Where does it belong? Building A Text Classification System

Murali Manohar Kondragunta **Xi Yu** **Duygu Bayram**
{5397294, 5425786, 5416752}

Abstract

Text classification is important for businesses and social scientists to quickly gather information about public opinion. To build such a classification system, we start by evaluating the task difficulty by building a baseline with Long Short-Term Memory networks. Later, we evaluate different pre-trained language models and select the best one. We found that the BERT language model performs the best on test data among its variants like RoBERTa, DeBERTa, DistilBERT, and ALBERT.

1 Introduction

As the customer base of a company grows, it becomes increasingly hard to manually track and address each complaint/issue. This mandates the use of an automated system that can classify incoming complaints and forward them to the relevant department. Natural Language Processing has come a long way (Kowsari et al., 2019) in addressing this problem i.e text classification. Text classification with reviews can be further helpful for businesses as it can help them track customer satisfaction and perform competitor research. More complex applications of this task with different datasets can have other social impacts such as hate speech detection. It can further aid some social science research to analyze public opinion on certain issues and topics.

We have previously experimented with earlier approaches (Li et al., 2022) of machine learning applications without neural networks. These models employ a document-level approach which assumes a single review only contains an opinion on one entity and is often classified into positive or negative. There are more sophisticated implementations where the review is analyzed on a sentence level and includes a neutral sentiment to differentiate subjectivity from objectivity, and an aspect/feature-level approach where a review is recognized to have different entities and may contain different sentiments (Kowsari et al., 2019). Currently, classification tasks are most commonly performed using various Neural Network models (Li et al., 2022).

In our case, we aim to build a system that can categorize a review into any of the following categories namely, ”music”, ”dvd”, ”camera”, ”software”, ”health”, and ”books”. To this extent, we build a text classification system for category detection. We approach the problem in two steps:

1. We build a baseline with the Long Short-Term Memory (LSTM) model to estimate the difficulty of the problem. To arrive at the best model, we evaluate different architectural and hyperparameter-related decisions in this phase.

2. In the second phase, we evaluate the effectiveness of the pre-trained language model, BERT, on the task. Specifically, we finetune the language model on the downstream task, in this case, category detection. To achieve this, we remove the top layer of the pre-trained language model which is related to the pre-training task, and replace it with a new layer with random weights (which are later learned). In the process of finetuning, we experiment with different hyperparameters like the number of epochs, learning rate, etc. Later, we also evaluate successor models of BERT like RoBERTA and DeBERTA, which are known for better performance, and DistilBERT and ALBERT, which are known for faster processing.

From the experiments, we observed that

RoBERTA performs better in comparison to the other algorithms.

2 Related Work

Minaee et al. (2021) summarized the developmental progression of computational architectures used for text classification tasks under various subcategories. For our purposes, we will look at four: Feed Forward Neural Networks (FFNNs), Recurrent Neural Networks (RNNs), Attention, Transformers, and Pre-trained Language Models (PLMs).

2.1 Feed Forward Neural Networks (FFNNs):

FFNNs are the first-developed simplest deep learning models. They have a basic neural network architecture and are trained traditionally through weight calculations and backpropagation for weight adjusting. The input is often a word embedding, which is most commonly either word2vec (Mikolov et al., 2013) or GloVe (Pennington et al., 2014). One of the most-known FFNN models is the Multi-Layer Perceptron (MLP), a typical FFNN with multiple layers. For classification tasks, MLPs are trained traditionally and the classification is performed on the last layer's output. Deep Average Network (DAN) (Iyyer et al., 2015) is one such model built on this system and shows notable performance on syntactic understanding (Minaee et al., 2021). fastText (Joulin et al., 2016) is another similar model inspired by DAN that uses a bag of n-grams in addition to a bag of words. This further improves the model's performance on word order.

2.2 Recurrent Neural Networks (RNNs):

In contrast to FFNNs, which treat text as a bag of words, RNNs have a sequential approach. As such, these models perform better on dependencies due to the addition of a system that accounts for the previous context. While typical RNNs cannot outperform FFNNs in text classification tasks (Minaee et al., 2021), LSTM (Hochreiter and Schmidhuber, 1997) models do with their improved memory handling. Multi-timescale LSTMs (MT-LSTMs) (Liu et al., 2015) are improved LSTM architectures that account for varying timescales, through splitting the traditional LSTM hidden states into groups which are then responsible for separate time points. These models have been shown to achieve better performance on classification tasks (Minaee et al., 2021). Finally, Bidirectional LSTMs (Zhou et al., 2016a) are developed to better model text features by integrating a training stream running opposite to the traditional LSTM into the existing architecture.

2.3 Attention:

The addition of attention had a large impact on language models, as natural language itself is typically complex and the elements in a given input do not carry the same importance for learning a specific task. Attention, through the addition of importance weights, allows for different levels of reliance on different types of elements going into the model (Minaee et al., 2021). One of the best-performing attention models for text classification is the hierarchical attention network (Yang et al., 2016). The model is best selected for data containing documents with hierarchical structures as it relies on mirroring the structures found in the data. It is considerably robust as it includes word-level and sentence-level attention and has shown noticeable performance on appropriate tasks (Minaee et al., 2021). An improvement to this model is proposed by (Zhou et al., 2016b), by using LSTMs to handle cross-lingual classification (Minaee et al., 2021). Finally, self-attention networks also prove to be an impactful improvement in this area (Shen et al., 2018). This allows for an attention structure that is multi-dimensional and directional (Minaee et al., 2021).

2.4 Transformers and PLMs:

As reviewed, RNNs and RNN-based models such as LSTMs have brought notable improvements to NLP tasks, including text classification. However, such models have their own limitations, the main one being that they are strictly sequential, followed by vanishing and exploding gradient issues (Minaee et al., 2021). To solve these issues, Vaswani et al. (2017) built Transformers by making use of the developments in incorporating attention. To this end, they develop a model that contains a self-attention mechanism, allowing for parallel training and attending equally to all the words in the sentence. This in turn makes it possible to use bigger models and larger data, as the model is not slowed down by the sequential process (Minaee et al., 2021). Transformers also make way for the development of Pre-trained Language Models (PLMs) which provide a considerable impact on the text classification tasks as it allows for the use of the models that have been trained on large amounts of data (Minaee et al., 2021).

3 Methodology

3.1 Data

We used an English reviews dataset with 6,000 individual reviews labeled by category and sentiment. Based on our manual overview, we believe the data was collected from Amazon buyer reviews. The categories are namely, ”music”, ”dvd”, ”camera”, ”software”, ”health”, and ”books”. The sentiment is either positive or negative, titled ”pos” or ”neg”. Since the dataset has only 6,000 entries, we split the dataset into 80:10:10 train, dev, and test split. Table 1 shows the distribution of each class. As the dataset was already tokenized and lowercased, we did not perform any preprocessing steps.

	Train	**Dev**	**Test**
Book	798	91	104
Camera	796	100	92
DVD	826	94	92
Health	660	113	104
Music	832	100	95
Software	778	102	114
Total	4799	600	601

Table 1: Distribution of the instances in the dataset per class. The first column denotes six categories those reviews belong to. The next three columns represent the three sets we split the original data set into, the training set, the development set, and the test set.

3.2 Approach

We approach the problem in two phases. In the first phase, we build a baseline with LSTM. Specifically, we evaluate different architectural and hyperparameter-related decisions in this phase. In the second phase, we repeat the same evaluation procedure for the pre-trained language model, BERT. In addition to it, we also evaluate different pre-trained language models like BERT, RoBERTA, etc.

3.2.1 Baseline LSTM

The hyperparameter and architectural decisions in the first phase are listed below. It is important to note that each setting is tested in isolation, i.e, only one setting is changed at a time and everything else is set to defaults, unless otherwise specified:
(Trainable = False, Dense layer after embeddings = False, 0 LSTM, 0 units, dropout = 0, recurrent dropout = 0, Adam = 0.01, bidirectional LSTM = False)

1. **Trainable embeddings:** Pre-trained embeddings capture the semantic relations among words and integrating them into the models has proven (Jurafsky and Martin, 2000) to improve the model’s performance on downstream tasks. For this reason, in most cases, embeddings are kept constant throughout training. Although beneficial, one drawback of embeddings is that they do not support polysemy, i.e, each word has one only vector despite its polysemy. This can hurt a model especially when the downstream task’s domain is different from that of the corpus the embeddings are trained on. In such cases, it is beneficial to update the embeddings. So, we check both the settings, trainable and constant, to see which one fits our downstream task.

2. **Dense layer between embeddings and LSTM:** In the previous section, we discussed keeping the embeddings constant or updating them. Another hack would be to keep the embedding layer constant but stack a hidden layer on top of the embedding layer before passing it to LSTM. This acts as a projection layer that projects the pre-trained embeddings from their actual domain to that of the tasks. We check this hypothesis by setting the embeddings constant and adding a dense layer with 300 units (same as the embeddings’ dimension).

3. **Number of hidden units in LSTM:** The dimension mentioned for the dense layer above, 300, is arbitrary. Instead of continuing with the same arbitrary number for hidden units in LSTM, we experiment with different values, 300, 512, and 1024.

4. **Number of LSTM layers:** There is no definite rule of thumb for the number of layers one should choose. More layers can be better but also harder to train. Therefore, we

stacked extra one, two, and three LSTM layers respectively to check whether more layers would give better results. We need to change the configuration of the first layer to output the entire output sequence as input for the subsequent layer. This can be done by setting the `return_sequences` parameter on the layer to True (defaults to False). This will provide a 3D array and return one output for each input time step.

5. **Dropout:** Dropout can be helpful in handling overfitting issues, especially in cases where bias is present in the data. When dropout is added, the model randomly selects nodes to ignore during the training process based on the given ratio. This way, if certain categories of nodes are represented disproportionately with higher weights, this imbalance can be mitigated. In our previous experimentation with traditional classification models, we had found some bias towards the "health" category in our error analysis, as such, we expected adding dropout might improve performance. With the optimizer set to 'Adam', we experimented with dropout ratios between [0.02, 0.5] and found that using a dropout ratio of 0.2 improved our performance by approximately 1.5%. In addition, we experimented with using recurrent dropout (adding dropout to recurrent layers), both in replacement of dropout and in combination with dropout, and found that combining our previous dropout setting with a 0.2 recurrent dropout ratio improved performance by another 0.9%.

6. **Optimizer:** We experimented with the Stochastic Gradient Descent (SGD), Adagrad, Adadelta, and Adam optimizers. Optimizers aim to minimize the loss function and show different performances on different types of data and models. As a simple method, SGD functions by performing jumps on the loss function back and forth and adjusting the weight accordingly to the increase or the decrease. Following the slope, it increases the weight until the loss function also increases, in which case it goes back and readjusts the weight to a lower value. This way it finds the highest possible weight before it starts having a negative effect. However, language data is rarely this simple, and so it tends to get stuck in local minima with most models. Adagrad adjusts this jumping behavior according to the slope and the frequency of the parameters. As Adagrad has some unfavorable effects on the learning rate, Adadelta is developed to restrict gradient accumulation. Adam additionally adds momentum to the Adadelta optimizer to improve performance. In our experiments, we found using Adam as our optimizer improved our model performance, although not by a large margin (0.1%). We think this is because the data size is small and the distribution of the classes is relatively even.

7. **Bi-directionality:** Bi-LSTM usually has a better performance than LSTM as it gives the input from both directions, backwards and forwards, which preserves information from both past and future. So we change the LSTM to bidirectional to see whether it actually improves the performance. We also experiment with the number of hidden units (300, 512, and 1024) and the number of layers (1, 2, 3, and 4) in Bi-LSTM. While in our experiment Bi-LSTM did not improve the performance, so we continued to use LSTM for the sake of higher training speed.

8. **Model Checkpoint and Early Stopping:** When we keep training the model for too long, the model might start looking for shortcuts in the training data to minimize the loss and overfit on it. In order to avoid overfitting, at the end of each epoch, we test our model on the validation data (data that's not been seen before) and check if it's performing well on it. If the training loss and validation loss are coming down, one can continue training. However, if the validation loss is increasing, it means that the model is overfitting.

   And now comes the question, should I stop training the minute I see the validation error going bad? Not really. It could be the case that model was at local minima in this epoch and might get better in the next epoch. So, we define a limit. We see if the validation loss is continuously going bad over time. If the patience is set to 5 epochs, we check if the validation loss is continuously increasing for the last 5 epochs. If that's the case, we would stop training.

Table 2 shows the best hyperparameters obtained per each algorithm. We chose the macro F-score as the metric to compare different approaches because it will give equal importance to each class regardless of their frequency.

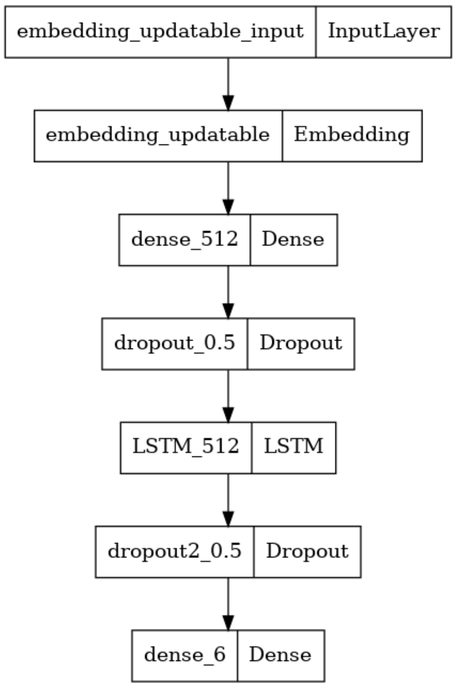

Figure 1: Architecture of the best performing LSTM baseline model after experimenting with multiple hyperparameters. Numbers mentioned in each layer represent the output dimension/dropout rate. For example, dropout_0.5 means a dropout layer with a dropout rate of 0.5. Note that embedding_updatable refers to the setting where we update embeddings while training.

In this case, we set the patience to 3 epochs and made `EarlyStopping` function monitor the validation loss. And to store the best model, we used `ModelCheckpoint`.

All the algorithms and features are implemented using `Keras` (Chollet and others, 2015) library.

3.2.2 Fine-tuning BERT

1. **Polynomial decay:** Extending the previous thought about training for too long, we do not want the model to update the weights as much as it is done in the initial epochs because the model has already learned most of the patterns. Therefore, we use a decaying learning rate, which reduces as we continue the training. We used `Polynomial Decay` from `Keras` library with an initial learning rate of 5e-5 and a final learning rate of 5e-7. And the rate at which the learning rate decays is given by a parameter called `decay_steps` where we mention the number of steps needed to get to the final learning rate. In our case, we spread the number of steps evenly across the training process i.e.

$$\#steps = \frac{len(Y)}{batchsize} \times \#epochs \qquad (1)$$

2. **Max sequence length:** The length of the longest sequence we can pass to the LSTM network is constrained by the maximum sequence length. We experiment with different values of max sequence length to see which one balances the performance and computational complexity best: 100, 128, 200, 256, 300, 400.

3. **Learning rate:** We also change the learning rate manually to find the one with the best performance. The learning rate decides the steps that the network takes for each iteration. A very high learning rate will make the learning skip the minima of a loss function and a too-low learning rate may get stuck in an undesirable local minimum, so we check other values of learning rate besides the default one (5e-5): 1e-4, 5e-4, 0.001, 0.005, 0.05, 0.01, 0.5, and 1.

4. **Batch size:** Batch size stands for how much of the data we pass onto the network at once, and the model iterates a number of times until all the data has been passed through for each epoch. For our model, we experimented with batch size values of [16, 32, 64], and found that a batch size of 64 yields the best performance with a 93.5% Macro-F1 score on our test data.

5. **Number of epochs:** While training a model for longer can give us more chances to adjust our weights, it is also possible that this will lead to overfitting and cause our model to be overly optimized for the training data that it starts to perform worse on unseen data. Therefore, it is good practice to optimize epochs to a point where the model has enough chances to adjust the weights, but not

Hyperparameter	Model state	Settings		Best	Score
Trainable embeddings	default	trainable, constant		trainable	0.875
Dense layer between embeddings & LSTM	default + embeddings = constant	yes / no (300 units)		256	0.977
Num. of hidden units	default	300, 512, 1024		512	0.875
Num. of layers	return_sequences = True	1, 2, 3		1	0.875
Dropout	optimizer = Adam	**dropout**	0.02, 0.05, 0.2, 0.5	dropout = 0.2 recurrent dropout = 0.2	0.892
		recurrent dropout	0.02, 0.05, 0.2, 0.5		
		both	0.02, 0.05, 0.2, 0.5		
Optimizers	default	SGD, Adagrad, Adadelta, Adam		Adam	0.868
Bi-directionality	hidden units = 300, 512. 1024 layers = 1, 2, 3, 4	yes / no		no	0.867

Table 2: A summary of our experiments for baseline LSTM while searching for the best hyperparameters. Model state refers to the initial model settings during the experiment. Macro-F scores are reported on the validation split.

so much that it overfits. To this end, we experimented with multiplies of 5 between [5, 20] and found that our model performed best at 5 epochs with a Macro-F1 score of 93.7% on our test set.

4 Models

LSTM: A major downside of FFNNs is that different input lengths are taxing to account for, and contextual prediction is challenging to define. In order to improve computation for these shortcomings, sequential models called RNNs were developed, wherein a Neural Network model contains a new construct part where the representation of the previous input is stored and is used as context for the calculations of the current input. These models allow for variable input sizes and their ability to store the previous context shows improvements on dependencies.

However, due to the vanishing gradients, RNNs have memory limitations and face issues with long-term dependencies. Consequently, LSTMs were developed to boost memory performance in RNN models, where another structure called the ”state” is added, alongside the previous memory system in RNNs, which stores previous input and controls how much of that storage needs to be forgotten, retained, or recalled for the incoming input. During the training, when a previous context is to be forgotten, the sigmoid activation function outputs a vector of zeroes so that the multiplication clears out the existing context. The context is stored as the model uses a sigmoid and a tanh activation function to store the new context in consideration of the previous context and the current input. The output is calculated through the sigmoid function and by sending the previous context through a tanh activation function. All of these processes are controlled by gates.

Language Model: Language models in NLP are probabilistic statistical models that calculate the probability of a given sequence of words occurring in a sentence based on the previous words. They analyze large text corpora to provide a basis for their word predictions. A number of NLP tasks can be solved by language models with an abstract understanding of natural languages, such as machine translation and question answering, as well as the task we are addressing now, text classification.

Currently, all state-of-the-art language models are neural networks. PLMs are large neural networks that operate under a pretrain-finetune paradigm: Models are first pre-trained over a large text corpus via unsupervised learning (also called self-supervised learning). The sentences are passed through the model and each word is predicted using the previous word as in traditional language models; however, the actual next word is known in this case and this information can be used during the training. This is why it is a self-supervised training process. Then, the models are finetuned on a specific task and this further adjusts the model's parameters through the use of a small amount of labeled data via supervised learning.

Autoregressive models and autoencoding models are two types of PLMs. The only difference between these two is in the way they are pretrained. Autoregressive models depend on the decoder part of the transformer model and use an attention mask on the top of the full sentence so that the model can only look at the tokens before the attention heads. On the other hand, autoencoding models correspond to the encoder part and they can look at all the tokens in the attention heads without any mask.

A typical example of autoencoding models is BERT(Devlin et al., 2018). BERT is built through stacking the encoder part of a Transformer, which was initially developed as a Machine Translation model to deal with the shortcomings of LSTMs. The issues were mainly that LSTMs were necessarily sequential and thus took a long time to train as the whole input had to pass through the model one-by-one and each output was generated one by one before the weights could be adjusted, and that their bidirectionality still relied on separate training. In contrast, the encoder part of a Transformer receives the entire input at once and produces embeddings to represent them. This part, separated from the decoder, deals strictly with language learning as opposed to generation.

BERT is trained for two tasks: Masked Language Modeling (MLM), and Next Sentence Prediction (NSP). In MLM, the model receives an input of sentences with some words in the sentences masked, and is trained to predict the masked words. In NSP, BERT predicts if given sentences follow each other. However, there are some limitations that come with the MLM training process. When users fine-tune BERT, most do not use masked tokens, and as such, this token is only seen during the pre-training process. Another issue is the context required to predict masked tokens, which results in a large data requirement for a small portion of the prediction, rendering it inefficient. Finally, the model assumes the masked tokens are independent, which may not be the case.

In our study, we evaluate four successor models besides BERT, which are RoBERTa, DeBERTa, DistilBERT, and ALBERT.

RoBERTa (Liu et al., 2019) was developed to improve issues evident in BERT. BERT was originally trained by making copies of the dataset with different masking patterns and sending those copies throughout the epochs. RoBERTa developers instead approach it differently, by sending randomly masked data every epoch during the training. Additionally, instead of using concatenated documents as the input, they use sequences of sentences and remove the NSP training task. Another big difference between these two models is that BERT uses word-pieces whereas RoBERTa uses byte-pair encodings on a character level. Of course, they also try different hyperparameter settings.

DeBERTa: When passing the input tokens to the Transformer model, the usual practice is to sum the token embeddings with that of the position embeddings. He et al. (2020) conjectured that summing up token embedding with position embedding can make it difficult for the transformer to disentangle them, especially in the higher layers. Therefore, they suggest maintaining two embeddings for each input token, token embedding, and position embedding, and propose an attention mechanism where both types of embeddings have individual attention matrices. The projected query, key vectors for content, and position are then joined in different combinations to ensure the information exchange. In addition to the relative position information, they also propose sending in absolute position information right after the transformer layers but before the softmax layer for enhanced mask word prediction.

DistilBERT (Sanh et al., 2019) is a small, fast, cheap, and light Transformer model based on BERT architecture. In the pre-training phase, knowledge distillation is performed, a compression technique in which a small model, DistilBERT in this case, is trained to reproduce the behavior of a larger model (BERT). The result of distillation is to reduce the size of BERT by 40%, making the inference 60% faster while retaining 97% of its performance. Besides, a triple loss combining language modeling, distillation, and cosine-distance losses is used to make use of the inductive biases learning by larger models during pre-training. DistilBERT is a trade-off between performance and computational cost that can be used with comparatively low-powered GPU devices.

ALBERT (Lan et al., 2019) is a lite version of BERT which reduces its footprint while maintaining its performance. It presents two techniques to achieve lesser sets of parameters: cross-layer parameter sharing and factorized embedding layer parameterization. In the first method, the parameter of only the first encoder layer is learned and the same is used with different weights across all encoders. In the first method, the parameters of all encoder layers are shared. This way, we have only one encoder layer and we apply that layer 12 times to the input (for 12 layers). Besides, instead of keeping the embedding size the same as the vector size that is passed between encoder layers like BERT, ALBERT reduces the size of embeddings using a matrix that multiplies with embeddings and enlarges to a size equal to the vector of the hidden layer. In addition to being a light version of BERT, ALBERT is trained on Sentence Order Prediction (SOP) instead of the NSP task BERT uses. The key difference is that the NSP task decides whether two sentences appear consecutively or not, while the SOP task is based on the coherence of sentences, i.e., whether the two sentences are in the right order or not.

5 Results

In this section, we report the scores obtained using LSTM baseline, BERT model, and its successors. Apart from reporting the scores, we discuss why we had to change the hyperparameters obtained after merging the best ones from isolated experiments.

5.1 LSTM Baseline:

After testing each hyperparameter and architectural choice in isolation, we arrived at the final set of hyperparameters and architectural choices:

(Trainable = True, 1 LSTM, 512 units, dropout = 0.2, recurrent dropout = 0.2, Adam = 0.01, bidirectional LSTM = False)

However, we noticed that simply merging the observations from isolated experiments resulted in scores lower, (81.3 F-score with learning rate 1e-2) than the default setting, (85 F-score) (Table 3).

Learning rate	Validation F-score	Test split F-score
1e-02	81.3	79.9
1e-03	88.5	**88.3**
1e-04	**89.5**	87.8
5e-05	87.8	87.8

Table 3: Benchmarking of LSTM with the best hyperparameters (after merging the observations from isolated testing). In this table, we evaluate the learning rate for the final set of hyperparameters.

Table 3 displays a curious case of overfitting on validation data: the setting which performs the best on the validation data (learning rate 1e-3 in our case) but not necessarily on the test data. In fact, it is the 1e-4 learning rate that performed best on the test data. In these cases, we can prefer 1e-4. Updated hyperparameters are:

(Trainable = True, 1 LSTM, 512 units, dropout = 0.2, recurrent dropout = 0.2, **Adam = 1e-4**, bidirectional LSTM = False)

Figure 2 presents the confusion matrix of the baseline model. It can be observed that the mispredictions are almost uniformly spread between the classes. We will analyze the reasons in detail in the discussion section.

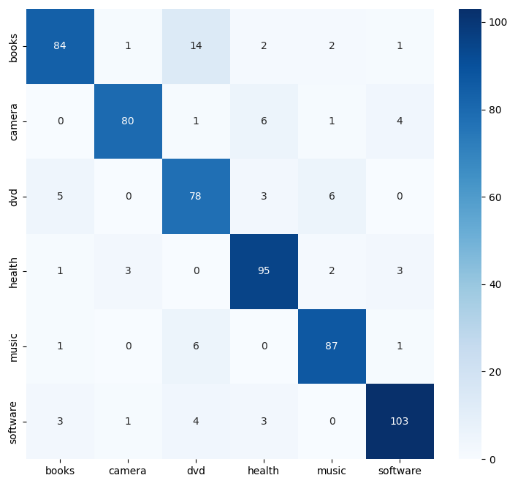

Figure 2: Confusion matrix of the baseline LSTM. The colour shade of each column represents the number of instances: the more instances it has, the darker the colour is.

5.2 BERT model

Hyperparameters obtained after merging the observations from isolated experiments are

(max sequence length: 200, learning rate: 5e-5, batch size: 64, number of epochs: 5)

Using the hyperparameters as it is resulted in memory issues. So, we had to either reduce `max sequence length` or `batch size`. Since it is good to have as much input as possible, we keep the `max sequence length` unchanged and reduce the batch size to 16. Updated hyperparameters on which we trained the model are

(max sequence length: 200, learning rate: 5e-5, **batch size: 16**, number of epochs: 5)

With the above-mentioned hyperparameters, we get an F-score of 94.4 on the validation set and 95.8 on the test set (refer to Table 4), which beats the LSTM baseline by 5 and 7 points on validation (89.5 F-score) and test splits (88.3 F-score). Figure 3 presents the confusion matrix of the model, with scores calculated per class. Reviews of the ”dvd” class are the most difficult to label as they are overlapping with all other classes (except health). We will analyze the reasons in detail in the discussion section.

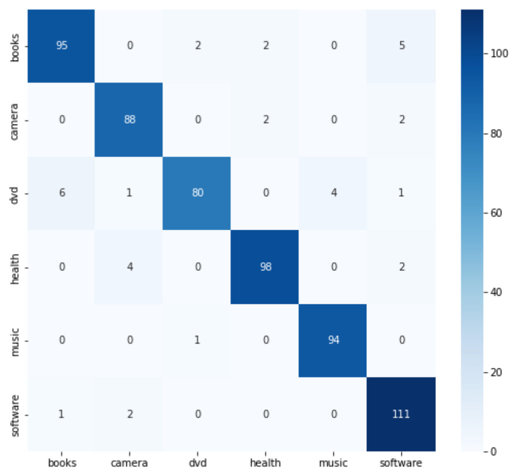

Figure 3: Confusion matrix of the pre-trained language model, BERT. The colour shade of each column represents the number of instances: the more instances it has, the darker the colour is.

5.3 BERT successor models

Now, we evaluate the BERT successor models, RoBERTa, DeBERTa, DistilBERT, and ALBERT. For these models, we use the same hyperparameters as BERT for comparability purposes. From table 4, we can see that DistilBERT, with 40% of the BERT’s size outperforms it with an F-score of 95.1 on the validation data. When it comes to the test split, BERT performs the best of all its successors.

Model	**Validation split**	**Test split**
BERT	94.4	**95.8**
DeBERTa	94.5	95
RoBERTa	94.5	94.5
DistilBERT	**95.1**	94.6
ALBERT	92.8	92.2

Table 4: Macro F1-Scores of BERT and its successor models on the validation data and test data.

6 Discussion

Since the classes have an approximately even distribution as shown in Table 1, and because their reviews contain relatively clearer identifying lexical items (such as ”movie” in ”dvd” and ”heart” in ”health”), it is not surprising to see the high scores for the baseline LSTM and BERT model.

Comparing the confusion matrices of LSTM (Figure 2)and BERT (Figure 3), we can see that the mispredictions across classes were reduced and that they can be narrowed down to two or three classes. We analyze these mispredictions in the error analysis section.

Error Analysis: We have identified a pattern of errors with our model (BERT) by observing the list of mislabeled instances. Based on the confusion matrix (Figure 3), we see that ”dvd” category overlaps with other classes like ”books” and ”music”. Upon inspecting these instances, we found that the reviews are indeed talking about both topics (ground truth and predicted class). For example, some reviews are about movie (dvd class) adaptions of books and some are about the dvds of music concerts. These examples are ambiguous to classify manually too. The same phenomenon can be extended to dvd and music

And there’s an interesting scenario where the words are too generic to map to any class. For example, ”this product” can refer to a dvd, book, camera, or software.

Upon analyzing the instances where health reviews are mispredicted as camera and software, we found that the predictions are indeed correct and that these were annotation errors. Another thing we noticed is that the health-related reviews were unclean and contained reviews from non-English languages, were too generic, and had a broad demographic, resulting in a variety in reviewer style and subtopics. We also conjecture that, because of errors like these, the model resorts to mapping miscellaneous reviews to the health class.

7 Conclusion

In this work, we evaluate LSTM, BERT, and its variants like RoBERTa, DeBERTa, DistilBERT, and ALBERT for building a text classification system. In the process of evaluating the models, we experiment with each hyperparameter in isolation to find the best hyperparameters. We found that we must tune the hyperparameters obtained after merging them from isolated experiments. We also observed a phenomenon of overfitting on validation data. Through the experimentation on BERT variants, we found DistilBERT to perform comparably or better (on the validation data) than the original BERT model while only being 40% of the BERT's size.

References

[Chollet and others2015] Francois Chollet et al. 2015. Keras.

[Devlin et al.2018] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

[Hochreiter and Schmidhuber1997] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11.

[Iyyer et al.2015] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. 2015. Deep unordered composition rivals syntactic methods for text classification. In *Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)*, pages 1681–1691.

[Joulin et al.2016] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. 2016. Fasttext. zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651.*

[Jurafsky and Martin2000] Daniel Jurafsky and James H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, USA, 1st edition.

[Kowsari et al.2019] Kamran Kowsari, Kiana Jafari Meimandi, Mojtaba Heidarysafa, Sanjana Mendu, Laura E. Barnes, and Donald E. Brown. 2019. Text classification algorithms: A survey. *CoRR*, abs/1904.08067.

[Lan et al.2019] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. ALBERT: A lite BERT for self-supervised learning of language representations. *CoRR*, abs/1909.11942.

[Li et al.2022] Qian Li, Hao Peng, Jianxin Li, Congying Xia, Renyu Yang, Lichao Sun, Philip S. Yu, and Lifang He. 2022. A survey on text classification: From traditional to deep learning. *ACM Trans. Intell. Syst. Technol.*, 13(2), apr.

[Liu et al.2015] Pengfei Liu, Xipeng Qiu, Xinchi Chen, Shiyu Wu, and Xuan-Jing Huang. 2015. Multi-timescale long short-term memory neural network for modelling sentences and documents. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 2326–2335.

[Liu et al.2019] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

[Mikolov et al.2013] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781.*

[Minaee et al.2021] Shervin Minaee, Nal Kalchbrenner, Erik Cambria, Narjes Nikzad, Meysam Chenaghlu, and Jianfeng Gao. 2021. Deep learning–based text classification: a comprehensive review. *ACM Computing Surveys (CSUR)*, 54(3):1–40.

[Pennington et al.2014] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

[Sanh et al.2019] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108.

[Shen et al.2018] Tao Shen, Tianyi Zhou, Guodong Long, Jing Jiang, Shirui Pan, and Chengqi Zhang. 2018. Disan: Directional self-attention network for rnn/cnn-free language understanding. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32.

[Yang et al.2016] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489.

[Zhou et al.2016a] Peng Zhou, Zhenyu Qi, Suncong Zheng, Jiaming Xu, Hongyun Bao, and Bo Xu. 2016a. Text classification improved by integrating bidirectional lstm with two-dimensional max pooling. *arXiv preprint arXiv:1611.06639*.

[Zhou et al.2016b] Xinjie Zhou, Xiaojun Wan, and Jianguo Xiao. 2016b. Attention-based lstm network for cross-lingual sentiment classification. In *Proceedings of the 2016 conference on empirical methods in natural language processing*, pages 247–256.