Detecting Bias in Online News Articles using Deep Neural Networks

Christen Ford

April 16th, 2021

1 Introduction

Recent global events including the attack on the US capital building in January 2021, the mass imprisonment and reeducation of the Uyghur people in China as well as the rampant suppression of the pro-democratic movement as well as the Democratic pushes in Malaysia have shown how governments and well-funded private groups can influence public opinion through mass media have shown that it is vital to differentiate between biased media sources and unbiased media sources. There have been many academic papers published in recent years that attempt to provide solutions to this topic such as Aires et al. [2] and Mokhberian et al. [15] have proposed approaches to this that utilize Moral Foundation Theory and Information Theory respectively. The bias towards a particular frame of reference is known as *framing bias* and is what this paper examines.

This paper seeks to provide a solution this problem based on Deep Learning and Neural Networks. As a baseline, this paper will analyze the neural network implementation in terms of three ensemble classifiers: Adaboost, Gradient Boosting and Extremely Random Forests. The neural network presented in this paper will be considered successful if it can outperform the best performing ensemble method. The comparison between the ensemble methods and the neural network will be performed using classification accuracy, F1 score, precision and recall. Noting that: F1 score measures the accuracy of testing, precision measures how many selected instances are relevant and recall measures how many relevant instances were selected. While precision and recall seem like they are the same measurement, they are in fact different from a statistical perspective.

2 Background

An embedding for a particular word w is an n-dimensional vector of real-valued components where each value specifies the distance between other words in a corpus or collection of words. Past methods of computing vectors such as the

Bag-of-Words model and Word2Vec utilize sliding window mechanisms over a document to generate word vectors. While these methods are extremely effective at doing so, they are inefficient in terms of run-time. Modern NLP sentence representations have moved onto the WordPiece embedding model first introduced by Wu et al. in their paper "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation" [24]. WordPiece seeks to provide an answer to an NLP optimization problem by breaking each word into two pieces - "Given a corpus and a number of desired tokens D, the goal of the optimization problem is to select D word pieces such that the resulting corpus is minimal in the number of wordpieces when segmented according to the chosen wordpiece model." WordPiece is used liberally in the state-of-the-art NLP architecture BERT introduced by Devlin et al. [6], in their work "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". This paper utilizes the Word2Vec model to generate word embeddings as the primary advantage of BERT is that it incorporates both left-to-right and right-to-left context into the embeddings which is not needed by the model presented in this paper.

As stated by Simpson in their work, Moral Foundations Theory [20], Moral Foundations Theory (MFT) is an approach to measuring human values, judgments and morality in light of opposing moral categories. In MFT, a person's morals can be summed up to how far they align towards vices or virtues of the Harm, Fairness, Ingroup, Authority, Purity, and General Morality MFT categories. A person's leanings towards the virtues or vices of both categories can be determined mathematically through the use of semantic axes as presented by An et al. in their 2018 work "SemAxis: A Lightweight Framework to Characterize Domain-Specific Word Semantics Beyond Sentiment".

Let a semantic axis S be defined as two vectors of n word embeddings for a single MFT category such that $S^+ = \{s_0, s_1, ..., s_{n-2}, s_{n-1}\}$ and $S^- = \{s_0, s_1, ..., s_{n-2}, s_{n-1}\}$. Then a semantic axis for a given MFT axis m can be computed as:

$$V_m = V^+ - V^- \tag{1}$$

where V^+ and V^- are the average values of the S^+ and S^- vectors respectively. Utilizing a given semantic axis m, Mokhberian et al. [15] define the *framing bias* for a given document D as:

$$B_m^D = \frac{\sum_{d \in D} f_d * s(V_m, d)}{\sum_{d \in D} f_d} \tag{2}$$

Where $s(V_m, d)$ is the cosine similarity between the semantic axis and the embedding for word d and f_d is the frequency of word d in the document D. Lastly, the *framing intensity* of a document along a specific moral axis m can be compute as:

$$I_m^D = \frac{\sum_{d \in D} f_d * (s(V_m, d) - B_m^T)^2}{\sum_{d \in D} f_d} \tag{3}$$

where B_m^T is the baseline *framing bias* for the entire corpus T along a given semantic axis m. B_m^D and I_m^D are generated per semantic axis per document in my corpus and make up one of the embeddings that are fed into the neural network, E_{mft}.

Aires et al. [2] propose an information theoretic approach to determining framing bias in their work, "An Information Theory Approach to DetectMedia Bias in News Websites". In their work, the authors propose a solution to classifying the bias of online news articles based on entropy, mutual information, Cosine Distance, Jaccard Distance and Jensen-Shannon Divergence.

As it relates to the model presented in this paper, let the entire corpus D be represented by a probability mass function $p^{(D)}$ such that $p^{(D)} = (p_1^{(d)}, p_2^{(d)}, ..., p_n^{(d)})$ maps every word in every document in the corpus to the probability of it appearing in any given document in the corpus. Furthermore, let a given document d be represented by a probability mass function $p^{(d)}$ that maps every word w in the document to it's probability of appearing in the document such that $p^{(d)} = (p_1^{(w)}, p_2^{(w)}, ..., p_n^{(w)})$. These probability mass functions are then used to calculate the Cosine Distance, Jaccard Distance and Jensen-Shannon Divergence to serve as embeddings for the neural network.

Given a discrete random variable X with possible outputs $x_1, x_2, ..., x_n$, the entropy of X is defined as $H(X) = -\sum_{i=1}^{n} P(x_i)\, log\,(P(x_i))$. Given this, the Jensen-Shannon Divergence which is used to measure the similaritty between two probability distributions can be determined by:

$$jsd(p,q) = \frac{H(p) + H(q)}{2} - H(\frac{p+q}{2}) \tag{4}$$

noting that in this case, p is the overall pmf for the entire corpus and q is the pmf for a specific document from the corpus.

The next metric, cosine distance is equivalent to the Pearson correlation and represents the proportional angle between two points in a vector space. The cosine distance is determined as:

$$cos(p,q) = 1 - \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2}\sqrt{\sum_{i=1}^{n} q_i^2}} \tag{5}$$

The last information theory metric is the jaccard distance. Utilized in Computer Science to measure the difference between vectors in $\mathbf{R}^n$ spaces, the jaccard distance is calculated as:

$$jac(p,q) = 1 - \frac{\sum_{i=1}^{n} min(p_i, q_i))}{\sum_{i=1}^{n} max(p_i, q_i))} \tag{6}$$

These three metrics are calculated on a per-document basis and make up E_{it} for purposes of being fed into the neural network.

Lastly, sentiment analysis is a branch of NLP that looks at determining the relationships between words and using these relationships to determine the cardinality of the sentence containing the words. Here cardinality is also known as valence and can be rated in terms of positive, negative or neutral valence indicating how emotionally positive, negative or neutral a sentence is respectively.

Again, as mentioned earlier, Google BERT by Devlin et al. [6] is the current state-of-the-art in this area. However, due to time constraints, I decided to not utilize Google BERT and instead use VADER from Hutto and Gilbert [12] instead.

First presented in "VADER: A Parsiminious Rule-based Model for Sentiment Analysis of Social Media Text" by Hutto and Gilbert [12], VADER is "... a simple rule-based model for general sentiment analysis ... [that combines] gold standard lexical features [with] five general rules that embody grammatical and syntactical conventions for expressing and emphasizing sentiment intensity." Vader utilizes a configurable lexicon of annotated English words to construct a semantic model for determining the valence-based intensity of English sentences. Because VADER was built to handle English in a social media context (i.e. a very noisy environment), VADER is extremely flexible in the sentences it can analyze. This makes it a perfect candidate for generating sentiment analysis features for use as embeddings (referred to as E_{sa}) for the neural network presented in this paper.

3 Data Acquisition and Preprocessing

This paper utilizes the *All the News* dataset from kaggle.com [21]. Consisting of 200,000 news articles from 15 different online news outlets, *All the News* incorporates articles from a variety of political bias leanings including: Far–Left, Left, Moderate, Right and Far–Right. In it's raw form, each article consists of an articleid, title, publication, authors, date of publication, year of publication, month (as an integer), url (which is an empty field), and the content of the article. For the purposes of this paper, features are only generated from the content of the article.

Now, the goal of the neural network and the other models I employ in this paper is to correctly classify the articles bias from one of: Far–Left, Left, Moderate, Right and Far–Right. Now, it is important to note that the dataset does not come with bias labels. Utilizing mediabiasfactcheck.com, a reputable (it has been utilized by published academic papers) website that lists the bias of major news outlets, I looked up the bias of each news outlet in the dataset and assigned this bias to the corresponding articles by each publication. For instance, all of the articles that were published under the New York Times were labelled as having a Left bias. This was a compromise, as I did not have the time to manually annotate all 200k articles with bias labels. It is very possible that some articles were mislabelled as it is known that some publications host articles with biases that fall outside the bias of the overall publication however, for the majority of the articles in the dataset, assigning them the bias label of the publication should be fine.

This dataset was downloaded as three separate comma separated value (CSV) files. To process this dataset for use by my classification models, I wrote several scripts that first cleaned the text of each article and then generated E_{mft}, E_{it} and E_{sa} for each of them. It's important to note that as I describe this process, it's not something you can do on a normal computer. My machine has 32 Gigabytes of RAM, of which pre-processing the entire dataset takes 11.5 Gigabytes of RAM because I load the entire dataset at once all into memory using a Pandas dataframe. This is done because Pandas in combination with Numpy [11] and SciPy [22] allow me to apply transformation operations to the text of each article in parallel, therefore resulting in shorter processing time overall.

Feature generation requires the use of word vectors as described in the background section. Aires et al. [2] and Mokhberian et al. [15] utilize a reference corpus in addition to the dataset they are analyzing - the Stanford Global Vectors for Word Representation (GloVe) dataset [17]. I utilize the same reference corpus in this paper (Common Crawl 42B) which consists of word embeddings generated from uncased 42B tokens, 1.9M token unique vocabulary sourced from a combination of Wikipedia pages and Twitter data. The reference corpus is utilized by both Aires et al. and Mokhberian et al. to provide a generalized high-quality corpus to leverage in key calculations of metrics from both papers. To represent the GloVe and All the News word vectors in a format that my pre-processor scripts can work with I utilize the well-known Gensim vector space modelling library [18] which implements functionality for generating word vectors and modelling them in memory.

E_{mft} and E_{it} are generated per article from my own implementation of the corresponding Moral Framing and Information Theory metrics mentioned in the *Background* section. E_{sa} is generated utilizing the VADER implementation provided by the Natural Language Toolkit [4]. I utilize the GloVe word vectors and All the News word vectors where appropriate.

E_{mft} is an embedding consisting of 12 features (2 features for each moral axis defined by the Moral Foundations Theory framework). The first feature is framing bias. This indicates how biased an article is w.r.t. a specific moral axis. Values closer to zero indicate less biased, while values further from zero along the positive or negative axis indicate a positive (virtue) or negative (vice) bias towards the axis respectively. The second feature is the framing intensity which attempts to capture how passionate the article is when speaking to the vices or virtues of a specific moral axis.

E_{it} is an embedding consisting of 3 features generated for each article. The features generated for this embedding utilized what is known as a vocabulary of reference. To generate the vocabulary of reference, I generate the pmf of all the words in the entire *All the News* dataset. Utilizing this pmf, I then calculate the Shannon entropy for each word in the dataset. Sorting the words by entropy in ascending order and taking the top-n words yields the vocabulary of reference. A Shannon entropy metric as close to zero as possible indicates that the information gain from that particular word is high. In essence, the vocabulary of reference is the top-n most important words in the *All the News* dataset. I then utilize this vocabulary of reference along with the word vectors for the GloVe and *All the News* dataset to generate the cosine distance, Jaccard distance and Jensen-Shannon divergence for each article in the dataset thus yielding E_{it}.

E_{sa} is generated by applying the VADER algorithm from Hutto et al. [12] to all the articles in the dataset. The embedding consists of 24 metrics that describe the compound, negative, neutral and positive sentence valences for each sentence in each article. For each article, I compute the minimum, maximum, median, mean, standard deviation and variance valence. These metrics describe the overall shape of the compound, negative, neutral and positive sentence valence distributions for each article and are the metrics utilized by the E_{sa} embedding.

Prior to being fed into the classifiers, I perform min-max normalization on each embedding to bring each feature from each embedding into the [0, 1] range. This is especially important for the neural net architecture proposed in this paper, as neural nets expect input features to be in the range [0, 1]. Feeding data into a neural network outside of this range will prevent the neural network from properly learning the weights for each feature.

Note that training data and testing data is sampled 80/20 from the entire dataset. This sampling attempts to correct for undersampling and oversampling using the SMOTE algorithm followed by cleaning with the ENN algorithm. First introduced by Chawl et al. in the research paper "SMOTE: Synthetic Minority Over-sampling Technique" [5], SMOTE adjusts for this by creating synthetic examples of the minority classes. Per Chawl et al. "The minority class is over-sampled by taking each minority class sample and introducing synthetic examples along the line segments joining any/all of the k-minority class nearest neighbors." [5] The generated synthetic examples cause the target classifier to create larger and less specific decision regions whereas a poorly sampled dataset would produce densely clustered regions that are harder to classify.

First introduced in the journal article "Edited Nearest Neighbor Rule for Improving Neural Networks Classifications", Edited Nearest Neighbors adds an additional rule to the K-Nearest Neighbors algorithm whereby an instance in the clustering space S is removed if it does not agree with the the majority of it's K-nearest neighbors. Alejo et al. state that "This edits out noisy instances as well as close border cases, leaving smoother decision boundaries." [3]. ENN is utilized as a post-cleanup step by the SMOTEENN algorithm to remove noisy examples from the SMOTE-sampled training set.

4 Model

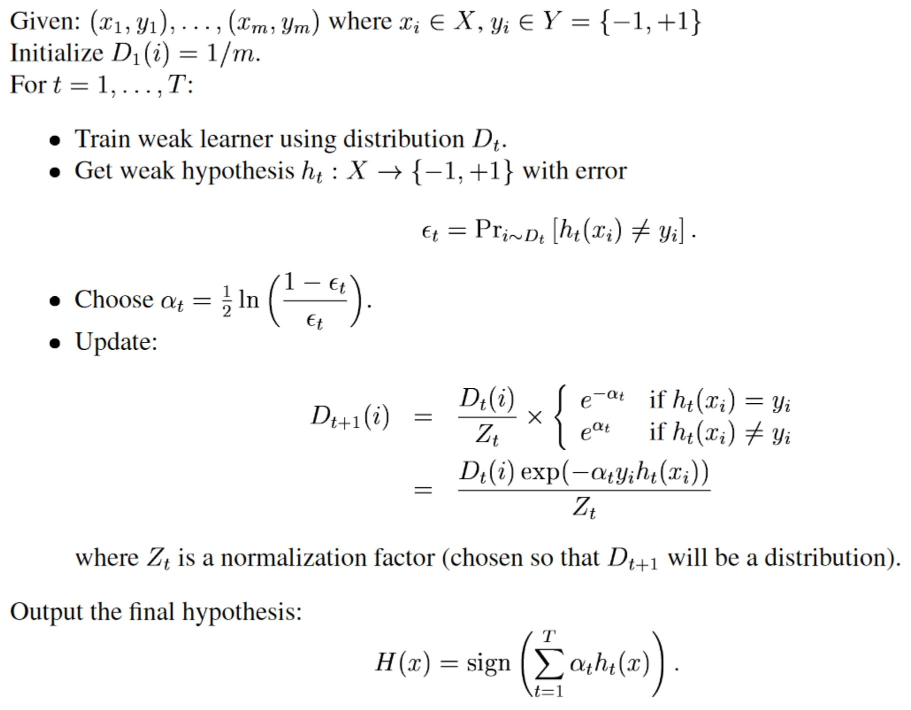
4.1 AdaBoost Ensemble Learning

As described by Freund and Schapire in their 1999 article "A Short Introduction to Boosting" [8], AdaBoost is a meta-estimator that repreatedly fits copies of the same classifier on-top of the input dataset adjusting for incorrectly classified instances such that their weights are given higher preference in the underlying classifier. According to Freund and Schapire, AdaBoost takes as input a training set $(x_1, y_1), ..., (x_m, y_m)$ where each x_i belongs to some domain or instance space X, and each label y_i is in some label set Y. AdaBoost functions by repeatedly calling a wear learner (a classifier that needs boosted) in a series of $t = 1, ..., T$

rounds. AdaBoost thus maintains a distrubution of weights $D_t(i)$ over the course of training, which are initially equal, but are adjusted each round such that the weights of incorrectly classified examples are increased so the weak learner is forced to focus on hard examples from the training set. Overall, the goal of AdaBoost is to make a weak learner less prone to outliers in a dataset – a challenge many classifiers must deal with in some way.

According to Freund and Schapire [8], the weak learner must find a *weak hypothesis* $h_t : X \to \{-1, +1\}$ for the distribution D_t in the binary classification task. Noting that AdaBoost does generalize well to the multi-class classification task as presented by the problem in this paper. A *weak hypothesis* is measured by the error function defined in equation 7.

$$\epsilon_t = Pr_{i \sim D_t}[h_t(x_i) \neq y_i] = \sum_{i:h_t(x_i) \neq y_i} D_t(i) \tag{7}$$



Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in X$, $y_i \in Y = \{-1, +1\}$
Initialize $D_1(i) = 1/m$.
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : X \to \{-1, +1\}$ with error

$$\epsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i] .$$

- Choose $\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$.
- Update:

$$\begin{aligned} D_{t+1}(i) &= \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases} \\ &= \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t} \end{aligned}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right) .$$

Figure 1: The AdaBoost Algorithm as described by [8]

The AdaBoost algorithm is defined in figure 1. α_t can be likened to the learning rate in a traditional classification algorithm. Within the context of AdaBoost, α_t measures the importance that is assigned to the hypothesis h_t. The algorithm iterates the weights each round by increasing the weights for incorrectly classified instances and decreasing the weights for correctly classified instances. The final hypothesis H_t is determined by majority vote of the ensemble. Thus AdaBoost overall is an ensemble learner that focuses a weak learner on hard or outlier training instances. Note that this paper employs the AdaboostClassifier from the sklearn scientific computing library with default parameters.

4.2 Gradient Boosting

First proposed by Jerome H. Friedman in his seminal article "Greedy Function Approximation: A Gradient Boosting Machine" [7], gradient boosting, as the name implies, is a greedy approach to function approximation where the goal is to apply gradient descent based boosting to fitting criterion such as least-squares and mean-squared error. Friedman developed this as a general approach and thus it is applicable to areas such as logistic regression, binary classification, multi-class classification and anywhere else loss functions are applied.

Gradient boosting breaks down into three components:

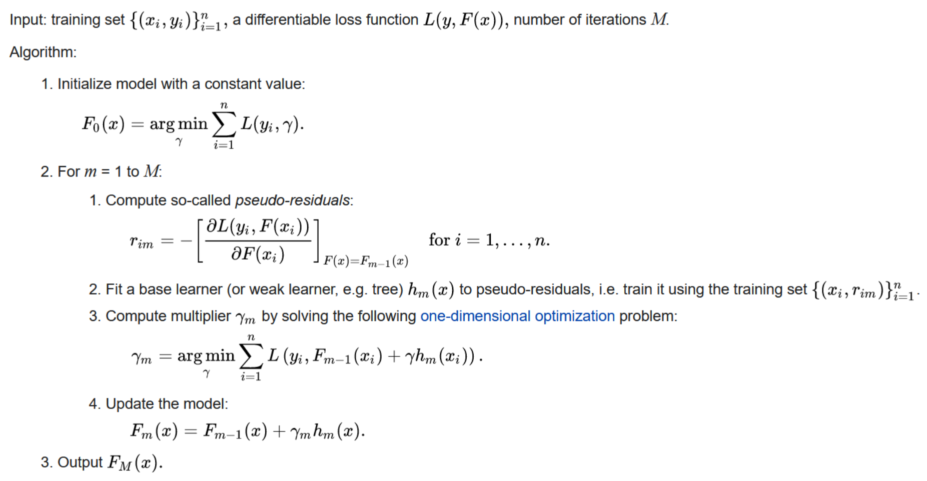
1. A loss function.
2. A weak learner.
3. An additive model.

The loss function depends on the problem being solved and it must be differentiable as it will have gradient descent applied to it during the optimization step. Any standard statistical loss function can be applied with gradient boosting and in fact, as long as it is differentiable, users of gradient boosting can even define their own on a per-problem basis.

The weak learner is always a decision tree in gradient boosting. Similar to how AdaBoost repeatedly changes the weights on classification instances from a weak learner, gradient boosting repeatedly applies a greedy optimization algorithm to find the best splits in the input decision tree. Specifically, gradient boosting will utilize the additive model to add the outputs of subsequent models together and correct the residuals leftover in predictions.

While gradient boosting is running, it will repeatedly add one decision tree at a time to the model, leaving existing trees unchanged. Gradient descent is then applied during the additive process to minimize the results of the loss function when taken across trees. After determining the loss, the gradient descent procedure parameterizes all of the candidate trees that could be added and determines the best tree to add that would move the overall model in the right direction that best minimizes the residual loss. The new best tree is then added to the ensemble. This process repeats until either a fixed number of trees have been added or a certain number of iterations has been reached or a certain training criteria has been reached.

Algorithmically, the Gradient Boosting algorithm is defined in figure 2.



Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

   $$F_0(x) = \underset{\gamma}{\arg\min} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For *m* = 1 to M:
   1. Compute so-called *pseudo-residuals*:

      $$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

   2. Fit a base learner (or weak learner, e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

      $$\gamma_m = \underset{\gamma}{\arg\min} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

   4. Update the model:

      $$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Figure 2: The Gradient Boosting Algorithm as described by the Wikipedia page on Gradient Boosting.

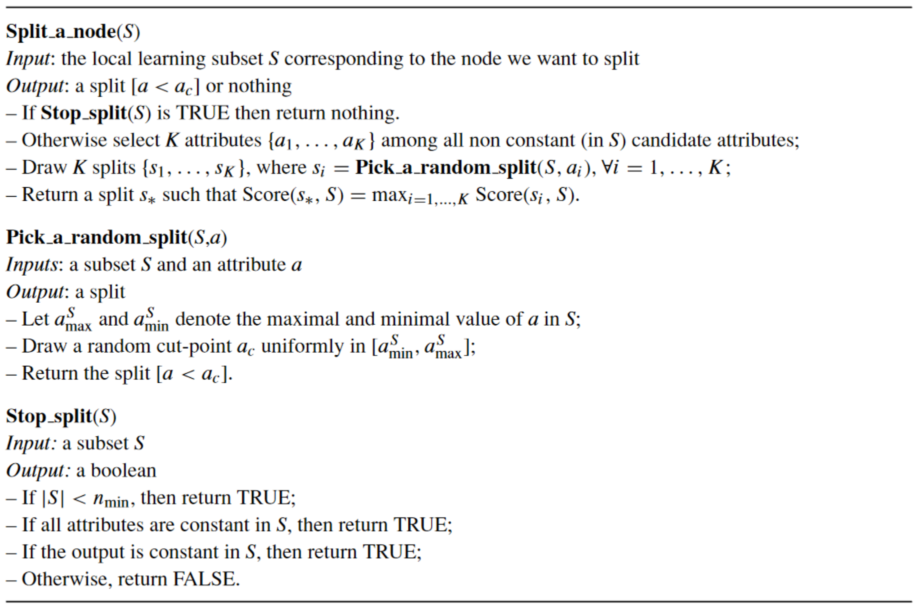
I utilized the algorithm as presented on the Wikipedia page because this algorithm describes what is going on in a fashion that is more comprehensible than the version presented in [7]. The two algorithms are in fact identical. Note that this paper employs the GradientBoostingClassifier class from the sklearn scientific computing library with default parameters.

4.3 Extremely Random Forests

First proposed by Pierre Geurts, Damien Ernst and Lousi Wehenkel in their article "Extremely Randomized Trees" [9], an extremely randomized tree is a decision tree that utilizes a stochastic process to create tree splits. An extremely randomized forest is constructed in the same way that a random forest is in that a subset of candidate features is utilized. Instead of looking for the most discriminative thresholds that classify instances for each candidate feature, an extremely random tree will instead draw thresholds at random for each candidate feature and the best thresholds are thus utilized as the split criterion. This process is in fact quite similar to a Monte-Carlo process and in general is proven to reduce the classification variance of a random forest at the expense of slightly increased bias.

An extremely random forest is a meta-estimator that attempts to fit a number of extremely random tree classifiers on top a various samples of a dataset. The construction of the extremely random forest classifier is guided by averaging in order to improve accuracy and control over-fitting wherein a classifer learns an inherent bias towards a particular training instance and as a result biases towards classifying instances as the over-fitted instance. Extremely random trees are controlled via appropriate choice of a randomization parameter. This parameter is of course situation dependent.

Per Geurts et al. [9], the primary extremely random tree algorithms can be summarized in figure 3. Because the focus of this paper is on the neural network model presented in the next subsection, I will not describe in detail what a decision tree is or how the splits are normally computed.



```
Split_a_node($S$)
Input: the local learning subset $S$ corresponding to the node we want to split
Output: a split $[a < a_c]$ or nothing
– If Stop_split($S$) is TRUE then return nothing.
– Otherwise select $K$ attributes $\{a_1, \dots, a_K\}$ among all non constant (in $S$) candidate attributes;
– Draw $K$ splits $\{s_1, \dots, s_K\}$, where $s_i$ = Pick_a_random_split($S, a_i$), $\forall i = 1, \dots, K$;
– Return a split $s_*$ such that $\text{Score}(s_*, S) = \max_{i=1,\dots,K} \text{Score}(s_i, S)$.

Pick_a_random_split($S$,$a$)
Inputs: a subset $S$ and an attribute $a$
Output: a split
– Let $a^S_{\max}$ and $a^S_{\min}$ denote the maximal and minimal value of $a$ in $S$;
– Draw a random cut-point $a_c$ uniformly in $[a^S_{\min}, a^S_{\max}]$;
– Return the split $[a < a_c]$.

Stop_split($S$)
Input: a subset $S$
Output: a boolean
– If $|S| < n_{\min}$, then return TRUE;
– If all attributes are constant in $S$, then return TRUE;
– If the output is constant in $S$, then return TRUE;
– Otherwise, return FALSE.
```

Figure 3: Extremely random trees as presented in Geurts et al. [9].

Note that the splitting procedure has two parameters: K which represents the number of attributes randomly selected at each extremely random tree node and n_{min} which represents the minimum sample size for splitting a node. This sample size is used to generate an ensemble model. When the classifier must make a prediction, all of the trees in the forest simultaneously come to a prediction regarding an input instance and the output classification is decided by majority vote as is common with ensemble methods.

4.4 The Proposed Model

The model presented in this section underwent extensive redesign as I tested each architecture with various parameters. While the architecture changed, the input and output from the model did not change. Let the political bias classification problem presented in this paper be a multi-class classification problem. Furthermore, let the dataset be split into two sets: a training set T and a validation set V consisting of n and m labelled instances respectively. Additionally, let T and V be split into a number of inputs and outputs, (X_T, Y_T) and (X_V, Y_V) respectively. The overall goal is to train a supervised model on (X_T, Y_T) such that the model can correctly classify all instances of (X_V, Y_V).

As a reminder, input for each classification instance consists of three embeddings: E_{mft}, E_{it} and E_{sa} which correspond to vectors containing metrics for Moral Foundation Theory, information theory and sentiment analysis respectively. Please see the *Background* section for information on how each of these vectors are calculated. Output for each classification instance consists of one of five bias labels: $B = \{\text{Left-Center}, \text{Left}, \text{Moderate}, \text{Right-Center}, \text{RightFar-Right}\}$. The classifier must learn the individual aspects of each bias class from (X_T, Y_T) and correctly apply these characteristics during validation of (X_V, Y_V). Note that there is an imbalance in the dataset in that it does not contain any examples from the Far-Left.

The models presented in the subsequent subsections were constructed using PyTorch [16] and Tensorflow [1]. The implementation is highly parallelized and is setup to utilize the Compute Unified Device Architecture (CUDA) if the device running the model has a dedicated Nvidia graphics card. Each of these models utilize Cross-Entropy Loss as the loss criterion. They also utilize the Adam optimizer to guide weight updates. Cross-Entropy loss is applicable in situations with either multi-class classification (the problem proposed in this paper) as well as multi-label classification (when a single input instance can be multiple outputs).

Cross-Entropy loss is defined in equation 8 as stated in [10]. Note that t is the target vector, and C are the classification classes.

$$CE = -\sum_{i}^{C} t_i log(f(s_i)) \tag{8}$$

In equation 8, t_i and s_i represent the ground-truth and the score calculated by the classifier respectively. Note that the models presented in the subsequent functions do not apply softmax or log softmax to the prediction logits because internally cross-entropy loss within the PyTorch framework applies softmax to them before calculating the loss. Applying the softmax or log softmax function to the output logits of the model before determining loss would result in improper training. Note that in a multi-class classification environment, the output labels from the ground-truth as well as the output labels from the classifier are one-hot encoded (PyTorch constructs the one-hot encoding for the classifier output on the fly when you give the cross-entropy loss function the truth index).

PyTorch applies the cross-entropy loss function per class as seen in figure 4.

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

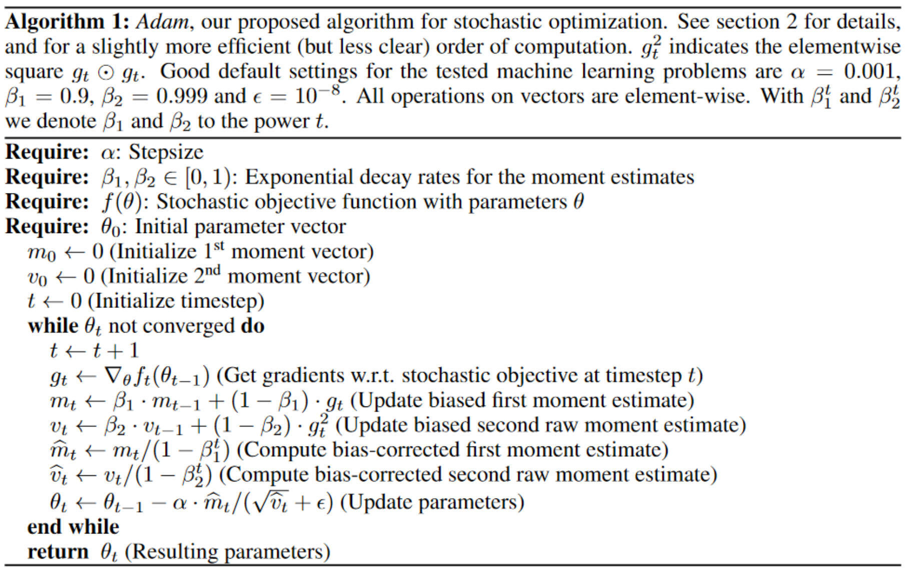
Figure 4: Class-level CLE as calculated by PyTorch.

PyTorch then averages the Cross-Entropy loss across each observation for each input batch (in this paper, the batch size is 64). This results in the loss function defined in figure 5.

$$\text{loss} = \frac{\sum_{i=1}^{N} loss(i, class[i])}{\sum_{i=1}^{N} weight[class[i]]}$$

Figure 5: Averaged CLE as calculated in PyTorch.

First introduced by Diederik P. Kingma and Jimmy Lei Ba in 2014, the Adam optimization algorithm is a first-order gradient-based optimizer capable of optimizing stochastic objective functions based on adaptive estimates of lower-order moments [14]. Adam computes individual adaptive learning rates for different parameters from both first and second moments of the gradients of the objective functions it is optimizing. This model utilizes the Adam optimizer to optimize the Cross-Entropy loss function defined above to train the weights of the proposed neural networks. Adam's algorithm can be seen in figure 6.



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Figure 6: The Adam objective function optimization algorithm as described in [14]

.

Adam accepts four parameters. First, α is the step size and it determines how the first and second moments are calculated. β_1 and β_2 are exponential decay rates that are used to apply decay to the exponential moving averages from the first and second moments (a configurable parameter in PyTorch's implementation of Adam). $f(\theta)$ is the objective function being optimized with parameters θ. Lastly, θ_0 is the initial parameter vector for use by Adam.

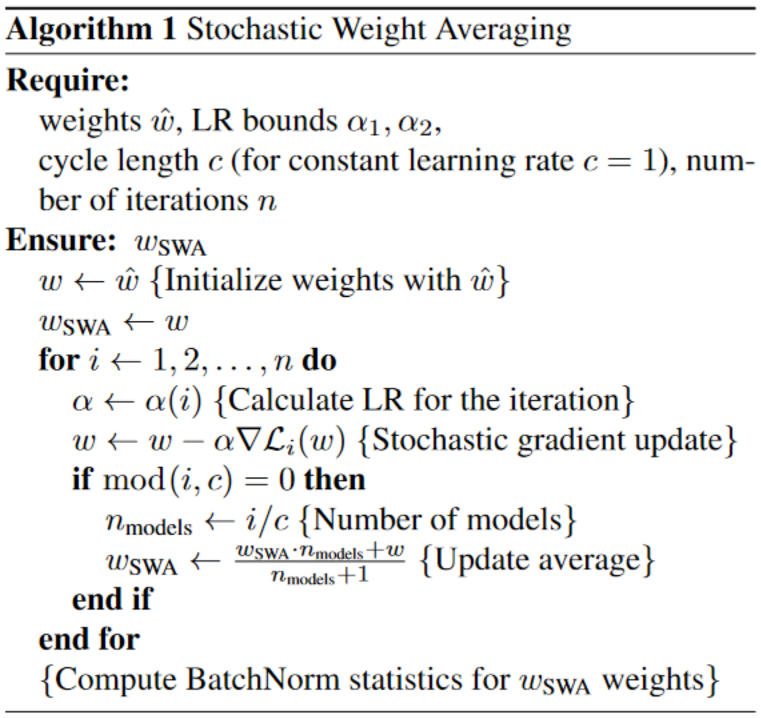
At each time-step, Adam first computes the first and second moment gradients of the objective function. Adam then computes a biased first and second moment estimate. This estimate is then corrected to produce an unbiased first and second moment estimate. These estimates are then used to update Adam's parameter list. The entire point of Adam is to build an exponential moving average of the gradient and squared gradient of the objective function w.r.t. Adam's hyper-parameters. At each time step, these gradients are applied to optimize the parameters of the objective function. Noting that the objective function in a classification context is generally a loss function, this results in a loss function that can overall guide the classification algorithm to produce less and less loss as it's neural network weights are adjusted based on instances from the training space (X_T, Y_T). W.r.t this model, the loss function being optimized is the multi-class cross-entropy loss.

In the following sections it is important to note that I utilized the entire dataset for training and validation. Training and validation was performed on a 80/20 split and running each network consisted of running 100 training iterations followed by a single validation instance. Any input examples in the dataset that contained null or NaN values were dropped from the dataset before it was fed into the neural network model. Note that Adam optimization was only performed on the Linear/Bilinear/Dropout layers. Optimization was not performed on the ReLU activation layers as it does not make sense to optimize an activation function.

Stochastic Weight Averaging (SWA) is similar to the ensemble methods presented earlier in this paper. SWA attempts to optimize the training of a DNN by applying a cyclical learning rate schedule over a series of training epochs to an optimization algorithm such as gradient descent or Adam. The main goal of SWA is to average the weights of the model during each epoch. SWA optimizes the scheduling process of the target optimizer by jumping to and from the minimum/maximum learning rates. This is done to minimize the convex hull surrounding the Gaussian distribution centered around the local optimum weight.

Per Izmailov et al. in their paper "Averaging Weights Leads to Wider Optima and Better Generalization" [13], SWA can be summarized in two meanings: on the one hand, it is an average of SGD weights. On the other, with a cyclical or constant learning rate, SGD proposals are approximately sampling from the loss surface of the DNN, leading to stochastic weights. SWA is applied to the third model in this paper in an attempt to glean more accuracy from the model.

The SWA algorithm can be seen in figure 7. This algorithm accepts a set of weights, a lower and upper learning rate bound, a cycle length and number of iterations. First, the algorithm initializes a set of weights using the weights it received. It then initializes the set of SWA weights. For each iteration, the algorithm then calculates the learning rate for the iteration, performs a stochastic gradient update all of the SWA weights and lastly updates the average weights if the current iteration number is mod 0 of the cycle length. The algorithm then computes and returns the BatchNorm statistics for each of the SWA weights.



Algorithm 1 Stochastic Weight Averaging

```
Require:
  weights $\hat{w}$, LR bounds $\alpha_1, \alpha_2$,
  cycle length $c$ (for constant learning rate $c = 1$), number of iterations $n$
Ensure: $w_{\text{SWA}}$
  $w \leftarrow \hat{w}$ {Initialize weights with $\hat{w}$}
  $w_{\text{SWA}} \leftarrow w$
  for $i \leftarrow 1, 2, \ldots, n$ do
    $\alpha \leftarrow \alpha(i)$ {Calculate LR for the iteration}
    $w \leftarrow w - \alpha \nabla \mathcal{L}_i(w)$ {Stochastic gradient update}
    if $\mathrm{mod}(i, c) = 0$ then
      $n_{\text{models}} \leftarrow i/c$ {Number of models}
      $w_{\text{SWA}} \leftarrow \frac{w_{\text{SWA}} \cdot n_{\text{models}} + w}{n_{\text{models}} + 1}$ {Update average}
    end if
  end for
  {Compute BatchNorm statistics for $w_{\text{SWA}}$ weights}
```

Figure 7: The Stochastic Weighted Averaging algorithm as described in [13].

4.5 Iteration One

The first iteration of the neural network in presented in this paper was based on the work conducted by Clemen Winter in ”Mastering Real-Time Strategy Games with Deep Reinforcement Learning: Mere Mortal Edition” [23] where he rigorously developed a DNN capable of playing a simplified real-time strategy game CodeCraft. The architecture is presented in figure 8.

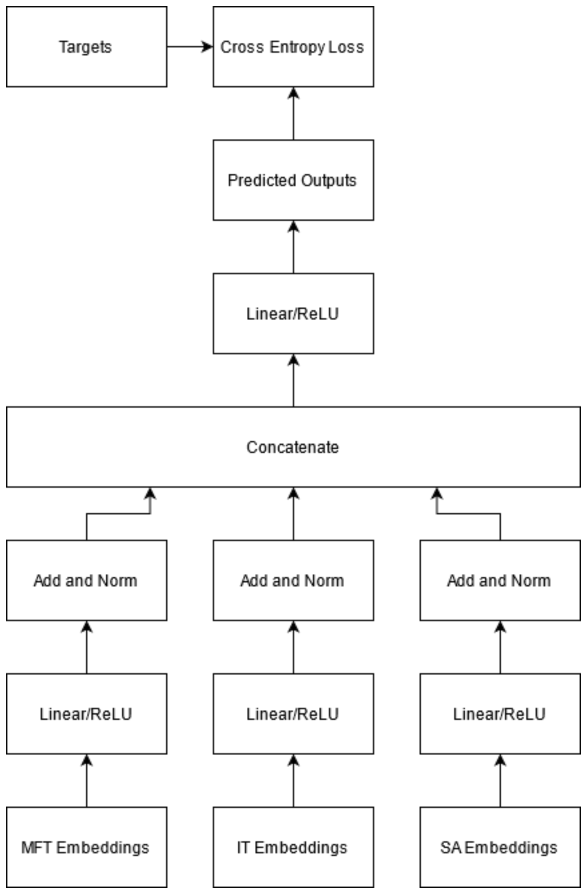


Figure 8: The first iteration of the proposed model.

I designed this model such that the input was still split into the three separate embeddings E_{mft}, E_{it} and E_{sa}. This architecture consisted of a fully connected deep neural network with four hidden layers. The first hidden layer consists of three Linear/ReLU modules that accepted as input 12, 3 and 24 features respectively (for the three embeddings). This layer output the same amount of output to the next layer consisting of a tensor addition and normalization operation. The addition of this layer consists of adding the output of the first hidden layer to the original embeddings (though it is not shown in the figure). This produced output of 24, 6, and 48 values tensors respectively. The next hidden layer just concatenates the three tensor it receives from the previous layers into a single 1D tensor consisting of 78 values. This is then passed as input to the next hidden layer consisting of a Linear/ReLU activation function that outputs a single 1D tensor consisting of 78 values. These values are then arg-maxed to retrieve the classification index for use in the cross-entropy function.

4.6 Iteration Two

The second iteration of this model was a stripped down variant of the first model. In this model I attempted to reduce complexity of the network and therefore increase accuracy as it can be shown that for many classification problems, even complex problems, you only need a single hidden layer. This model is shown in figure 9.

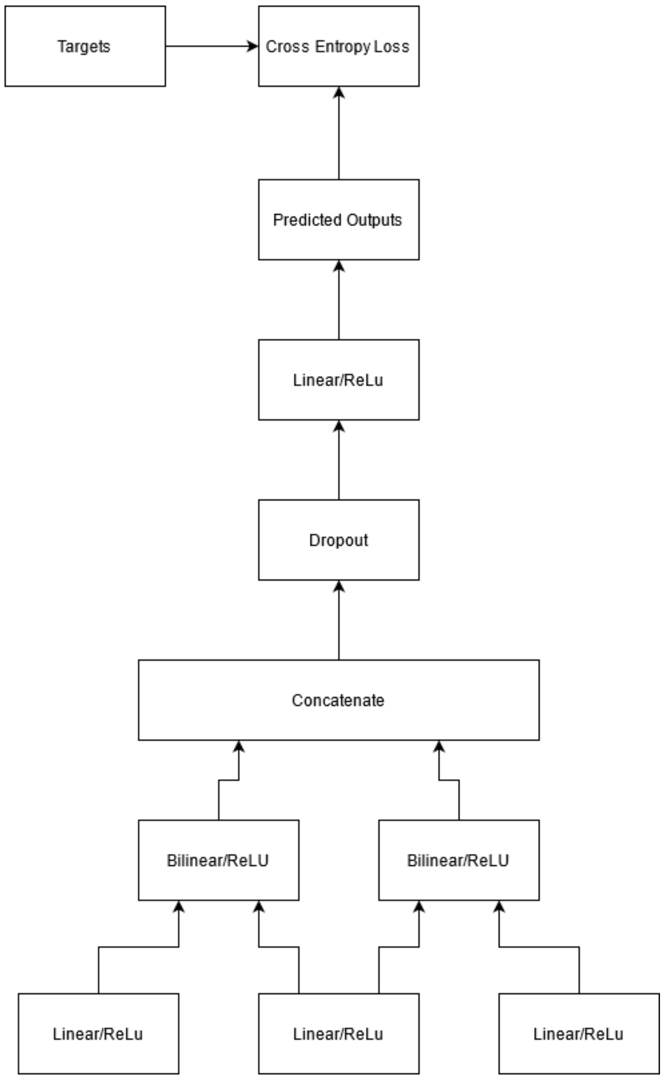


Figure 9: The second iteration of the proposed model.

This iteration still treats the embeddings as seperate inputs to three different Linear/ReLU activation functions. However, this iteration saw the replacement of the Add/Norm layer with two Bilinear/ReLU layers that accept as input the output of the MFT/IT and IT/SA Linear/ReLU activation layer respectively. The first hidden layer accepted as input 12/3/24 features respectively and output 100/100/100 features respectively. The Bilinear/ReLU layer accepted as input 200/200 input features respectively and output 100/100 features respectively. The concatenation layer accepts 100/100 features as input and outputs a single 200 feature tensor. The dropout layer accepts a 200 feature tensor and was configured to randomly drop inputs with a probability of 0.25% and outputs 100 features. The next Linear/ReLU layer accepts 100 features and outputs 5 features.

4.7 Iteration Three

The third iteration feeds all features into the neural network as a single embedding of 39 features. Likewise, the first layer has been replaced by a single Linear/ReLU activation function that accepts as input 39 features and outputs 300 features. The next layer is a Linear function that accepts as input 300 features and outputs 6 features. This simplistic model, as will be seen in the results section, outperforms the other two models by a strong margin. The model itself can be seen in figure 10.

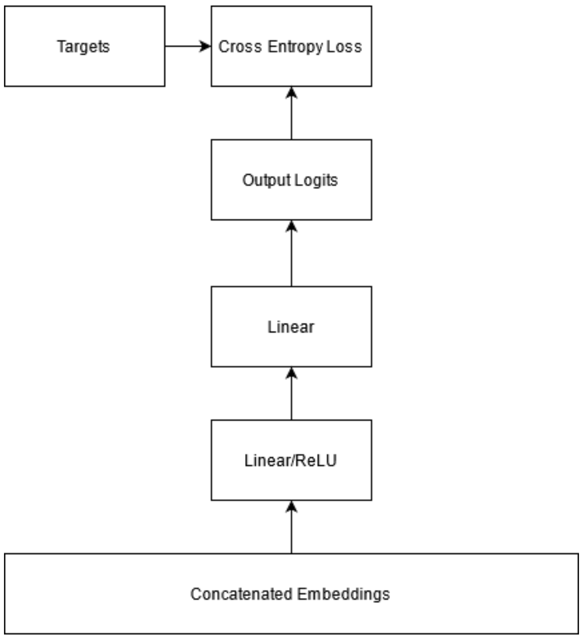


Figure 10: The third iteration of the proposed model.

This model, as I hinted to earlier, performed the best because it is the least complicated. For a majority of classification problems, even tough problems, they can be solved with a single hidden layer neural network.

4.8 SWA and STA

I also implemented two different versions of model three. The first modification implements Stochastic Weighted Averaging (SWA) while the second modification replaces the ReLU activation function with a Stochastic Threshold Activation (STA) function. For a description of SWA please see the background section.

Introducing randomness to an activation function is not a new idea. Shridhar et al. introduce a probabilistic activation function in "ProbAct: A Probabilistic Activation Function for Deep Neural Networks" [19] that maps the well-known Rectified Linear Unit (ReLU) activation function to a stochastic regularizer. Their results are just as effective as ReLU and in fact improve upon it.

This model re-implements the PyTorch threshold activation function. Specifically, after each forward pass of the module, the threshold value is recalculated utilizing a uniform random walk process as defined in listing 1. Note that the forward function as seen on the line containing $y = forward(i)$ is the forward function as defined by the PyTorch Threshold module.

Listing 1: The STA Algorithm

```
Algorithm STA(i, t, v, s):
  // Input:
  //   i: The input tensor.
  //   t: The threshold value.
  //   v: The value to replace with.
  //   s: The step size.
  // Output:
  //   x: The adjusted threshold value.
  //   y: The output Tensor.
  p = uniform(-s, s)

  if t + p > 1:
    x = 1
  elif t + p < 0:
    x = 0
  else:
    x = t + p

  y = forward(i)
  return x, y
```

The intuition in listing 1 is to allow the threshold value to fluctuate up and down based on a provided step size s. The fluctuation is based on a one-dimensional random walk process that applies the output of a uniform distribution $X \sim Uniform(-s, s)$ to a threshold activation value t. p is then added to the prior threshold value t to generate the new threshold value x.

5 Experimental Results

This section discusses the results of running the benchmark ensemble classifiers as well as all three models. Note that the ensemble classifiers and models are best compared from a holistic viewpoint. While accuracy is certainly important for any classifier, precision, f1 score and recall are all also important metrics. These metrics are layed out in tables 1 and 2.

All training and validation was performed on a machine with the following parameters (all at stock, factory settings): 8-Core AMD Ryzen 7 3700X, 32GB DDR4 RAM 3200Mhz, Asus RTX 3080 FTW3 Ultra, 1TB Samsung EVO 970 Pro NvME SSD, 1TB Samsung 860 EVO SATA SSD. Training and validation hardware is presented here for reproducability of this study.

Classifier	Accuracy	F1-Score	Precision	Recall
AdaBoost	0.4370	0.4370	0.4370	0.4370
Grad. Boos.	0.4946	0.4946	0.4946	0.4946
Extr. Rand. Fore.	0.4996	0.4996	0.4996	0.4996

Table 1: Metrics for the Ensemble Classifiers

As can be seen in tables 1 and 2, the AdaBoost and Gradient Boosting classifiers perform roughly similar to the best performing model. I did not expect this outcome especially since the AdaBoost and Gradient Boosting in sklearn utilize a decision tree as the weak classifier. I expected the accuracy of the Extremely Random Trees classifier to be much higher than it was. Generally speaking, decision trees are extremely good classifiers. In fact, prior iterations of the implementation of this project saw an accuracy of 90% for the extremely random trees classifier. The data presented here showcases the accuracy of the classifier at default settings after I corrected how I was calculating two of the features.

Model	Accuracy	Loss	F1-Score	Precision	Recall
M1	0.4425	2675.4782	0.4397	0.4397	0.4397
M2	0.4603	0.2643.4456	0.4603	0.4603	0.4603
M3	0.5114	554.5768	0.5117	0.5117	0.5117
M3: SWA	45.9100	624.5940	0.4591	0.4591	0.4591
M3: STA	0.3933	705.0575	0.3930	0.3930	0.3930

Table 2: Metrics for the Neural Network Models

The results presented in table 2 showcases the principle that I have stated multiple times in this paper that simpler neural networks often work better for classification problems. Loss can be interpreted as how well the model is doing w.r.t. learning the weights needed to model the dataset. Generally, the lower the loss value, the better. It should be made known that the more complicated a DNN is (the more layers there are), the harder it is for gradient descent or any other optimization algorithm to optimize the weights across the network. A neural network with a single hidden layer works well with this task because there is only one layer of weights to update and as a result the optimization function knows exactly which parameters affect the learning process. Effectively, the more layers there are in a network, the more parameters the optimization algorithm has to optimize, this is known to reduce the overall effectiveness of the optimization algorithm.

It should be noted that with both the ensemble learners and the neural network models, the precision, recall and f1-scores are all the same. F1-score is essentially a combination of both precision and recall. To the best of my ability, I want to say that this oddity is due to the way the computation of these metrics are performed by the sklearn library.

A confusion matrix shows a heatmap of the ground truth to the predicted outputs of a classifier. The higher the value, the more testing instances the classifier predicted as the corresponding specific ground truth class. The desired outcome for a classifier is to have perfect classification down the left-to-right diagonal as partially seen in figure 13. The confusion matrices for the three ensemble learners can be seen in figures 11, 12, and 13.

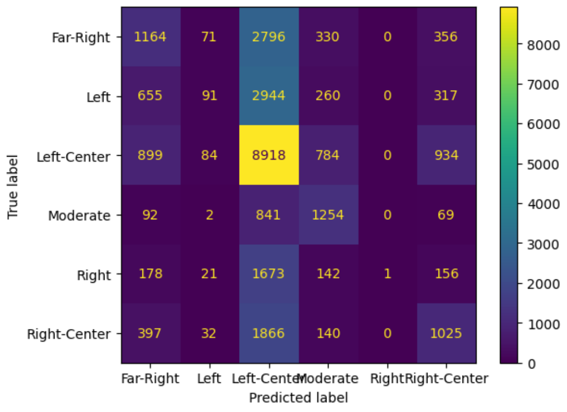
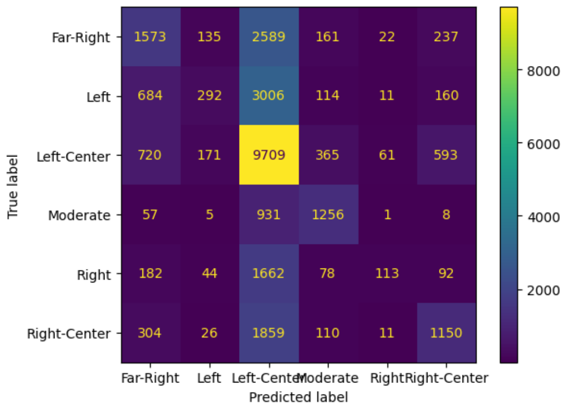
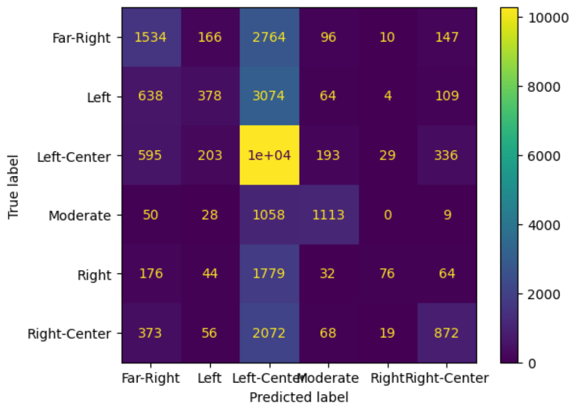


Figure 11: The confusion matrix for the AdaBoost ensemble learner.



True label \ Predicted label	Far-Right	Left	Left-Center	Moderate	Right	Right-Center
Far-Right	1573	135	2589	161	22	237
Left	684	292	3006	114	11	160
Left-Center	720	171	9709	365	61	593
Moderate	57	5	931	1256	1	8
Right	182	44	1662	78	113	92
Right-Center	304	26	1859	110	11	1150

Figure 12: The confusion matrix for the Gradient Boosting ensemble learner.



True label \ Predicted label	Far-Right	Left	Left-Center	Moderate	Right	Right-Center
Far-Right	1534	166	2764	96	10	147
Left	638	378	3074	64	4	109
Left-Center	595	203	1e+04	193	29	336
Moderate	50	28	1058	1113	0	9
Right	176	44	1779	32	76	64
Right-Center	373	56	2072	68	19	872

Figure 13: The confusion matrix for the Extremely Random Trees ensemble learner.

The above confusion matrices showcase that the issue does indeed lie with the feature set. Another contributing factor is that the Left-Center class significantly outweighs the other classes in terms of overall population from the dataset. I attempted to correct for this by undersampling the majority class and oversampling the minority classes using the SMOTEENN algorithm. The application of SMOTEENN produced worst results than randomized sampling.

6 Conclusion

This was an extremely challenging task for me to complete. As stated in my proposal, this was intended as a mini-application of the research I was intending to conduct as a part of my dissertation research. Overall, this project yielded mixed results towards determining if this is a good approach or not. On one hand, the neural network models vastly underperformed from what I expected. On the other hand, the extremely random forests classifier proved that this task could be accomplished with a much simpler (yet still stochastic) model.

My best guess as to why the neural network models underperformed was either because a.) I implemented the calculation of the MFT, IT and SA metrics incorrectly or b.) I did not design the architecture of the models correctly in the first place. I double-checked my implementation of the metrics, so I do not believe (a) is the problem. So, I strongly suspect that it the cause of the poor performance was due to issues with the model. This was my first time working with PyTorch and Tensorflow and I had to figure everything out on the fly with what is an extremely complicated library. I'm actually surprised I was able to get the models implemented at all as there was a point where I had considered dropping the neural network portion of the project altogether and instead using just the ensemble classifiers.

For future work, I would like to find more multi-class classification models in the literature and apply what they have done to my work. I am of the opinion that implementing a custom loss function that incorporates the three categories of metrics would also greatly assist the classification process. I would have done so for this project but frankly, I ran out of time.

References

[1] Martın Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] Victoria Patricia Aires et al. "An Information Theory Approach to Detect Media Bias in News Websites". In: *WISDOM 20: Workshop on Issues of Sentiment Discovery and Opinion Mining*. Association for Computing Machinery, Aug. 2020.

[3] R. Alejo et al. "Edited Nearest Neighbor Rule for Improving Neural Networks Classifications". In: *Advances in Neural Networks - ISNN 2010*. Ed. by Liqing Zhang, Bao-Liang Lu, and James Kwok. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 303–310. ISBN: 978-3-642-13278-0.

[4] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc., 2009.

[5] N. V. Chawla et al. "SMOTE: Synthetic Minority Over-sampling Technique". In: *arXiv e-prints*, arXiv:1106.1813 (June 2011), arXiv:1106.1813. arXiv: `1106.1813 [cs.AI]`.

[6] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: `10.18653/v1/N19-1423`. URL: `https://www.aclweb.org/anthology/N19-1423`.

[7] Jerome H. Friedman. "Greedy function approximation: A gradient boostingmachine." In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. DOI: `10.1214/aos/1013203451`. URL: `https://doi.org/10.1214/aos/1013203451`.

[8] Yoav Fruend and Robert E. Schapire. "A Short Introduction to Boosting". In: *Journal of Japanese Society for Artificial Intelligence* 14.5 (1999), pp. 771–780.

[9] Pierre Geurts, Damien Ernst, and Louis Wehenkel. "Extremely Randomized Trees". In: *Machine Learning*. Vol. 63. Springer Science, 2006, pp. 3–42. DOI: `10.1007/s10994-006-6226-1`.

[10] Raúl Gómez. May 2018. URL: `https://gombru.github.io/2018/05/23/cross_entropy_loss/`.

[11] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`. URL: `https://doi.org/10.1038/s41586-020-2649-2`.

[12] Clayton J. Hutto and Eric Gilbert. "VADER: A Parsimonious Rule-Based Model for Sentiment Analysis of Social Media Text." In: *ICWSM*. The AAAI Press, 2014. URL: `http://dblp.uni-trier.de/db/conf/icwsm/icwsm2014.html#HuttoG14`.

[13] Pavel Izmailov et al. "Averaging Weights Leads to Wider Optima and Better Generalization". In: *arXiv e-prints*, arXiv:1803.05407 (Mar. 2018), arXiv:1803.05407. arXiv: `1803.05407 [cs.LG]`.

[14] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv e-prints*, arXiv:1412.6980 (Dec. 2014), arXiv:1412.6980. arXiv: `1412.6980 [cs.LG]`.

[15] Negar Mokhberian et al. "Moral Framing and Ideological Bias of News". In: *Social Informatics* (2020), pp. 206–219. DOI: `10.1007/978-3-030-60975-7_16`.

[16] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[17] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: `http://www.aclweb.org/anthology/D14-1162`.

[18] Radim Rehurek and Petr Sojka. "Gensim–python framework for vector space modelling". In: *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic* 3.2 (2011).

[19] Kumar Shridhar et al. "ProbAct: A Probabilistic Activation Function for Deep Neural Networks". In: *arXiv e-prints*, arXiv:1905.10761 (May 2019), arXiv:1905.10761. arXiv: `1905.10761 [cs.LG]`.

[20] Ain Simpson. "Moral Foundations Theory". In: *Encyclopedia of Personality and Individual Differences*. Ed. by Virgil Zeigler-Hill and Todd K. Shackelford. Cham: Springer International Publishing, 2017, pp. 1–11. ISBN: 978-3-319-28099-8. DOI: `10.1007/978-3-319-28099-8_1253-1`. URL: `https://doi.org/10.1007/978-3-319-28099-8_1253-1`.

[21] Andrew Thompson. *All the News*. 2017. URL: `https://www.kaggle.com/snapcrack/all-the-news`.

[22] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: `10.1038/s41592-019-0686-2`.

[23] Clemen Winter. *Mastering Real-Time Strategy Games with Deep Reinforcement Learning: Mere Mortal Edition*. Mar. 2021. URL: `https://clemenswinter.com/2021/03/24/mastering-real-time-strategy-games-with-deep-reinforcement-learning-mere-mortal-edition`.

[24] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *arXiv e-prints*, arXiv:1609.08144 (Sept. 2016), arXiv:1609.08144. arXiv: `1609.08144 [cs.CL]`.