Impact of Various Techniques on Minority Subject Area News Text Classification

Al Hogan, Anadi Jaggia, Nick Torvik, Isaac Sutor
Georgia Institute of Technology
ahogan32@gatech.org, ajaggia3@gatech.edu, ntorvik3@gatech.edu, isutor3@gatech.edu

Abstract

In many news classification datasets, underreported news domains are not represented, manifesting as severe class imbalance. With the many niche news domains out there, we explored whether or not they could be detected and properly classified, possibly to serve a more niche population interested in such content. We proposed a multifaceted approach to solving the class imbalance issue through various pre-existing methods to solve this problem. Preprocessing methods were used to manipulate the sampled data before training, different loss functions and weighting schemes were applied, and architectural adjustments were made to better handle class imbalance in the data. Preprocessing, some loss functions, and their weighting schemes did not improve the model's performance, whereas leveraging LDAM as a loss function did improve the model's accuracy.

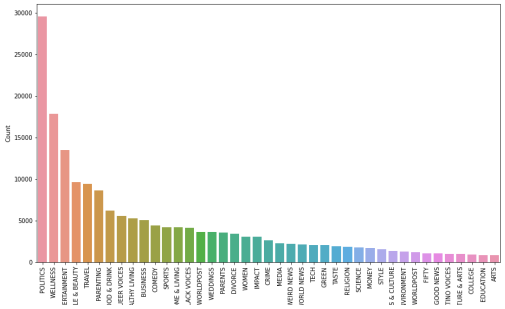


Figure 1. Categories by Volume

1. Introduction/Background/Motivation

Class imbalance is common in real-world data, and techniques for mitigating the impact of that imbalance are a hot research topic. To help with research, we used a news category dataset from Kaggle, with over 180 thousand records, three top-heavy categories, and a more even distribution after to test different methods for mitigating the effects of class imbalance [11]. We wanted to explore the current benchmarks while creating our own model to address the questions above. In the process, we aim to look at which strategies improved our model performance the most. This project presents a powerful learning opportunity for our team, with the possibility that we could contribute to the research being done in this important field.

There were two examples we studied to understand benchmarks today. Neither re-categorize nor change the data, and both use embedding, LSTM, and dropout to model. The first approach adds linear layers to the model and sees a validation accuracy of 65% [12], while the second, LTSM with attention, also scores a 65% [14]. Neither of these focus on applying methods to deal with imbalance. Our goal is to improve accuracy by focusing on handling the class imbalance. We will use similar techniques to these baseline examples, show our accuracy, and then expand and experiment. Class imbalance is a huge problem in real-world data. Fraud, spam, early warning detection systems, and personnel hiring are real-world examples where class imbalances is an issue. A Deep Learning model could approve fraud because it can get 99% accuracy by approving every transaction. However, that 1% or less of transactions the model fails to classify contains fraudulent transactions, resulting in a negative impact to customers. By finding better solutions to deal with class imbalances, biases in the data can be mitigated, ensuring that each class is equally represented. It also means very rare occurrences can impact a model's decision-making more than the distribution of data elements would typically allow.

2. Approach

For this project, an LSTM model that evaluates our chosen dataset was chosen as a baseline from Kaggle. The model contained no methods for curtailing the imbalance effects in the data and used the cross-entropy loss function. Since the dataset has imbalance, this baseline provided the opportunity to add data preprocessing,

loss functions, and architecture to observe what positively impacted the model by improving model accuracy and reducing loss. First, various methods for preprocessing the imbalanced data were used, such as removing stopwords or lemmatization. Next, using PyTorch, various loss functions were used, along with various loss function weighting schemes. Finally, text embedding was leveraged to see how that approach stacked up to the other methods. Looking at the research available, we believed at least one of these approaches would be successful because they had worked for class imbalance in other multi-class datasets . Many of the strategies we pursued have been used for computer vision tasks, so the approach to adapt these to a text classification problem is somewhat new.

We anticipated some issues with adapting the various methods to the baseline model. Loss functions had to be tweaked to work with this implementation because the baseline was not built with these auxiliary methods in mind. There were some issues with getting the loss functions adapted. The original focal loss code did not work, so a different implementation had to be found and adapted for our use. Ultimately, almost none of the methods we tried worked. Improving the model's performance using class-balancing methods proved more difficult than originally thought. The preprocessing methods did not yield better results, and only the Dice Loss and LDAM DRW functions made improvements. None of the weighting schemes used to augment the loss functions helped. We also tried changing pre-trained word embeddings, which yielded no positive results, but did point us toward another way that architecture can be used to reduce class imbalance by using Facebook's FastText architecture. Overall, taking a multi-faceted approach to improving a model using class-imbalanced data was difficult and highlighted the importance of trying many different approaches. If we had only tried one type of method for improving the model, we might not have seen any improvement at all.

3. Experiments and Results

3.1. Preprocessing/Dataset Manipulation

There were three independent analyses done before training the text classifier. This section discusses the reason and impact of each and uses code from Heng Zheng's Kaggle example [14], where the main architecture was LSTM with attention. There is minimal preprocessing in this baseline code, and no data augmentation or sampling was done. In fact, there were only three preprocessing steps done:

- Combine the headline and description into one line
- Use Keras tokenizer, which by default removes punctuation and makes letters lowercase
- Removes any headline+description < 5 words

3.1.1 Data Preprocessing

The first analysis created additional preprocessing by doing three things:

- Removing stopwords
- Stemming
- Lemmatization

Stopwords are commonly used words such as "the", "me" or "and" that provide little to no value in natural language processing. For this research, NLTK(Natural Language Toolkit) in python was used to remove stopwords [2]. Stemming removes or stems the last characters of words, such as -ing or -ly. Lemmatization converts words to the correct base form based on vocabulary and morphological analysis of words. For example, we convert *see* to *saw* based on verb or noun context [10]. The preprocessing vs. the baseline results were not positive, suggesting that the authors' use may have been distorted, making the model perform worse.

3.1.2 Undersampling

The approach used to test undersampling was the simplest form: random undersampling. Random undersampling involves removing events from the majority classes. The advantage of this analysis is that removing records corrects the imbalance, requiring less storage and time to run. However, by eliminating records, we are also losing potentially valuable information. Since it is also random, the selection chosen could be biased. Results show that the model underperforms substantially to both the baseline and preprocessed versions.

3.1.3 Oversampling

Opposite to undersampling, where we remove records, random oversampling creates duplicate records of the minority classes. A well-known con to this approach is that making exact copies of the majority class may increase the likelihood of overfitting occurring [1]. The results show that the random oversampling overfit quite substantially. The second approach to Oversampling was SMOTE, which synthesizes new examples for the minority class and provides further information to the model. Specifically, it picks a random example from a minority class, and the k nearest neighbors are found. The synthetic example is 'created by choosing one of the k nearest neighbors b at random and connecting a and b to form a line segment in the feature space. The synthetic instances are generated as a convex combination of the two chosen instances a and b' [5]. The downside to this approach is that if there is ambiguity between classes, the synthetic example could resemble a majority class.

The results show that the downside to this (ambiguity) is present.

Table 1 summarizes the results from the preprocessing and dataset manipulation methods. Graphs for different experiments can be found in Appendix section 8.

3.2. Loss Methods/Weighting

3.2.1 Focal Loss

The Focal Loss function can be used to address class imbalance in input data for deep learning models. It improves on the Cross-entropy function by adding a modulating factor, giving it the ability to focus model training on hard negatives while down-weighting easy examples [8]. Typically it is used for object detection, but for this project, Focal Loss was used for text classification. Because the source dataset has a large imbalance between the minority class and majority class (education and politics, respectively), it was hypothesized that this method could help compensate for the imbalance, improving the accuracy while mitigating loss. There was no research found for leveraging Focal Loss specifically for text classification tasks.

3.2.2 Weighting methods

In addition to utilizing different loss functions to help overcome the class imbalance in data, various weighting schemes can be used to adjust how a loss function operates. Weighting the loss function allows the loss computed for the different classes to be computed differently based on the number of occurrences of each label class in the input data [13]. The loss functions used here are all related to research in cost-sensitive learning [4]. In addition to vanilla Focal Loss, the loss function was adjusted to allow the use of various loss weighting schemes. The same weighting schemes were also plugged into the Cross-entropy loss to observe whether the original model could benefit from only adding weights to the loss function out of the box. These weighting methods have been mainly used for computer-vision related tasks rather than text classification tasks, and no research was found regarding this use case. Research leveraging these weighting methods typically uses them for sampling the data in preprocessing steps, but can these methods produce a superior loss function that allows the model to beat the baseline?

Class Balance Weighting First, class balance weighting was used. This method calculates an effective number of samples, or the expected volume of each of the samples, to create a set of weights to add to a loss function [8]. A hyperparameter, β, can be tuned to adjust the reweighting effects of this method. The original authors found that a high β close to 1 is the most effective. Several values were experimented with, and a value of 0.9999 was used for these experiments because it produced the best results [4] [13]. This method of weighting was applied to both focal loss and cross-entropy loss, similarly to how it was utilized by Cui et al.

Inverse Class Frequency Weighting This simple weighting method takes the inverse of the total number of samples of each class to produce weights. While simple, it is commonly used and is generally effective [4] [13]. The weights produced by this method are similar to the weights produced by the class balance method when it has a high β value because the class balance problem becomes very similar to a simple inverse class problem as β approaches 1. While class balance and inverse class methods did produce similar weights, they did not produce identical results when used in the loss functions.

Inverse Square Class Frequency Weighting Another simple method, inverse square weighting, takes the inverse of the square root of class frequency to produce a distribution of weights [13]. This method results in a similar weighting distribution to inverse class frequency weighting, but produces larger values with a smaller overall spread.

Proportional Weighting A final set of weights was created by taking a ratio of each class frequency to the total number of records in the dataset and subtracting the value from 1. This method created a small spread and gave much less weight to the majority class while giving very similar weights to the middle and minority weighted classes.

Results None of the methods for weighting had a positive effect on the baseline classifier. While manipulating the loss functions may have helped classify more minority records correctly, it may have been unable to classify the majority class well enough to make the tradeoff worthwhile. Focal loss with inverse class weighting performed the best with the highest accuracy and lowest test loss of the methods used, but it still fell short of the baseline performance. Table 2 summarizes the results from the weighting methods. Graphs for different experiments can be found in Appendix section 7.1.2.

3.2.3 Dice Loss

Where cross-entropy fails with imbalanced datasets is that it averages the loss across all examples of the dataset. Therefore, the more unbalanced the categorical distributions are, the more difficult it will be to generalize. While you can introduce an array of weights to cross-entropy loss to undermine overrepresented classes and give more credence to the underrepresented classes, finding this exact balance is not trivial and is not guaranteed to work. This is where the Dice-Sørensen coefficient comes in as seen in [7]. It is used to gauge the

similarity between two samples X and Y.

$$DSC = \frac{2|X \cap Y|}{|X| + |Y|}$$

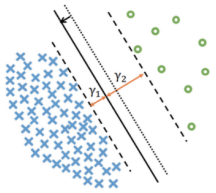
Here, we can view X as the ground truth and Y as our predictions. The DSC is equal to the number of elements common to both sets multiplied by 2 and then divided by the sum of the total number of elements in both sets. The main advantage of such a formula is that we do not need to try and obtain an optimal weighting strategy. Since this is a measure of similarity, and we want to minimize the difference, we subtract this DSC from 1 to get our Dice Loss function: 1 – DSC. By aiming to minimize this cost function, we can find more optimal local minima. In our experiments however, we found this to be false, and failed to beat the baseline test accuracy of 65.65% with our dice loss's test validation score of 51.16%.

3.2.4 LDAM DRW Loss

Label Distribution Aware Margin Loss with deferred reweighting, or LDAM-DRW loss, aims to alleviate this imbalanced dataset problem and performed better than Dice Loss in our experiments. The formula is described as follows:

$$\mathcal{L}_{\text{LDAM}}((x, y); f) = -\log \frac{e^{z_y - \Delta_y}}{e^{z_y - \Delta_y} + \sum_{j \neq y} e^{z_j}}$$
$$\text{where } \Delta_j = \frac{C}{n_j^{1/4}} \text{ for } j \in \{1, \ldots, k\}$$

where (x, y) is a datapoint, f is the model, z_j is the jth output of the model for the jth class and C is a hyperparameter to tune which affects the class-dependent margins. A margin is defined as the minimum distance of a given data point to the decision boundary and is denoted as γ. The paper shows that in order to improve the prediction power of the model towards underrepresented data points, we minimize this distance to the margin for these same underrepresented samples. What makes this algorithm 'label distribution aware' is that the margin's value is set to be inversely proportional to the number of samples of the underrepresented classes as shown below:



The deferred reweighting (DRW) process works by initially having one method for weighting the different classes up to and including a tuned epoch number, followed by a second reweighting method for the rest of the epochs. The first is naïve and gives uniform weights to all classes, whereas the second reweighting method reweights "according to the inverse of the effective number of samples in each class, defined as $(1 - \beta^{n_i})/(1 - \beta)$." [3]. This is the same reweighting method as Focal Loss. Unfortunately, the paper does not go into how to exactly tune this parameter C. In the open-source implementation, there were two parameters: max_m, which corresponds to the max enforced margin value, and s, which corresponds to the C value based on the descriptions in the paper. Via hyper-parameter tuning, for the first 5 epochs, s was set to be 0.25 and max_m was set to be 10. After that point, from the 6th epoch onward, the values were the default with s at 30 and max_m at 0.5. In the end our test accuracy was 62.79%, which is comparable performance to our baseline. The key here is that the first stage of training brings the model to a better position in time for training the second half compared to just starting from scratch with the second stage. Nevertheless, from the 6th epoch onwards the loss increased, and the validation accuracy stagnated. This is possible because accuracy is just measuring how many samples' max probability output in the end was correct, while loss is measuring how far off it was from the ground truth. The model can still be correct, but just not as sure, which explains this phenomenon.

In an attempt to get around this, an experiment was run without DRW at all, allowing the model to train for all 10 epochs without DRW ever kicking in. In the end, this proved to be our most improved final test accuracy being 67.44%, narrowly beating out the baseline without the spike in loss. This means that utilizing Adam with the LR scheduling is sufficient for this use case paired with LDAM instead of LDAM-DRW.

3.2.5 Overfitting and Mitigation

When first implementing LDAM-DRW and Dice Loss, the model would overfit when training. To mitigate this problem, a two-pronged approach was implemented by pairing regularization with a tailored learning rate sched-

uler. For our optimization algorithm, Adam was used. The learning rate scheduler was different for Dice Loss and LDAM. On a high level, regularization helps avoid overfitting by reducing the importance given to certain weights that are quite high, or weights that are outliers in other words. Regularization prevents us from learning a more complex model, which is what tends to overfit our data in the first place. It does so by shrinking these higher coefficients closer to 0, but never 0, in the L2 norm case. L2 regularization is especially useful when we have more parameters than samples (which is the case for our dataset), and L2 helps reduce model complexity, which reduces overfitting and leads to higher average accuracy.

3.2.6 LDAM DRW Overfitting Mitigation

To regularize Adam, we first made use of the $weight\_decay$ parameter, which implements L2 regularization. The regularization term is the $weight\_decay$ parameter multiplied by each of the weights. This is then added to the cost function, which is then derived to calculate the gradients for back propagation. As for the learning rate scheduler, we implemented ReduceLROnPlateau for LDAM, which reduces the rate depending on certain parameters that we can tune and set. Via hyperparameter tuning, the patience was set to 1, the threshold to 0.5 and all other settings were left at default. This means the model waits for 1 epoch in which it sees a training loss improvement less than 0.5 before annealing the learning rate by a factor of 0.1.

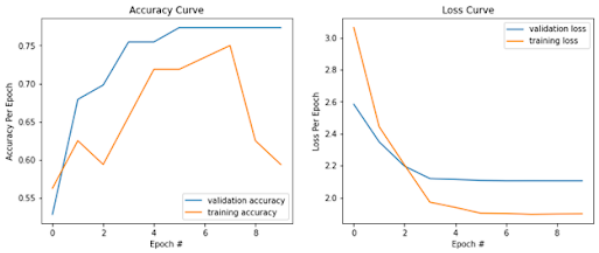


Figure 2. Overfit Migation with LDAM (no DRW) + ReduceLROnPlateau + Adam L2 Regularization

Reference Figures in 7.1.3 for additional graphs that were underperforming prior to mitigation.

3.2.7 Dice Loss Overfitting Mitigation

Here, it was empirically sufficient to simply utilize the CosineAnnealingLR scheduler without any weight decay implementation. Its formula is described as follows:

$$n_t = n_{min} + \frac{1}{2}(n_{max} - n_{min})(1 + cos(\frac{T_{cur}}{T_{max}}\pi))$$

N_t is the current learning rate, n_{Min} is the lowest possible rate, n_{Max} is the highest possible learning rate, and T_{max} is max number of iterations the function is called for. In short, this formula changes the learning rate such that it follows the path of a cosine wave. The advantage of this is that when the learning rate is low and stuck in a local minimum, the switch to the larger rate allows it to move on to potentially find a better spot. While it may be that the jump here isn't enough to escape this minimum, it is unlikely to know if this will happen without trial and error. Via tuning, Tmax was set to 2, and all other settings were the same as default.

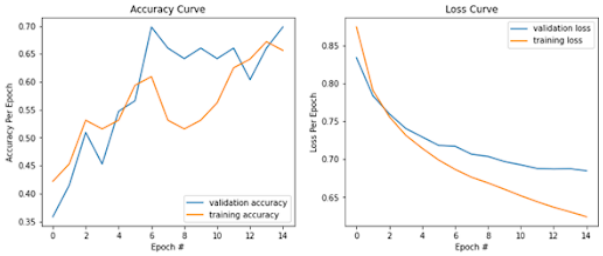


Figure 3. Overfit Mitigation with Dice Loss + CosineAnnealingLR

Table 3 summarizes the results from the LDAM, LDAM DRW, and Dice Loss methods. Figure 7.1.3 highlights an additional graph of Dice Loss overfitting.

3.3. Architecture Discovery

Following loss methods, preprocessing, and reweighting classes, we tried looking at the impact that architecture has on class imbalance. In this step, we tried using Facebook's FastText library which uses continuous bag of words or n-grams. We wanted to see how much of class imbalance can be impacted by architecture without any changes to the dataset used.

3.3.1 Word Embeddings with FastText

FastText uses pre-trained word embeddings to improve training time and accuracy. Word embeddings tie related words mathematically, such that the number of columns in a word embedding dictionary doesn't equal the number of unique words in a text corpus. This makes the size of the learning corpus smaller, making the speed of training and evaluation much faster.

3.3.2 Bag of N-Grams with FastText, Reimplemented in Pytorch

FastText uses a Continuous Bag of Words or N-Grams architecture. It builds trees to connect related words for faster lookup. FastText uses a Huffman algorithm for the trees. This algorithm helps account for class imbalances.

Due to the algorithm adding deeper nodes to account for frequency, FastText can handle class imbalances much better than other architectures. It trains on CPU, but does use more memory than other similar methods like traditional word2vec and glove.

The combination of the word embeddings and the architecture allowed the FastText model to train very quickly with good accuracy. The model trains on the full dataset in 4.5 seconds on an Intel Core i7-4790K (released in 2015). It also performs very well with over 70% accuracy and recall. Further, the model can produce top 5 results, achieving the correct class 93% of the time in the top 5 classes. These results are without hyperparameter tuning and using the defaults provided by FastText. When increasing epochs to 25 and adjusting the learning rate, the model can achieve 74% accuracy within 20 seconds. Adjusting the learning rate and epoch allow a greater learning of underrepresented classes even more. FastText also allows the ability to tune specifically for a certain class, by using underrepresented classes as the autotune metric, we could adjust the weighting to correct answers for the minority classes, balancing accuracy more evenly throughout the dataset as shown in Figure X. The best result was from our own hyperparameter tuning which involved increasing the wordNgrams to two which increases the size of the text that the model looks at either side of a given word for context, along with increasing epochs and upping the learning rate to attempt to let minority classes learn more rather than being diminished to nothing if they end up getting trained last. This result provided a good distribution of results from the highest to the lowest frequency classes, causing only a 30% difference rather than the almost 60% difference in baseline FastText run.

To separate the impact of the pretrained embeddings from the learning architecture, we needed to find a way to change the architecture while still using FastText embeddings. We attempted to use FastText Embeddings in place of the GloVe embeddings used in the baseline PyTorch model. This resulted in nearly identical results to the original, leading to the belief that the benefit of FastText is in the underlying architecture more than the pre-trained embeddings.

4. Future Work

4.1 One thing that could be tried in the future to further improve model performance is Adam with decoupled weight decay (ADAMW). As discussed above, the regularization term is the $weight\_decay$ parameter multiplied by each of the weights, which is then added to the cost function, which is explained further in the appendix 9.

The issue with vanilla Adam from step 6 (highlighted in purple in the appendix) is that the weight decay term is eventually divided by the square root of velocity as well. Since the velocity is derived from momentum, which is in turn derived from the addition in step 6 followed by obtaining the gradient, the velocity can turn out to be quite large if the weights are quite large. This means when we square it, those weights' gradients are changing slower than its counterparts which are smaller. This is not the true meaning of L2 regularization as only some of the weights are getting improved instead of all and not at the same rate. Therefore, as seen in [9], the authors propose adding the same regularization terms after we update the weights instead. This allows for only adapting the gradient of the loss function in isolation instead of the loss function and the regularization gradient, hence the term decoupled (step 12). In theory, this can yield even better results. However, since we achieved good numbers with just Adam, this further optimization is left for future work.

4.2 Future work may also include a combination of FastText architecture, namely their utilization of the Huffman Algorithm to reduce imbalance and LDAM/Loss function techniques. FastText builds a tree for relational embeddings using a Huffman algorithm. This process of storing vectors of related words and phrases reduces the impact of class imbalance by accounting for the frequency of words in the depth of the node. This combination may lead to interesting results, though the combination could prove to overfit the minority class depending on how the two methods interact and feed each other.

5. Conclusion

While the baseline notebook did not aim to handle imbalance, a few of the data imbalance handling methods we tried proved useful. The first thing that worked was Dice Loss. This was subsequently beaten by LDAM DRW, followed by LDAM without DRW, which was the top loss method attempted. However, they didn't work right away and were overfitting. This required some additional methods for remediation, such as the implementation and hyper-parameter tuning of various learning rate schedulers and loss optimizers. Another successful method to solve this imbalance issue was FastText's continuous bag of words architecture. LDAM and FastText individually proved to have the best results in improved accuracy and reduction in bias against the minority class. It would be interesting to utilize the benefits of the Huffman algorithm in learning architecture found in FastText along with LDAM to maximize the impact on imbalance datasets in one combined model as a future work endeavor to build upon the model further.

6. Work Division & Project Code

Table 5 summarizes the contributions of each teammate. Project code can be found on Github [6].

References

[1] Fernandez Alberto, Garcia Salvador, Mikel Galar, Ronaldo C. Prati, Bartosz Krawczyk, and Francisco Herrera. *Learning from Imbalanced Data Sets*. Springer International Publishing, 2018. 2

[2] Bird, Steven and Klein, Ewan and Loper, Edward. Accessing text corpora and lexical resources. https://www.nltk.org/book/ch02.html, Last accessed on 2021-12-13. 2

[3] Kaidi Cao, Colin Wei, Adrien Gaidon, Nikos Arechiga, and Tengyu Ma. Learning imbalanced datasets with label-distribution-aware margin loss, 2019. 4

[4] Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie. Class-balanced loss based on effective number of samples, 2019. 3

[5] Haibo He and Yunqian Ma. *Imbalanced learning: foundations, algorithms, and applications*. John Wiley Sons, Inc., 2013. 2

[6] Hogan, Al and Jaggia, Anadi and Torvik, Nick and Sutor, Isaac. Dl_fall2021, 2021. http://github.com/Jaggia/DL_Fall21/, Last accessed on 2021-12-15. 7

[7] Xiaoya Li, Xiaofei Sun, Yuxian Meng, Junjun Liang, Fei Wu, and Jiwei Li. Dice loss for data-imbalanced nlp tasks, 2020. 3

[8] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018. 3

[9] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. 6

[10] Manning, Christopher D and Raghavan, Prabhakar and Schütze, Hinrich. Stemming and lemmatization. https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html, Last accessed on 2021-12-13. 2

[11] Rishabh Misra. News category dataset, 06 2018. 1

[12] Muhammad Atiqi. News classification lstm 65% accuracy, Jul 2019. https://www.kaggle.com/arutaki/news-classification-lstm-65-accuracy, Last accessed on 2021-12-15. 1

[13] Ishan Shrivastava. Handling class imbalance by introducing sample weighting in the loss function, Dec 2020. https://medium.com/gumgum-tech/handling-class-imbalance-by-introducing-sample-weighting-in-the-loss-function-3bdebd8203b4, Last accessed on 2021-12-13. 3

[14] Heng Zheng. News category classifier (val_acc 0.65), Jul 2018. https://www.kaggle.com/hengzheng/news-category-classifier-val-acc-0-65, Last accessed on 2021-12-15. 1, 2

7. Appendix

7.1. Graphs

7.1.1 Preprocess and Dataset Manipulation

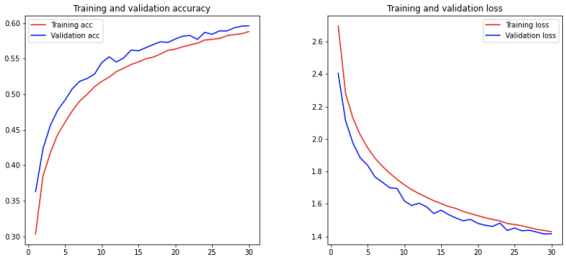


Figure 4. Baseline

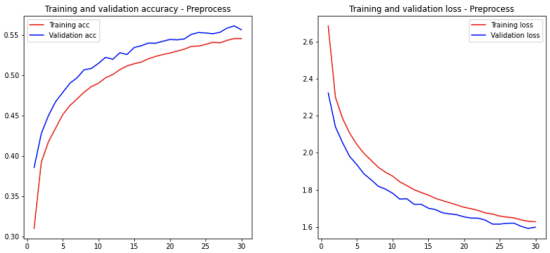


Figure 5. Preprocess

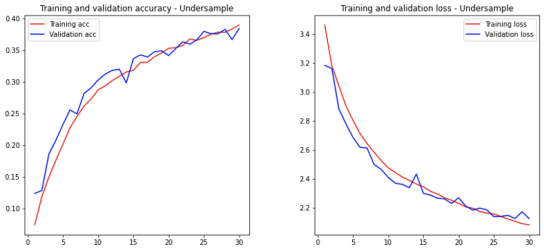


Figure 6. Undersampling

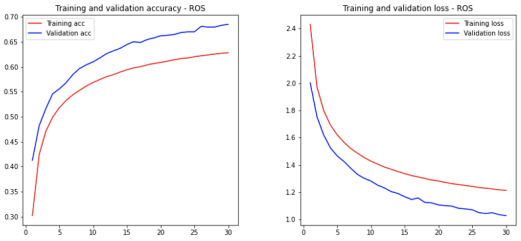


Figure 7. Oversampling-ROS

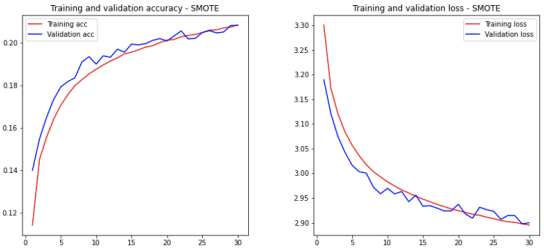
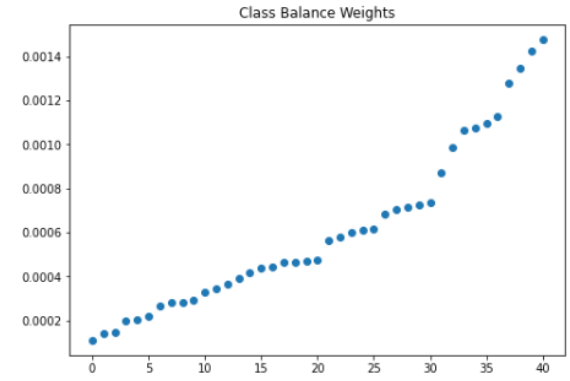
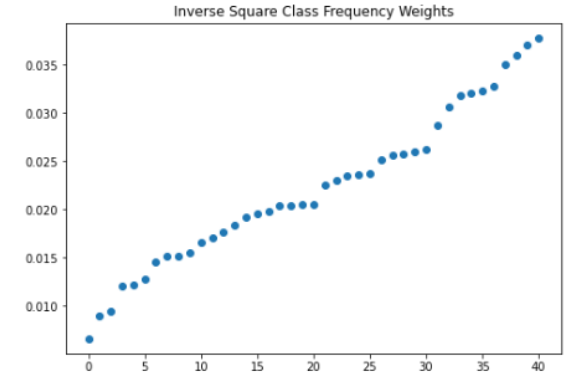
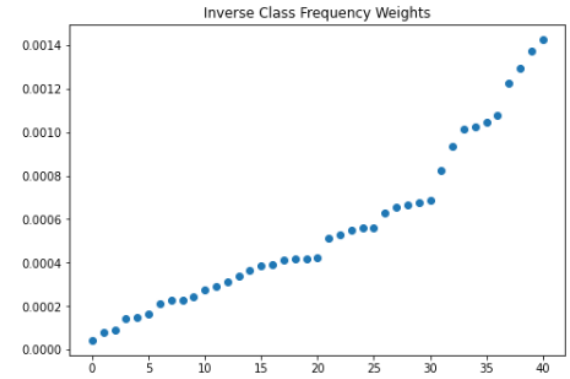


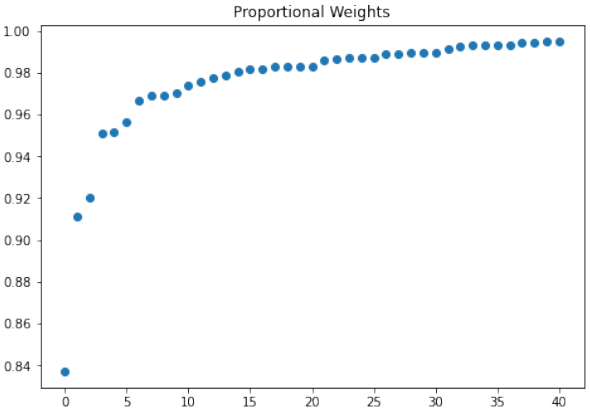
Figure 8. Oversampling-SMOTE

7.1.2 Weights

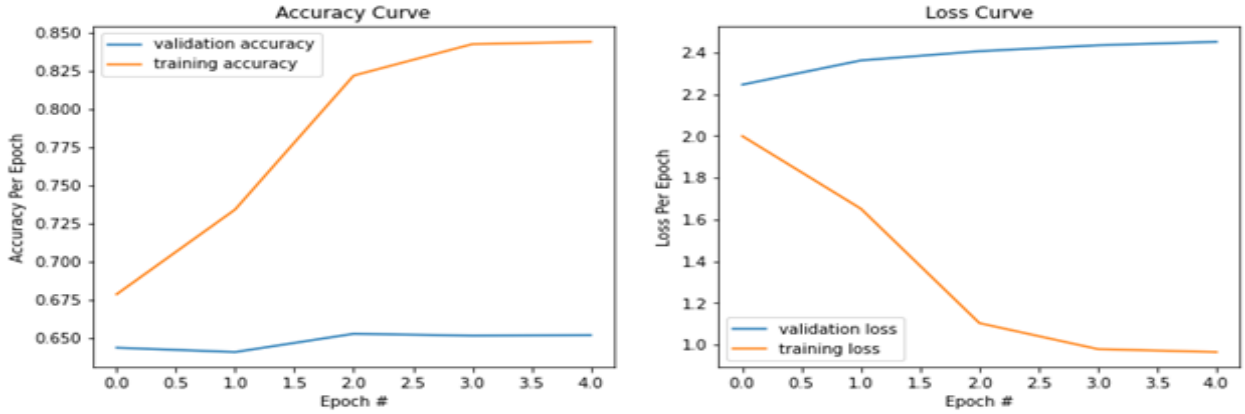




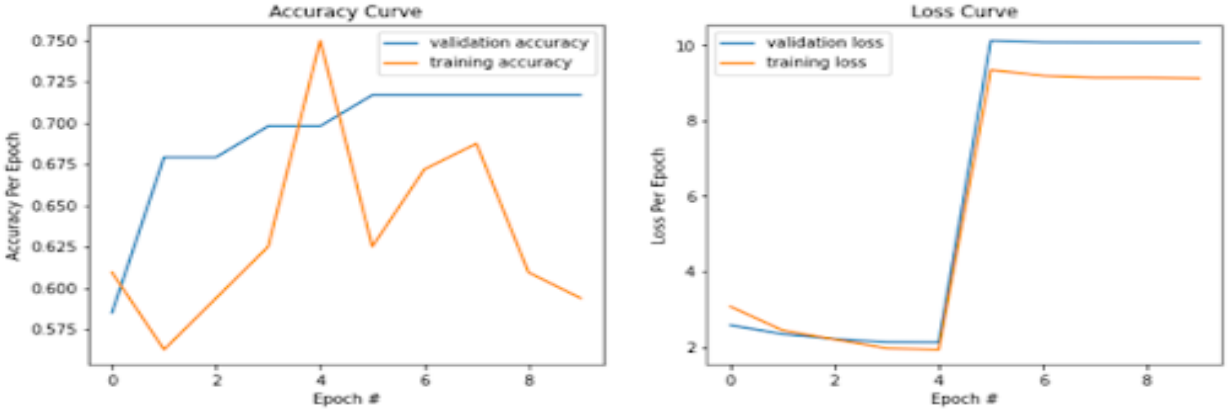




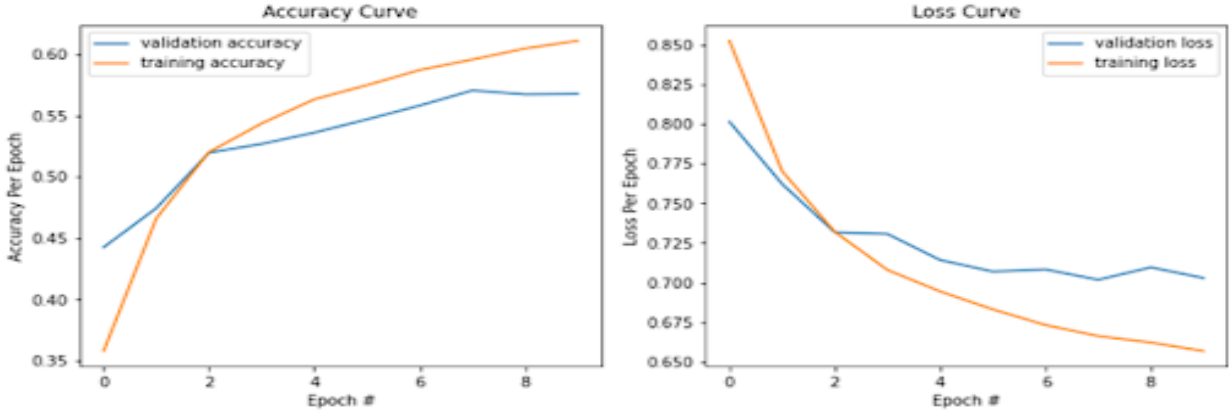
7.1.3 Losses



Overfit LDAM



Overfit mitigation with LDAM DRW + ReduceLROnPlateau + Adam L2 Regularization



Overfit Dice Loss

7.2. Algorithms-ADAMvADAMW

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda \boldsymbol{\theta}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

Figure 9. ADAM v ADAMW

7.3. Tables

Adjustment	Loss			Accuracy		
	Train	Val	Test	Train	Val	Test
Baseline	1.4285	1.4152	1.415	58.82	59.61	59.77
Preprocess	1.6266	1.5984	1.6049	54.55	55.65	55.67
Undersample	2.0814	2.1233	2.0839	38.98	38.48	38.80
Oversample-ROS	1.2132	1.0283	1.0255	62.83	68.52	68.71
Oversample-SMOTE	2.8952	2.8999	2.8973	20.83	20.81	20.94

Table 1. Preprocess and Dataset Manipulation Results

Weights	Loss	Loss			Accuracy		
		Train	Val	Test	Train	Val	Test
ICFW	CE	0.665	2.018	2.89	79.35	60.13	61.33
ISCFW	CE	0.295	2.649	3.197	89.94	61.1	61.12
CB	CE	0.171	3.517	3.958	93.63	60.48	60.81
Prop	CE	1.043	1.263	1.536	69.83	65.18	63.65
ICFW	focal	0.001	0.067	0.067	95.97	59.91	59.97
ISCFW	focal	0.97	1.214	1.332	66.68	63.14	63.32
CB	focal	0.003	0.049	0.05	92.14	61.28	61.47
Prop	focal	17.044	99.049	124.294	87.19	62.05	61.81

Table 2. Weight Results

	LDAM DRW	LDAM (no DRW)	Dice Loss
Training Loss	9.122	1.899	0.624
Training Accuracy	59.4	59.4	65.6
Validation Loss	10.071	2.105	0.685
Validation Accuracy	71.7	77.4	69.8
Test Loss	10.100	2.119	0.687
Test Accuracy	62.8	67.4	51.2

Table 3. LDAM DRW, LDAM (no DRW), Dice loss

Model Parameters	Total Accuracy	Accuracy of Majority Class	Accuracy of Minority Class
Base FastText	70.2	73.1	16.6
25 Epochs, LR = 1.0, wordNgrams =2	73.3	75.4	45.3
Autotuned Base 10 minutes	72.4	76.2	32.4
Autotuned 5 minutes (Minority Emphasized	72.0	76.1	36.6

Table 4. FastText Experiments

Student Name	Contributed Aspects	Details
All	Planning, Implementation, Analysis, Review	Includes: • Attending meetings each week. • Discussing optimal strategies and research summaries. • Individual coding and code review. • Analysis. • Paper reviews.
Al Hogan	Implementation, Analysis, LaTeX Writeup	• Experimented with Bag of Words approach as well as different architectures. • Researched and implemented preprocessing, over-sampling, and under-sampling techniques. • Wrote parts of Introduction and all of Section 3.1. • Helped translate collaborative google-doc into LaTeX.
Anadi Jaggia	Implementation and Analysis	• Researched and implemented two loss methods, namely LDAM DRW and Dice Loss. • Mitigated overfitting that was present in the model after using CE, LDAM DRW and Dice Loss via different learning rate schedulers and optimizers. • HyperParameter tuning for the losses, learning rate schedulers and optimizers. • Wrote about all these findings in Section 3.2.3 to 3.2.7 inclusive and produced graphs for report and appendix. Lastly also wrote about ADAMW (Section 4.1) in the FutureWork section.
Nick Torvik	Implementation and Analysis	• Researched and implemented many loss functions, including several PyTorch defaults. • Researched, implemented, and tuned (where necessary) class weighting techniques to augment loss functions. • Implemented and tuned Focal Loss, adding capability to adjust weighting scheme. • Wrote sections 2, 3.2.1, and 3.2.2
Isaac Sutor	Implementation and Analysis	• Researched alternate architectures/embedding solutions • Trained the FastText Model on the Imbalanced dataset and attempted to tune the hyperparameters • Implemented the introduction of the FastText Embeddings in the baseline architecture • Wrote parts of the introduction/background, all of 3.3, 4.2, and 5.

Table 5. Contributions of team members.