BERT, the Machine Learning Muppet, and the Kaggle Toxic Comment Challenge

Faramarz Munshi
ID# 82245616
fmunshi@uci.edu

Janus Varmarken
ID# 67495924
jvarmark@uci.edu

Kevin Choi
ID# 90084056
k.choi@uci.edu

Group Name: Faramarz plz join

December 2018

1 Introduction

Bidirectional Encoder Representations from Transformers (BERT) has generated great enthusiasm from the machine learning community since its introduction a little over a month ago. For the toxic comment challenge, we decided to train a task-specific classification layer on top of BERT's pre-trained model to test BERT's state-of-the-art bi-directional Transformer encoder. We believe the toxic comment challenge offers fertile conditions for analyzing both BERT's strengths and weaknesses. Given BERT's novelty, the toxic comment classification layer will be one of the first applications outside the original paper. From the data analysis, we compute a lower bound accuracy for the toxic comment challenge of 90%. For an upper bound, we will use the Kaggle high score of 98.90%. We will evaluate BERT based on these bounds as well as make comments regarding potential future work with regards to BERT optimizations.

1.1 Data Analysis and Visualization

1.1.1 Basic Structure

To get a basic sense of the what the dataset looks like, we first load the entire training set (`train.csv`) into a Pandas dataframe, `full_tr`, and output its shape using the code in App. A.1. The resulting shape is a $(159571, 8)$ tuple. The first element of the tuple is the number of data points in the dataset, and the second element is the number of features of each data point. As such, we are working with a dataset where each data point has eight features. We next output an example entry in `full_tr` to visualize what a single data point looks like (see Fig. 1). As evident from Fig. 1, column one contains a unique identifier, column two contains the entire text of the comment, and columns three through eight are binary flags that denote if a particular kind (as per the corresponding column's label) of toxicity is present in the comment.

```
id                                      c70efd55724be549
comment_text     === I AM GLEN AND I LOVE BEING A FAG===
toxic                                                 1
severe_toxic                                          0
obscene                                               0
threat                                                0
insult                                                0
identity_hate                                         0
```

Figure 1: An example data point. Note that the output format "transposes" the data point's row vector (making the row vector appear as a column vector) and includes column names.

1.1.2 Toxicity Distribution: Lower Bound on Accuracy of Classifier

We next get a basic sense of the distribution of toxic vs. non-toxic comments in the dataset by counting the number of data points in `full_tr` (see Sect. 1.1.1) that are labeled as containing *any* or *none* (for toxic and non-toxic, respectively) of the six possible toxic behaviors, and dividing each of these two counts by the total number of data points (see code in App. A.2). The output is shown in Fig. 2. Evidently, approximately 90% of all data points (comments) in the provided training data are non-toxic. This imposes a lower bound on the accuracy of our classifier when evaluated against the training set: a classifier that *always* guesses non-toxic will achieve 0.9 accuracy, so our classifier must do better than this lower bound.

```
Non-toxic comments count: 143346
Toxic comments count: 16225
Non-toxic fraction (of total): 0.8983211235124177
Toxic fraction (of total): 0.10167887648758234
```

Figure 2: Output from code in App. A.2 showing the distribution of toxic and non-toxic comments.

1.1.3 Correlation Between Toxicity Labels

In order to get an idea of any potential correlation between the different kinds of toxicity, we generated a matrix plot (see Fig. 3 and code in App. A.3) that illustrates the number of times toxicity label l_1 occurs

alongside toxicity label l_2 in the full training set[1]. Of all the label pairs, no pair indicated significant correlation. For instance, the order of labels with the highest probability of pairing with toxic is obscene, insult, severe_toxic, identity_hate, threat. This also happens to be the order of labels from most frequent to least frequent.

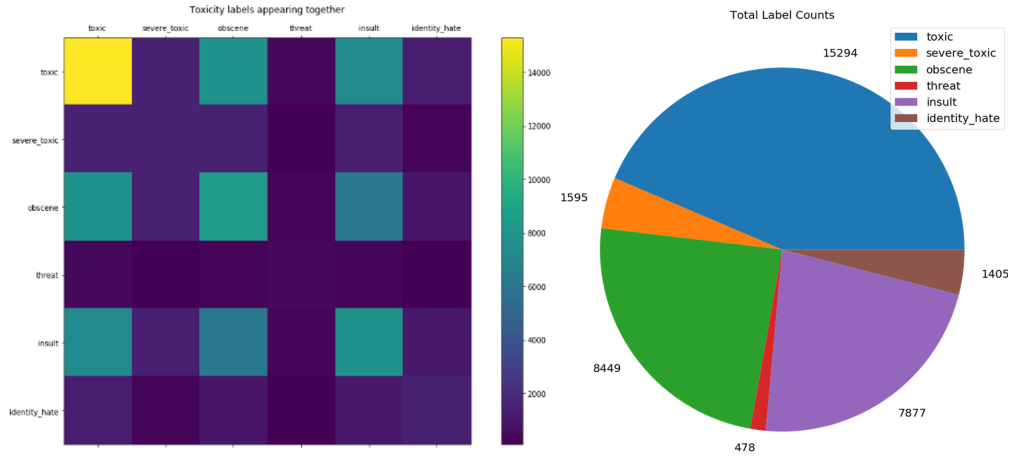

Figure 3: Output from code in App. A.3, showing the pairwise correlation between different kinds of toxicity (left) and the corresponding total label counts for reference (right).

1.1.4 Correlation Between Source Text and Toxicity Labels

Besides understanding the relationships between the toxicity labels, we sought visual relationships between the source texts and their respective labels. We first analyzed the relationship between the presence of *individual* words and the toxicity labels. To compare the vocabulary used in non-toxic comments to the vocabulary used in toxic comments, we generated the word clouds shown in Fig. 4. The word clouds indicate that very few words from the two cases overlap. In fact, several of the words in the toxic word cloud should automatically trigger toxicity labels: while words like "jew", "kill", and "die" may be helpful in the correct context, words such as "fuck", "shit", "cunt", "faggot", and "bitch" are inflammatory in almost any context. Consequently, we adjusted our baseline classifier from Sect. 1.1.2 (predict all zero labels) with the simple rule that any source text containing the above inflammatory language be automatically labelled. This rule increases the lower bound of our expected accuracy to 91.29%.

We hypothesized that various other metrics could be devised to segregate toxic and non-toxic comments. These attributes included word count, sentence count, uppercase-to-lowercase ratio, and word-to-sentence ratio. The only attribute to illustrate some interesting correlation to toxicity was the uppercase-to-lowercase ratio. We had hypothesized that poor grammar may be indicative of toxicity, but the word-to-sentence ratios showed that both non-toxic and toxic comments were prone to run-on sentences. Ultimately, we were only able to demonstrate the positive correlation between toxicity and an abnormal amount of uppercase letters in Fig. 5.

2 Choosing a Model

2.1 Introduction to NLP Modeling

Historically, one of the primary challenges with NLP has been the transformation of large bodies of text into features suitable for machine learning. One of the simplest and most fundamental techniques is appropriately named Bag-of-Words as it first creates a dictionary by parsing every word in the corpus. Relevant features can then be generated for each row of text data by counting the frequency that dictionary words appear in the text.

[1]The diagonal in Fig. 2 represents the total number of data points for which the corresponding toxicity label had a value of 1, irrespective of the values of all other labels.

Figure 4: Word clouds populated with the most common words from non-toxic comments (all zero labels) and most-toxic comments (all one labels).

2.1.1 Frequency-Based Embeddings: Bag-of-Words

The Bag-of-Words technique is limited by its inability to extrapolate context and meaning from words. Well-known synonyms are treated the same as all other pairs of words. In addition, the binary nature of Bag-of-Words means every word is weighed the same, whether the word is a proper noun like "Eiffel" or an article like "a." This latter issue was mitigated by the introduction of TF-IDF, or Term Frequency-Inverse Document Frequency. TF-IDF calculates each word's significance by comparing its frequency in the text to its frequency in the corpus.

Other pre-processing techniques include tokenization, stop-words-filtering, stemming, and lemmatization [8]. Tokenization parses corpuses into meaningful words and phrases (tokens) instead of words alone. Stop-words-filtering removes stop-words like "the," "a," and "an," stemming removes stems like "-ing" and lemmatization removes additional grammar rules so that "ran," "running," etc. reduce to "run." Despite the pre-processing techniques, the resulting word frequency vector or word embedding is inherently sparsely populated. Sparsely populated matrices are prone to being inefficient and inaccurate. One way to avoid the pitfalls of sparsely populated matrices is by calculating the co-occurrence matrix using principal component analysis (PCA) or singular value decomposition (SVD) [4]. By choosing k components out of V components so that the new matrix is of the form $V \times k$, a single word will be represented in k dimensions instead of V dimensions while capturing the same semantic meaning. In addition, latent semantic analysis (LSA) extrapolates certain semantic relationships between words like "man" and "woman" that can be interpreted by a trainer to form more probability-based predictions than the otherwise frequency-based ones.

2.1.2 Prediction-Based Embeddings: Word2Vec

While calculating a co-occurrence matrix enables word vectors to encapsulate some contextual context, the method remains entirely deterministic. On the contrary, probabilistic methods use underlying neural networks to learn weights that best predict target vectors based on the input word(s). Such methods fall under the category of predictive embedding models. Word2Vec is one such model [2].

Word2Vec is not a single algorithm but the combination of two techniques: Continuous Bag of Words (CBOW) and Skip-Gram. Outside of using different loss functions (negative log-likelihood instead of mean squared error) and different activation functions (linear vs. sigmoid), CBOW and Skip-Gram do not deviate far from the default multilayer perceptron (MLP). CBOW and Skip-Gram also do not deviate far from each other. The Skip-Gram architecture is the exact mirror reflection of the CBOW architecture. While CBOW aims to predict a word given a context (a bag of words), Skip-Gram aims to predict a context given a word.

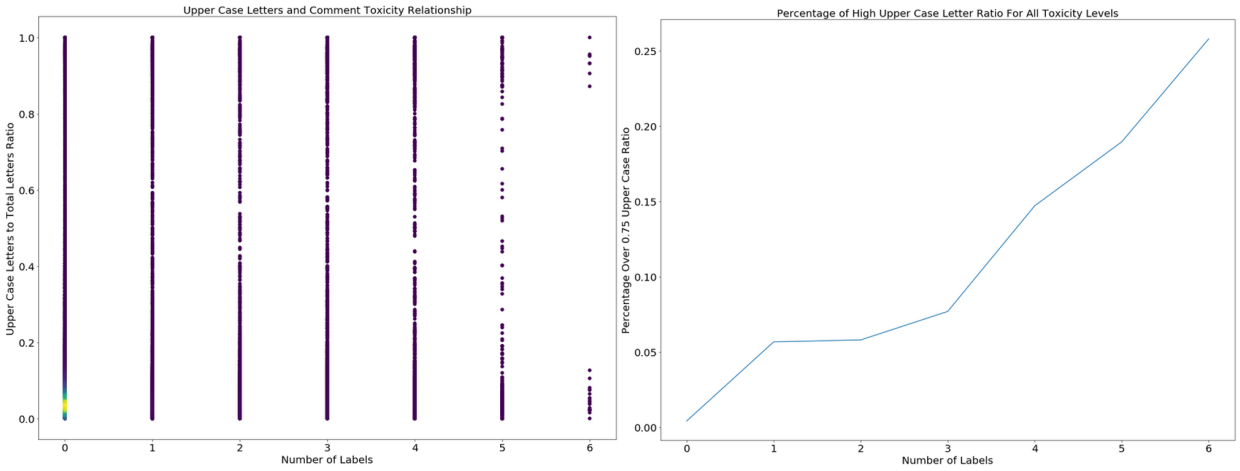

Figure 5: Plotting the uppercase ratio to the number of labels (left) shows that the texts with higher numbers of labels also have more extreme uppercase-to-lowercase ratios. We further illustrate the correlation by plotting the percentage of source texts with very high uppercase ratios to the number of labels (right).

2.2 Current State of NLP Modeling

Word2Vec utilizes shallow neural networks to make predictions; newer NLP neural architectures use deep feed forward networks to improve language classification performance. Other developments that have occurred since Word2Vec are Convolution Neural Networks (CNN) [5], Recurrent Neural Networks (RNN) [6], and Attention [9] and Copy Mechanisms [3].

CNNs are comprised of three primary repeating layers or operations and one final classification step. The three layers of operations are the convolution, non-linearity, and pooling layers. The convolution step in which CNNs derive their name extracts features from the input text. The non-linearity step performs the Rectified Linear Unit (ReLU) operation so that negative feature values map to zero, and the pooling step reduces the dimensionality of the resulting feature set. RNNs build on CNNs by not only further emphasizing significant words in text, but by also emphasizing the ordering and dependencies among the words. This is achieved with the idea of cell states, input gates, forget gates, and output gates. The intuition is that information in the cell state persists as information is continuously added through the input gates, removed through the forget gates, and returned through the output gates. Finally, attention mechanisms provide a means of weighing the contextual impact of each input vector on the output prediction of the RNN, and copy mechanisms reduce overfitting to the input text. The attention mechanism helps delineate which parts of the comment triggered which elements of the ultimate prediction, and the copy mechanism determines whether a word prediction should be generated from the source text or the general vocabulary embedding.

2.2.1 Choosing BERT

While the original plan consisted of comparing SVD on Bag-of-Words to Word2Vec in order to document the performance differences between a frequency-based embedding and a prediction-based embedding, it soon became clear that neither method reflected current NLP classification standards; not to mention, similar comparisons were already well-documented [7].

So instead, we turned our attention to integrating all the above techniques to get a sense of the current state of NLP. Coincidentally, the BERT paper was published a little over a month ago with lofty claims of posting state-of-the-art results for eleven distinct NLP tasks. Intrigued, we began researching BERT's functionality and reviewing the open-source repository with the ultimate goal of training a BERT model to solve the toxic comment classification task. A summary of our findings follows.

2.2.2 BERT Summary

The most novel aspect of BERT's architecture is the bi-directional Transformer encoder [1]. A Transformer applies an attention mechanism by focusing on specific words in the source sentence; for example, in the sentence "The robber picked the lock," the Transformer can immediately attend to the word "robber" because it minimizes the context possibilities of the sentence. In the next step, the Transformer would compare the word "robber" to every other word in the sentence. Intuitively, the next two words to receive high attention scores would be "lock" and "picked."

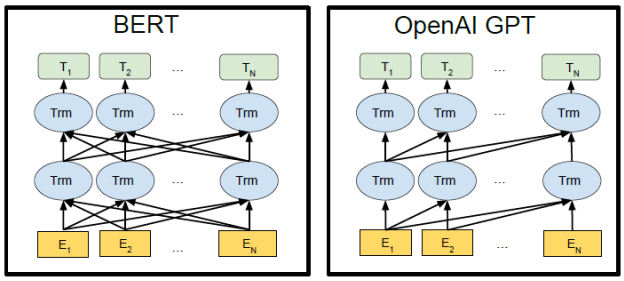

Figure 6: A comparison of BERT's bi-directional transformer encoder to OpenAI GPT's left-to-right transformer encoder.

BERT was not the first model to use a Transformer encoder. OpenAI GPT was implemented with the left-to-right Transformer encoder seen in Fig. 6; however, given the example above, we can clearly see the benefits of a bi-directional process. Moving from left-to-right, we would be unable to attend to "lock" prior to "picked" since "picked" comes first in the sentence despite "lock" contributing more to the context. It is plausible to replace "lock" with "fruit" given the verb "picked," but it is not clear how "picked" could be replaced given the noun "lock." BERT further bolsters its deep bidirectional word representations by introducing the pre-training objective: the "masked language model" (MLM). MLM randomly masks 15 percent of input tokens during training and presents the BERT model with the objective of predicting the masked token.

The theories behind transfer learning have been proven in the image classification field with the release of very complex CNNs like the Inception v3 Model also released by Google. The thought behind transfer learning was to develop a model that could be potentially transferred to another application. Google, with its access to some of the largest data stores of both images and words (BERT was pre-trained on 40 epochs totalling over 3.3 billion words), will have the most robust architecture and most fair weights when large amounts of data have already been used to pre-train the model. BERT additionally, was built as a task-agnostic operator, behaving almost like a CNN in image classification, spitting out vector transformations of original documents, sentences, or words using the novel techniques briefly mentioned above.

Table 1: BERT vs State-of-the-art [1]

System	MNLI-(m/mm)	CoLA	MRPC	RTE	Average
Pre-OpenAI SOTA	80.1	35.0	86.0	61.7	65.7
BiLSTM+ELMo+Attn	76.1	36.0	84.9	56.8	63.45
OpenAI GPT	81.4	45.4	82.3	56.0	66.27
BERT Base	83.4	52.1	88.9	66.4	72.7
BERT Large	**85.9**	**60.5**	**89.3**	**70.1**	**76.45**

Finally, the most compelling reason for the use of BERT in this application is the almost 5% increase in accuracy achieved over any other previously state-of-the-art mechanisms. The results from the paper shown in Table 1 demonstrate the disparity between BERT and the previous state-of-the-art systems in word-related, sentence-related, and whole document-related tasks. From these results, we can conclusively show that BERT is the best fit of model for NLP related tasks in the modern day.

3 Results

As we are using the BERT model, we have a lot of options regarding customization and hyper-parameters. The first main parameter that we can customize is the use of BERT large (cased only) or BERT regular (cased or uncased). For the sake of this experiment where case or capitalization matters, we settle for using both the BERT large cased model and the BERT regular cased model. The main hyper-parameters that are inputted directly into the model are as follows: training batch size, evaluation batch size, the learning rate, the number of training epochs, the warm-up proportion, the max sequence length, the number of iterations per loop, and the number of TPU Cores used. The last of these, although seemingly inconsequential, actually does affect the validation accuracy as BERT, when evaluating the validation batches, truncates the last batches further on more TPU cores, affecting the validation accuracy by as much as .5%. The main parameters that we focused on optimizing, both because of time limitations and the theory that these would improve the accuracies the most, were the learning rate as well as the max sequence length.

Table 2: Results

MSL / **LR**	128				512				1024			
1e-2	94.1	94.0	95.0	94.8	94.2	93.0	92.0	93.8	93.8	94.2	94.0	94.0
5e-3	95.1	96.3	96.4	96.5	96.0	96.6	96.8	95.5	96.4	97.3	97.8	97.5
1e-3	96.6	96.4	96.8	97.1	96.1	97.3	97.8	97.5	97.0	97.0	98.8	96.5
5e-4	97.1	96.5	97.3	98.0	97.2	96.4	97.1	97.9	98.1	96.6	97.4	98.2
1e-4	98.4	97.4	98.0	97.7	98.3	97.0	98.3	98.5	**98.2**	**97.4**	**99.0**	**98.8**

The learning rate is a very standard hyper-parameter in Machine Learning that controls how much we are adjusting the weights of our network with respect to the gradient of the loss function. Theoretically, a higher learning rate means the model will get to a better validation accuracy quicker, but will plateau or bounce around the bottom of the curve without ever finding the local or true minima. A lower learning rate means it will take more iterations to get to the minima but will more likely produce a higher validation and test (in this case we refer to test as the actual blind accuracy from Kaggle) accuracy. The Max Sequence Length is also a very important hyper-parameter, dictating how many sequences or pieces/slices of words in each comment are actually processed into features before truncation or zero-padding. Shorter comments are zero padded, longer ones are truncated. A longer comment with the toxicity located at the end will be miscategorized if the sequence length is not long enough, but a larger sequence length also means a larger input vector and a longer time training the model. A longer sequence length seems to be valuable, but the training time may not be worthwhile.

In practice, we can also see with both BERT Large and BERT regular that the higher the learning rate, on average, the lower the validation and Kaggle accuracy appear. This also means that our tuning for the hyper-parameter, number of iterations, is high enough to arrive at a good local minima. With further iterations and a lower learning rate, a higher accuracy on validation data could be possible.

In Table 2, Learning Rate is abbreviated LR and Max Sequence Length as MSL. The accuracies follow the formatting:

BERT Regular Validation Acc. | *BERT Regular Kaggle Acc.* | *BERT Large Validation Acc.* | *BERT Large Kaggle Acc.*

The accuracies reported for the validation accuracies for both BERT Large and Regular are five-fold cross-validated to account for variance and bias in the data. The highest Kaggle accuracies we found were for a learning rate of 1e-4 and a Max Sequence Length of 1024.

Although not included in a table or chart, we did a very preliminary investigation of the misclassifications BERT made and the majority of misclassification took place in the threat and insult categories, where the model seemed to not understand the difference between a threat or insult. Certain comments were classified as both threats and insults instead of only threats or only insults. The obscene, toxic, severe_toxic, identity_hate labels had a very high rate of successful prediction in all cases. It is vital to also mention that the labels we used were a combination of all of the categories, so a non-toxic comment would be labeled 000000 rather

than each of the separate binary components. If we were to train the model for each category, i.e. one for toxicity, one for severe toxicity, etc. and use an ensemble of BERT classifiers to predict each category, we may have been able to do better than the results we achieved currently. All of the code to reproduce these results for a single hyper-parameter tuning arrangement are included in Appendix A4.

4 Conclusion

It is evident that even with simply fine-tuning the last layer of BERT, we can achieve enviable results. The current highest accuracy on Kaggle is 98.90%; our model with only two optimized hyper-parameters accomplished a 98.80% accuracy, placing us in the top 14 submissions. BERT Large clearly demonstrates its value, accomplishing what a large team did over countless hours and multiple submissions, in the short span of 20 hours and 15 submissions. With further tuning, we are confident we can surpass the 98.90% accuracy and set a new standard for toxic comment classification. We are grateful that this project gave us the opportunity to learn the BERT internals and run a model on intriguing data. We are excited to be part of the dawn of BERT and cannot wait to begin a series of future papers using BERT.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[2] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[3] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. In *Advances in Neural Information Processing Systems*, pages 1693–1701, 2015.

[4] Thomas Hofmann. Probabilistic latent semantic analysis. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 289–296. Morgan Kaufmann Publishers Inc., 1999.

[5] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[6] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[7] MUNEEB TH, Sunil Sahu, and Ashish Anand. Evaluating distributed word representations for capturing semantics of biomedical concepts. *Proceedings of BioNLP 15*, pages 158–163, 2015.

[8] Alper Kursat Uysal and Serkan Gunal. The impact of preprocessing on text classification. *Information Processing & Management*, 50(1):104–112, 2014.

[9] Wenpeng Yin, Hinrich Schütze, Bing Xiang, and Bowen Zhou. Abcnn: Attention-based convolutional neural network for modeling sentence pairs. *arXiv preprint arXiv:1512.05193*, 2015.

Appendices

A Code

A.1 Exploring the Basic Structure and Size of the Dataset

```
# Import the data loader code (we assume that
# a copy resides in the current dir).
import data_loader as dl

import numpy as np
np.random.seed(42)

# Load the entire training set:
# we don't want to split it into training and validation for
# this part as we just want to investigate what the data looks
# like, which is the reason valid_rate is set to 0.
full_tr, _ = dl.load_train_data('../dataset/train.csv', valid_rate=0.0)


# Let's first inspect the shape of our data
print(">> DATA SHAPE <<")
print(f"full_tr.shape={full_tr.shape}")
print()

# Let's output the first 3 data points to
# get an idea of how the data frame looks
# (e.g. colums/feature mappings). Note that
# row is printed horizontally so that each
# column of the row looks like a row in the
# output.
print(">> EXAMPLE ROWS (first 3) <<")
for i in range(0,3):
    print(full_tr.iloc[i])
    print()
```

A.2 Code for Exploring the Distribution of Toxic vs. Non-toxic Datapoints

```
# Let's get an idea of the distribution of
# toxic vs non-toxic comments...

def select_nontoxic(df):
    """
    Subsamples df, selecting only those rows
    that represent a non-toxic comment.
    """
    return df.loc[(df['toxic'] == 0) & \
                  (df['severe_toxic'] == 0) & \
                  (df['obscene'] == 0) & \
                  (df['threat'] == 0) & \
                  (df['insult'] == 0) & \
                  (df['identity_hate'] == 0)]

def select_toxic(df):
    """
    Subsamples df, selecting only those rows
    that represent a toxic comment.
    """
    return df.loc[(df['toxic'] != 0) | \
                  (df['severe_toxic'] != 0) | \
                  (df['obscene'] != 0) | \
                  (df['threat'] != 0) | \
                  (df['insult'] != 0) | \
                  (df['identity_hate'] != 0)]

def print_simple_dist(df_nontoxic, df_toxic, df_full):
```

```
    print(f"Non-toxic␣comments␣count:␣{df_nontoxic.shape[0]}")
    print(f"Toxic␣comments␣count:␣{df_toxic.shape[0]}")
    nontoxic_frac = float(df_nontoxic.shape[0]) / \
                    float(df_full.shape[0])
    toxic_frac = float(df_toxic.shape[0]) / \
                 float(df_full.shape[0])
    print(f"Non-toxic␣fraction␣(of␣total):␣{nontoxic_frac}")
    print(f"Toxic␣fraction␣(of␣total):␣{toxic_frac}")
    return (nontoxic_frac, toxic_frac)

full_tr_nontoxic_rows = select_nontoxic(full_tr)
full_tr_toxic_rows = select_toxic(full_tr)

# Sanity checks
assert full_tr.shape[0] == full_tr_nontoxic_rows.shape[0] + \
                           full_tr_toxic_rows.shape[0]

# Compute and print the distribution of toxic vs. non-toxic
full_tr_nontoxic_frac, full_tr_toxic_frac = print_simple_dist(full_tr_nontoxic_rows,
                                                              full_tr_toxic_rows,
                                                              full_tr)
```

A.3 Code for Generating a Toxicity Correlation Matrix

```
import numpy as np
import matplotlib.pyplot as plt

def construct_comb_matrix(df, cols=None):
    """
    Compute a matrix that holds the number of times
    that two toxicity labels in df were both set to
    1, for all combinations of labels.

    cols: list of column names to include.
    """
    # Reduce dataframe to subset matching specified columns.
    df = df if cols is None else df[cols]
    # Prepare result matrix
    m_width = len(df.columns.values)
    matrix = np.zeros((m_width, m_width))
    for i, cname1 in enumerate(df.columns.values):
        for j, cname2 in enumerate(df.columns.values):
            filtr = (df[cname1] == 1) if cname1 == cname2 \
            else (df[cname1] == 1) & (df[cname2] == 1)
            filtered_df = df.loc[filtr]
            # The row count in the filtered matrix is the
            # value we are looking for for this i,j
            matrix[i][j] = filtered_df.shape[0]
    return matrix

def plot_comb_matrix(matrix, cols):
    """
    Creates a matrix plot that visualizes a
    matrix created using construct_comb_matrix.
    """
    f, ax = plt.subplots(1, 1, figsize=(15, 10))
    mplot = ax.matshow(matrix, interpolation='nearest')
    f.colorbar(mplot)
    ax.set_xticklabels(['']+cols)
    ax.set_yticklabels(['']+cols)
    ax.set_title("Toxicity␣labels␣appearing␣together")
    plt.show()

# Load the entire training set:
# we don't want to split it into training and validation as
# we just want to plot what the data looks like.
full_tr, _ = dl.load_train_data('../dataset/train.csv', valid_rate=0.0)
```

```
# Toxicity labels are the last 6 columns.
lbls = list(full_tr.columns.values)[2:]
cmat = construct_comb_matrix(full_tr, lbls)
plot_comb_matrix(cmat, lbls)
```

A.4 BERT Model Applied to the Kaggle Toxic Comment Challenge Dataset

First step is to install Kaggle in order to be able to actually download the data:

```
! pip install kaggle
```

As the second step, change the environment variables to authenticate with Kaggle, and download the dataset to the correct file for future processing in our `toxic_classification` task:

```
import os
os.environ['KAGGLE_USERNAME']='faramarzmunshi'
os.environ['KAGGLE_KEY']='PLACEHOLDER'

! cd kaggle_data || mkdir kaggle_toxic_classification || cd kaggle_toxic_classification ||
    kaggle competitions download -c jigsaw-toxic-comment-classification-challenge

! mkdir kaggle_data
! mkdir kaggle_data/kaggle_toxic_classification
! cd kaggle_data/kaggle_toxic_classification
! kaggle competitions download -c jigsaw-toxic-comment-classification-challenge
! mv *.zip kaggle_data/kaggle_toxic_classification
! ls -la kaggle_data/kaggle_toxic_classification

! unzip -o kaggle_data/kaggle_toxic_classification/train.csv.zip -d kaggle_data/
    kaggle_toxic_classification/
! unzip -o kaggle_data/kaggle_toxic_classification/test.csv.zip -d kaggle_data/
    kaggle_toxic_classification/
! unzip -o kaggle_data/kaggle_toxic_classification/test_labels.csv.zip -d kaggle_data/
    kaggle_toxic_classification/
```

Thirdly, split the dataset into the correct validation, test, and training data, along with changing the file format to a friendly format for input to BERT, removing and cleansing special characters thatw would break the tab separated value input that BERT accepts:

```
import csv
import numpy as np
import random
import pandas as pd

PERCENTAGE_TRAINING_SET = .8

def simple_csv2tsv(file):
  with open("kaggle_data/kaggle_toxic_classification/{0}".format(file)) as csvin, open("
      kaggle_data/kaggle_toxic_classification/{0}.tsv".format(file[:-4]), 'w') as tsvout:
    csvin = csv.reader(csvin)
    tsvout = csv.writer(tsvout, delimiter='\t')

    for row in csvin:
      tsvout.writerow([row[0], row[1].replace('\n','.␣')])


df = pd.read_csv("kaggle_data/kaggle_toxic_classification/train.csv", header=0)
df = df.sample(frac=1)
df['*'] = (['*'] * len(df['id']))
df['comment_text'] = df['comment_text'].map(lambda x: x.strip().replace('\n', '.␣').replace(
    '\t', '␣').replace('"', ''))
msk = np.random.rand(len(df)) < PERCENTAGE_TRAINING_SET
df['label'] = df['toxic'].apply(str) + df['severe_toxic'].apply(str) + df['obscene'].apply(
    str) + df['threat'].apply(str) + df['insult'].apply(str) + df['identity_hate'].apply(str
    )
df = df[['id', 'label', '*', 'comment_text']]
train = df[msk]
```

```
test = df[~msk]
train.to_csv("kaggle_data/kaggle_toxic_classification/train.tsv", quoting=csv.QUOTE_NONE,
    index = False, sep='\t', header=None)
test.to_csv("kaggle_data/kaggle_toxic_classification/dev.tsv", quoting=csv.QUOTE_NONE, index
     = False, sep='\t', header=None)

# convert_csv_train("train.csv")
simple_csv2tsv("test.csv")
```

Now, we must set up the Google Colab TPU running environment, verify a TPU device is successfully connected, and upload and authenticate credentials for the TPU on my personal GCS bucket. If running at home, you will be asked to input your Google Credentials that will allow this to run on your free TPU or GCE instance:

```
import datetime
import json
import os
import pprint
import random
import string
import sys
import tensorflow as tf

assert 'COLAB_TPU_ADDR' in os.environ, 'ERROR: Not connected to a TPU runtime; please see
    the first cell in this notebook for instructions!'
TPU_ADDRESS = 'grpc://' + os.environ['COLAB_TPU_ADDR']
print('TPU address is', TPU_ADDRESS)

from google.colab import auth
auth.authenticate_user()
with tf.Session(TPU_ADDRESS) as session:
  print('TPU devices:')
  pprint.pprint(session.list_devices())

  # Upload credentials to TPU.
  with open('/content/adc.json', 'r') as f:
    auth_info = json.load(f)
  tf.contrib.cloud.configure_gcs(session, credentials=auth_info)
  # Now credentials are set for all future sessions on this TPU.
```

We must then prepare, download, and import BERT's modules and add it to system path for faster and more convenient access:

```
import sys

!test -d bert_repo || git clone https://github.com/google-research/bert bert_repo
if not 'bert_repo' in sys.path:
  sys.path += ['bert_repo']
```

Now there are three key steps:

- Specify the task and the location of the training data implicitly from task.
- Specify which of the three BERT pretrained models we would like to use.
- Specify which specific GS bucket, and create an output directory on that bucket for the model's checkpoints and its evaluation results.

```
TASK = 'kaggle_toxic_classification' #@param {type:"string"}

TASK_DATA_DIR = 'kaggle_data/' + TASK
print('***** Task data directory: {} *****'.format(TASK_DATA_DIR))
! ls $TASK_DATA_DIR

# Available pretrained model checkpoints:
```

```
#   uncased_L-12_H-768_A-12: uncased BERT base model
#   uncased_L-24_H-1024_A-16: uncased BERT large model
#   cased_L-12_H-768_A-12: cased BERT large model
BERT_MODEL = 'cased_L-12_H-768_A-12' #@param {type:"string"}
BERT_PRETRAINED_DIR = 'gs://cloud-tpu-checkpoints/bert/' + BERT_MODEL
print('***** BERT pretrained directory: {} *****'.format(BERT_PRETRAINED_DIR))
!gsutil ls $BERT_PRETRAINED_DIR

BUCKET = 'bucket_faramarz_ml2' #@param {type:"string"}
assert BUCKET, 'Must specify an existing GCS bucket name'
OUTPUT_DIR = 'gs://{}/bert/models/{}'.format(BUCKET, TASK)
tf.gfile.MakeDirs(OUTPUT_DIR)
print('***** Model output directory: {} *****'.format(OUTPUT_DIR))
```

Now that everything is set up, we can finally specify the model's parameters. We are using the Cola prebuilt task as a baseline, modifying it heavily for our own classification needs:

```
#@title Model Parameters { form-width: "350px" }
# Setup task specific model and TPU running config.

import modeling
import optimization
import run_classifier
import tokenization


# Model Hyper Parameters
TRAIN_BATCH_SIZE = 32 #@param {type:"integer"}
EVAL_BATCH_SIZE = 8 #@param {type:"integer"}
LEARNING_RATE = 1e-3 #@param {type:"number"}
NUM_TRAIN_EPOCHS = 3.0 #@param {type:"number"}
WARMUP_PROPORTION = 0.1 #@param {type:"number"}
MAX_SEQ_LENGTH = 128 #@param {type:"integer"}
# Model configs
SAVE_CHECKPOINTS_STEPS = 1000 #@param {type:"integer"}
ITERATIONS_PER_LOOP = 1000 #@param {type:"integer"}
NUM_TPU_CORES = 8 #@param {type:"integer"}
VOCAB_FILE = os.path.join(BERT_PRETRAINED_DIR, 'vocab.txt')
CONFIG_FILE = os.path.join(BERT_PRETRAINED_DIR, 'bert_config.json')
INIT_CHECKPOINT = os.path.join(BERT_PRETRAINED_DIR, 'bert_model.ckpt')
DO_LOWER_CASE = BERT_MODEL.startswith('uncased')

processors = {
  "cola": run_classifier.ColaProcessor,
  "mnli": run_classifier.MnliProcessor,
  "mrpc": run_classifier.MrpcProcessor,
}
processor = processors['cola']()
label_list = ['000000', '000001', '000010', '000011', '000100', '000101', '000110', '000111'
    , '001000', '001001', '001010', '001011', '001100', '001101', '001110', '001111', '
    010000', '010001', '010010', '010011', '010100', '010101', '010110', '010111', '011000',
     '011001', '011010', '011011', '011100', '011101', '011110', '011111', '100000', '100001
    ', '100010', '100011', '100100', '100101', '100110', '100111', '101000', '101001', '
    101010', '101011', '101100', '101101', '101110', '101111', '110000', '110001', '110010',
     '110011', '110100', '110101', '110110', '110111', '111000', '111001', '111010', '111011
    ', '111100', '111101', '111110', '111111']
tokenizer = tokenization.FullTokenizer(vocab_file=VOCAB_FILE, do_lower_case=DO_LOWER_CASE)

tpu_cluster_resolver = tf.contrib.cluster_resolver.TPUClusterResolver(TPU_ADDRESS)
run_config = tf.contrib.tpu.RunConfig(
    cluster=tpu_cluster_resolver,
    model_dir=OUTPUT_DIR,
    save_checkpoints_steps=SAVE_CHECKPOINTS_STEPS,
    tpu_config=tf.contrib.tpu.TPUConfig(
        iterations_per_loop=ITERATIONS_PER_LOOP,
        num_shards=NUM_TPU_CORES,
        per_host_input_for_training=tf.contrib.tpu.InputPipelineConfig.PER_HOST_V2))
```

```
train_examples = processor.get_train_examples(TASK_DATA_DIR)
num_train_steps = int(
    len(train_examples) / TRAIN_BATCH_SIZE * NUM_TRAIN_EPOCHS)
num_warmup_steps = int(num_train_steps * WARMUP_PROPORTION)

model_fn = run_classifier.model_fn_builder(
    bert_config=modeling.BertConfig.from_json_file(CONFIG_FILE),
    num_labels=len(label_list),
    init_checkpoint=INIT_CHECKPOINT,
    learning_rate=LEARNING_RATE,
    num_train_steps=num_train_steps,
    num_warmup_steps=num_warmup_steps,
    use_tpu=True,
    use_one_hot_embeddings=True)

estimator = tf.contrib.tpu.TPUEstimator(
    use_tpu=True,
    model_fn=model_fn,
    config=run_config,
    train_batch_size=TRAIN_BATCH_SIZE,
    eval_batch_size=EVAL_BATCH_SIZE)
```

Now we actually train the model on the training data:

```
# Train the model.
train_features = run_classifier.convert_examples_to_features(
    train_examples, label_list, MAX_SEQ_LENGTH, tokenizer)
print('*****␣Started␣training␣at␣{}␣*****'.format(datetime.datetime.now()))
print('␣␣Num␣examples␣=␣{}'.format(len(train_examples)))
print('␣␣Batch␣size␣=␣{}'.format(TRAIN_BATCH_SIZE))
tf.logging.info("␣␣Num␣steps␣=␣%d", num_train_steps)
train_input_fn = run_classifier.input_fn_builder(
    features=train_features,
    seq_length=MAX_SEQ_LENGTH,
    is_training=True,
    drop_remainder=True)
estimator.train(input_fn=train_input_fn, max_steps=num_train_steps)
print('*****␣Finished␣training␣at␣{}␣*****'.format(datetime.datetime.now()))
```

And finally, we can evaluate the model on the custom hold-out set we created above:

```
# Eval the model.
eval_examples = processor.get_dev_examples(TASK_DATA_DIR)
eval_features = run_classifier.convert_examples_to_features(
    eval_examples, label_list, MAX_SEQ_LENGTH, tokenizer)
print('*****␣Started␣evaluation␣at␣{}␣*****'.format(datetime.datetime.now()))
print('␣␣Num␣examples␣=␣{}'.format(len(eval_examples)))
print('␣␣Batch␣size␣=␣{}'.format(EVAL_BATCH_SIZE))
# Eval will be slightly WRONG on the TPU because it will truncate
# the last batch.
eval_steps = int(len(eval_examples) / EVAL_BATCH_SIZE)
eval_input_fn = run_classifier.input_fn_builder(
    features=eval_features,
    seq_length=MAX_SEQ_LENGTH,
    is_training=False,
    drop_remainder=True)
result = estimator.evaluate(input_fn=eval_input_fn, steps=eval_steps)
print('*****␣Finished␣evaluation␣at␣{}␣*****'.format(datetime.datetime.now()))
output_eval_file = os.path.join(OUTPUT_DIR, "eval_results.txt")
with tf.gfile.GFile(output_eval_file, "w") as writer:
  print("*****␣Eval␣results␣*****")
  for key in sorted(result.keys()):
    print('␣␣{}␣=␣{}'.format(key, str(result[key])))
    writer.write("%s␣=␣%s\n" % (key, str(result[key])))
```