COMP9417 21T2 Assignment

S. Chen, J. Garratt, A. Kataria., K. Mashimo

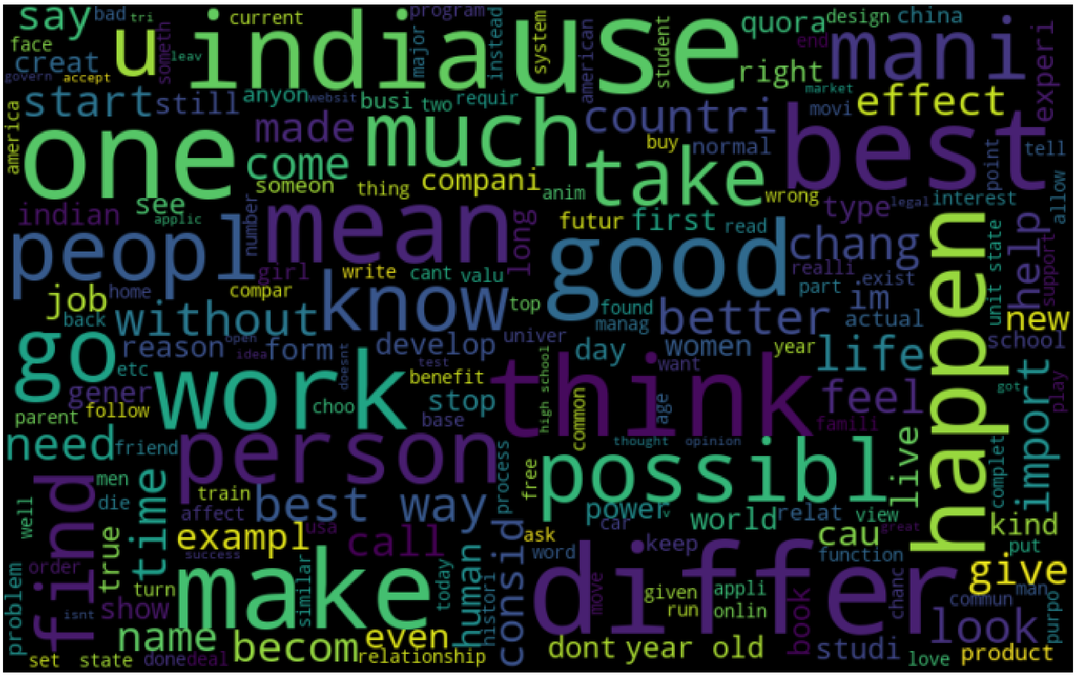
Introduction

The Internet has expanded and created many platforms where society can interact with each other in many different ways. It has transformed the way people communicate with each other making it our preferred medium of everyday communication. However, one of the biggest issues in today's online society is that many platforms suffer from abusive and toxic actions from their user base. On this account, the objective of this report will be to build classifiers to help detect divisive content on a social question-and-answer website.

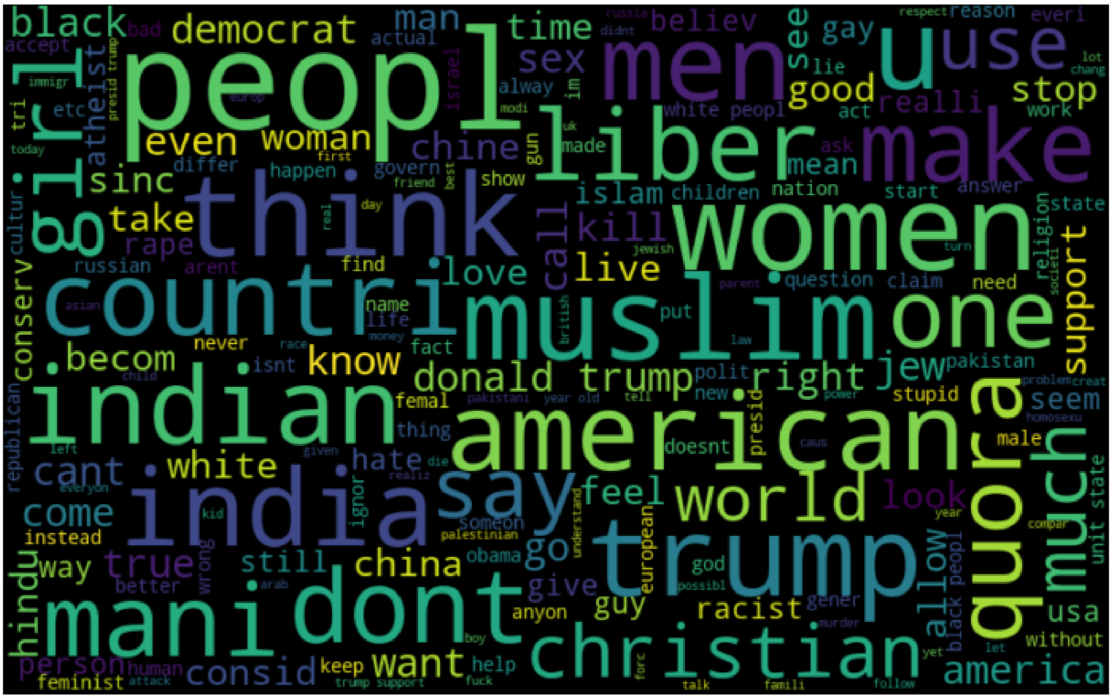
In this group project, we have chosen the Kaggle competition, [Quora Insincere Questions Classification](). Quora is a community-based question-and-answer website where users aim to find legitimate solutions to their questions. The goal of this competition is to create a model that can detect insincere questions posted by users. The competition has defined insincere as text that is of non-neutral tone, disparaging or inflammatory, falsified message, sexual content (incest, bestiality, pedophilia) for shock value, and users not seeking genuine answers. The data was provided via the competition and has 3 data fields which are the qid (unique question identifier), question_text (Quora question text) and target (a question labelled "insincere" has a value of 1, otherwise 0).

To have a better visualisation of the data, we have generated a word map that displays word frequencies present in the data for both insincere and sincere questions:

Sincere



Insincere



The central objective of this report is to determine if rule-based trees provide more stable and accurate results than models that can absorb and contextualise semantic connections of the dataset provided. We have hypothesised that both tree-based models and neural networks should produce reasonable results but due to the nature of neural networks and their ability to detect deep patterns as well as training on larger datasets, neural network type models may have an edge on tree-based models. Thus, the contributions of this report will compare the performances of logical tree-based models with more complex models such as neural networks and conclude which model produces the most optimal results.

Competition Evaluation

The Kaggle competition's leaderboard listings are based on F-score: in particular, F_1-score. F-score is an accuracy measure defined as the harmonic mean of precision and recall - that is:

$$
\begin{aligned}
\mathrm{F}_1 &= \frac{2}{\frac{1}{p}+\frac{1}{r}} \\
&= \frac{2 \cdot pr}{p+r} ,
\end{aligned}
$$

.

where precision is the ratio of positive values correctly identified to all positives classified and recall is the ratio of true positives identified and all positives in the testing set:

$$\text{precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}},$$

and hence,

$$\text{F}_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

Why F-score?

The harmonic mean is a useful operation as it penalises extreme values. A heuristic reason for which the harmonic mean is preferable over the arithmetic mean is the given example:

Take a naive binary classification model which always predicts A, and a dataset consisting of exactly one element belonging to class A and a large number of elements in class B. Then, this classifier’s precision is asymptotically 0 and recall is 1. This model is completely inaccurate as its accuracy is almost zero; the arithmetic mean would score this model at 50% while the harmonic mean would score 0%. This is therefore a useful metric for evaluation, particularly for imbalanced datasets where seemingly high accuracies can be achieved from naive classification.

While there exists the downside of the F-score not including True Negative classifications, overall it provides a strong evaluation metric. Other measures such as Cohen’s kappa or informedness could be used here.[1]

[1] Powers, D. M. (2015). What the F-measure doesn't measure: Features, Flaws, Fallacies and Fixes. *arXiv preprint arXiv:1503.06410*.

Pre-model Steps

Prior to the development and training of the ideated models, it was clear that some level of common preprocessing was required for many model types.

The implemented techniques are traditional statistical natural language processing (NLP) steps for many model types. The following steps will use the same example, “What was machine learning’s biggest achievement in 1958?”:

1. Original:

   What was machine learning’s biggest achievement in 1958?
2. Remove special characters:

   What was machine learnings biggest achievement in 1958
3. Change characters to lowercase:

   What was machine learnings biggest achievement in 1958
4. Mask numbers (eg. 2005 -> XXXX):

   what was machine learnings biggest achievement in XXXX
5. Convert sentences to lists of tokens:

   ‘what’, ‘was’, ‘machine’, ‘learnings’, ‘biggest’, ‘achievement’, ‘in’, ‘XXXX’
6. Remove stopwords:

   ‘machine’, ‘learnings’, ‘biggest’, ‘achievement’, ‘XXXX’
7. Stem/lemmatise each token:

   ‘machin’, ‘learn’, ‘biggest’, ‘achiev’, ‘XXXX’
8. Recombine tokens in each row of the data:

   machin learn biggest achiev XXXX

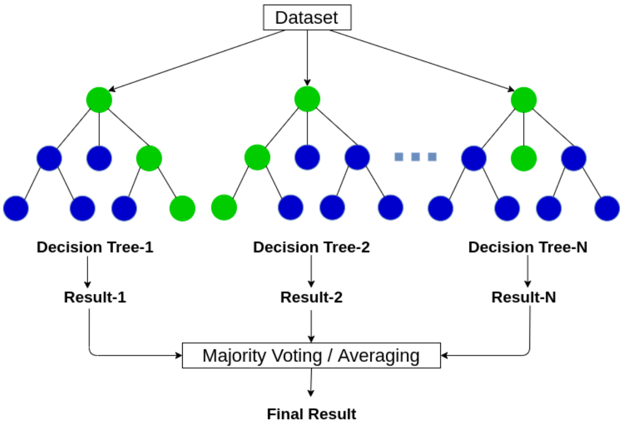
These steps have the primary focus of removing small yet (from a statistical linguistics perspective) meaningless variations in grammatical inflection, possession, pluralisation, comparison, superlative inflection, figures and punctuation. This means that when statistical models are applied to/trained on the data points, the preprocessing maximises the potential of each row of the data.

The code for the preprocessing can be found here. The original and processed data can be found here.

Implementation

Random Forest

Random Forest Tree is a supervised learning model. It is an ensemble method that trains several decision trees in parallel which is performed with the bagging method. This model utilises bagging and feature randomness to create an uncorrelated forest of decision trees. In general, the Random Forest model is a learning method that generates multiple decision trees and combines them to improve performance significantly via variance reduction. Random Forest Trees have three key hyperparameters which are the number of trees to generate, the number of features to be sampled when training and the size of the nodes in the trees. This model also creates additional randomness by searching for the best feature in a random subset of features rather than searching for the most important feature while splitting a node.



[2]As we can see from the diagram above, the Random Forest model randomly creates multiple decision trees and at each node of the tree, the tree will be training on a random subset of the features to produce the final predictions of the data.[3]

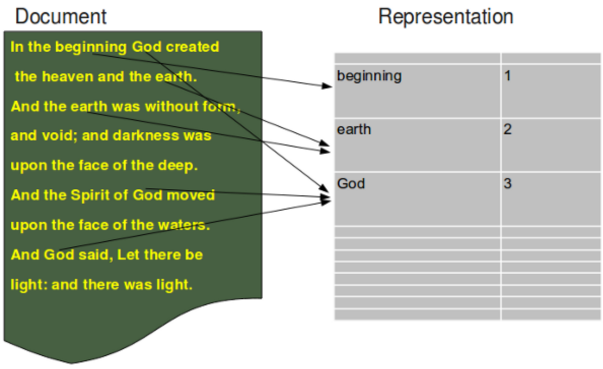
The reason why we chose this as one of our models for the Quora Insincere Questions Classification Kaggle Competition, is that it provides key benefits such as:

- Maintains high accuracy rates while also being flexible as the model utilises feature bagging, which creates very accurate estimations of missing values.
- Minimises the risk of overfitting, it is known that decision trees are prone to overfit as they tightly fit train the data in one tree, whereas a random forest utilises the number of decision trees in the forest and averages the uncorrelated trees, reducing the prediction error and variance.[4]

Using the processed trained data, we have split the data into 80:20 training and testing sets. We then use the bag-of-words approach to classify the text data. This approach would vectorise the text data by generating a vocabulary list in which we calculate the frequency of

[2] Niklas Donges (June, 2019) A complete guide to the random forest algorithm
[3] Abishek Sharma (May, 2020) Decision Tree vs. Random Forest – Which Algorithm Should you Use?
[4] IBM Cloud Education (Dec, 2020) Random Forest

each word and store it in the vector. For instance, if the word “Frisbee” has 7 occurrences in the data, it will be listed as 7 in the vector.


Document
In the beginning God created
the heaven and the earth.
And the earth was without form,
and void; and darkness was
upon the face of the deep.
And the Spirit of God moved
upon the face of the waters.
And God said, Let there be
light: and there was light.
Representation
beginning
1
earth
2
God
3

First, we split the data into a normal 80:20 split and begin to vectorise the data to generate a bag-of-words dictionary by using $CountVectorizer$ from SKLearn which provides a simple procedure to tokenize a group of text data and generate a vocabulary of known words. It also has the ability to encode new documents using that vocabulary. This will allow us to vectorise the processed data into a bag-of-words vocabulary.[5]

Hyperparameter Tuning

To get the best estimator for the Random Forest Model, we have implemented RandomizedSearchCV. This function constructs a search space that is bounded by the domain of the hyperparameter inputs and then begins to randomly sample the points in the domain. This will allow us to choose a set of parameters that deliver the best estimators in terms of the scoring which is set to ‘f1’.[6]

Using the following parameter ranges for the Random Forest Model:

- **n_estimators = [int(x) for x in np.linspace(start = 50, stop = 1000, num = 50)]**

This parameter will set the number of trees in the forest to be generated. Setting a higher value in this setting will not cause overfitting, however the more trees you add the longer it will take to train.

- **max_depth = [int(x) for x in np.linspace(start = 6, stop = 14, num = 2)]**

This parameter will set the maximum depth of the tree

- **min_samples_split = [2, 5, 10]**

Min_samples_split determines the minimum number of samples that is required to split an internal node. This parameter will allow the forest to become more constrained since it has to consider an increase in samples at each node

- **min_samples_leaf = [1, 2, 4]**

In a decision tree, a leaf is the last node of that tree, a smaller leaf decision tree will generally make the model more prone to creating more variance in the predictions as it captures more noise from the training data.Thus, setting a minimum leaf size will allow the

[5] Charles Rajendran (Apr, 2018) Text classification using the Bag Of Words Approach with NLTK and Scikit Learn

[6] Will Koehrsen (Jan, 2018) Hyperparameter Tuning the Random Forest in Python

model to have the most optimum results since it won't get influenced heavily by the noise in the data.

Results

Here we can see a table of the results that RandomizedSearchCV produced:

N_estimators	Min_samples_split	Min_samples_leaf	Max_depth	Mean_test_score	Model Number
340	2	4	14	0.447958	1
50	5	1	14	0.389613	2
786	5	1	6	0.450051	3
844	2	2	6	0.443427	4
864	2	4	6	0.447459	5
767	2	1	14	0.454405	6
495	2	4	6	0.440471	7
243	5	1	6	0.429827	8
651	10	4	6	0.447623	9
263	10	2	14	0.445724	10

RandomizedSearchCV outputted model #6 as the best model with the best estimators in terms of the scoring 'f1':

- Max_depth = 14
- Min_samples_split = 2
- n_estimators=767
- Min_samples_leaf = 1

Running the model to predict the training set and testing set produced the following results:

	Accuracy	F1-Score
Training Set	88.71%	0.4554
Testing Set	88.45%	0.4456

As we can examine the results, the training F1-Score is very similar to the testing accuracy F1-Score. This indicates that the model isn't overfitting the data. In similar experiments, there were times where the training F1-score was a lot higher than the testing F1-score which depicted that overfitting was occurring in the model. After further research, setting the max_depth too high and not including min_sample_split as well as min_sample_leaf, caused the model to overfit the data.

To show comparison, this is the results of one of the previous Random Forest Models:

	Accuracy	F1-Score
Training Accuracy	95.14%	0.6834
Testing Accuracy	91.29%	0.5002

This model had set max_depth to 30, which led to this overfitting issue as seen in the difference of the training and testing F1-Score. Due to this overfitting, the F1-score is completely unreliable making the model unreliable as well.

Thus in the final model, in an 80:20 split of the training set. As seen above, the results have not been very optimal as the Random Forest Model has limitations in creating a contextual understanding of the data producing inaccurate predictions affected by high variance.

XGBoost

Extreme Gradient Boosting (XGBoosting) is a method that builds upon existing boosting methods and is an additive tree ensemble model. The basis of XGBoost, additive tree modelling, is an iterative and sequential process of adding decision trees one step at a time where each iteration reduces the loss function value of the model. XGBoost has two primary goals: execution speed and model performance. It achieves the former by storing sorted data in column blocks in a compressed format and parallelizing tree construction through capitalisation of all CPU cores during training. To achieve model performance, it features three methods of overfitting avoidance: regularized learning, shrinkage and column sampling.[7] Besides the regularization term, shrinkage scales newly added weights by a factor of η after each iteration of tree boosting subsequently reducing the influence of each tree, leaving space for future trees to improve the model [8].

Although XGBoost primarily targets structured and tabular data, we can apply the XGBoost technique to the Quora corpus. Implementation of the work is done using python 3.7 version with anaconda version 3. For experimenting on Quora corpus that consists of *insincere* and *sincere* questions, figure 1 shows sentiment distribution in the training data set.

The implementation consists of two main parts, data preparation for input and subsequent evaluation. To train the classifier we first split the Quora questions. Here we will experiment with a balanced train test split, where the number of insincere questions matches the number of sincere questions, and an 80:20 train test split on the entire data set, where we blindly partition 80% of the data for training and 20% for testing. After the split we vectorize the words to later add into a hstack.

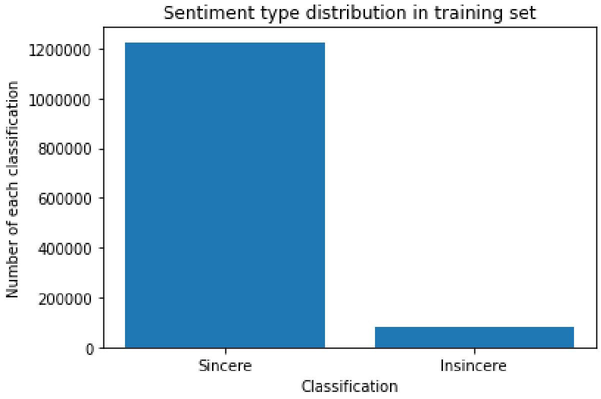


Figure 1. Sentiment type distribution in training data set

TF-IDF Vectorization

TF-IDF vectorization is an acronym that stands for Term Frequency - Inverse Document Frequency which are the components of the scores assigned to each word or character. It is a method of frequency scoring that attempts to highlight words that are more interesting. Here we use the TF-IDF vectorizer in combination with feature engineering to create a

[7] Chen & Guestrin, *XGBoost* 2016
[8] ibid., p. 2

combined input that we will feed into the XGBoost model, allowing us to make use of both the qualitative and quantitative aspects of the corpus.

We then begin feature engineering by considering important features like the length of each question, the number of question marks, the number of exclamation marks and the number of unique words in the question among others. These values are then appended to the respective row from which it was calculated. We then convert all of this training data into a SciPy sparse matrix. Using sparse matrices will allow us to save a significant amount of memory and time.

Hyperparameter Tuning

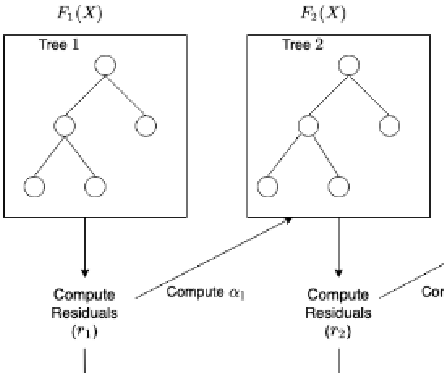
XGBoost has a considerable number of parameters. Hence, we tuned the parameters min_child_weight, gamma, max_depth and the learning rate using GridSearchCV, and use them as the model parameters on which we train our model.

- **Min_child_weight = 6**

In a regression, the loss of each point in a node is $\frac{1}{2}(y_i - \hat{y}_i)^2$. Taking the second derivative with respect to y_i we get 1. From this, we can conclude that if we sum the second derivative overall points in the node, we get the number of points in the node. This is reflective of the splitting process. Once the sum of all weights in a node is less than the min_child_weight, we give up further partitioning.

- **Gamma = 0.1**

Gamma is another one of XGBoosts' regularization parameters that works by regularising the α_i across tree parameters. In particular, by observing the typical loss changes we can adjust gamma such that we instruct our tree to add nodes if and only if the associated gain of adding the node is greater than the complexity cost of adding the leaf [9]. Since gamma is a regularization parameter applied across the tree, it is clear that regularising with deeper trees will create an unnecessary burden on our learning procedure.



- **Learning Rate = 0.1**

Learning rate is another one of XGBoost's regularization parameters. It is a technique to slow down the learning of the gradient boosting model through the application of a weighting factor or "shrinkage factor." This reduces the effect of each correction made by the addition of adding new trees when we run the model, allowing us to avoid overfitting.

- **Max_depth = 4**

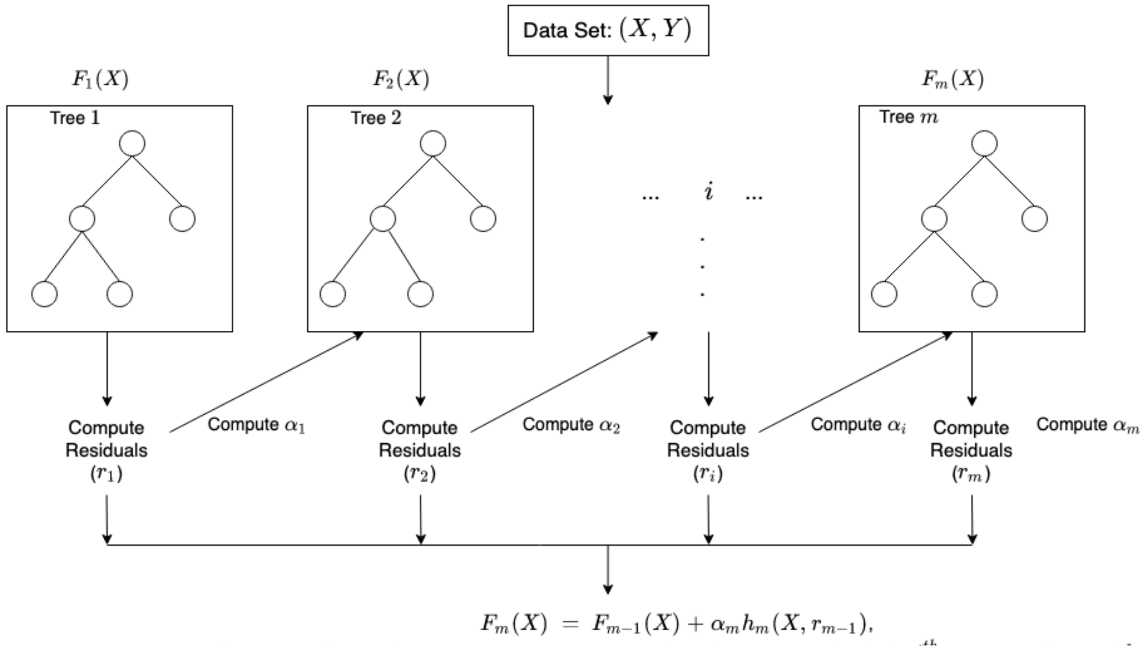
Another one of XGBoosts' regularization parameters that assists with overfitting avoidance. The deeper the tree, the more complex and likely it is that we overfit. This makes intuitive sense.

[9] Chen & Guestrin, *XGBoost* 2016

XGBoost Algorithm

At each boosting step we have a current estimate of the binary result at a certain leaf, is it insincere or sincere. We want to improve this estimate by replacing each estimate with weak learners that correspond to the scaling factor we mentioned previously η.

At a high level, the XGBoost algorithm looks like this:


Data Set: (X, Y)
$F_1(X)$
Tree 1
$F_2(X)$
Tree 2
... i ...
$F_m(X)$
Tree m
Compute Residuals (r_1)
Compute α_1
Compute Residuals (r_2)
Compute α_2
Compute Residuals (r_i)
Compute α_i
Compute Residuals (r_m)
Compute α_m
$F_m(X) = F_{m-1}(X) + \alpha_m h_m(X, r_{m-1}),$

[10]

Here α_i and r_i are the regularization parameters and residuals computed for the corresponding i^{th} tree. To compute α_i we use the we use the residuals r_i with the predictive function h_i and execute the following loss function:

$$arg \min_{\alpha} = \sum_{i=1}^{m} L(Y_i, F_{i-1}(X_i) + \alpha h_i(X_i, r_{i-1}))$$

where $L(Y, F(x))$ is a differentiable loss function [11].

Results

The table below shows data trained on an **even split**, where there were an equal number of test and training samples.

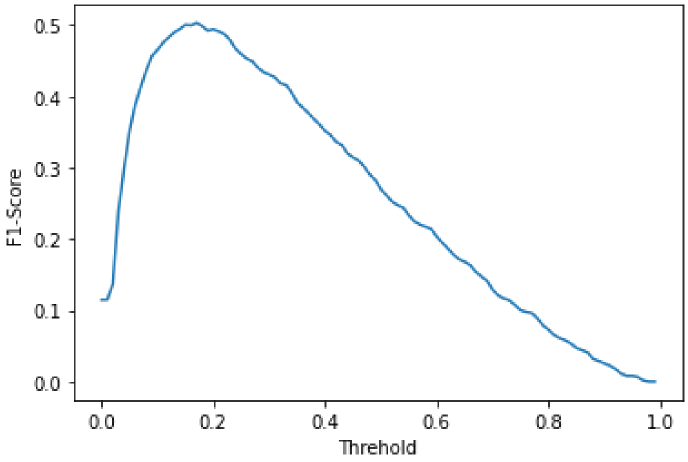
Prediction on validation set

Cross Validation (Folds)	Train Loss (5dp)	Validation Loss (5dp)	Validation Accuracy (5dp)	Validation F-score (5dp)	Threshold (2dp)
1	0.97196	0.89000	0.81799	0.48983	0.2

[10] Hudgeon & Nichol, *How XGBoost Works* 2020
[11] ibid., p. 1

2	0.95279	0.89151	0.82537	0.49419	0.19
3	0.95766	0.89348	0.82801	0.48219	0.17
4	0.94474	0.88787	0.81882	0.47829	0.15
Average	0.95679	0.89072	0.82255	0.48613	0.18

To find the optimal threshold for f1-score we iterate over probabilities in increments of 1%. By varying a numeric threshold value we are able to numerically identify an optimal threshold that maximises f1-score.



Note: Thresholding diagram from from Kfold 2 of 4 for an even trained data set

Compared to an 80:20 split, our f-score is considerably lower as we will see later. This makes intuitive sense, as we have less data to train on with an even split. From the corpus, we can see that 80800 questions are insincere compared to the 1306122 total number of questions.

Prediction on test set

Accuracy (5dp)	F-score (5dp)	Threshold
0.82079	0.48755	0.17

The table below shows the 4 different stratified k folds of the model, all trained on an **80% training set and 20% validation set**

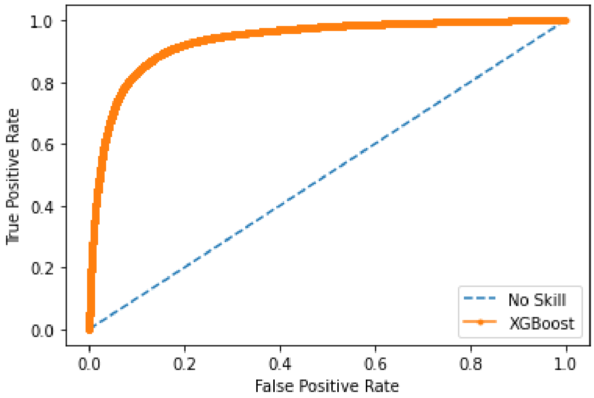
Prediction on Validation Set

Cross Validation	Train Loss (5dp)	Validation Loss (5dp)	Accuracy (5dp)	F-score (5dp)	Threshold
1	0.96851	0.93744	0.87202	0.57774	0.21
2	0.96370	0.93394	0.86618	0.57324	0.21

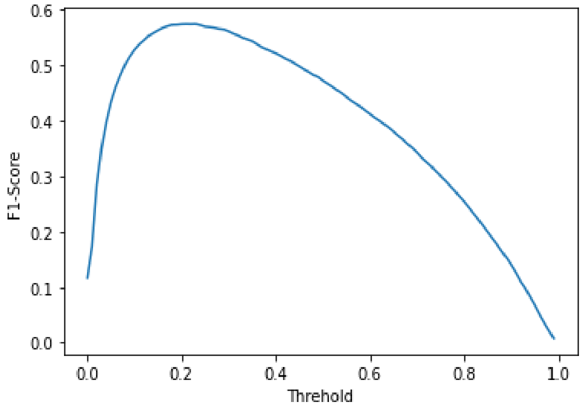
3	0.96747	0.93586	0.86865	0.57418	0.22
4	0.96375	0.93477	0.86860	0.57079	0.21
Average	0.96586	0.93550	0.86886	0.57399	0.21

Prediction on the test set

Accuracy (5dp)	F-score (5dp)	Threshold
0.87018	0.57426	0.23



Note: We see that as the number of iterations increases that the true positive rate begins to plateau. After 2 iterations of no improvement in train_loss, we choose the best model. Here we are plotting the precision recall curve for our unbalanced data.



Note: Thresholding the predictions of our unbalanced data set

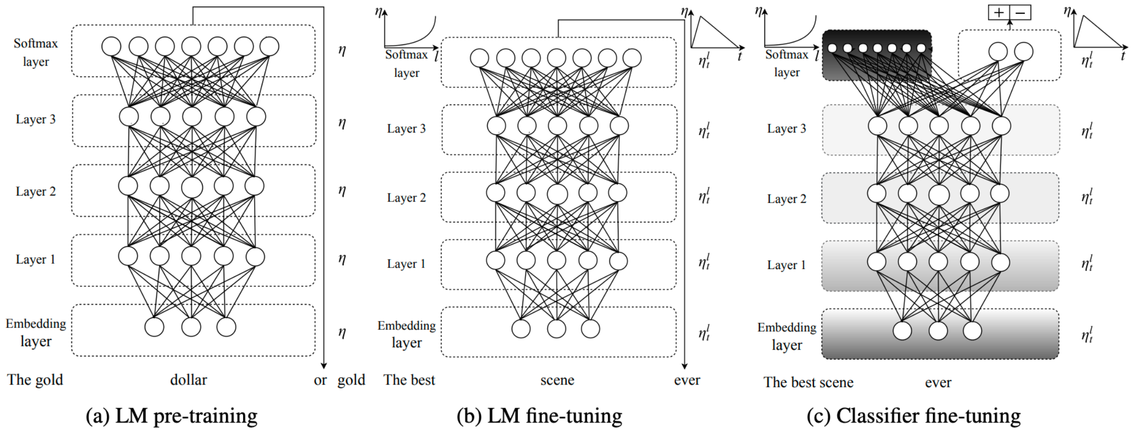
In total the second 80:20 model took 4 hrs and 45 minutes to run with Kaggle GPU boosting, holding all other things constant aside from sample size. In comparison to the 34 minute run time of the re-scaled balanced training set, it is evident that one of XGBoost's limitations is its lack of performance when scaled upwards.

ULMFiT

Universal Language Model Fine-tuning for Text Classification (ULMFiT) proposes a new method for generalized inductive transfer learning for NLP problems. The method consists of pretraining a language model (LM) on a large general corpus and then fine-tuning on target specific data (in this case the Quora corpus). The vocabulary of the resulting language model is then used to train a classifier [12]. In this case, the role of the LM is to capture the idiosyncrasies of the Quora corpus which, more formally, has been demonstrated to induce a hypothesis space that can be applied to other NLP problems[13].

In our approach, we use the fastai library which leverages automatic preprocessing (using the Spacy library tokenizer) and high level access to deep learning algorithms including the ULMFiT algorithm.

In detail, our implementation of the algorithm can be separated into three stages.



Note: The three stages of ULMFit. a) LM is pre-trained on a general domain corpus to identify general features. b) Resultant LM is fine-tuned on target specific corpus using discriminative fine-tuning ("Discr") and slanted triangular learning rates (STLR). c) Classifier is trained on target specific corpus using gradual unfreezing, "Discr", and STLR (shaded: unfreezing stages; black:frozen). Adapted from Howard, J., & Ruder, S. (2018). Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. arXiv preprint arXiv:1801.06146.

General-domain LM

As introduced by the fastai package, we use the WikiText103 model with a vocabulary size of 267,735 [14] as first presented by Merity et al in 2016.

Target task LM fine-tuning

Since fastai is a highly optimized library including automatic cuda management and parallel processing, we decided to use a larger amount of data, specifically, a 80-20 train-test split, with a further sub-set of 20% of the training corpus withheld for validation. Likewise, with

[12] Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. arXiv preprint arXiv:1801.06146.
[13] Baxter, J. (2000). A model of inductive bias learning. *Journal of artificial intelligence research*, *12*, 149-198.
[14] Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2016). Pointer sentinel mixture models. arXiv preprint arXiv:1609.07843.

experimentation on Kaggle kernel's, we found that a batch size of 32 allowed for sufficient memory whilst balancing training speed.

Our model utilises ASGD Weight-Dropped LSTM (AWD LSTM) as recommended by the fastai package with no other sophistication other than tuned dropout hyperparameters. In alignment with the fastai package, we chose to use the default hyper-parameters for AWD LSTM [15]. Likewise, we chose Adam for optimisation, and cross-entropy loss for our objective.

Target task classifier fine-tuning

The classifier is the only model built from scratch and incorporates an augmented version of the pre-trained LM with two additional linear blocks, both with batch normalisation and dropout. Likewise, for the intermediate layer we use ReLU activations and a softmax activation for the final layer to output the class probabilities over our target space.

In regards to implementation, we used the same train-test split ratio as the LM with the same seed. However, from our own research we discovered that we could significantly increase the batch size to 512 to halve our training time (from 30 mins to 15 mins per epoch). Similarly, we chose an augmented version of cross entropy loss, namely, cross entropy loss flat with a focal parameter γin order to help mitigate class imbalance [16].

In addition to the ULMFit architecture, introduced in their original paper [17] Howard & Ruder incorporate 3 novel techniques which are defined below.

Discriminative fine-tuning ("Discr")

Instead of using the same learning rate for every layer, "Discr" proposes a new learning rate to tune each layer.

Thus, given the traditional stochastic gradient descent (SGD) algorithm, with parameter θat step t, learning rate $\eta$and the gradient $\nabla_{\theta}J(\theta)$:

$$\theta_{t} = \theta_{t-1} - \eta\nabla_{\theta}J(\theta)$$

Define θ^{l}, the parameter of the model at layer l. Hence, SGD with discriminative fine-tuning:

$$\theta^{l}_{t} = \theta^{l}_{t-1} - \eta^{l}\nabla_{\theta^{l}}J(\theta)$$

This method is used particularly in the target task classifier, which by following empirical evidence from Howard & Ruder, we fine tune only the last layer and use learning rate $\eta^{l-1} = \eta^{l}/2.6$ as the learning rate for lower layers.

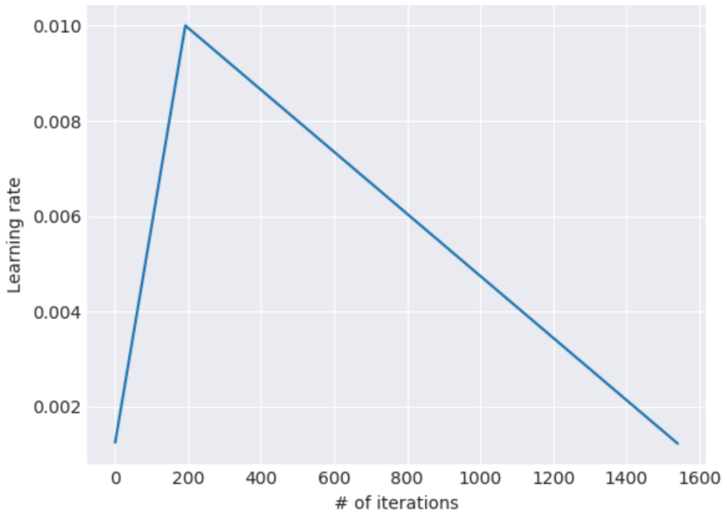
[15] https://docs.fast.ai/text.models.awdlstm.html#AWD_LSTM

[16] Lin, T. Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. In Proceedings of the IEEE international conference on computer vision (pp. 2980-2988).

[17] Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. arXiv preprint arXiv:1801.06146.

Slanted triangular learning rates (STLR)

In order to quickly converge to a suitable parameter space and then continue to refine, fastai first linearly increases, then linearly decays the learning rate as seen in the figure below.

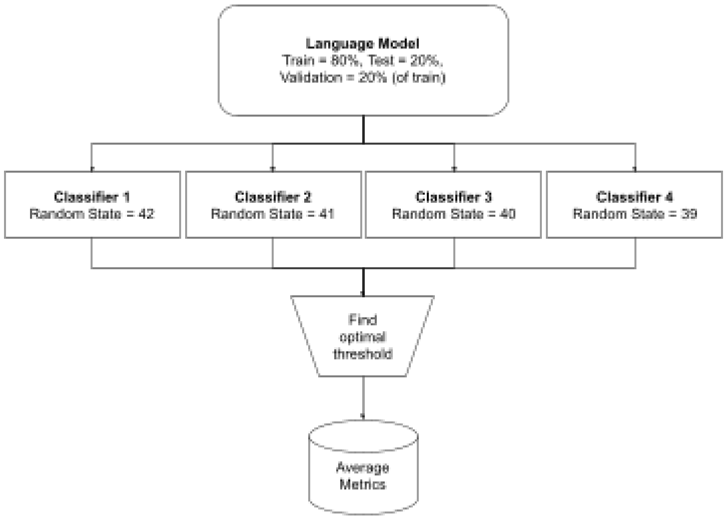


Note: STLR used for ULMFit. Adapted from Howard, J., & Ruder, S. (2018). Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. arXiv preprint arXiv:1801.06146.

Gradual Unfreezing

As opposed to fine-tuning all models in unison, we implement a new method called "gradual unfreezing" that minimises the risk of catastrophic forgetting [18]. Effectively, the method first unfreezes and fine-tunes the last layer. The method is completed iteratively, unfreezing the next lower layer and fine-tuning all unfrozen layers (i.e last and second-last layer). This is repeated until all layers have been fine-tuned.

Results

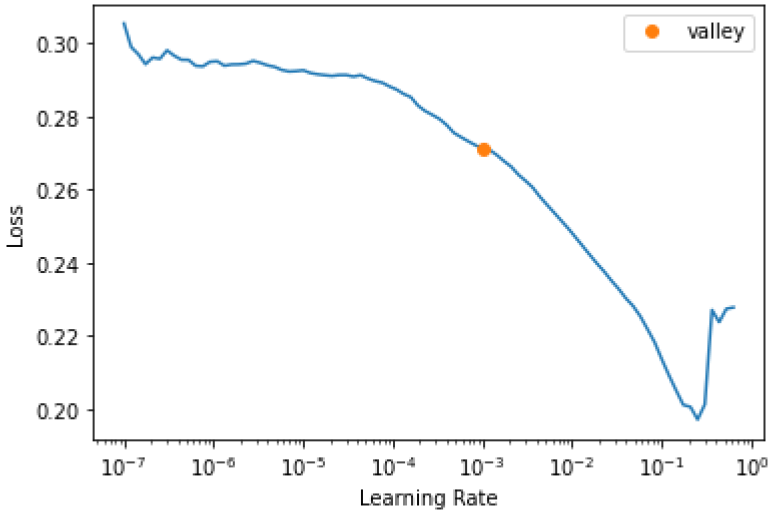


Note: The above illustrates a high-level overview of our implementation of ULMFit.

[18] Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. arXiv preprint arXiv:1801.06146.

Hyper-parameter Tuning

Besides the additional techniques previously described, the learning rate (LR) for each model was discovered by scheduling different LRs through a single epoch. We then approximated the optimal LR by visually determining a valley where loss begins to decrease. In other cases where a valley was not clear we used a LR around 1e-2 and continued to decrease the LR upon no decrease in loss after two epochs.

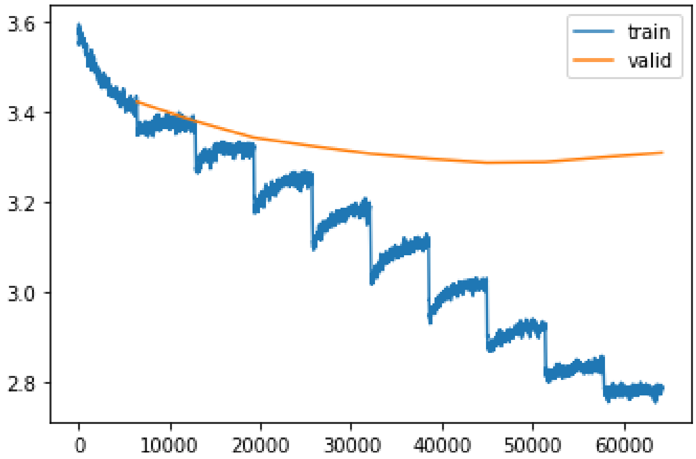


Note: The learning rate finder technique is implemented in fastai and allows for visual confirmation of an approximately optimal learning rate.

Language Model

To maximise the amount of time spent training and experimenting on the classifier, we decided to minimise the amount of time experimenting on the language model, instead opting to trust empirical evidence from the fastai community. The purpose of the language model is to generate a sufficiently large vocabulary of words to then transfer to the classifier, hence, leveraging fastai's sophisticated pre-processing capabilities, we can be assured that with minimal experimentation we can generate a sufficient vocabulary. The results below show our results.

Epoch (out of 10)	**Train set proportion**	**Train Loss**	**Validation Loss**	**Accuracy**	**Perplexity**	**Total train time**
6	80% (30% dev set)	3.019791	3.288078	0.412260	26.791323	4hrs 30 mins



Note: x-axis=number of iterations, y-axis = cross-entropy loss. The number of epochs has a diminishing effect on loss, so by saving the model wherever loss decreases, we choose the best model based on min valid loss at epoch 6.

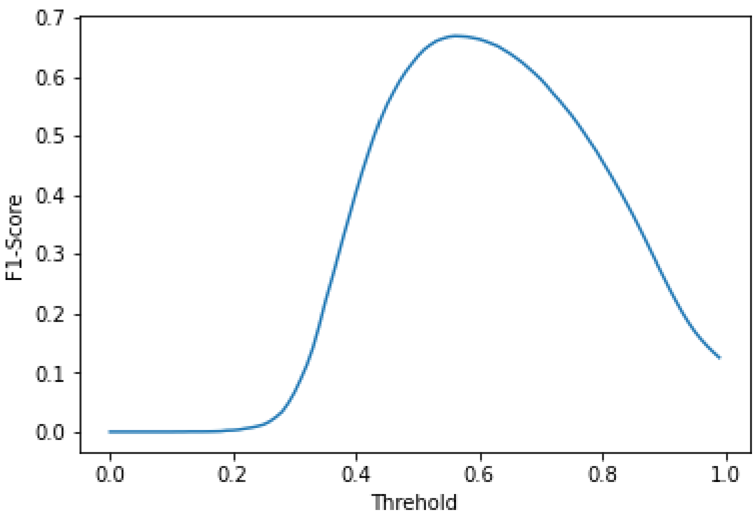
Classifier Training

Initial experiments (using cross-entropy loss) demonstrated that increasing the training set size significantly increased the f-score, and did not affect the time per epoch if we scaled the batch size. Some handpicked examples that demonstrate this phenomenon can be seen in the table below.

Train set proportion	**Batch Size**	**Train Loss**	**Valid Loss**	**Validation Set F1-Score**	**Time per epoch (avg)**
40% (20% dev set)	64	0.141759	0.120475	0.483601	20 mins
80% (20% dev set)	512	0.095271	0.098894	0.632827	19 mins

Furthermore, additional research demonstrated that incorporating a focal parameter into the loss function did not significantly boost the f-score, however, we continued to use it for our final 4 models to help mitigate class imbalance across multiple different re-samples to minimise the variance between out-of-sample f1-scores. In essence, focal loss adds a factor $(1-p_t)^{\gamma}$to cross-entropy loss whereγadjusts the rate at which easy examples are down-weighted and $p_t$is the probability of predicting the ground truth class[19]. To further maximise f1-score we used a technique to threshold probabilities, or in other terms, by varying the probability threshold, we could numerically find the optimal threshold to maximise f1-score.

[19] Lin, T. Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. In Proceedings of the IEEE international conference on computer vision (pp. 2980-2988).



Note: Threshold for mapping probability of a given question as sincere or insincere (0 or 1 respectively).

The table below shows the final results of the 4 re-sampled models and their properties (all models were trained on 80% of the data set with a further 20% for validation and batch size=512).

Seed	Train Loss	Valid Loss	Validation Set F1-Score	Out-of-sample f1-score	Optimal Threshold	Out-of-sample f1-score with optimal threshold
42	0.024911	0.026418	0.638958	0.640916	0.570000	0.678270
41	0.023964	0.026666	0.642887	0.647754	0.563000	0.673915
40	0.025293	0.026864	0.636740	0.634809	0.560000	0.668700
39	0.025103	0.026131	0.647679	0.643553	0.560000	0.675673
Average	**0.024818**	**0.026520**	**0.641566**	**0.641758**	**0.563250**	**0.674140**

Transformers

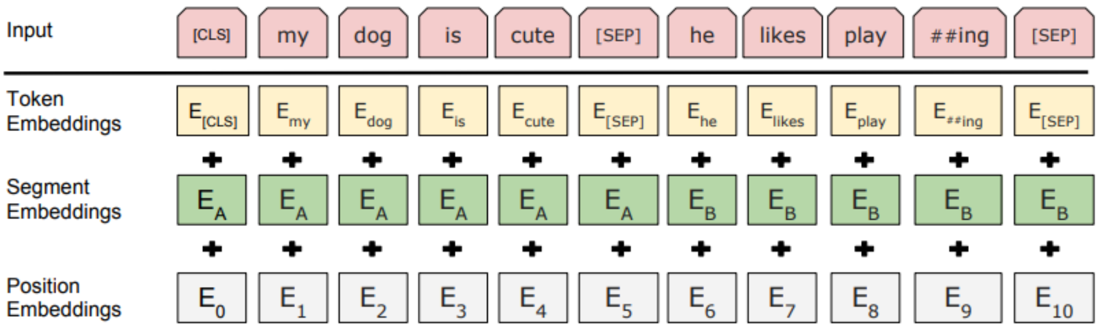
Motivation

This approach makes heavy use of transformer architectures for deep learning. Originally implemented for machine translation tasks[20], these models are extremely useful for sequential text data. However, there are three key differences from a comparable method - recurrent neural networks, specifically long-short term memory models:

Attention mechanisms. As initially introduced in the paper *Attention Is All You Need*[21], the Transformer is "based solely on attention mechanisms", a novel architecture that no longer requires recurrent or convolutional architectures. Transformer models elevate important parts of the input to devote more computing power, while paying less attention to inconsequential tokens.

Non-sequential processing. Transformers process each sentence as a whole non-directionally so there is no risk of "forgetting" previous information. This is a strong improvement over directional RNN models as transformers do not require past hidden states to understand contextual dependencies between words earlier in the sequence. Bidirectionality also provides the added benefit of being able to learn the context of a word based on its surroundings, not just preceding words.

Triple embeddings. The interpretation of text input is as follows:


Input
[CLS] my dog is cute [SEP] he likes play ##ing [SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} E_{cute} $E_{[SEP]}$ E_{he} E_{likes} E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10}

Note: The multiple embeddings used by BERT-like models to numerically represent the semantics and positions of each token. [CLS denotes a sentence-level classification, used for pooling. [SEP] separates two distinct sequences/sentences. "The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings." Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Each word is represented by a combination of three embeddings:

- **Token:** Embeddings to represent the meaning of the word.
- **Segment:** Marks which sentence the token belongs to - this is useful for question answering and sentence similarity tasks.
- **Position:** Expresses the order of words so BERT can capture the sequence of tokens.

[20] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998-6008). arXiv.

[21] ibid.

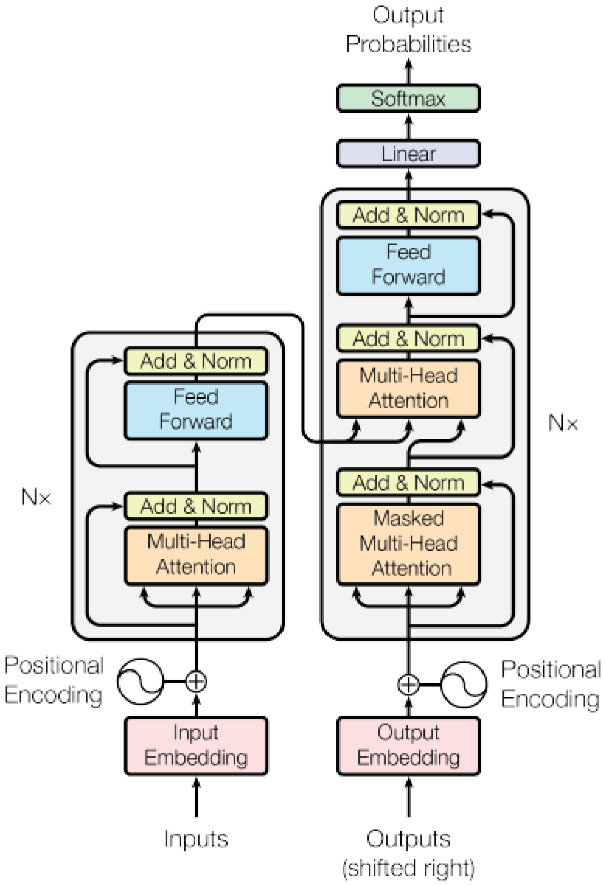
The particular NLP transformer group of models explored in this investigation is the BERT family - BERT, RoBERTa, DistilBERT and DistilRoBERTa. The BERT series of models are unique by their bidirectionally encoded representations - during pretraining and processing they process each token, alongside the preceding and proceeding tokens.

Transformers can handle polysemy - words having multiple meanings - an issue that was impacting the performance of shallow networks eg. Word2Vec (W2V). Non-directionality and attention enable transformer models to capture the multiple nuances of each word in a string.

In this context, the transformer with a sequence classification/regression head (a linear layer on top of the pooled output) is used.[22]

Architecture

In order to learn contextual links between words, transformers have highly complex encoder-decoder structures. BERT is a pre-trained stack of encoders: bert-base, the model used in this task has twelve layers. These are joined with 769 feedforward networks and 12 attention heads.



[22] https://huggingface.co/transformers/model_doc/bert.html

Implementation

As BERT has been pre-trained on massive corpora of unlabelled human text (BooksCorpus and English Wikipedia - totalling to ~3300M words[23], performing linguistic preprocessing to remove stopwords or lemmatise would be counterproductive to the classification performance.

As a result, the only preprocessing that was required was to map words to lower-case if the transformer model in question was an uncased variant.

The primary goal of this experiment was to determine the effects of three dummy hyperparameters: class weighting, model distillation and RoBERTa-like models.

Class Weighting

Most machine learning classification algorithms perform well with balanced class distributions. However, for detection-type problems, this is not always the circumstance. The motivation is to penalise misclassifications of the minority while decreasing the emphasis of mistakes made on the majority class.

Intuitively we want a weighting that is inversely proportional to the observed frequency. To calculate the weight of each class, we apply the formula:

$$w_i = \frac{|D|}{c \cdot n_i},$$

That is, the weighting of class *i* is equal to the number of total data samples *D* divided by the number of classes *c* multiplied by the number of data samples of class *i*, n_i. In the Quora dataset, the ratio of the negative to the positive class is approximately 15:1. Hence we would expect a training weighting of roughly [0.53,8].

Distillation

When it was released in 2018 by Google, BERT was the avant-garde of NLP in terms of performance.[20] However, a drawback associated with its cutting-edge performance was its size. BERT-base has 110M parameters which drastically impacts its training, testing, and loading time.

Distillation is the application of a suite of techniques to reduce the inference time and size of a given model. One such technique is knowledge distillation,[24] where a smaller “student” model is trained to reproduce the large “teacher” model’s behaviour. The paper by Sanh et al. demonstrates that BERT’s size can be reduced by 40% while maintaining its language understanding (97% performance) and increasing inference time by 60%. As a result, training is faster and more lightweight, enabling more data to be used in the fine-tuning process in less time. The nature of the results below indicates that the models were likely not training enough as using more of the training data (in distillation) improved performance.

[23] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*. arXiv.

[24] Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*. arXiv.

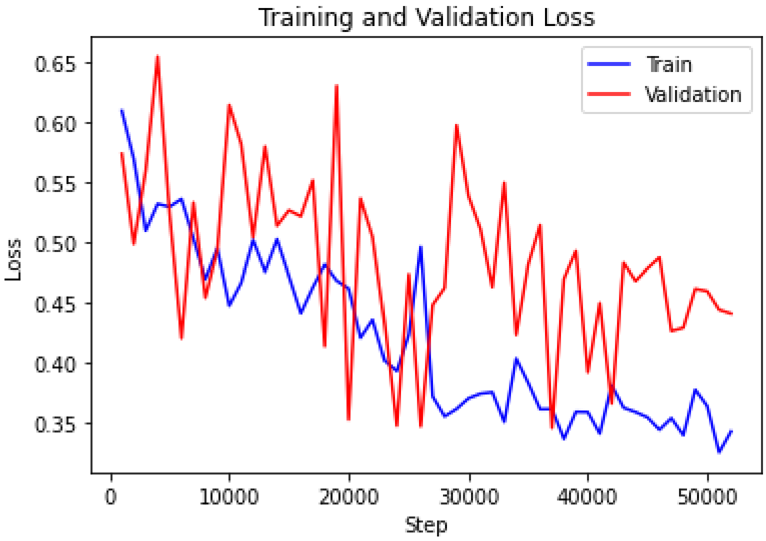
RoBERTa

Liu et al.'s paper *RoBERTa: A Robustly Optimized BERT Pretraining Approach*[25] provides strong evidence that pre-training hyperparameters have a significant impact on the performance of transformer models. Specifically, "BERT was significantly undertrained" and by optimising the pre-training process by considering new training design choices, can produce a model with a stronger state of the art performance.

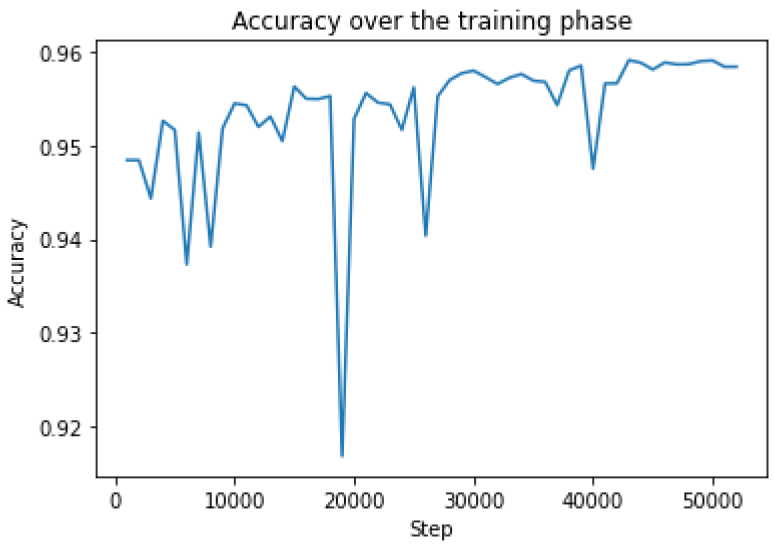
Each combination of the three hyperparameters was tested, with the main training process remaining consistent:

1. Read in data and split into train and test sets. For this, an 80:20 split with a set seed was used for reproducibility. 20% of the train set was withheld as a validation set.
2. Select the transformer model.
3. Calculate class weights from the training set for custom loss in the fine-tuning process.
4. Load in the model and tokenizer.
5. Further subdivide training set into training and validation.
6. Tokenize the train and validation sets and convert them to a dataset object for fine-tuning.
7. Define a custom loss function incorporating class weights.
8. Fine-tune and evaluate.

One such example of training is given below:



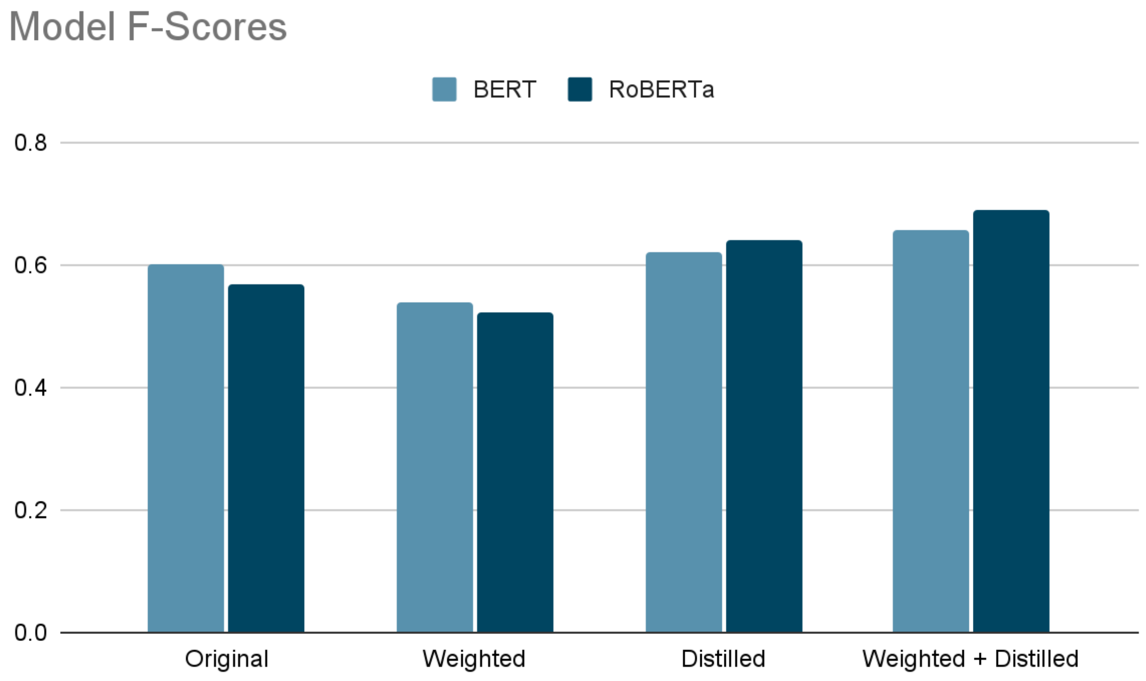
[25] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... & Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*. arXiv.



Results

The following results depict the f-scores of each type of model, calculated on the testing set.

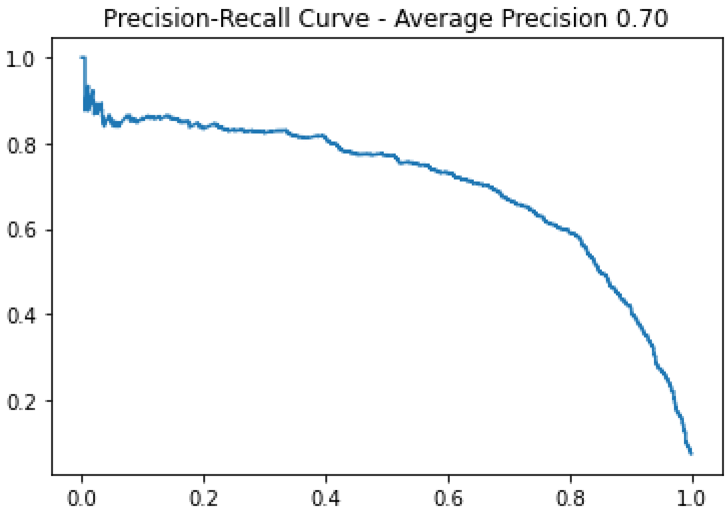
Model	Raw	Weighted	Distilled	Distilled + weighted
BERT	0.600	0.540	0.622	0.658
RoBERTA	0.567	0.521	0.642	0.689



From this experiment, we can conclude that the distilRoBERTa variant with class weights had the best classification performance in the f-score. Interestingly, BERT outperformed

RoBERTa in their base configurations, while distillation showed a better performance overall and significantly improved RoBERTa. Further optimisation experiments were conducted below.

Below is a precision-recall curve of the classifier's scores, with an AUC of 0.7.



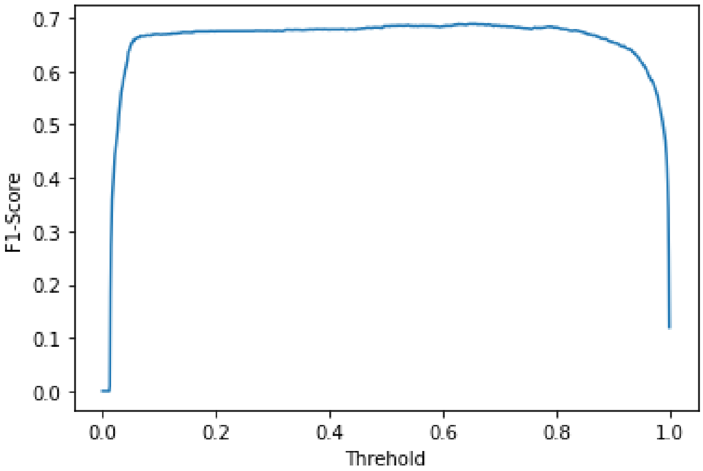
Prediction Probability Threshold Optimization

By default, the classification prediction is chosen as the argument (in this case index) that maximises the tensor returned by the model. As a result, the classification threshold occurs at 0.5 as the tensor is in the form $[P(c = 0), 1 - P(c = 0)]$. However, this may not be the optimal location due to skewed class distribution and the importance of classifying positive classes.

Thresholding, here, involved performing a grid search, sweeping from zero to one in increments of 0.001 to determine the value that maximises the f-score. For the weighted distilRoBERTa model, the average increase in f-score was 0.0086.

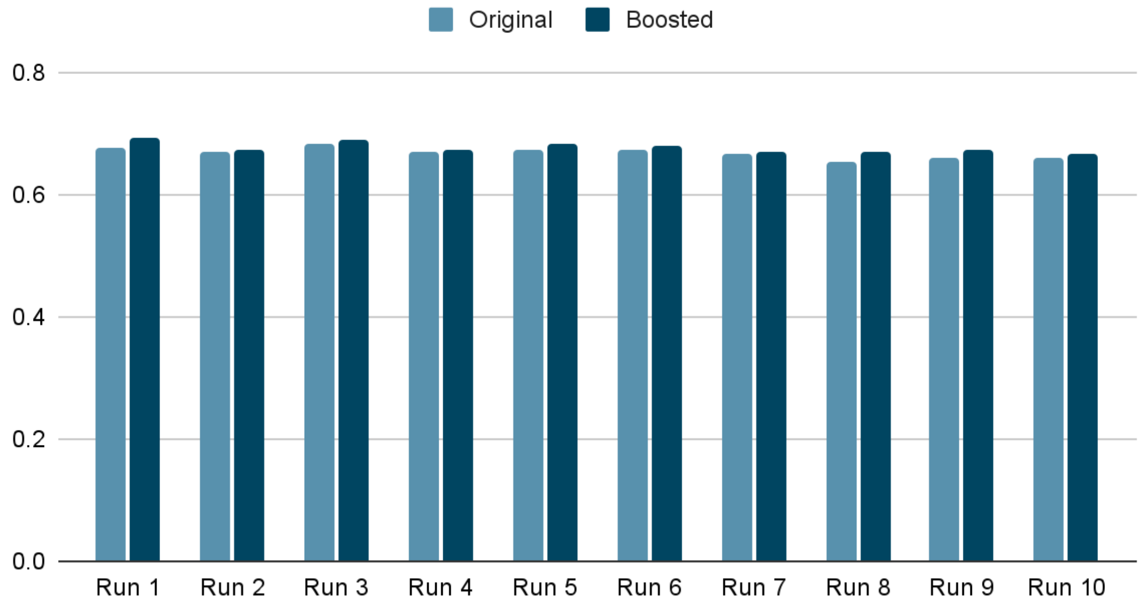
A single instance of this optimization is below.

F-score threshold curve.



Evidently, the model is confident in its decisions as changing the prediction threshold has minimal impact on the final score. Furthermore, this is a retrospective technique: the improvement can only be determined if the true values are known. Thus, the process will be run several times and the mean of the optimal thresholds will be selected.

Improvements from Thresholding



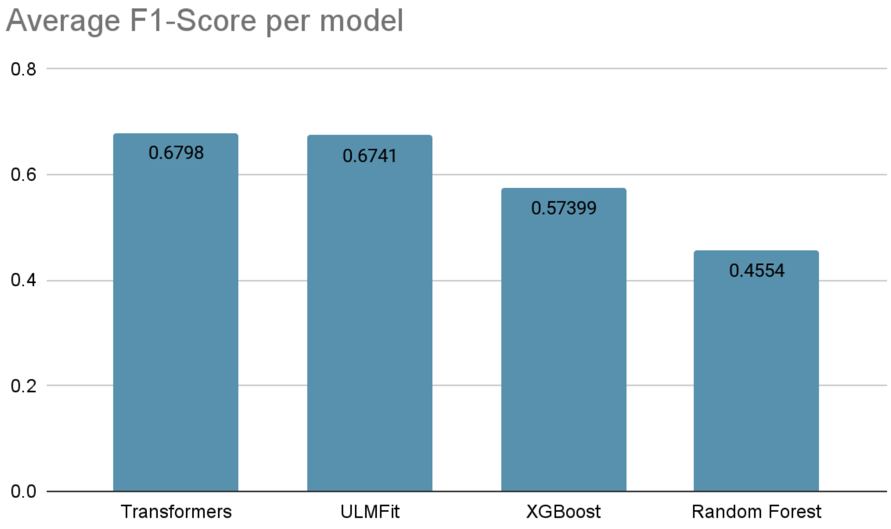
Descriptive Statistics of the Thresholding Process

Threshold Mean	Threshold Standard Deviation	Mean Improvement
0.6798	0.0556	0.0086

A copy of the best performing model can be found here.

Discussion

Overall, increasing the sophistication of the model enhanced scores, with the Transformers method achieving the highest f1-score. This is unsurprising as neural network techniques abstract semantic representation (as opposed to engineering features manually) in order to learn a deeper understanding of the underlying relationships in the given target corpus.



Note: F1-Scores for each model.

Initially, we implemented the random forest model, which provides a baseline for our more complex techniques although it achieved a relatively low f1-score. Effectively, since the random forest implementation uses bag-of-words and hence only considers single word sentiment values, it cannot account for the overall context of the question. This issue is amended by LSTM (via ULMFit) and by extension Transformer techniques. Nevertheless, certain techniques can be used to boost the performance of random forests, including using word2vec to better capture relationships between text, and experimentation with the number of trees to optimize the trade-off between complexity and train time.

Perhaps the most famous model applied in Kaggle Competitions, our implementation of XGBoost lives up to its name and provides a far more powerful model than bagging techniques, namely, Random Forest. On the other hand, via our experimentation with XGBoost on the Quora corpus, our implementation can be extended beyond its current capabilities by utilising a greater number of folds and further analysis to engineer additional features to better capture the various nuances in the data, for instance, perhaps the number of exclamation marks could be an indicator for insincerity.

Neural networks capture far greater complexities in text data that is difficult to replicate with feature engineering, reflected by empirical results from high f1-scores. Specifically, along with considerably larger f1-scores than XGBoost, ULMFit converges quickly to a stable f1-score within a few epochs via the “Gradual Unfreezing” methodology. Likewise, additional use of the LM allows the classifier to capture semantic representation beyond LSTM architectures. Conversely, the model could be significantly improved by ensemble methods such as stacking. Likewise, as ULMFit requires training multiple models, the process is highly resource-intensive and may not be a viable business solution.

Attaining the highest results, the Transformers technique reinforces its reputation as state-of-the-art in NLP. This can be attributed to attention mechanisms to optimise computation time and power, non-sequential processing so as not to lose previous contextual dependency information and, triple embedding systems to better understand contextual meaning by position in a sequence as well as the order of the sequences themselves. On the contrary, attention mechanisms add more complexity to parameters and pretraining can bias the model's fine-tuning. With further research, more advanced techniques may resolve issues of fine-tuning via self-ensemble and self-distillation methods[26].

Conclusion

Kaggle's Quora Insincere Question Classification Challenge is a complex research problem that we have tackled via the implementation of several machine learning methods.

The performance of the four algorithms were primarily measured by F1-score, with ULMFit's and Transformers' performance exceeding our implementations of Random Forest and XGBoost. Our most successful F1-scores were all trained on an 80:20 split of 80% of the original data, giving us an average 0.6741 F1-score for ULMFit and 0.6798 F1-score for Transformers compared to 0.5740 F1-score for XGBoost and 0.4554 for Random Forest. From this we can distill that models with the ability to learn semantic relationships were vastly superior than those models that required feature engineering.

This is somewhat surprising since we had believed that logical or rule-based trees may have been able to compete against more complex models such as neural networks. However, we have come to learn that this is not at all the case. Models that are more complex, able to train on large quantities of data and are suited to detecting deep patterns, such as Transformers, can be conclusively superior for text sequence classification. Logical or rule-based trees appear to be insufficiently complicated to detect patterns of insincerity as they have the same accuracy as network-based learnings. We have come to conclude that this is a limitation of logical and rule-based trees, that they are unable to create valuable inferences about the context of a given question.

Parallel processing through the python pool library should have been used to facilitate the implementation of more complex ensemble techniques. Combining parallel processing with greater fold values and deeper feature engineering would have ultimately given our rule-based trees a greater chance at competing with models with the ability to learn semantic relationships. This is because feature engineering increases the amount of valuable information logical and rule-based models can use to predict. Not only could adjustments be made to improve run-time and complexity, but more advanced techniques were also required to handle the imbalanced data set problem. For example, defining the appropriate metrics,

[26] Xu, Y., Qiu, X., Zhou, L., & Huang, X. (2020). Improving bert fine-tuning via self-ensemble and self-distillation. arXiv preprint arXiv:2002.10345.

thresholding for predictions and using more advanced or simpler loss functions for a more complete idea of our models' F1-scores should be done in future.

With further development of our current study we can expand our models to not only classify the insincerity of a question but also potential toxicity that may be present. We may also be able to extract our current classification models and repurpose them for the classification of fake news or problems with similar sentiments.

Appendix

GitHub Repository:
https://github.com/MillenniumForce/COMP9417-Group-33

Links to access models:

- Random Forest

https://www.kaggle.com/ashsweg/randomforesttree-9417-assignment

- XGBoost model and code found at:

https://www.kaggle.com/stanleychennn/fork-of-xgboost-baseline

- ULMFiT

Model binaries and code found at:
https://www.kaggle.com/juliangarratt/comp9417-quora-insincere-classification

- Transformers

Model files found here. Kaggle notebooks here but are also in the code .ZIP file.