

✓ Deep Learning Final Project

For my final project, I've chosen to work on a text classification problem that lends itself to using recurrent neural networks. I'd like to practice and better understand that area more.

The prediction problem is focused on identifying whether or not a comment is toxic. Data and more details can be found on the Kaggle competition page: <https://www.kaggle.com/c/jigsaw-unintended-bias-in-toxicity-classification/overview>

The data comes from an online repository of comments and has been annotated by proficient English speakers. In this analysis, I will focus on the comment text and the rating for whether or not the comment is toxic. The dataset includes many other annotations, including whether or not the comment pertains to particular identity groups (women, LGBTQ, specific religions, etc.) as well as the type of toxic comment (threats, insults, etc.). The training set alone includes over 200,000 records. I found this to be a rich and interesting dataset to work with and would encourage you to consider it for future projects.

Github link: <https://github.com/blockee/cu-deep/tree/main/final-comments>

✓ Setup and Data Reading

```
import pandas as pd
from sklearn.model_selection import train_test_split

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer, WordNetLemmatizer
from nltk.tokenize import word_tokenize
import string

import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
from tensorflow.keras.initializers import Constant

def plot_metrics(history):
    metrics = ['accuracy', 'precision', 'recall']

    fig, axes = plt.subplots(3, 1, figsize=(8, 6)) # 3 rows, 1 column
    for n, metric in enumerate(metrics):

        axes[n].plot(history.history[metric], label=metric)
        axes[n].plot(history.history[f'val_{metric}'], label=f'val_{metric}')
        axes[n].set_title(f'Subplot {metric}')
        axes[n].set_xlabel('Epoch')
        axes[n].set_ylabel(metric)
        axes[n].legend(loc='lower right')

    # Adjust layout to prevent overlapping
    plt.tight_layout()

    # Show the plot
    plt.show()

# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('punkt_tab')

[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
```

```
[nltk_data] Unzipping tokenizers/punkt_tab.zip.
True
```

```
df = pd.read_csv('/content/train.csv', on_bad_lines='skip')
df.shape
```

```
(260416, 45)
```

▼ Data Exploration

```
df.head(5)
```

```
↗
```

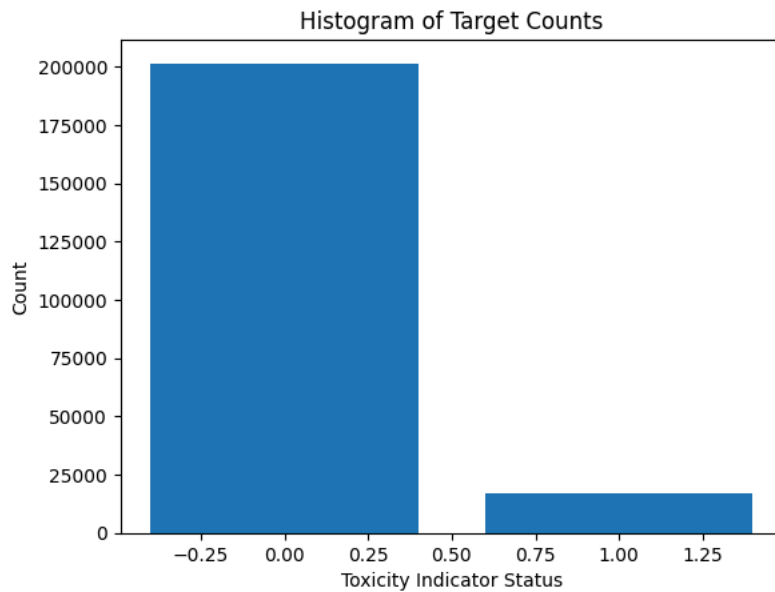
	id	target	comment_text	severe_toxicity	obscene	identity_attack	insult	threat	asian	atheist	...	article_id	rating	fur
0	59848	0.000000	This is so cool. It's like, 'would you want yo...	0.000000	0.0	0.000000	0.000000	0.0	NaN	NaN	...	2006	rejected	
1	59849	0.000000	Thank you!! This would make my life a lot less...	0.000000	0.0	0.000000	0.000000	0.0	NaN	NaN	...	2006	rejected	
2	59852	0.000000	This is such an urgent design problem; kudos t...	0.000000	0.0	0.000000	0.000000	0.0	NaN	NaN	...	2006	rejected	
3	59855	0.000000	Is this something I'll be able to install on m...	0.000000	0.0	0.000000	0.000000	0.0	NaN	NaN	...	2006	rejected	
4	59856	0.893617	haha you guys are a bunch of losers.	0.021277	0.0	0.021277	0.87234	0.0	0.0	0.0	...	2006	rejected	

```
5 rows × 45 columns
```

```
# Creating an indicator column for toxicity so we can treat as a binary classification problem.
```

```
df['toxic_ind'] = np.where(df['target'] >= 0.5, 1, 0)
```

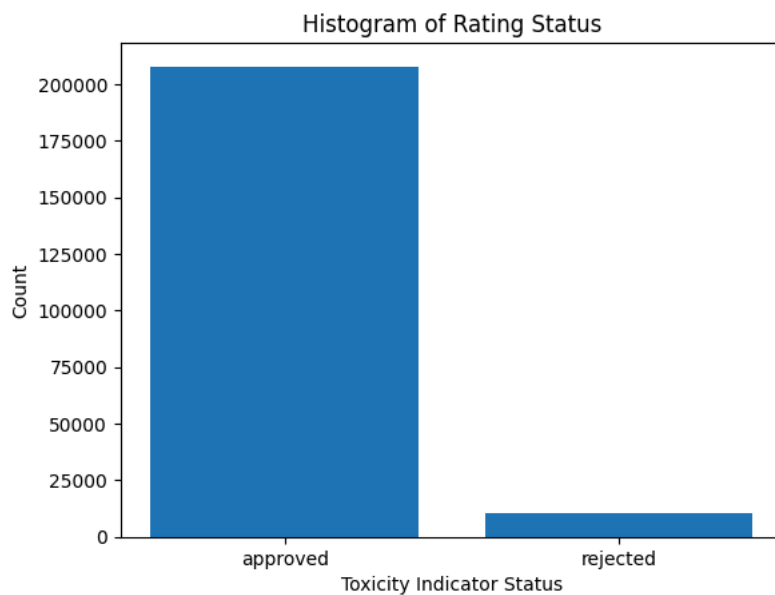
```
category_counts = df['toxic_ind'].value_counts()
# Create the histogram
plt.bar(category_counts.index, category_counts.values)
# Add labels and title
plt.xlabel('Toxicity Indicator Status')
plt.ylabel('Count')
plt.title('Histogram of Target Counts')
# Show the plot
plt.show()
```



The dataset is highly imbalanced. I'll have to take that into consideration when it comes time to pick my evaluation metric.

I noticed this "rating" column which seems like it might be an evaluation of the toxicity rating. I reviewed the data documentation and didn't see this column mentioned. Taking a closer look.

```
category_counts = df['rating'].value_counts()
# Create the histogram
plt.bar(category_counts.index, category_counts.values)
# Add labels and title
plt.xlabel('Toxicity Indicator Status')
plt.ylabel('Count')
plt.title('Histogram of Rating Status')
# Show the plot
plt.show()
```



```
# Are a lot of the rejected ratings also identified as being toxic?
len(df.loc[(df['rating'] == 'rejected') & (df['toxic_ind'] == 1)])

# A higher proportion of toxic comments do have this rejected rating
```



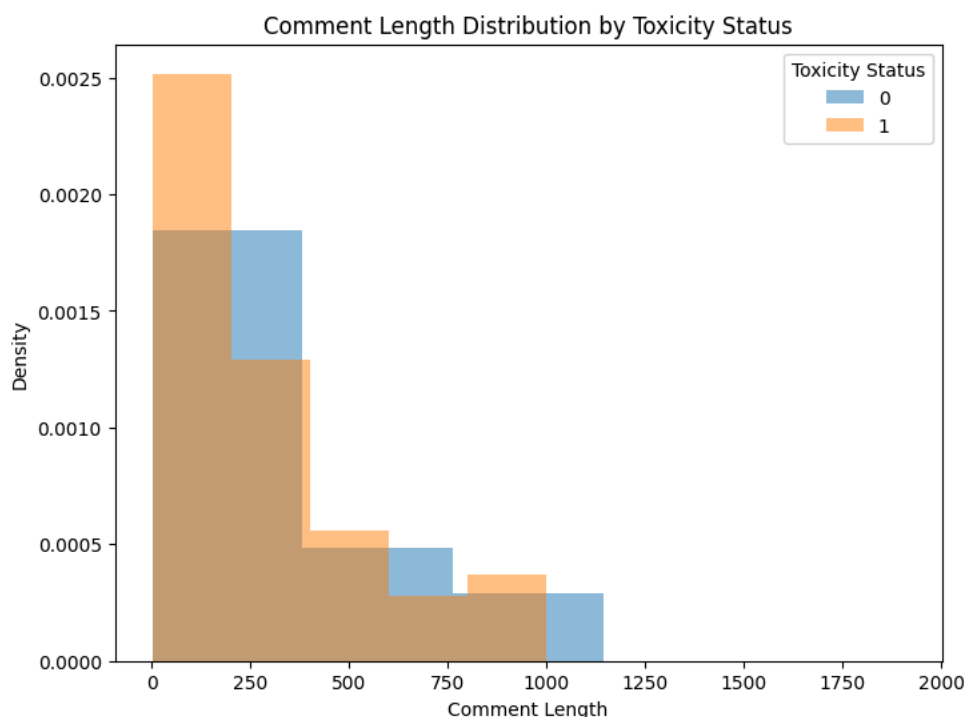
3082

```
# Took a look at some example using the head function
# All the comments I reviewed do seem truly toxic so I'm not sure what role the
```

```
# rating field plays. Won't exclude based on this field for now.

# df.loc[(df['rating'] == 'rejected') & (df['toxic_ind'] == 1)].head(20)

df['comment_len'] = df['comment_text'].apply(len)
grouped = df.groupby('toxic_ind')
# Plot histograms for each category
plt.figure(figsize=(8, 6))
for name, group in grouped:
    plt.hist(group['comment_len'], bins=5, alpha=0.5, label=name, density=True)
plt.xlabel('Comment Length')
plt.ylabel('Density')
plt.title('Comment Length Distribution by Toxicity Status')
plt.legend(title='Toxicity Status')
plt.show()
```



The first thing I notice is that these comments vary quite a lot in length. It may make sense to filter down to comments of a certain length but that is something I can come back and experiment with.

It makes sense to see a spike of very short, toxic comments. It doesn't take many characters to all-caps an expletive or insult.

Exploration takeaways

- Added the toxicity indicator column.
- Optimization metric needs to consider the class imbalance.
- I'm unsure the role that the rating column plays but it seems safe to ignore it.
- Toxic comments do tend to be shorter than non-toxic.
- The comment text is quite messy and doesn't seem to have been cleaned prior.

✓ Text Cleaning

```
def clean_text(text):
    # Convert to lowercase
    text = text.lower()

    # Remove URLs, mentions, hashtags, and special symbols
    text = re.sub(r'http\S+|www\S+|https\S+', '', text)
    text = re.sub(r'@\w+|#\w+', '', text)

    # Remove punctuation and numbers
    text = re.sub(r'[%s]' % re.escape(string.punctuation), '', text)
```

```
text = re.sub(r'\d+', '', text)

# Tokenize
tokens = word_tokenize(text)

# Remove stopwords
stop_words = set(stopwords.words('english'))
tokens = [word for word in tokens if word not in stop_words]

# Stemming and Lemmatization
stemmer = SnowballStemmer("english")
lemmatizer = WordNetLemmatizer()
tokens = [stemmer.stem(lemmatizer.lemmatize(word)) for word in tokens]

# Join tokens back to string
clean_text = ' '.join(tokens)

# Remove extra whitespace
clean_text = re.sub(r'\s+', ' ', clean_text).strip()

return clean_text

def process_comments(df):
    df['clean_comment'] = df['comment_text'].astype(str).apply(clean_text)
    return df

df = process_comments(df)
```

df.loc[:,['comment_text', 'clean_comment']].head(20)

	comment_text	clean_comment
0	This is so cool. It's like, 'would you want yo...	cool like would want mother read realli great ...
1	Thank you!! This would make my life a lot less...	thank would make life lot less anxieti induc k...
2	This is such an urgent design problem; kudos t...	urgent design problem kudo take impress
3	Is this something I'll be able to install on m...	someth abl instal site releas
4	haha you guys are a bunch of losers.	haha guy bunch loser
5	ur a sh*tty comment.	ur sh tti comment
6	hahahahahahahhhha suck it.	hahahahahahahhhha suck
7	FFFFUUUUUUUUUUUUUUUU	ffffuuuuuuuuuuuuuuuu
8	The ranchers seem motivated by mostly by greed...	rancher seem motiv most greed one right allow ...
9	It was a great show. Not a combo I'd of expect...	great show combo expect good togeth
10	Wow, that sounds great.	wow sound great
11	This is a great story. Man. I wonder if the pe...	great stori man wonder person yell shut fuck e...
12	This seems like a step in the right direction.	seem like step right direct
13	It's ridiculous that these guys are being call...	ridicul guy call protest arm threat violenc ma...
14	This story gets more ridiculous by the hour! A...	stori get ridicul hour love peopl send guy dil...
15	I agree; I don't want to grant them the legiti...	agre want grant legitimaci protestor greed sm...
16	Interesting. I'll be curious to see how this w...	interest curious see work often refrain commen...
17	Awesome! I love Civil Comments!	awesom love civil comment
18	I'm glad you're working on this, and I look fo...	glad work look forward see play comment sectio...
19	Angry trolls, misogynists and Racists", oh my....	angri troll misogynist racist oh take iq see s...

Dataset-specific Encoding

I plan to experiement with encoding during this project. I'll start by deriving the encoding from the dataset but will then move on to use a publically-availabl encoder to assess the difference.

```
# Utility function for converting pandas to a TF dataset
```

```
def pandas_to_dataset(df, text_column, target_column, batch_size=64):
    """Converts a Pandas DataFrame with one text input column + target column
    to a TensorFlow Dataset."""

    texts = df[text_column].values

    if target_column is None:
        dataset = tf.data.Dataset.from_tensor_slices((texts))
    else:
        labels = df[target_column].astype(int).values
        dataset = tf.data.Dataset.from_tensor_slices((texts, labels))

    dataset = dataset.batch(batch_size)
    return dataset

train_df, valid_df = train_test_split(df, test_size=0.2, random_state=121, stratify=df.toxic_ind)

print(train_df.shape)
print(valid_df.shape)

↗ (208332, 47)
   (52084, 47)

train_tf = pandas_to_dataset(train_df, 'clean_comment', 'toxic_ind')
valid_tf = pandas_to_dataset(valid_df, 'clean_comment', 'toxic_ind')

example, label = next(iter(train_tf))
print('Text:\n', example.numpy()[0])
print('\nLabel: ', label.numpy()[0])

↗ Text:
  b'love mani stoner smoke lot green ick glad live anchorag rather get cancer deal stuff'

Label: 0

encoder = tf.keras.layers.TextVectorization(max_tokens=15000)
encoder.adapt(train_tf.map(lambda text, _: text))

# Extracting the vocabulary from the TextVectorization layer.
vocabulary = np.array(encoder.get_vocabulary())

# Encoding a test example and decoding it back.
original_text = example.numpy()[0]
encoded_text = encoder(original_text).numpy()
decoded_text = ' '.join(vocabulary[encoded_text])

print('original: ', original_text)
print('encoded: ', encoded_text)
print('decoded: ', decoded_text)

↗ original: b'love mani stoner smoke lot green ick glad live anchorag rather get cancer deal stuff'
  encoded: [ 185  23 6134  727 103 1211   1  769  54  290  302   6 1700  234
    687]
  decoded: love mani stoner smoke lot green [UNK] glad live anchorag rather get cancer deal stuff
```

▼ Model Build

I've decided to create an LSTM model to set the baseline for this task. I'm adding in a dropout layer after the dense layer as well as dropout and recurrent dropout during the LSTM layers. These will prevent overfitting. The memory component of the LSTM layers are well-suited to the task of putting words in context with one another to determine whether or not the comment is toxic or not.

I've also chosen to use Binary Focal Crossentropy as the loss function to address the class imbalance. Binary focal crossentropy downweights easily-classified examples and weighs by the inverse of class frequency, resulting in a greater emphasis on the minority class. This loss function is much more appropriate for this data than simple binary crossentropy.

Additional details: <https://blog.dailydoseofds.com/p/focal-loss-vs-binary-cross-entropy>

```
model = tf.keras.Sequential([
    tf.keras.Input(shape=(1,), dtype=tf.string),
    encoder,
    tf.keras.layers.Embedding(len(encoder.get_vocabulary()), 64, mask_zero=True),
```

```

tf.keras.layers.Bidirectional(
    tf.keras.layers.LSTM(64, return_sequences=True, dropout=0.3, recurrent_dropout=0.2)),

tf.keras.layers.Bidirectional(
    tf.keras.layers.LSTM(32, dropout=0.3, recurrent_dropout=0.2)),

tf.keras.layers.Dense(64, activation='relu',
    kernel_regularizer=tf.keras.regularizers.l2(0.01)),

tf.keras.layers.Dropout(0.4),
tf.keras.layers.Dense(1, activation='sigmoid')
])

```

```

# Summary of the model
model.summary()

```

```

# Compile the model
model.compile(
    loss=tf.keras.losses.BinaryFocalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy', 'precision', 'recall']
)

```

```
# BinaryFocalCrossentropy
```

🔗 Model: "sequential_1"

Layer (type)	Output Shape	Param #
text_vectorization (TextVectorization)	(None, None)	0
embedding_1 (Embedding)	(None, None, 64)	960,000
bidirectional_2 (Bidirectional)	(None, None, 128)	66,048
bidirectional_3 (Bidirectional)	(None, 64)	41,216
dense_2 (Dense)	(None, 64)	4,160
dropout_1 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 1)	65

Total params: 1,071,489 (4.09 MB)
 Trainable params: 1,071,489 (4.09 MB)
 Non-trainable params: 0 (0.00 B)

```

history = model.fit(
    train_tf,
    epochs=4,
    validation_data=valid_tf,
)


```

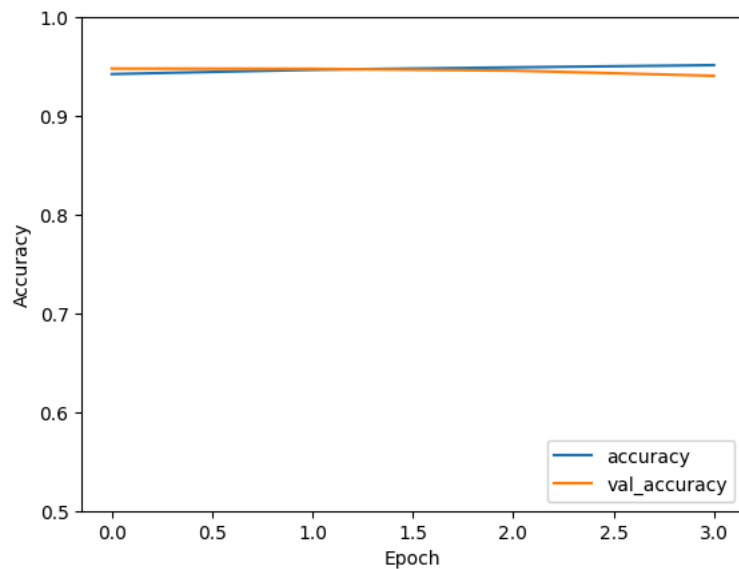
🔗 Show hidden output

```

plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

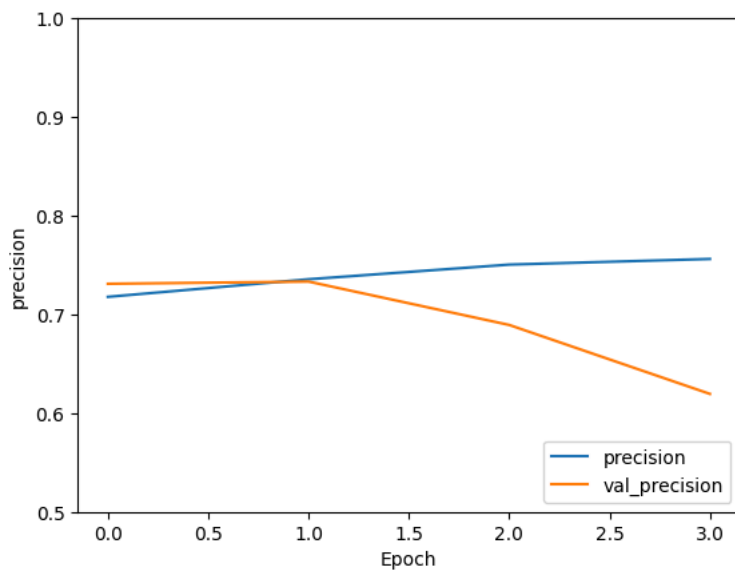
```

 <matplotlib.legend.Legend at 0x79338afa89d0>



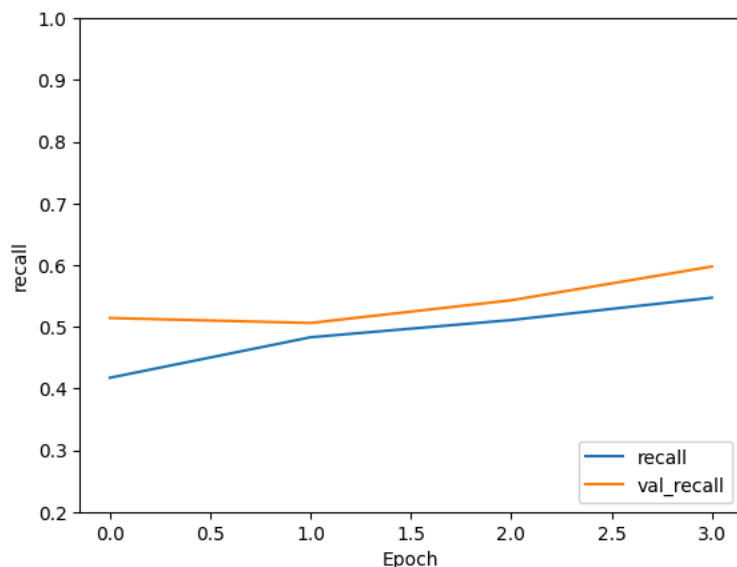
```
plt.plot(history.history['precision'], label='precision')
plt.plot(history.history['val_precision'], label = 'val_precision')
plt.xlabel('Epoch')
plt.ylabel('precision')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
```

 <matplotlib.legend.Legend at 0x7933892b0910>



```
plt.plot(history.history['recall'], label='recall')
plt.plot(history.history['val_recall'], label = 'val_recall')
plt.xlabel('Epoch')
plt.ylabel('recall')
plt.ylim([0.2, 1])
plt.legend(loc='lower right')
```


 <matplotlib.legend.Legend at 0x79338902f9d0>



plot_metrics(history)

 [Show hidden output](#)

The overall performance of the model seems quite strong. Results seem to be best after the second epoch. 73% precision with 51% recall for such a heavily-imbalanced classification task means that a positive prediction represents ~15x risk enhancement. We go from a baseline toxicity rate of ~5% to nearly 75% toxicity among predicted-positive examples.

✓ Switching to GloVe Embeddings

I would also like to test the performance of the GloVe algorithm. GloVe is an unsupervised algorithm for obtaining vector representations of words.

From the model overview section: "GloVe is essentially a log-bilinear model with a weighted least-squares objective. The main intuition underlying the model is the simple observation that ratios of word-word co-occurrence probabilities have the potential for encoding some form of meaning."

Glove available here: <https://nlp.stanford.edu/projects/glove/>

```
MAX_NUM_WORDS = 15000
MAX_SEQUENCE_LENGTH = 100
EMBEDDING_DIM = 100      # Must match GloVe dimension (e.g. 50, 100, 200, 300)
```

```
# 1. Tokenization and padding
tokenizer = Tokenizer(num_words=MAX_NUM_WORDS)
tokenizer.fit_on_texts(df['clean_comment'])
sequences = tokenizer.texts_to_sequences(df['clean_comment'])
```


```
word_index = tokenizer.word_index
print(f'Found {len(word_index)} unique tokens.')
```

```
X = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)
y = df['toxic_ind'].values
```

 Found 52960 unique tokens.

```
# 2. Load GloVe embeddings
embeddings_index = {}
with open('/content/glove.6B.100d.txt', encoding='utf-8') as f:
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs
print(f'Found {len(embeddings_index)} word vectors in GloVe.')
```

```
# 3. Create embedding matrix
embedding_matrix = np.zeros((MAX_NUM_WORDS, EMBEDDING_DIM))
for word, i in word_index.items():
    if i < MAX_NUM_WORDS:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
```

 Found 400000 word vectors in GloVe.

```
# 4. Build RNN model with pre-trained embeddings
model = Sequential([
    Embedding(input_dim=MAX_NUM_WORDS,
              output_dim=EMBEDDING_DIM,
              embeddings_initializer=Constant(embedding_matrix),
              input_length=MAX_SEQUENCE_LENGTH,
              trainable=False),
    tf.keras.layers.Bidirectional(
        tf.keras.layers.LSTM(64, return_sequences=True, dropout=0.3, recurrent_dropout=0.2)),

    tf.keras.layers.Bidirectional(
        tf.keras.layers.LSTM(32, dropout=0.3, recurrent_dropout=0.2)),

    tf.keras.layers.Dense(64, activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l2(0.01)),

    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

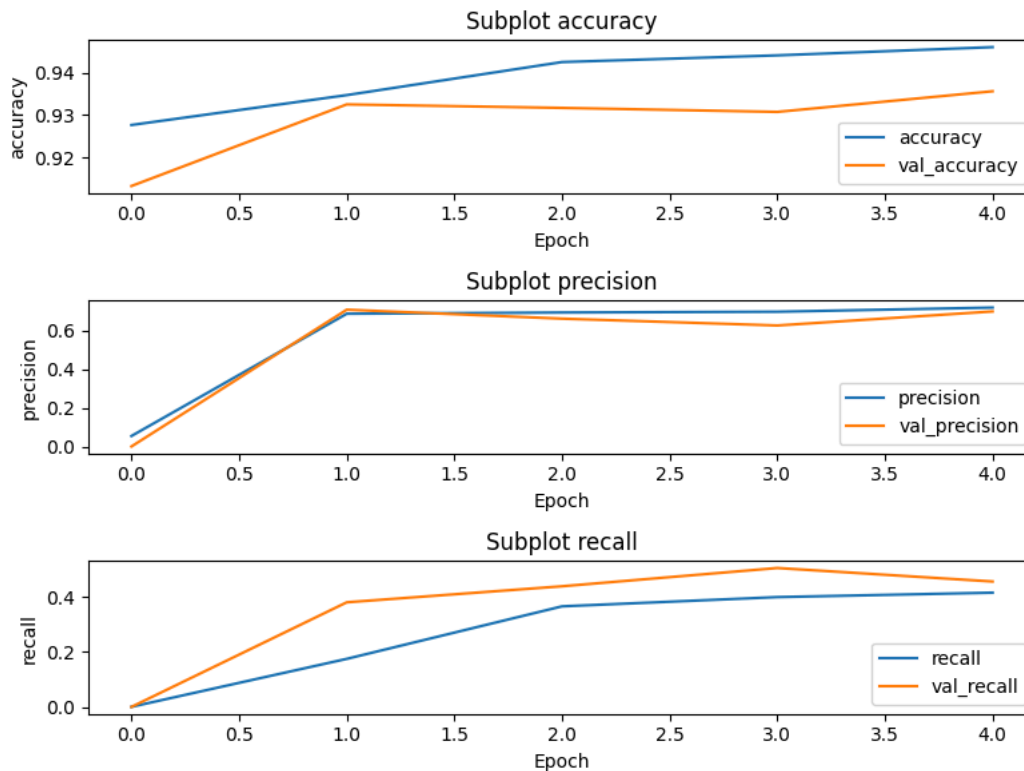
model.compile(
    loss=tf.keras.losses.BinaryFocalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy', 'precision', 'recall']
)

model.summary()
```

```
# 5. Train the model
history = model.fit(X, y, batch_size=32, epochs=5, validation_split=0.2)
```

 [Show hidden output](#)

```
plot_metrics(history)
```



The model performs similarly when using GloVe for embeddings rather than embedding from scratch. Overall, it seems to perform slightly worse. The GloVe model never has a validation loss lower than 0.05 while the other model has a best validation loss under 0.04. The precision of 0.73 and recall of 0.51 of the original model is also better than the 0.62 / 0.5 validation split of the GloVe model.

✓ GloVe Text Cleaning Experiment

As I was reading more about GloVe, I came across notes that text cleaning may not always be beneficial to task accuracy. Words with different stem endings can have importantly different meanings and, if possible, it would be great to include that context for the model prediction task. Below, I re-run the model with GloVe embeddings but start with the raw text instead of the clean text.

```
# Parameters
MAX_NUM_WORDS = 15000
MAX_SEQUENCE_LENGTH = 100
EMBEDDING_DIM = 100      # Must match GloVe dimension (e.g. 50, 100, 200, 300)
```

```
# 1. Tokenization and padding
tokenizer = Tokenizer(num_words=MAX_NUM_WORDS)
tokenizer.fit_on_texts(df['comment_text'])
sequences = tokenizer.texts_to_sequences(df['comment_text'])
```

```
word_index = tokenizer.word_index
print(f'Found {len(word_index)} unique tokens.')
```

```
X = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)
y = df['toxic_ind'].values
```

Found 157811 unique tokens.

```
embeddings_index = {}
with open('/content/glove.6B.100d.txt', encoding='utf-8') as f:
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs
print(f'Found {len(embeddings_index)} word vectors in GloVe.')
```

```
# 3. Create embedding matrix
embedding_matrix = np.zeros((MAX_NUM_WORDS, EMBEDDING_DIM))
for word, i in word_index.items():
```

```

if i < MAX_NUM_WORDS:
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector

```

Found 400000 word vectors in GloVe.

```

model = Sequential([
    Embedding(input_dim=MAX_NUM_WORDS,
              output_dim=EMBEDDING_DIM,
              embeddings_initializer=Constant(embedding_matrix),
              input_length=MAX_SEQUENCE_LENGTH,
              trainable=False),
    tf.keras.layers.Bidirectional(
        tf.keras.layers.LSTM(64, return_sequences=True, dropout=0.3, recurrent_dropout=0.2)),

    tf.keras.layers.Bidirectional(
        tf.keras.layers.LSTM(32, dropout=0.3, recurrent_dropout=0.2)),

    tf.keras.layers.Dense(64, activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l2(0.01)),

    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

```

```

model.compile(
    loss=tf.keras.losses.BinaryFocalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy', 'precision', 'recall']
)

```

model.summary()

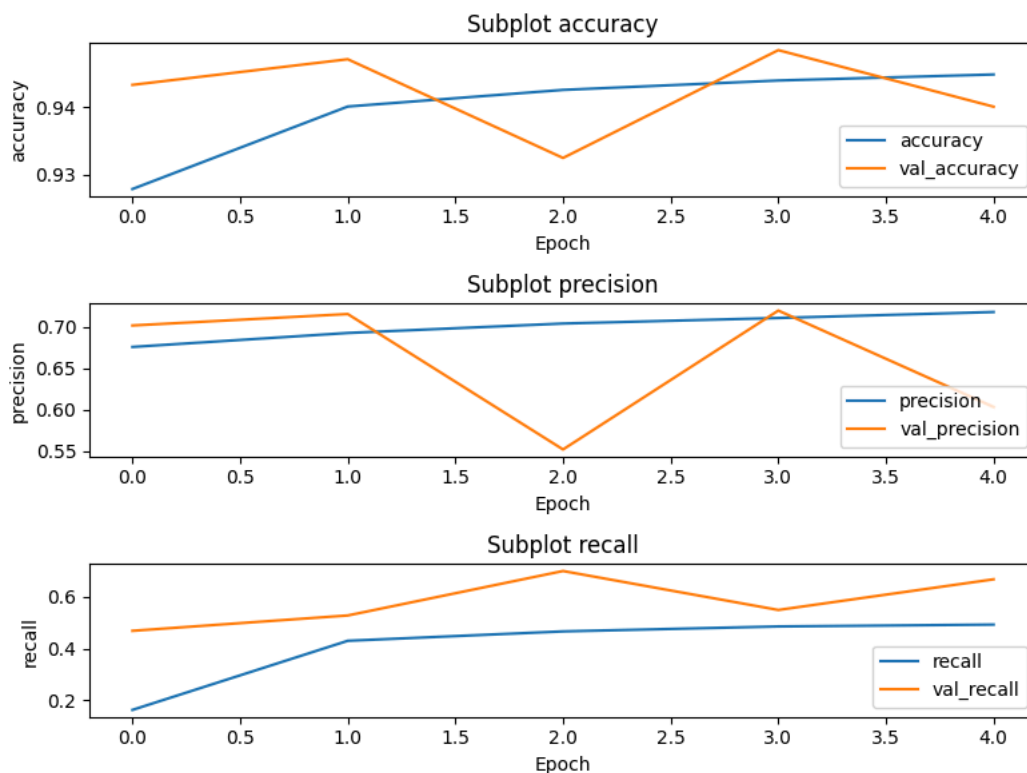
```

# 5. Train the model
history = model.fit(X, y, batch_size=32, epochs=5, validation_split=0.2)

```

Show hidden output

plot_metrics(history)



Using GloVe embeddings on the raw comment text performs better than using the embeddings on cleaned text. The performance of this second GloVe model achieved a best validation result of 72% precision, 55% recall. This is very similar to the best results achieved by the naive-embedding model built on cleaned text. This model clearly outperformed the GloVe model built on cleaned text.

On this uncleaned text, the vocab tripled to ~150,000 unique tokens compared to ~55,000 unique tokens on the cleaned data. This uncleaned version took ~5,000 per epoch compared to ~3,400 seconds per epoch of the cleaned version.

✓ GRU Model Experiment

Since the GloVe models don't offer a clear performance boost for this task, I decided to assess the performance of using Gated Recurrent Units rather than LSTM units. GRU is a simpler architecture and is appropriate when the task doesn't require memory for long sequences. I believe these comments are short enough that model performance won't be negatively affected by this switch. I tried to keep all other architecture components consistent with prior models.

```
model = tf.keras.Sequential([
    tf.keras.Input(shape=(1,), dtype=tf.string),
    encoder,
    tf.keras.layers.Embedding(
        len(encoder.get_vocabulary()), 64, mask_zero=True),
    tf.keras.layers.GRU(units=64, return_sequences=True, dropout=0.3, recurrent_dropout=0.2),
    tf.keras.layers.GRU(units=32, dropout=0.3, recurrent_dropout=0.2),
    tf.keras.layers.Dense(64, activation='relu',
        kernel_regularizer=tf.keras.regularizers.l2(0.01)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(1, activation = 'sigmoid')
])

model.summary()

# Compile the model
model.compile(
    loss=tf.keras.losses.BinaryFocalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy', 'precision', 'recall']
)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
text_vectorization (TextVectorization)	(None, None)	0
embedding (Embedding)	(None, None, 64)	960,000
gru (GRU)	(None, None, 64)	24,960
gru_1 (GRU)	(None, 32)	9,408
dense (Dense)	(None, 64)	2,112
dropout (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 1)	65

Total params: 996,545 (3.80 MB)
 Trainable params: 996,545 (3.80 MB)
 Non-trainable params: 0 (0.00 B)

```
history = model.fit(
    train_tf,
    epochs=4,
    validation_data=valid_tf,
)
```

Show hidden output

```
plot_metrics(history)
```

