Appendix

1 More Background on GPT-2

Here, we review some key concepts used in GPT-2.

1.1 The Self-Attention Mechanism

The Self-Attention mechanism was introduced to transform text features [Vaswani, 2017]. The mechanism takes as an input a sequence of tokens represented by vectors $(a_1, a_2, ..., a_n)$ and using matrix multiplication with three different trained matrices, it computes, for each vector a_i, three resulting vectors: a query vector q_i, a key vector k_i, and a value vector v_i. Then, to transform the token i, the mechanism computes $q_i K^T$ Which are the scores for all the other tokens when transforming the token i. Then the mechanism uses these scores to compute a weighted average of all the value vectors of these tokens. If we group all the vectors for all the tokens into the matrices (Q, K, V), then we can summarise the attention mechanism by:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{|k_i|}}\right)V$$

Where the scaling factor $\frac{1}{\sqrt{|k_i|}}$ is used to avoid making the dot-product too large for long vectors.

1.2 The Decoder-Only Transformer

In a Decoder Transformer, when transforming a term to the next layer, we only compute the weighted average from the previous terms, and we don't allow it to access values from future words. This is done in the $softmax$ step, where we set the scores for future words to be 0 (or $-\infty$ if using a logit $softmax$). This is done to observe the auto-regressive property of the model.

1.3 Auto-Regression Generation: Top-K Sampling

To generate text using GPT-2, we start with some initial tokens. Then, we run the tokens through the model to calculate the probabilities for the next word. Then, we choose the next word and add it to our input. We repeat this step until we generate an ending token, which means that this document is done.

Once we calculate the probabilities, we don't simply pick the word with the most probability. Rather, a better method for generation is to pick the top K words, and then randomly sample one of these words. This turns out to perform better than other methods, like greedy sampling or beam search [Platen, 2020], and it allows us to generate a lot of different examples instead of a deterministic generator.

2 Full Results

Here are the full results of all the models, on all training sets, on all test sets:

Model	Training Dataset	Accuracy on Train	F1 on Train	Accuracy on Full Test 80:20	F1 on Full Test 95:5	Accuracy on Imbalanced Test 80:20	F1 on Imbalanced Test 95:5
TF-IDF + log	A. Original 80:20	0.8929	0.7016	0.8799	0.662	0.9409	0.4891
TF-IDF + log	B. Imbalanced 95:5	0.9575	0.3038	0.821	0.246	0.9545	0.2273
TF-IDF + log	C. GPT2 80:20	0.9446	0.8502	0.8366	0.4209	0.9452	0.3515
TF-IDF + log	D. GPT2 90:10	0.9489	0.6842	0.8326	0.3585	0.952	0.3203
TF-IDF + log	E. EDA 80:20	0.9682	0.9203	0.8189	0.2715	0.9475	0.2264
Tiny Bert	A. Original 80:20	0.9	NA	0.8886	0.7002	0.9399	0.5162
Tiny Bert	B. Imbalanced 95:5	0.9644	NA	0.8443	0.4198	0.9573	0.3955
Tiny Bert	C. GPT2 80:20	0.9587	NA	0.8403	0.4289	0.949	0.3684
Tiny Bert	D. GPT2 90:10	NA	NA	0.8327	0.3521	0.9538	0.332
Tiny Bert	E. EDA 80:20	0.999	NA	0.7931	0.0038	0.949	0.0046
Base Bert	A. Original 80:20	0.981	NA	0.9193	0.7865	0.9592	0.6433

Table 1: Full results of all models

Note that there are some "NA"s in the table. This is because for Bert models, the batches in the training dataset are shuffled every time we iterate over them. This improves batch training but makes it hard to measure performance after the training is done. We aren't interested in results on training data anyway so we didn't try to overcome this problem.