Chapter 9: The Prism of Toxic Comment Classification

9.1 Glowing Greeting: An introduction.

Since childhood, we have been told to focus on our goals, ignore the negativity around us, and always root for positivity. Everybody has faced some type of toxicity irrespective of the size of it. Some may have faced it with a bunch of friends, some may have it in offices, etc. The important thing is it leaves a lasting effect on us in terms of mental peace. Some stop taking part in discussions, some may join the negative side, change the job/school, lose confidence, some face anxiety/depression, etc. Now that the web is blasting, to land in the snare of harmful remarks is exceptionally simple and thus more straightforward to confront its repercussions. It has always been difficult to run away from listening to what people are saying about us whether it's online or face-to-face. With this much poisonousness around an individual, zeroing in on positivity is not so basic. The type of tormenting utilizing electronic means is called Cyberbullying.

When a white ray passes through a prism, it gives us seven different tones. In the same way, toxicity is also an umbrella term for all negative comments. This chapter focuses on detecting different types of toxicity when going through a machine-learning model.

One of the challenges by platform owners is to effectively facilitate conversations, leading many communities to limit or completely shut down user comments. In 2018 Kaggle ("Toxic Comment Classification Challenge," n.d.) presented a competition named Toxic Comment Classification Challenge, where the task was to build a multi-headed model that's capable of detecting different types of toxicity like threats, obscenity, insults, and identity-based hate comments for comments provided with a large number of Wikipedia comments which have labeled by human raters for toxic behavior.

In all the sub-section titles the part before 'or' is named for fun and limited to this chapter only and the part after 'or' is the official name. The beginning of every section has names of jupyter notebooks expected for the part. All the required jupyter notebooks can be found on Github. The goal of this chapter is not to present the best result but to get hands-on with different methods to solve this problem statement. You can get the best answer by playing with hyperparameters. Only a few models are used in this chapter (SVM, Random forest, etc), you can try some other models too. This chapter splits data into train, valid and test sets. You can also go for cross-validation. The training and validation loss and accuracy curves can be found in notebooks including the confusion matrix, f1_score, recall, and precision scores. The required libraries for this chapter are:

```
Anaconda (if using mac-os M2 chip using miniconda)
Python 3.9
Tensorflow 2.x
Jupyter
Numpy
Pandas
Nltk
sci-kit learn
Genism
Kaggle
Seaborn
Tensorflow-text
```

The chapter is organized in the following manner. 9.2 takes you around the analysis or EDA of data, 9.3 gives you some general understanding of the solutions, 9.4 revolves around TF-IDF, 9.5 works with word embeddings (Gensim) especially with word2vec, 9.6 focuses on character embedding, 9.7 shows how to work with BERT, 9.8 gives augmentation techniques for NLP, 9.9 builds word2vec from scratch using TensorFlow while 9.10 builds with NumPy, 9.11 presents a way to get sentence embeddings using LSTM encoder-decoder architecture and 9.12 takes you through some concepts in brief (word2vec, BERT).

9.2 Data Lighthouse or analysis

Notebooks: beam me up, data (download data).ipynb and data lighthouse (analysis).ipynb It is very important to understand the data and extract insights as much as we can. There are two main files in the data, train.csv, and test.csv. The training data has in-total 159571 rows and 8 columns.

There are no null values in the data. Multi-label data means that the output for a particular example is not only one label. Say, a movie can be labeled as multiple genres. We have six labels, 1 represents that the comment belongs to a particular column and 0 means not. Also, this data has comments which do not belong to any columns meaning 0 for labels, consider them as non-toxic comments. Now, make a single string of labels for every comment, by simply joining the labels, and naming that column as 'label'.

We have 41 (000000 means - non-toxic) unique labels but we have no idea how many comments per label we have in this dataset, plotting seems necessary Figure 9.1 - number of comments vs. label count. The majority of comments are non-toxic and there is a clear imbalance between toxic and non-toxic comments, if we consider only toxic comments there is also a class imbalance present there as shown in Figure 9.2 - total comments vs. toxic types.

Figure 9.1: number of comments vs. label counts

Figure 9.2: total comments vs. toxic types

The percentage count of each class w.r.t total comments in data and w.r.t toxic comments can be seen using stats() function described in the code, i.e., what percentage of the total comments are non_toxic; toxic; toxic + severe_toxic, etc. For example, w.r.t total comments(159571), non-toxic comments - 89.83% and count is - 143346. Toxic comments - 3.55% and the count is - 5666. We calculated the percentages w.r.t toxic comments and non-toxic comments. For any combination of the toxic comments, the percentages are calculated w.r.t. 16225 (i.e., toxic comments). Now, specific_stats() function calculates the percentages w.r.t particular class, say, 20 comments are under a toxic umbrella, and there is a total of 10 comments that belong to threat, now for every possible combination of class starting with threat the percentage will be calculated w.r.t. 10 (threat + insult; threat + insult + obscene etc.), while in the previous function the same combinations were calculated w.r.t 20.

Table 9.1: Important analysis

It's very clear in this data set that there are more capital letters and exclamations in toxic comments than in non-toxic comments. You can do a deep analysis of the columns in Table 9.1 for different types of toxic and non-toxic comments. There are other non-alphanumeric characters also present in the data, sometimes people draw toxic things using exclamations and other non-alphanumeric characters. So it's very important to remove those from this dataset and change the comments to lowercase. After removing non-alphanumeric characters from the comments, if there some comments turn out to be empty strings, remove them too.

Zipf's regulation states that given a few corpus of NLP, the frequency of a phrase is inversely proportional to its rank withinside the frequency table. In this way, the most successive word will happen around two times as frequently as the second most regular word, multiple times as frequently as the third most continuous word, and so forth. In the Brown corpus of American English text, the word 'the' will be the most often happening word, and without help from anyone else represents almost 7% of all word events. Consistent with Zipf's regulation the runner up expression 'of' represents somewhat more than 3.5% of words, trailed by 'and'. Just 135 jargon things are expected to represent a portion of the Earthy colored Corpus. On the off chance that we don't eliminate these words then the most spotlight will be on these words while grouping rather than the other significant words as these words have extremely low-level data. These words are called stop words. Table 9.2 shows the statistics after removing non-alphanumeric characters, empty strings, stop words, empty spaces, and new lines. Now, even though we have comments under a toxic umbrella, most words used for specific toxic categories can be different. The figure shows the top 50 words for 6 classes. For getting word plots for a particular category we need to count the appearances of the words and select the top 50. Figures 9.3 to 9.8 are the word plots for each class.

Figure 9.3: clean comment only

Figure 9.4: toxic comment only

Figure 9.5: obscene comment only

Figure 9.6: threat comment only

Figure 9.7: insult comment only

Figure 9.8: identity_hate comment only

Table 9.2: Analysis after removing non-alphanumeric characters, empty strings, stop words, empty spaces, and new lines.

Let’s take a look at the length of the comments in the data. Table 9.3 shows the percentage of comments that have more than 10 words, 50 words, etc. For this, the words need to be tokenized first. We have a list 'all_words', containing all the unique words from the comments. Using the above list dictionary 'word_index' is made which has every unique word and the index. Function tokenize() forms the tokenized list for any sentence. Example,

Text = [‘its me hi i am the problem its me’, ‘i am the problem’, ‘hi’]

all_words = [‘its’, ‘me’, ‘hi’, i’, ‘the’, ‘problem’]

word_index = { ‘its’ : 0, ‘me’:1 , ‘hi’:2, ‘i’:3, ‘am’:4, ‘the’:5, ‘problem’:6}

Tokenized text will be: [0, 1, 2, 3, 4, 5, 0, 1]; [3, 4, 5, 6]; [2]

Now when we get the tokenized text, we can see that the max length of the sentences is 1250.

Table 9.3: length of comments

There are no comments only belonging to the sever_toxic class.

9.3 Sparkling techniques: A starry sky of solutions

Now, let’s dive into our classification problem. Here, we will be looking at this problem statement in two ways: binary classification (toxic and non-toxic) and multi-label classification and sometimes we will convert multi-label into multi-class classification.

In every technique the first important thing is to get a good representation for the comments i.e., computers can only understand the language of numbers, so a good number representation for the text is necessary, we will see different types of representations. Multi-label classification problems can be solved in two ways: we have 41 classes and now the classification is a vanilla hardcore classification where every input belongs to just one class only (classic multi-class problem, output nodes will be 41 and loss will be categorical cross entropy). For all 41 classes, the values will be between 0 and 1 (using softmax), and a maximum of 41 values will be selected. Another way to look at this (multi-label), the output nodes will be 6 (number of real classes), and the loss will be binary cross entropy. This means that each node will give the probability of belonging to that class.

The cleaning steps followed in the above sections will be done for any incoming new data point at the time of inference and if the string ends up empty then that output will be calculated as a random guess. If you are following the 41-class approach, generate a random number between 0 to 40, that will be the output. If you are following a 6-class approach, generate a random number 6 times from 0 and 1 and that will be the prediction. For all experiments 'clean.csv' will be used that was saved at the end in our analysis notebook.

9.4 Totaling Trek or TF-IDF

Notebooks: Totaling trek (TF-IDF Binary).ipynb and Totaling Trek (TF-IDF multi label).ipynb. The full form is term frequency-inverse document frequency.

Term frequency (TF) - How important a word is in the text. We have document D and the word W1, then the number of times W1 is repeated in D is called frequency. The count of W1 is divided by the total words in D is TF.

Document frequency (DF) - This is the same as TF, the only difference is now we have N documents and we want to check the word W1 is repeated in how many documents. In other words the number of documents in which W1 is present Number of documents W1 have divided by the total number of documents.

Inverse document frequency (IDF) - It says how relevant the word is. The quantity of records in the corpus isolated by the recurrence of the text is IDF. The corpus is made from all the documents. Total documents divided by DF given IDF. The more common word is considered to be less significant. We always use the log of IDF with base 2.

Term frequency-inverse document frequency (TF-IDF) is the product of TF and IDF.

Let's start with binary classification. When we save data in the clean.csv file, the label columns have some values starting with 0, e.g., '000001'. While saving those 0s are not saved. Hence, to deal with this issue we will be adding 0s where the string length is not equal to 6. Now, we have a label column as we have before saving it to a .csv file. For binary labels replace values in the label with 0 where all non-toxic comments and 1 for all toxic comments. Divide the data into 3 non-equal parts (train, valid and test). For tf-idf we are using TfidFVectorizer from sci-kit learn. We will be fitting it on only train data. After transforming the output shape will be (number of data points, features), and features represent the tf-idf score for different unigrams and bigrams.

```
tf_idf = TfidfVectorizer()
tf_idf.fit(x_train)
X_train = tf_idf.transform(x_train)
```

The experiments are done using random forest, support vector machine, naive Bayes, and logistic regression.

Now for multi-label classification, we will convert our problem statement into multi-class classification (classes = 41). There are a few classes that have only one data point, if we are to split the data then it is mandatory to have more than 1 data point in all the classes when using 'stratify'. Stratify arranges the data points in such a way that all the classes are present in the test and train set as the random splitting of data may lead to shifting all data points to either train or test when we have very few data points for a class. Label encoder has the disadvantage that algorithms can interpret the numeric values as some sort of order or hierarchy in them, and we have 41 classes here so, instead of label encoder, a one-hot encoding strategy is used via LabelBinarizer(). Random forest and SVM (one vs. rest) models are used for multi-class classification.

9.5 Shimmering words or word2vec embeddings.

Notebooks: All 4 notebooks starting with shimmering words. The above method is fully based on the count of the words. Let's look for some advanced techniques. Humans can easily understand the similarity between words. If we come across a new word then we approximately understand the meaning of that word by neighboring words, i.e., by its context. Such embeddings are more powerful as it extracts the meaning of the words. One such example is Word2Vec. It detects similarity mathematically. It places similar words together in the vector space.

We will be using the Genism library for word embeddings. It has a lot of pre-trained models and other facilities which you can find on its official page. It has fasttext, word2vec (Mikolov et al. 2013), glove embedding pre-trained on different datasets. For our experiments, we will be using word2vec trained on google news with dimension 300. One disadvantage of word2vec is, it can not handle out-of-vocabulary words.

```python
import gensim
vectors = gensim.downloader.load('word2vec-google-news-300')
vectors['computers']
```

The above code will give the 300-dimensional embedding for word computers. Once we extract the embedding for all the words in the comment, the important thing is to get the embedding for the whole sentence, as at the end we want to predict whether the whole sentence is toxic or not. This can be dealt with in two ways: Mean magic and Word watchtowers.

Mean magic simply means taking the average of all the embeddings for a comment. Word watchtowers say instead of taking mean pass all embeddings for a comment to a LSTM/RNN model and the neural network will itself take care of it.

When using LSTM/RNN for sentence vectors, it's not necessary to deal with multi-label classification problems as muli-class. Instead, neural networks support multi-label simply by specifying the number of nodes in the output layer as the number of classes. Sigmoid activation for each node in the output layer and binary cross entropy loss function.

```
def rnn():
    …
    outputs = tf.keras.layers.Dense(6, activation =
'sigmoid')(x)
    …
model_rnn = rnn()
model_rnn.compile(..., loss='binary_crossentropy')
```

Two types of attention layers are implemented, simple and weighted in the notebook mentioned above.

Attention: the encoder generates annotations from the input sentence, which is fed to an alignment model and the previous hidden decoder state. Attention scores are generated by alignment model using this information. To normalise (values between 0 and 1) the values produced attention scores into weights values apply softmax function. These weights with prior computed annotations, generate context vector using a weighted sum of the annotations. Fed context vector and previous hidden decoder state and previous outcome to the decoder to have the final answer. Repeat until the end of sentence.

In multi-label classification (MLC) classes are not mutually exclusive as in multi-class classification. So, evaluating the classifier is a little different but very easy. Say, we have 3 classes: Apple, Banana, and Mango. We have 3 data points for which real and predicted values are:[[1 0 1], [0 0 1], [1 1 1]] and [[1 1 0], [0 0 1], [0 1 1]] respectively. Compute confusion matrix, recall, precision, and f1 score in all three classes separately. The first value in every sub-list represents class A, the next represents class B, and so on. In the end, we have 3 precision, 3 recall, and 3 F1-score. To check the overall performance of the classifier: aggregate the metrics like macro, micro, and weighted average. Simply taking the average precision A, precision B, and precision C is the final precision macro average. Micro refers to the summing of true positives of A, B, and C and the summing of false positives of A, B, and C and putting them in the precision formula.

```
preds_biattn = np.where(preds_biattn > 0.7, 1, 0)

p_avg, r_avg, f1_avg = 0,0,0
for i in range(6):
    real = Y_val[0:, i]
    reel = preds_biattn[0:, i]
    conf_mat = confusion_matrix(real, reel)

    print(conf_mat)
    p = precision_score(real, reel)
    r = recall_score(real, reel)
    f1 = f1_score(real, reel)

    p_avg += p
    r_avg += r
    f1_avg += f1

print(p_avg / 6, r_avg /6, f1_avg / 6)
```

9.6 The Letter Lab or Character embedding

Notebook: The letter lab (character level binary classification).ipynb Word vectors can not handle out-of-vocabulary words. (Zhang, Zhao, and LeCun 2016) If we have embedding for every letter then all misspellings, and OOV words can be handled easily. English has 26 alphabets or 52 including upper and lower case or even more if special characters are included, but the vocab still remains very small if letter embeddings are taken into account.

```
tk = tf.keras.preprocessing.text.Tokenizer(num_words=None,
char_level=True, oov_token='UNK')
tk.fit_on_texts(x_train)
```

Keras tokenizer builds the vocab of characters and tokenizes the text.

```
x_train =
tf.keras.preprocessing.sequence.pad_sequences(x_train,
maxlen=5000, padding='post')
```

Keras pad_sequences makes all the comments of equal length by padding 0.0 (by default, you can change the value).

```
embedding_weights = []
for char, i in tk.word_index.items():
    onehot = np.zeros(vocab_size)
    onehot[i-1] = 1
    embedding_weights.append(onehot)
embedding_weights = np.array(embedding_weights)

embedding_layer = tf.keras.layers.Embedding(
vocab_size, embedding_size, input_length = input_size,
weights = [embedding_weights]
)
```

Inputs will be passed to the embedding layer and then follows the Conv1D model or any other desired model. There are two ways to form inputs for embedding layers. 'tk' gives us the vocab size. We can form embedding weights using one hot coder. For all the words in the vocab form the one-hot coding and pass that matrix to weights.Another way is to simply define the embedding dimension along with vocab_size and the input_size, layer will produce (vocab_size, embedding_size) matrix.

```
embed_size = 100
input_size = 5000
vocab_size = len(tk.word_index)
embedding_layer = tf.keras.layers.Embedding(vocab_size,
output_dim = embed_size, input_length = input_size)

inputs = tf.keras.Input(shape=(input_size,), name='input',
dtype='int64')
x = embedding_layer(inputs)

x = tf.keras.layers.Conv1D(32, 7, activation = 'relu')(x)
```

9.7 Glittering attention or BERT model

Notebooks: Glittering attention (binary classification using Bert).ipynb and Glittering attention (multi-label classification using Bert).ipynb One thing to notice is that for specific tasks we need to get a large amount of data and train word embeddings for it. In computer vision, we have pre-trained vectors on the ImageNet dataset. NLP does not have such a thing. So researchers came up with general-purpose language representation with the data crawled from the web and trained them using the BERT model. Now these representations can be fine-tuned even for a smaller task-specific dataset.

Here we will be using a model that is trained on uncased English data. The output of this model will be 128 dimensions. There are a lot of other models much bigger in size than this and also other variations like XL-Net, Electra-net, etc.

```
import tensorflow_hub as hub
import tensorflow_text as text
bert_model_name = 'small_bert/bert_en_uncased_L-2_H-128_A-2'
tfhub_handle_encoder = map_name_to_handle[bert_model_name]
tfhub_handle_preprocess =
```

One is to perform preprocessing (tfhub_handle_preprocess) and the other one for encoding (tfhub_handle_encoder).

```
def build_classifier_model():
    text_input = tf.keras.layers.Input(shape=(),
dtype=tf.string, name='text')
    preprocessing_layer =
hub.KerasLayer(tfhub_handle_preprocess, name='preprocessing')
    encoder_inputs = preprocessing_layer(text_input)
    encoder = hub.KerasLayer(tfhub_handle_encoder,
trainable=False, name='BERT_encoder')
    outputs = encoder(encoder_inputs)
    net = outputs['pooled_output']
    net = tf.keras.layers.Dropout(0.1)(net)
    net = tf.keras.layers.Dense(1, activation='sigmoid',
name='classifier')(net)
    return tf.keras.Model(text_input, net)
```

We are not training the BERT model hence the trainability is kept False. We are joining a dropout and dense layer to the output of the BERT model. Then train the model as we did in previous examples. Here the whole dataset is taken while fine-tuning, you can try with a lesser amount of data also.

9.8 Jazzy Jitters - shake up your data with augmentation techniques

Notebooks: Starting with Jazzy jitters. Real-life data is not always balanced. Data augmentation always comes in handy. Sometimes data augmentation reduces accuracy, sometimes it helps in improving accuracy. So, data augmentation should not be used blindly. Here are some of the ways to handle imbalanced data or to augment data in NLP.

<u>Oversample minority class and undersample majority class</u>: remove data points from the majority class and duplicate some minority class datapoints. By doing this sometimes we lose valuable data points. Here for binary classification, we have 16225 data points for the toxic class and 143288 data points for the non-toxic class. We are deleting 100000 (random.choices

selects k points at random without repetition) datapoints randomly from a non-toxic class. Now we have 71251 data points for the non-toxic class. In the same way, you can duplicate a few data points from the toxic class.

```
index = range(0, non_toxic.shape[0])
index_list = random.choices(index, k = 100000)
non_toxic.drop(non_toxic.index[index_list], inplace=True)
```

Delete the words: From the comment remove a few words. Excessive removal of words leads to no meaning at all of the sentences and hence reduced accuracy. Select random as above and use random.choice() to select only 1 word to be deleted on them. You can remove multiple words.

```
def delete(text):
    text.remove(random.choice(text))
    return ' '.join(text)

temp_delete['comment_text'] =
temp_delete.comment_text.str.split(' ').apply(delete)
```

Shuffle random words: Remember jumbling sentences in our childhood, the same logic goes here also, jumbling the sentences should not change the gist of the sentences, and the model can learn where to look or what words to understand for the completion of a specific task.

```
def jumble(text):
    random.shuffle(text)
    return ' '.join(text)
```

Reverse the sentence: Reversing the datapoint gives a new sample. The idea is that order should not change the class of the sentence. index_list is the randomly selected k number of indices. Those data points should be reversed and added to the train set. Note we are not deleting the correct order comments from the train set. Split the sentence, reverse the list, form the string again and we have new data points.

```
temp_reverse['comment_text'] =
temp_reverse.comment_text.str.split(' ').apply(lambda x: '
```

Add synonyms: One advanced technique is to replace a few words with their synonyms. In pre-trained vector space, the words which are of similar meaning are kept together. We can find similar words from that space and replace the words in the comments (new points).

9.9 Shine with word2vec

Notebook: Shine with word2vec.ipynb We saw how to use pre-trained word vectors. Now let's see how to train on our corpus using the Gensim library.

```
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
```

Import word2vec model from Gensim. We have two sentences in our corpus. Now split them to word level but keep sentence separation intact.

Now initialize the word2vec model with embedding size and min_count. Min_count ignores all words with a total frequency lower than specified. The model will build vocab from the words and then train the model for the desired number of epochs.

```
model = Word2Vec(min_count=1, vector_size = 10)
model.build_vocab(word_sents)
model.train(word_sents, total_examples = len(sentences),
epochs=3)
```

The trained word vectors are stored in a KeyedVectors instance, as model.wv. You can load anytime and can extract embedding for words.

```
word_vectors = model.wv
word_vectors.save("word2vec.wordvectors")
wv = KeyedVectors.load("word2vec.wordvectors", mmap='r')
vector = wv['chapter']
```

Building a model from scratch gives a better understanding. So, let's build word2vec from scratch using tensorflow. In skip-gram architecture the center word is the input to the model and context words are predicted. 'this section focuses on skip-gram architecture', the input will be of the form [(target, context), label] with window size 1: [(section, [this, focuses]), 1], etc. Label will be 1 if target and context words are valid, else 0. For negative samples, random context words from the corpus are selected for target words.

First, we will convert the dataframe to a Tensorflow dataset. TextVectorization layer of TensorFlow maps text features to integer sequences. We get vocab and the total number of sentences using tensorflow dataset and vectorization, including the vocabulary and inverse vocabulary which is very important. Next, we want to form target and context words using skipgrams. We only form positive pairs. For negative samples, we use log_uniform_candidate_sampler (samples words at random from vocabulary). In the end, we will concat positive and negative words to form context.

Word2vec model: initialize the word and target embedding matrix in the init(). The dot product takes place in call(). Target embedding: looks up the embedding of a word when it appears as a target word, Context embedding: looks up the embedding of a word when it appears as a context word.

```
class Word2Vec(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim):
        super(Word2Vec, self).__init__()
        self.target_embedding =
tf.keras.layers.Embedding(vocab_size,  embedding_dim,
input_length = 1,  name = 'w2v_embedding')
 self.context_embedding =
tf.keras.layers.Embedding(vocab_size, embedding_dim,
input_length = num_ns + 1)
    def call(self, pair):
        target, context = pair
        if len(target.shape) == 2:
            target = tf.squeeze(target, axis = 1)
        word_emb = self.target_embedding(target)
        context_emb = self.context_embedding(context)
        dots = tf.einsum('be,bce->bc', word_emb, context_emb)
        return dots
```

9.10 Ground zero

It's always fun to understand how a model works from scratch. The backpropagation is handled by the TensorFlow libraries. Now let's build word2vec using numpy. V is for vocabulary, N is for embedding dimensions, x represents input, W and W' are weight matrices and y is the predicted output.

The dimensions are as follows: $x = (V, 1), W = (V, N), h = (N, 1), W' = (N, V), y = (V, 1)$.

The feedforward propagation is as follows:

$$h = W^T . x \tag{9.1}$$

$$u = W'^T . h \tag{9.2}$$

$$u_j = V_{w_{ij}}{}^T . h \tag{9.3}$$

$$P(w_j | w_i) = y_j = \frac{e^{u_j}}{\Sigma_{j'=1}{}^v e^{u_{j'}}} \tag{9.4}$$

Multiply one hot encoding of the center word (x) with the first weight matrix W to get the hidden layer matrix h (9.1). To get the new matrix u multiply the hidden layer vector with weight matrix W' (9.2). We have to apply the softmax to the output of (9.2) to have output layer y. In (9.3) let j be the neuron of layer u and in vocab, it is the jth indexed word. V_{wj} with jth column matrix W', i.e., the column corresponding to a word. (9.4) applies softmax to (9.3) and we will get the probability that the word is a context word. So, now our goal is to maximize $P(w_{j*} | w_i)$, j* represents the indices of context words. In terms of equation maximize (9.5) and j* is the vocab indices of context words ranging from c = 1,2,3…C

$$\mathbf{\Pi}_{c=1}{}^{C} \frac{e^{u_{jc*}}}{\Sigma^{v}{}_{j'=1} e^{u_{j'}}} \tag{9.5}$$

(9.6) takes the negative log-likelihood of this function to get a loss function, which we want to minimize. Now the actual output vector from training data for a particular center word will have 1's at the positions of context words and 0's at remaining places, let's name it t and t_{j*c} are the 1's of the context words, (9.7) multiples them and solving that will give us (9.8) our loss function.

$$E \ = \ -log\{\mathbf{\Pi}_{c=1}{}^{C} \frac{e^{u_{jc*}}}{\Sigma^{v}{}_{j'=1} e^{u_{j'}}}\} \tag{9.6}$$

$$E \ = \ -log(\mathbf{\Pi}^{C}{}_{c=1}\, e^{u_{jc*}}) \ + \ log(\Sigma^{v}{}_{j'=1}\, e^{u_{j'}})^{C}$$

(9.7)

$$E \ = \ -\Sigma^{C}{}_{c=1}\, e^{u_{jc*}} \ + \ C\,.log(\Sigma^{v}{}_{j'=1}\, e^{u_{j'}}) \tag{9.8}$$

```
def feed_forward(data, W, W1, embedding_dim):
    h = np.dot(W.T, data).reshape(embedding_dim,1)
    u = np.dot(W1.T,h)
    y = softmax(u)
    return y, u, h
```

Now let's take a look at backpropagation:

In the matrices W and W' the parameters need to be adjusted, so find the partial derivatives of our loss function w.r.t to W and W' to apply the gradient descent algorithm. Find $\frac{\partial E}{\partial W'}$ and $\frac{\partial E}{\partial W}$.

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial w_{ij}} \tag{9.9}$$

$$\frac{\partial E}{\partial u_j} = \; -\Sigma^{C}{}_{c\,=\,1}\, e^{u_{jc*}} \;+\; C\,.\; \frac{1}{\sum^{v}{}_{j'=1} e^{u_{j'}}} \cdot \frac{\partial}{\partial u_j} \sum{}^{V}{}_{j\,=\,1} e^{u_j} \tag{9.10}$$

$$\frac{\partial E}{\partial u_j} = \; -\Sigma^{C}{}_{c\,=\,1} 1 \;+\; \sum{}^{V}{}_{j\,=\,1} y_j \tag{9.11}$$

$$\frac{\partial E}{\partial u_j} = y_j \;-\; t_j \;=\; e_j \tag{9.12}$$

$$\frac{\partial E}{\partial w'_{ij}} = e_j \,.\, \frac{\partial u_j}{\partial w_{ij}} \;=\; e_j \,.\, \frac{\partial w'_{ij} * h_i}{\partial w'_{ij}} \tag{9.13}$$

$$\frac{\partial E}{\partial w_{ij}} \;=\; e_j \,.\, h_i \tag{9.14}$$

$$\frac{\partial E}{\partial w_{ij}} \;=\; \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial w_{ij}} \tag{9.15}$$

$$\frac{\partial E}{\partial w_{ij}} \;=\; \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial h_i} \cdot \frac{\partial h_i}{\partial w_{ij}} \tag{9.16}$$

$$\frac{\partial E}{\partial w_{ij}} = \; e_j \;.\; w'_{ij} \,.\, \frac{\partial w_{ij*} x_i}{\partial w_{ij}} \tag{9.17}$$

$$\frac{\partial E}{\partial w_{ij}} \;=\; e_j \,.\, w'_{ij} \,.\, x_i \tag{9.18}$$

```
def backpropagation(xdata, ydata, y, vocab_len, alpha, h, W,
W1):
    e = y - np.asarray(ydata).reshape(vocab_len, 1)
    dLdW1 = np.dot(h, e.T)
    x = np.array(xdata).reshape(vocab_len, 1)
    dLdW = np.dot(x, np.dot(W1, e).T)
```

9.11 Twinkling word weaving

Notebook: twinkling word weaving (lstm autoencoder).ipynb We saw how to build word2vec for word embeddings from scratch using tensorflow and numpy. Also for sentence embedding, we saw averaging the word vectors and passing them through LSTM layers. For sentence vectors there are other methods, like Doc2vec, an extension of Word2Vec, SentenceBERT, but these are not the only way to get word vectors. LSTM encoder-decoder models can be used to directly get the sentence embeddings. We will be using one-hot encoding just as done for word2vec and character embedding.

Build the vocabulary using the cleaned data and tokenize the data. Pad sentences to make them of equal length. Change every token to one hot encoding to get (total sentences, steps, features) or (sentences, words, and vocab size). Pass them through the LSTM autoencoder model. For any new clean and tokenized data point, change it to (sentence, words, vocab size) form and predict sentence representation using the encoder model.

9.12 Brightened clarity

Understanding keras embedding layer: A table contains all possible vectors and every input integer is used as the index to access that table. Therefore, to initialize the table, vocabulary size as the argument is needed. Train corpus: 'This chapter focuses on comment classification' and 'This chapter has multi-label classification. Now, let's encode these sentences using a unique integer. [0, 1, 2, 3, 4, 5] [0, 1, 6, 7, 8, 5]. Now train the model with the first layer as a Keras embedding layer. The input dimension is 9 (maximum words in vocabulary), the output dimension is the embedding size and input_length is the size of the sentence. After the network is trained, the weights of the embedding layer will be (input_dim, output_dim) i.e., (vocab_size, embedding_dim). Here it is (9,2) 9 vocab words, each having an embedding of size 2. Imagine a trained matrix of (9,2) size which maps integers to embedding vectors. To convert [0, 1, 2, 3, 4, 5] go to the 0 index in the matrix and get the embedding. Same for all other integers.

BERT: It is a language model trained bidirectionally using transformers (Devlin et al. 2018). It is a language model trained bidirectionally. A language model would have looked at the sentence sequences, generated the next word, appended that word to the sequence, and predicted the next word until the end of a sentence. BERT is a self-supervised model trained with a technique called masked language model (MLM) and next sentence prediction (NSP). This masks the words randomly and then tries to predict them. To predict the word, the model looks in both directions and it uses the full context. BERT takes two sentences and it determines if the next sentence follows the first like binary classification. Previous and next tokens are considered at the same time, unlike previous methods.

Word2vec comes in the category of context-free models. They generate word embedding for each in the vocab not based on the surrounding words. 'bank of river' and 'bank account', both have the word bank but the meaning is different, and hence the embedding should be different. Context-based embeddings give representations based on surrounding words. Transformers are faster than previous models as words can be processed simultaneously. Let's say the task is to translate English sentences into German. The 2 important blocks of transformers are encoder and decoder. The encoder takes the English words simultaneously and generates embeddings for those words. The decoder takes the embeddings and the previously translated German words and uses them to generate the German words. The translation is generated one word at a time until the end of the sentence. The encoder learns what is English, grammar and what is context. Decoder learns how English words relate to German words. Both have some understanding of language. On stacking decoders, we get GPT architecture. On stacking encoders we get BERT. BERT can be used for any number of language-related tasks (sentiment analysis, text summarization, etc.). Pre-training in BERT is basically for understanding the language and context. Fine-tuning is related to the model knowing the language and using it for solving a particular task.

Fine-tuning: Replace the fully connected output layer of the network with a fresh set of output layers. Then perform supervised learning on the data for any NLP problem. This will be pretty fast.

BERT is trained on MLM and NSP simultaneously. Input is a set of two sentences with some words being masked. Each token is a word, convert each word into embeddings using pre-trained embedding. On the output side, one is binary output for NSP, 1 if sentence B follows sentence A else 0. We also get the word vectors for masked language word problems. The number of word vectors in the input is the same as the number of word vectors in the output.

Fine-tuning: For the question-answering task, train the model by modifying the input and output layers. Pass the question followed by a passage having answers as inputs, and the output layer would have start and end words that encapsulate the answer assuming the answer is in the same span of text.

BERT - generate embedding from word tokens: The initial embedding is generated by three vectors. Token embedding (pre-trained embeddings). The paper uses WordPiece embeddings that have a vocabulary of 30,000 tokens. The segment embeddings are the sentence number that is encoded into a vector. Position embedding is the position of every word within that sentence encoded into a vector. Adding all these vectors forms the input embedding for

BERT. Sentence and word embeddings are for preserving the temporal order, as they are processed simultaneously hence preserving order is necessary. All of the word vectors have the same size and all are generated at the same time. Take each word vector and pass it to a fully connected layer output with neurons as equal to tokens in vocabulary (30,000 here). Apply softmax activation hence converting a word vector to a distribution. The actual label of this distribution is a one-hot encoded vector of the actual word. So compare the two distributions and train the network using cross-entropy loss. Output has all the words even the non-masked words. Loss only considers the prediction of the masked ones and ignores all others, because more focus is given to predicting the masked values so it gets them correct and increases context awareness.

Positional embeddings: PE is sort of an identifier, telling the whereabouts of words to the transformer. We add PE to the word, telling the word x should move along a dimension towards a specific part of space to cluster with all the other first tokens in any other sequence. We move the initial representations in the direction of the other first tokens or second tokens, etc. PE should have the same identifier irrespectively of the sequence length or what the input is. The shift should not be too large otherwise they will be pushed into a different subspace where positional similarity or dissimilarity overtones the semantic similarity.

Let's say we have 4-dimensional word vectors. Now the PE should also be 4-dimensional. For the words coming in the sequence, number it from 1 to the total number of tokens, giving PE as [1, 0, 0, 0], [2, 0, 0, 0], etc. This means shifts are not small or not bounded. The sine or cosine function is bounded in terms of the different values it can take. It periodically returns values between 1 and -1 and they are defined till infinity. Sigmoid is also bounded but sines and cosines have variability even for big numbers. The sine function gives PEs as [0, 0, 0, 0.01], [0, 0, 0, 0.02], etc. This periodic function repeats the same output for different positions, so lower the sine frequency, such that the numbers will not repeat even for the biggest sequence length. This produces the same problem as we faced while numbering the tokens from 1, with a difference that every number is between 1 and -1. This will also push the words with great distance. The push should not be too large or too small. Let's use all the dimensions of PE. A cosine function for the next dimension but this also does not make two embeddings differ very much [0, 0, 0.99, 0.01], [0, 0, 0.98, 0.02]. So, if we increase the frequency of the cosine, the difference between the values will be more as the slope has increased. Now all the tokens are uniquely differentiating from one another including consecutive tokens. Doing this for every dimension, i.e, alternate between sines and cosines with increasing frequency, we give enough information to ensure that the transformer cannot miss this order of sequence [0.2, 0.7, 0.99, 0.01], [0.01, -0.8, 0.70, 0.02]. So, with an increase in the frequency along the dimension, we are narrowing the location of the words. To balance out positional and semantic information we have the division by 10,000 in the arguments of sines and cosines. There are other ways for getting PE such as relative, learned positional embeddings.

WordPiece embedding: For the OOV words, the older technique is to define an unknown token and define embedding for it, that's not much help. Also, slang and abbreviations are hard to handle with word-based tokenization. It fails where language does not use spaces for word separation like in Chinese, Korean, and Japanese languages (Schuster and Nakajima 2012). Character tokenization is resource hungry as more tokens mean more inputs. With no inherent meaning, learning with characters is like learning with no meaningful semantics. BERT uses a WordPiece model and it breaks the word embeddings into multiple subwords. BERT has no token for 'embedding' words but for 'em', 'bed', 'ding'. Fasttext is an improvement on the word2vec model that increases subword information. Fasttext averages over the subword embeddings to form the one-word embedding. BERT does not combine them. Breaking the word down based on a set of prefixes and suffixes is like 'unhappily': un##, happ, ##ily, double hash denotes prefixes and suffixes. The model puts these learned subwords together and creates other words. Hence memory requirement and effort in creating large vocab is solved. Some of the subwords created as defined rules may never appear in the text and for every language define new rules for subword creation.

Originally BPE is used for data compression i.e., to find the best way to identify the common byte pairs. In NLP we use it to find the best representation of text using the smallest number of tokens. BPE: add </w> at the end of each word so that the end of a word is identified and then calculate the word frequency in the text. Then split the characters and calculate the character frequency. For a decided number of epochs, count the frequency of the consecutive byte pairs and contact the most frequent byte pairing. Do this until there are no more pairs or till the end of the loop. In this, the original character token frequency will reduce and the new pair token frequency will ace in the token dictionary. The number of tokens created will increase first as new pairings will be created, but they will decrease after some epochs. Final tokens will vary as the total epochs vary. Hence for a single text, we will have different tokens and different embeddings. BPE is greedy as it runs for the highest combined frequency at every step.

There are methods for improving BPE algorithms. BERT uses a new algorithm named WordPiece, similar to BPE but with an additional layer of likelihood calculator to decide whether the merged token will make the final cut. WordPiece, BPE, Unigram, SentencePiece are some popular subword-based tokenization algorithms. Bottom-up and top-down are the two implementations of the WordPiece. Bottom-up is based on BPE. BERT uses top-down implementation.

Bottom-up approach: A step ahead of frequency is the probabilistic approach. WordPiece is doing the same, with one difference, the way pairs are added to the vocab. The algorithm will check if the probability of occurrence of 'lt' is more than the probability of occurrence of 'l' followed by 't'. If the probability of 'lt' divided by 'l', 't' is greater than any symbol pair then the concatenation will happen. So WordPiece is evaluating what it will lose by merging the two tokens to ensure that the step it is taking is worth it or not. WordPiece is greedy as it picks the best pair at every epoch to concat and builds new tokens (bottom-up - characters to pairs etc). So a model is trained on the vocab, picks the pair having the highest likelihood, adds that pair to the vocab, trains the model on the new vocab, and repeats the steps until the decided vocab size or set likelihood is reached.

Word2vec: One-hot encoding cannot understand the relationships between words, even incompatible with a large corpus. Word embeddings should help in establishing the association of a word with another similar meaning word through the created vectors. Word2vec is a self-supervised model which on giving corpus without any label information produces word embeddings. It internally works as a supervised classification model. *Continuous Bag of Words (CBOW)* architecture comprises a neural network classification model in which inputs are context words and output is a target single word. 'This section focuses on word2vec', inputs with window size 1 from both sides are [context, target] form: [[this, focuses], section], [[section, on], focuses], etc. The embedding layer is initialized using random weights, context words passed through this embedding layer, then embeddings are passed through a layer that averages out them. Finally, embeddings are passed through a softmax layer that predicts the target word. Compare the real target word with the predicted word and calculate the loss. Perform backpropagation with every epoch and update the embedding layer.

9.13 Flashing references

1. "Toxic Comment Classification Challenge." n.d. Kaggle.com. https://www.kaggle.com/competitions/jigsaw-toxic-comment-classification-challenge/data.
2. Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." ArXiv.org. October 11, 2018. https://arxiv.org/abs/1810.04805.
3. "The Evolution of Tokenization – Byte Pair Encoding in NLP." 2021. FreeCodeCamp.org. October 5, 2021. https://www.freecodecamp.org/news/evolution-of-tokenization/.
4. Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Efficient Estimation of Word Representations in Vector Space." ArXiv.org. September 7, 2013. https://arxiv.org/abs/1301.3781.
5. Zhang, Xiang, Junbo Zhao, and Yann LeCun. 2016. "Character-Level Convolutional Networks for Text Classification." *ArXiv:1509.01626 [Cs]*, April. https://arxiv.org/abs/1509.01626.
6. Zhang, Xiang, and Yann LeCun. 2016. "Text Understanding from Scratch." *ArXiv:1502.01710 [Cs]*, April. https://arxiv.org/abs/1502.01710.
7. Khanna, Chetna. 2021. "WordPiece: Subword-Based Tokenization Algorithm." Medium. August 18, 2021.
8. "A Fast WordPiece Tokenization System." n.d. Ai.googleblog.com. Accessed April 28, 2023. https://ai.googleblog.com/2021/12/a-fast-wordpiece-tokenization-system.html.
9. Schuster, Mike, and Kaisuke Nakajima. 2012. "Japanese and Korean Voice Search." IEEE Xplore. March 1, 2012. https://doi.org/10.1109/ICASSP.2012.6289079.
10. "Implement Your Own Word2vec(Skip-Gram) Model in Python." 2019. GeeksforGeeks. January 18, 2019. https://www.geeksforgeeks.org/implement-your-own-word2vecskip-gram-model-in-python/.
11. "Word2Vec | TensorFlow Core." n.d. TensorFlow. https://www.tensorflow.org/

tutorials/text/word2vec.

12. Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." ArXiv.org. 2017. https://arxiv.org/abs/1706.03762.