above uses the trained model_7 to make predictions on the validation sentences (val_sentences) and store the predicted probabilities and the predicted classes.
The model_7.predict(val_sentences) function computes the predicted probabilities for each sentence in val_sentences, resulting in an array of shape (len(val_sentences), 1) where each value is a probability b/w 0 & 1 indicating the likelihood of the sentence affliated to the +ve class.
The tf.squeeze function removes any dimensions of size 1, resulting in an array of shape (len(val_sentences),) containing the predicted probabilities for each sentence.
Finally, the tf.round function rounds the predicted probabilities to the nearest integer, resulting in an array of shape (len(val_sentences),) containing the predicted class labels (0 or 1) for each sentence.

The 24/24 in the output indicates that the model made predictions on all 24 batches of the validation set. The time taken depends on the length of the validation data and the complexity of the model, but in this case it took approximately 1 second to make predictions on the validation set.

**Task 65:**

The code calculates performance metrics of the trained model (model_7) on the validation set (val_sentences). The calculate_results() function is used to calculate the F1-score, precision, recall, and accuracy. The model_7_results dictionary contains these performance metrics.
The accuracy value of 82.02% indicates the ratio of accurately classified sentences from all of the sentences in the validation set.
The precision value of 82.26% indicates the proportion of true positive predictions (correctly classified positive examples) out of all positive predictions made by the model.
The recall value of 82.02% indicates the proportion of true positive predictions out of all positive examples in the validation set.
The F1-score value of 81.87% is the harmonic mean of precision and recall and provides an overall measure of the model's performance.
Overall, these results suggest that the model is performing reasonably well, but there may be some room for improvement.

**Task 66:**

In this chunk we are building Tensorflow Hub pretrained Universal sentence encoder with 10% of data. This code is splitting the training data into a smaller subset of 10% of the original size. Specifically, it calculates the index train_10_percent_split that corresponds to the 10% mark of the train_sentences list, and then slices the list from the beginning to that index (exclusive) to obtain the 10% subset of training sentences. Similarly, it slices the corresponding labels list train_labels to get the labels for the 10% subset. This is often done when you want to quickly test out different models and hyperparameters during the development process, without having to train on the entire dataset every time. Using a smaller subset allows for faster training and experimentation, while still providing some representative examples from the original dataset.

**Task 67:**

The clone_model() function creates a new model that has the same architecture and weights as the input model. In this case, model_8 is being created as a clone of model_7. This means that model_8 will have the same layers, activations, and number of parameters as model_7, and will start with the same initial weights. However, the two models will be independent of each other, so any changes made to model_8 will not affect model_7.

**Task 68:**

This code compiles the cloned version of an existing Keras model (model_7) with binary cross-entropy loss, Adam optimizer, and accuracy as a performance metric. The compiled model can be used for training and evaluation of binary classification tasks.

**Task 69:**

The model summary provides a summary of the layers in the model, their output shape, and the number of parameters in each layer.
**The model consists of three layers:**
The first layer is a KerasLayer with an output shape of (None, 512), which represents the output of the Universal Sentence Encoder (USE) pre trained on a big corpus of text data. This layer has 256,797,824 non-trainable parameters, which are the weights of the pre trained USE model.
The 2nd layer is a Dense layer with 128 units, which has 65,664 trainable parameters.
The third and final layer is also a Dense layer with a single output unit, representing the binary classification task of the model. This layer has 129 trainable parameters.
The total no. of parameters in the model is 256,863,617, with 65,793 trainable parameters and 256,797,824 non-trainable parameters.

**Task 70:**

The code above is training a NN for (model_8) on 10% of the training data for 5 epochs using 2 category classification with an activation funcn. (sigmoid) in its final output. It is assembled with 2 fold cross entropy as the function loss, along with the optimizer Adam, and accuracy is used as the evaluation metric during training and testing. These specified parameters enable the model to be trained and evaluated appropriately for its intended purpose.
The validation set is the remaining 1,000 samples that were set aside earlier.

The results show the training and val. Loss & accuracy for individual epoch. As we can see, the training accuracy raised from 0.7460 in the 1st epoch to 0.8438 in the final epoch. Similarly, the validation accuracy increased from 0.7493 in the 1st epoch to 0.7808 in the final epoch.
The final val. accuracy achieved by the model is 0.7808. This is slightly good than the base model but worse than the model_7 that was trained to the full dataset. However, the model was only trained on 10% of the data, so it's possible that increasing the amount of training data or training for more epochs could improve its performance.

**Task 71:**

In this code, the trained model_8 is used to make predictions on the validation set (val_sentences).
The model_8.predict() method returns the predicted probability values of the positive class for each example in the validation set. The tf.squeeze() method removes any dimensions that have a size of 1. The tf.round() method rounds the predicted probabilities to the nearest integer (0 or 1) since we are using binary classification.The resulting predictions are saved as model_8_preds.
The time taken for the prediction step is around 1.009 seconds.

**Task 72:**

The code model_8_results = calculate_results(val_labels, model_8_preds) calculates the evaluation measures f1-score, precision, recall, and Accuracy for the predictions made by model_8 on the validation.
The results show that model_8 achieved an accuracy of 78.08%, precision of 78.15%, recall of 78.08%, and f1 score of 77.95% on the validation data. Comparing the results of model_7 and model_8, model_7 performed slightly better on the validation set with an accuracy of 82.02%, precision of 82.26%, recall of 82.02%, and f1-score of 81.87%. However, it is worth noting that model_8 was trained on only 10% of the training set while model_7 was trained on the full training set.

**Task 73:**

Next, we'll compare the model by storing them in a data frame named "all_model_results" which stores the evaluation results (F1 , precision, recall, and Accuracy) of various models for the sentiment analysis task. These models include a baseline model, a simple dense model, an LSTM model, a GRU model,

an LSTM-Bidirectional model, a Conv_1D model, and two models using the TF HUB USE encoder, one trained on the entire dataset and one trained on a 10% subset of the data. The evaluation results for each model are stored in a dictionary and then added to the DataFrame using the dictionary keys as column names.

**Task 74:**

The code transposes the DataFrame all_model_results, which was previously created to store the evaluation results of different models. By transposing the DataFrame, the rows and columns are interchanged, so that each row now represents a model and each column represents a metric (i.e.F1 score, precision, recall, and Accuracy). This makes it easier to judge the performance levels of different models across different metrics.

**Task 75:**

This code divides the "accuracy" column of the all_model_results DataFrame by 100 to convert the values from percentages to decimal fractions.
The accuracy column of all_model_results contains the accuracy scores of various models on a binary classification task. By dividing these scores by 100, we can represent them as decimal fractions ranging from 0 to 1, which is the standard format for accuracy scores in machine learning.
Dividing by 100 is necessary because the accuracy scores were originally calculated as percentages using the calculate_results() function. The all_model_results DataFrame contains these scores in percentage format, so this code converts them back to decimal format for consistency and ease of use.

**Task 76:**

The table represents the F1 score, , precision, recall, and accuracy of various models evaluated on the same dataset. The models are:

- Baseline: A simple machine learning model that predicts the majority class, achieving an accuracy of 79.3%.
- Simple_dense_model: A neural network model with one dense layer, achieving an accuracy of 78.1%.
- LSTM: A neural network model with one LSTM layer, achieving an accuracy of 76.4%.

- GRU: A neural network model with one GRU layer, achieving an accuracy of 75.9%.
- LSTM-Bidirectional: A neural network model with one bidirectional LSTM layer, achieving an accuracy of 76.5%.
- Conv_1D: A neural network model with one 1D convolutional layer, achieving an accuracy of 52%.
- TF HUB USE Encoder: A neural network model that uses a pre-trained Universal Sentence Encoder (USE) from TensorFlow Hub, achieving an accuracy of 82%.
- TF HUB USE Encoder_10%: A variant of the TF HUB USE Encoder model trained on only 10% of the data, achieving an accuracy of 78.1%.

From the results, it can be seen that the USE Encoder model performs the best, followed by the baseline and the variant of the USE Encoder model trained on only 10% of the data. The Conv_1D model performs the worst.

**Task 77:**

This code generates a bar plot of the performance metrics (accuracy, precision, recall, f1) for each of the trained models. The plot() function is called on the all_model_results DataFrame, which has the different models as rows and the metrics as columns. The kind parameter specifies the type of plot, which is a bar plot in this case. The figsize parameter sets the size of the plot. Finally, the legend() function is called to add a legend to the plot, and the bbox_to_anchor parameter specifies the position of the legend outside the plot. And as discussed above USE Encoder model performs the best, followed by the baseline and the variant of the USE Encoder model trained on only 10% of the data. The Conv_1D model performs the worst as per the plotted graph.

## Conclusion

In this project, we aimed to classify tweets as either discussing an environmental disaster or not. We used various models, starting from a simple machine learning model to deep learning ones, including feed-forward, CNN, RNN using Long short-term memory, and bi-directional gated recurrent units, and TensorFlow Hub pre-trained feature extractor. We pre-processed the text data by tokenizing and learning word embeddings, and then performed text vectorization using numeric weights assigned to individual tokens.
After evaluating the models on the test dataset, we found that the Universal Sentence Encoder (USE) model performed the best with an accuracy of 82%. This model used a pre-trained encoder from TensorFlow Hub to encode the

text data into fixed-length vectors that were then used to train a classification model. The baseline model that predicts the majority class achieved an accuracy of 79.3%, while the simple dense model achieved an accuracy of 78.1%. The LSTM, GRU, and bidirectional LSTM models achieved accuracies of 76.4%, 75.9%, and 76.5%, respectively. The Conv_1D model performed the worst with an accuracy of 52%.

In conclusion, the USE Encoder model is the most effective at classifying tweets discussing environmental disasters. It outperformed all other models by a significant margin. The results also showed that even a simple baseline model can achieve a reasonable accuracy, and the performance can be improved by using more advanced deep learning models. However, it is important to note that the success of these models depends on the quality and size of the dataset, and the pre-processing steps used to prepare the text data for training. Further experimentation can be done by fine-tuning the pre-trained encoder or exploring other deep learning models that improves the execution of the categorization task.

The notebook serves as an excellent educational aid for individuals who are not well-versed in NLP. It delivers a comprehensive overview of diverse models and their efficacy on a real-world dataset, accompanied by code implementations and detailed explanations of the different stages in the process. All in all, it provides an in-depth NLP tutorial for professionals who lack expertise in the subject matter.

The appropriateness of the dataset for the task seems reasonable given that it contains tweet messages related to environmental disasters. However, the presence of missing data for the keyword feature could potentially impact the performance of some of the models. It would be beneficial to have a larger dataset with more complete features for further analysis.

Alternative approaches or technologies that might have worked better by the use of pre-trained language models like BERT, GPT-2 or T5, which have shown state of the art evaluations on on a wide range of NLP task. Additionally, ensemble of multiple models or using active learning techniques to improve the training data could also improve performance.

Suggestions for next steps in the reader's learning process include:

- "Natural Language Processing with Python" by Steven Bird, Ewan Klein, and Edward Loper - this book provides a comprehensive introduction to NLP with Python, covering topics such as tokenization, POS tagging, and sentiment analysis.
- "Deep Learning for Natural Language Processing" by Palash Goyal, Sumit Pandey, Karan Jain, and Karan Hundal - this book provides an overview

of deep learning techniques for NLP, including LSTM, CNN, and attention-based models.

- "The Illustrated BERT, ELMo, and co." by Jay Alammar - this blog post provides an intuitive explanation of pre-trained language models like BERT & ELMo, along with their applications and limitations.

## References:


1. Bird, S., Klein, E., & Loper, E. (2009). Natural Language Processing with Python. O'Reilly Media, Inc.
2. Goyal, P., Pandey, S., Jain, K., & Hundal, K. (2018). Deep Learning for Natural Language Processing. Packt Publishing Ltd.
3. Alammar, J. (2018). The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning). Retrieved from https://jalammar.github.io/illustrated-bert/

# NLP Project Report

ORIGINALITY REPORT

| 0% | 0% | 0% | 0% |
| --- | --- | --- | --- |
| SIMILARITY INDEX | INTERNET SOURCES | PUBLICATIONS | STUDENT PAPERS |

PRIMARY SOURCES

| | | | |
| --- | --- | --- | --- |
| Exclude quotes | Off | Exclude matches | < 1% |
| Exclude bibliography | Off | | |