Censoring toxic comments using fastai v2 with a multi-label text classifier

Making the internet a safe space one word at a time

Vinayak Nayak Sep 19, 2020 · 12 min read ★

The internet has become a basic necessity in recent times and a lot of things which happen physically in our world are on the verge of being digitised. Already a substantial proportion of the world population uses the internet for day to day chores, entertainment, academic research etc. It then is a big responsibility to keep the internet a safe space for everyone to come and interact because there are all sorts of people posting stuff on the internet without being conscious of its consequences.

This post goes through the process of making a text classifier which takes in a piece of text (phrase, sentence, paragraph any length text) and tells if the text falls under a range of different types of malignant prose. The topics covered in this post are

- Language Model in fastai v2
- Making inferences
- References

You can click on any of the above bullet points to navigate to the respective section.

Disclaimer: The dataset used here contains some text that may be considered profane, vulgar, or offensive.

Photo by Jon Tyson on Unsplash

Introduction

Natural Language Processing is a field that deals with understanding the interactions between computers and human language. Since a lot of things are going online or digital, and since these services are democratised to the whole world, the scale at which this data is generated is humongous. In these times where everyone on the planet is putting their opinions, thoughts, facts, essays, poems etc. online, monitoring and moderating these pieces of text is an inhuman task (even when we think of humans as a community and not an individual being).

Thanks to the advent of high capacity GPUs and TPUs and the recent advances in AI for text applications, we have come up with a lot of techniques to tackle this problem. Recurrent Neural Networks are the key element with the help of which these problems are addressed. fastai, a deep learning library built on top of PyTorch, developed by Jeremy Howard and Sylvain Gugger makes building applications for tasks like these very user-friendly and simple.

Let's the get started and learn how to do text classification using fastai.

Getting Data from Kaggle

Our model will be responsible for detecting different types of toxicity like threats, obscenity, insults, and identity-based hate. The dataset consists comments from Wikipedia's talk page edits. Without any further ado, let's get started with downloading this dataset.

```
import os

# Set your username and key from the kaggle API from your account
os.environ["KAGGLE_USERNAME"] = "###"
os.environ["KAGGLE_KEY"] = "###"
os.chdir("/content/drive/My Drive/Colab Notebooks")
os.mkdir("Toxic Comments Data")
os.chdir("./Toxic Comments Data")

# Download the data from kaggle to the directory you just created
!pip install kaggle
!kaggle competitions download -c jigsaw-toxic-comment-classification-challenge

# Unzip all the csv files and remove the zip files
!unzip train.csv.zip && rm train.csv.zip
!unzip test.csv.zip && rm test.csv.zip
!unzip test_labels.csv.zip && rm test_labels.csv.zip
!unzip sample_submission.csv.zip && rm sample_submission.csv.zip
```

kaggle_data_download.py hosted with ❤️ by GitHub view raw

You can either download the dataset from Kaggle manually or use the API provided by kaggle using the above commands.

To use the API, you'll have to create an account with Kaggle and generate the API key which allows you to use shell commands for downloading the dataset from kaggle and also making submissions for predictions from the working notebook or from the shell. Once you create a Kaggle account and create the API key, you will get a json file which contains both your username and key. Those need to be input in the above code as per your unique credentials.

Data Exploration

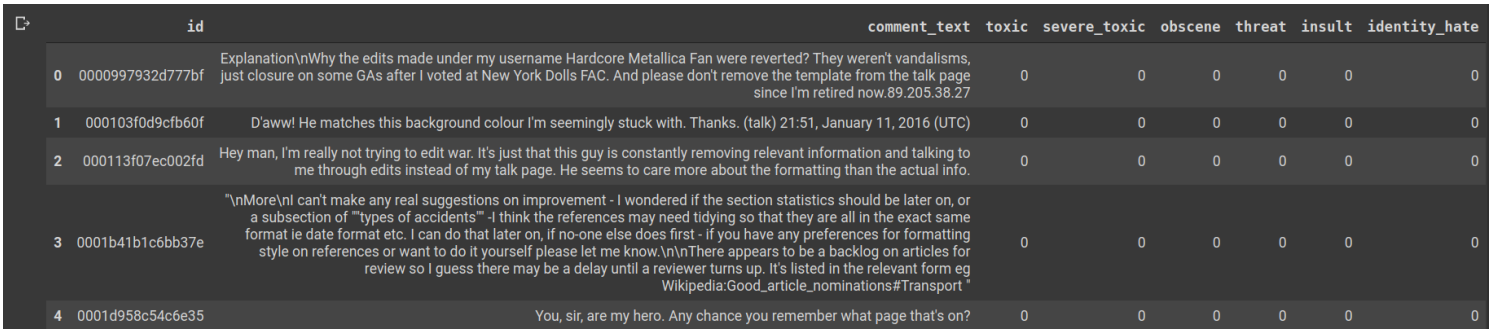
Let's read in both the train and test sets and get a hang of what data is contained in them.

```
import os

# Read in the train and test sets.
os.chdir("/content/drive/My Drive/Colab Notebooks/Toxic Comments Data")
train_df = pd.read_csv("train.csv")
test_df = pd.read_csv("test.csv")

# Look at a few entries from the dataset
train_df.head()
```

read_toxic_comments_data.py hosted with ❤️ by GitHub view raw



	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
0	0000997932d777bf	Explanation\nWhy the edits made under my username Hardcore Metallica Fan were reverted? They weren't vandalisms, just closure on some GAs after I voted at New York Dolls FAC. And please don't remove the template from the talk page since I'm retired now.89.205.38.27	0	0	0	0	0	0
1	000103f0d9cfb60f	D'aww! He matches this background colour I'm seemingly stuck with. Thanks. (talk) 21:51, January 11, 2016 (UTC)	0	0	0	0	0	0
2	000113f07ec002fd	Hey man, I'm really not trying to edit war. It's just that this guy is constantly removing relevant information and talking to me through edits instead of my talk page. He seems to care more about the formatting than the actual info.	0	0	0	0	0	0
3	0001b41b1c6bb37e	"\nMore\nI can't make any real suggestions on improvement - I wondered if the section statistics should be later on, or a subsection of ""types of accidents"" -I think the references may need tidying so that they are all in the exact same format ie date format etc. I can do that later on, if no-one else does first - if you have any preferences for formatting style on references or want to do it yourself please let me know.\n\nThere appears to be a backlog on articles for review so I guess there may be a delay until a reviewer turns up. It's listed in the relevant form eg Wikipedia:Good_article_nominations#Transport "	0	0	0	0	0	0
4	0001d958c54c6e35	You, sir, are my hero. Any chance you remember what page that's on?	0	0	0	0	0	0

Image by Vinayak

There are several fields in the dataframe.

- **id**: An identifier which is associated with ever comment text. Since this is picked up from Wikipedia's talk page, it could probably be the identification of a person who has commented, or an HTML DOM id of the text that they've posted.
- **comment_text**: The text of the comment which the user has posted.

represented by zeros else they're represented by a 1.

These elements are independent in the sense, they're not mutually exclusive for eg. A comment can be *both toxic and insulting,* or it's not necessary that if a comment is toxic it couldn't be obscene and so on.

```
# Look at the distribution of comments
f = lambda x: train_df[x].sum()

comment_types = train_df.columns[2:]
comment_counts = [f(x) for x in comment_types]

fig, ax = plt.subplots(1, 1, figsize = (10, 6))
plt.bar(comment_types, comment_counts)
ax.set_title(f"Comment distribution of {len(train_df)} wikipedia comments.")
plt.tight_layout();
```

visualize_toxic_counts.py hosted with ❤️ by GitHub view raw

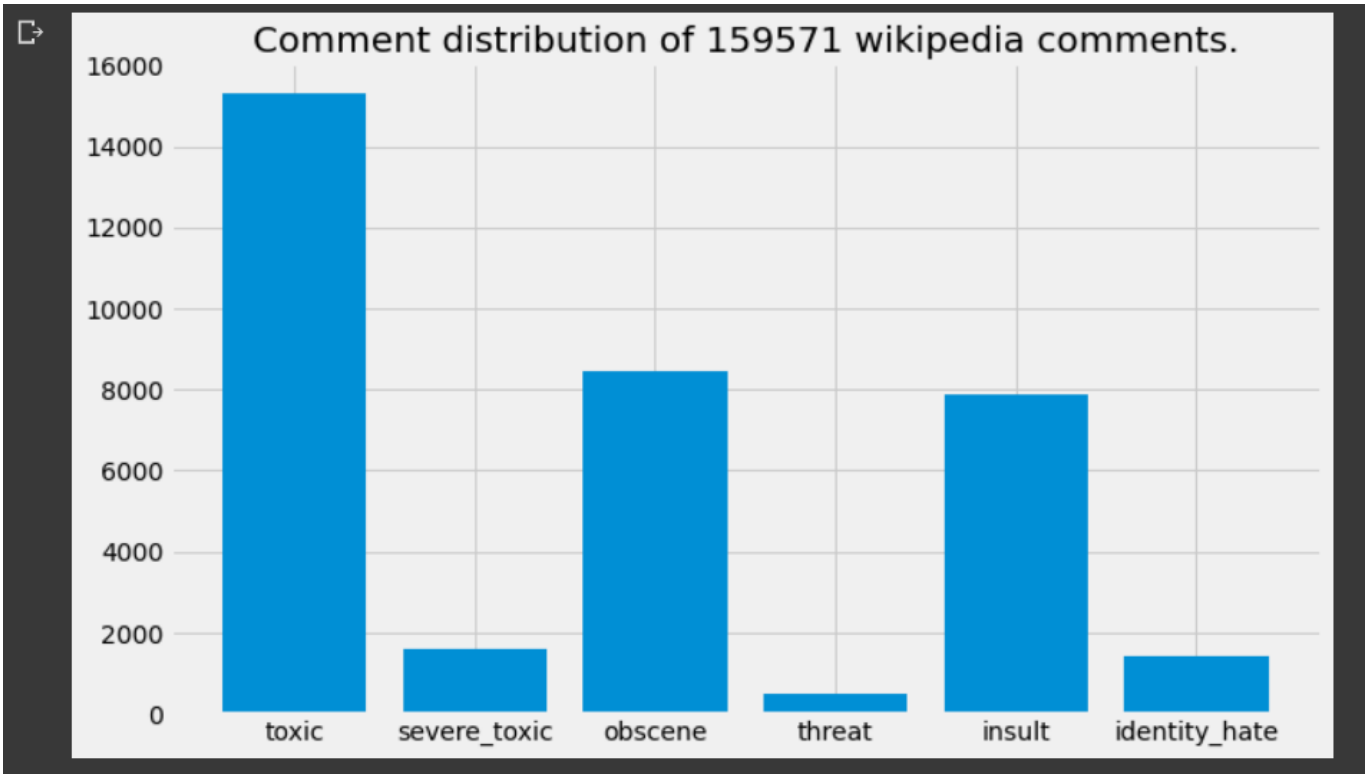


Image by Vinayak

objectionable categories (except toxic which is just a few more). This is good to know but it would be for the best if there were still fewer texts of this kind.

Also, the text was annotated by humans to have these labels. This task of annotation is a huge task and a lot of human interpretation and bias will have come along with these annotations. It's something that needs to be remembered and we'll talk about this in closing thoughts.

Approach toward multilabel text classification

Text or sentences are sequences of individual units — words, sub-words or characters (depends on the language you speak). Any Machine Learning algorithm is not capable of handling anything other than numbers. So, we will first have to represent the data in terms of numbers.

In any text related problems, first we create a vocabulary of words which basically is the total corpus of words which we will consider; any other word will be tagged with a special tag called unknown and put in that bucket. This process is called ***tokenization.***

Next, we map every word to a numerical token and create a dictionary of words that stores this mapping. So every prose/comment/text is now converted into a list of numbers. This process is called ***numericalization.***

Most certainly the comments will not be of equal length, because people are not restricted to comment in exactly a fixed number of words. But when creating batches of text to feed to our network, we need them all to be of the same length. Therefore we pad the sentences with a special token or truncate the sentence if it's too big to constrict to a fixed length. This process is called ***padding.***

While doing all the above, there are some other operations like lowercasing all the text, dealing with punctuation as separate tokens, understanding capitalization in spite of

- **xxpad**: For padding, this is the standard token that's used.
- **xxunk**: When an oov (out of vocabulary) word is encountered, this token is used to replace that word.
- **xxbos**: At the start of every sentence, this is a token which indicates the start/beginning of a sequence.
- **xxmaj**: If a word is capitalised or title cased, this token is prefixed to capture that information.
- **xxrep**: If a word is repeated, then in the tokenized representation, we will have that word followed by xxrep token followed by number of repetitions.

There's some more semantic information handled with more such tokens but all of this makes sure to capture precious information about the text and the meaning behind it.

Once this preprocessing is done, we can then right of the bat build an LSTM model to classify the texts into the respective labels. Words are represented as n-dimensional vectors which are colloquially called encoding/embedding. There's a construct for Embedding in PyTorch which helps lookup the vector representation for a word given it's numerical token and that's followed by other RNN layers and fully connected layers to build an architecture which can take sequence as an input and return a bunch of probabilities as the output. These vector embeddings could be randomly initialized or borrowed from commonly available GLoVE or Word2Vec embeddings which have been trained on a large corpus of text so that they have a good semantic word understanding about context in that particular language in a generic sense.

However, there's a trick which could improve the results if we perform it before building a classifier. That's what we'll look at next.

Language Model in fastai v2

In a nutshell, what they say is if you have a set of word embeddings which were trained on a huge corpus, they have a very generic understanding of the words that they learned from that corpus. However, when we talk about classification of hate speech and obnoxious comments and toxic stuff, there's a specific negative vibe associated with these sentences and that semantic context is not yet present in our embeddings. Also, many words/terms specific to our application (it may be medicine or law or toxic speech) may not be encountered often in that huge corpus from which we obtained the word embeddings. Those should be there included and represented well in the embeddings that our classifier is going to use.

So, before building a classifier we'll finetune a Language Model which has been trained on wikipedia text corpus. We will bind the train and test dataset comments together and feed them to the language model. This is because we're not doing classification but simply guessing the next word of a sequence given the current sequence; it's called a self supervised task. With the embeddings learned this way, we'll be able to build a better classifier because it has an idea of the concepts specific to our corpus.

Let's see how we can instantiate and fine-tune a language model in fastai v2.

```
[ ]  # For building a language model, we can basically use both train and test data so that the embeddings are
     # created and refined for many observed words.
     lm_df = train_df[["id", "comment_text"]].append(test_df).reset_index(drop = True)
```

```
[ ]  lm_df
```

	id	comment_text
0	0000997932d777bf	Explanation\nWhy the edits made under my username Hardcore Metallica Fan were reverted? They weren't vandalisms, just closure on some GAs after I voted at New York Dolls FAC. And please don't remove the template from the talk page since I'm retired now.89.205.38.27
1	000103f0d9cfb60f	D'aww! He matches this background colour I'm seemingly stuck with. Thanks. (talk) 21:51, January 11, 2016 (UTC)
2	000113f07ec002fd	Hey man, I'm really not trying to edit war. It's just that this guy is constantly removing relevant information and talking to me through edits instead of my talk page. He seems to care more about the formatting than the actual info.
3	0001b41b1c6bb37e	"\nMore\nI can't make any real suggestions on improvement - I wondered if the section statistics should be later on, or a subsection of ""types of accidents"" -I think the references may need tidying so that they are all in the exact same format ie date format etc. I can do that later on, if no-one else does first - if you have any preferences for formatting style on references or want to do it yourself please let me know.\n\nThere appears to be a backlog on articles for review so I guess there may be a delay until a reviewer turns up. It's listed in the relevant form eg Wikipedia:Good_article_nominations#Transport "
4	0001d958c54c6e35	You, sir, are my hero. Any chance you remember what page that's on?
...	...	...
312730	fffcd0960ee309b5	. \n i totally agree, this stuff is nothing but too-long-crap
312731	fffd7a9a6eb32c16	== Throw from out field to home plate. == \n\n Does it get there faster by throwing to cut off man or direct from out fielder? \n Were the out fielders in the Mickey mantle era have better arms? \n Rich
312732	fffda9e8d6fafa9e	" \n\n == Okinotorishima categories == \n\n I see your changes and agree this is ""more correct."" I had gotten confused, but then found this: \n :... while acknowledging Japan's territorial rights to Okinotorishima itself ... \n However, is there a category for \n :... did not acknowledge Japan's claim to an exclusive economic zone (EEZ) stemming from Okinotorishima. \n That is, is there a category for ""disputed EEZ""s? "

Image by Vinayak

tokenize these texts, do all the numericalization, padding and preprocessing before feeding it to the language model.

```
# Tokenize the dataframe created above to have all the descriptions tokenized properly a
# For creating a language model
dls_lm = TextDataLoaders.from_df(lm_df,
                                 # Specify the column that contains the comments
                                 text_col='comment_text',
                                 # Specify how much data is within train and validation
                                 valid_pct = .2,
                                 # Mention explicitly that this dataloader is meant for
                                 is_lm = True,
                                 # Pick a sequence length i.e. how many words to feed th
                                 seq_len = 72,
                                 # Specify the batch size for the dataloader
                                 bs = 64)
```

dataloader_toxic.py hosted with ❤️ by GitHub view raw

1

That's how simple it is in fastai, you just have to wrap all the arguments in a factory method and instantiate the TextDataLoaders class with it. This would have otherwise taken at least a hundred lines of codes with proper commenting and stuff but thanks to fastai, it's short and sweet. We can have a look at a couple of entries from a batch.

	text	text_
0	xxbos " \n\n_ = = sockpuppetry case== \n_ { | align=""left " " \n_ || \n_ | } \n_ xxmaj you have been accused of sockpuppetry . xxmaj please refer to for evidence . xxmaj please make sure you make yourself familiar with notes for the suspect before editing the evidence page . " xxbos : technically both , as xxmaj xxunk contains only xxmaj xxunk . xxbos 3rd xxmaj term \n\n	" \n\n_ = = sockpuppetry case== \n_ { | align=""left " " \n_ || \n_ | } \n_ xxmaj you have been accused of sockpuppetry . xxmaj please refer to for evidence . xxmaj please make sure you make yourself familiar with notes for the suspect before editing the evidence page . " xxbos : technically both , as xxmaj xxunk contains only xxmaj xxunk . xxbos 3rd xxmaj term \n\n xxmaj
1	and a detailed one below ? xxbos " \n\n_ = = xxmaj we need pie = = \n\n_ { | style=""background - color : # fdffe7 ; border : 1px solid # fceb92 ; " " \n_ |style=""vertical - align : middle ; padding : 5px ; " " | \n_ |style=""vertical - align : middle ; padding : 3px ; " " | xxmaj life throws us all curve balls ,	a detailed one below ? xxbos " \n\n_ = = xxmaj we need pie = = \n\n_ { | style=""background - color : # fdffe7 ; border : 1px solid # fceb92 ; " " \n_ |style=""vertical - align : middle ; padding : 5px ; " " | \n_ |style=""vertical - align : middle ; padding : 3px ; " " | xxmaj life throws us all curve balls , sorry

Image by Vinayak

As we can see the output is just offsetting the given sequence by one word which is in alignment with what we want, i.e. given a sequence, predict next word of a sequence. Once we have this dataloader, we can create a language model learner which can tune the encodings as per our corpus instead of the previous corpus of text.

```
                                AWD_LSTM,
                                # Dropout percentage for regularization
                                drop_mult = .3,
                                # Metrics to understand the performance
                                metrics = [accuracy, Perplexity()]).to_fp16()

# To go a little easy on the GPUs, we will save the weights of model with 16-point float
# Precision instead of the regular 32-point floating precision
```

language_model.py hosted with ❤️ by GitHub view raw

```
[ ]  # Fit one cycle with
     learn.fit_one_cycle(1, 2e-2)
```

epoch	train_loss	valid_loss	accuracy	perplexity	time
0	4.048460	3.809381	0.328461	45.122501	28:22

```
[ ]  # Save the encoder part for the sake of building a classification model later on
     learn.save_encoder("finetuned_language_model_encoder")
```

```
[ ]  # Unfreeze the model and then train the learner again for 4 more epochs
     learn.unfreeze()
     learn.fit_one_cycle(4, 1e-3)
```

epoch	train_loss	valid_loss	accuracy	perplexity	time
0	3.550328	3.587820	0.357523	36.155155	30:55
1	3.526780	3.476392	0.373162	32.342804	30:49
2	3.427127	3.419326	0.381164	30.548828	30:44
3	3.288719	3.410187	0.382664	30.270905	30:48

```
[ ]  # Save the encoder. It'll be used in training the classifier portion
     learn.save_encoder("finetuned_language_model_encoder")
```

Image by Vinayak

After we have the language model learner, we can fit the learner over several epochs and save the encodings using the save_encoder method. We can see that the language model can on an average predict with a 38% accuracy what the next word would be given the current sequence of words which is pretty decent for this dataset.

Classification Model in fastai v2

Before we move to creating a classification model, there's some bit of preprocessing that we need to perform in order to build a proper dataloader. At the time of writing this post, there's an issue with the DataBlocks API for text which avoids it from inferring all the dependent variables properly, hence we have to resort to this method.

Basically, we will have to create another column in our dataframe which indicates the presence or absence of individual label using a fixed delimiter. So, if a comment is obscene and toxic, our new column will show obscene;toxic where delimiter is ";". Also for the rows which don't have any objectionable text, we will call them sober for now for the sake of giving a label (without any label, fastai won't create the dataloader).

```
# Make a list of columns that would serve as your labels for this task
label_cols = list(train_df.columns[2:])

# Create a column of texts which has a list of all the categories.
# When all the entries are zeros, let's call the txt sober
def get_labels(row):
    indcs = np.where(row == 1)[0]
    if len(indcs) == 0:
        return "sober"
    return ";".join([label_cols[x] for x in indcs])

# Get the labels all in a nicely formatted style
labels = train_df[label_cols].apply(lambda row: get_labels(row), axis = 1)

# Add the labels object to our dataframe
train_df["Labels"] = labels
```

preprocessing_labels.py hosted with ❤ by GitHub — view raw

```
train_df.tail(3)
```

	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate	Labels
159568	ffee36eab5c267c9	Spitzer \n\nUmm, theres no actual article for prostitution ring. - Crunch Captain.	0	0	0	0	0	0	sober

Image by Vinayak

So we can see that there's a column Labels added which contains a ";" delimited labels field where all our labels are denoted instead of the one-hot encoded format in which they're provided.

```
# Create a Dataloader to feed to the model
dls_blk = DataBlock(blocks = (TextBlock.from_df(text_cols = "comment_text", seq_len = 12
                              MultiCategoryBlock),
                    get_x = ColReader(cols = "text"),
                    get_y = ColReader(cols = "Labels", label_delim = ";"),
                    splitter = TrainTestSplitter(test_size = 0.2, random_state = 21))

dls_clf = dls_blk.dataloaders(train_df,
                              bs = 64,
                              seed = 20)
```

toxic_classifier.py hosted with ❤️ by GitHub view raw

Now, we create the dataloaders using the datablocks API using "comment_text" field for x and "Labels" field for y respectively. If we would have mentioned the names of 6 columns as a list in the get_y field, it always picks up only two fields; due to this incorrect inference on the dataloader's part, we have to go through the process of creating a separate label column for getting the dependent variable i.e. y's values. Next, once we have the dataloader, we can build a classifier model using an LSTM architecture. Also we need to load the language model encodings/embeddings to the classifier once have instantiated it.

```
# Create a classifier learner using the dataloader defined above
learn_clf = text_classifier_learner(dls_clf,
                                    # Specify a model architecture for the learner
                                    AWD_LSTM,
                                    # Specify the % in dropout layer for regularization
                                    drop_mult=0.5,
                                    # Specify a metric to evaluate performance while tra
                                    metrics = accuracy_multi).to_fp16()

```

```
[ ]  # Fit a small cycle to understand the performance
     learn_clf.fit_one_cycle(1, 2e-2)
```

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.190189	0.117679	0.960435	10:56

```
[ ]  # Unfreeze the last two layers and train for an epoch while monitoring the performance
     learn_clf.freeze_to(-2)
     learn_clf.fit_one_cycle(1, slice(1e-2/(2.6**4),1e-2))
```

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.095186	0.082094	0.972064	13:45

```
[ ]  # Unfreeze the last three layers and train for an epoch while monitoring the performance
     learn_clf.freeze_to(-3)
     learn_clf.fit_one_cycle(1, slice(5e-3/(2.6**4),5e-3))
```

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.078971	0.067934	0.976303	22:17

```
[ ]  # Unfreeze all the layers and train for a couple of epochs and monitor the performance
     learn_clf.unfreeze()
     learn_clf.fit_one_cycle(1, slice(1e-3/(2.6**4),1e-3))
```

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.060560	0.065158	0.976997	29:23

```
[ ]  # Export the model and save it for inference
     learn_clf.export("toxic_comment_classifier.pkl")
```

+ Code + Text

Image by Vinayak

Then we can start training the classifier. Initially, we will keep most of the network except the final FC layer frozen. This means that the back-propagation weight updates will only happen in the penultimate layer. Gradually we will unfreeze the previous layers until eventually we unfreeze the whole network. We do this because if we start with an unfrozen network, it will become difficult for the model to converge quickly to the optimal solution.

It can be seen that the accuracy has reached a pretty solid 98% buy the end of the training. Since both train and valid loss both are decreasing, we can ideally train for

Making inferences

Now that we have a trained model and we've stored it as a pkl, we can use it for making predictions on previously unseen i.e. test data.

We shall first load the model that we just created and trained on the GPU. (Since we have hundreds of thousands of comment texts, CPU inference will take a lot of time). Next we will tokenize the test_df and then pass it through the same transforms that were used for train and validation data to create a dataloader of test comments for inference.

Next we will use the get_preds method for inference and remember to pass the reorder method to False otherwise there's a random shuffling of the texts that happens which will lead to incorrect order of predictions at the end.

Finally, we shall format these predictions in the sample_submissions.csv style. So, after predictions, we get a set of 7 values one for each class and the probability of "sober" class is not needed since it was introduced by us as a placeholder. We remove that and get all the ids in proper order. This is how the final submission looks like.

Image by Vinayak

Finally we can submit these predictions using the kaggle API itself. No need to manually go to kaggle and submit the csv file. It could be done simply by this shell command.

```
!kaggle competitions submit -c jigsaw-toxic-comment-classification-
challenge -f submissions_toxic.csv -m "First submission for toxic
comments classification"
```

You can change the submissions file name and message as per your own convenience. The final submission score I got is as shown below

Image by Vinayak

The top score on the leaderboard is around .9885 so our score is somewhat good with such few lines of code and little to no preprocessing. We could've removed stopwords, cleaned html tags, tackled punctuation, tuned language model even more or used GloVE or Word2Vec embeddings and went for a complex model like Transformer instead of a simple LSTM. Many people have approached this differently and used some of these techniques to get to such high scores. However, with little effort and using the already implemented fastai library we could get a decent enough score right in our first attempt.

On a closing thought, it is worth mentioning that this dataset as annotated by humans may have been mislabelled or there could have been subjective differences between people which is also fair because it's a very manual and monotonous job. We could aid that process by building a model, then using it to annotate and have humans supervise the annotations to make the process simpler or crowd-source this work to multiple volunteers to get a large corpus of labelled data in a small amount of time. In any case, NLP has become highly instrumental in tackling many language problems in the real world and hope after reading this post, you feel confident to start your journey in the world of text with fastai!

2. How to use Kaggle API for downloading data
3. Github repo with all code for this post
4. Text classification notebook using fastai

Naturallanguageprocessing Data Science Python Programming Text Classification Fastai