# Predicting Book Genres using different Hyperparameter Optimizers

Sophie Sananikone (S4716671), Lennart August (S4800036)
Diana Todoran (S4788761), Laura M Quirós (S4776380)

July 2, 2023

Neural Networks Semester Project - Group 3


Department of Artificial Intelligence
University of Groningen
9747 AG Groningen, The Netherlands

## Abstract

This paper makes use of two hyperparameter space search algorithms (Random Search and TPE Search) with the aim of finding a Neural Network capable of predicting the genre of a book based on its summary. Our base architecture is a Multi-Layer-Perceptron, and our hyperparameter space contains variations on the learning and dropout rate, L2 regularization, model size, and data augmentation method and quantity. Results showed that TPE search more consistently found models with higher F1 scores compared to Random Search. However, the best models of each search algorithm performed similarly on both validation and testing data. When tested on validation data, the best models achieved an F1 score of around 0.7. However, the same architectures only achieved a final F1 score of around 0.17 when evaluated on testing data. Confusion matrices showed that the model mostly only predicted one genre ("fantasty") regardless of the true genre of the summary. The model's behaviour leaves room for improvements regarding data splitting and evaluation methods, augmentation methods, and data quantity.

# Contents

# 1 Introduction

People are of many different kinds, and so are our preferences for specific genres and what we enjoy the most. Genre refer to the tone or identity of the story, novel, music, etc. Humans cannot be a masters of all traits, despite their profession. Therefore, genre is significant in all forms of creativity and development.

Furthermore, as digital books and large-scale textual data become more widely available, machine learning algorithms are being developed and used to automatically classify books into different genres. Book genre classification is an active field of research with applications of machine learning models (ML) and stands at the centre of our research.

In this project, we attempt to create a neural network that classifies books into 10 different categories. The basis of our model is the Multi-Layer-Perceptron (MPL). A search through a defined hyperparameter space to find the best-performing architecture was conducted using both Random Search and the Tree-structured Parzen Estimator (TPE). We experiment with two data augmentation methods, synonym replacement (SR) and random deletion (RD). We also experiment with the size of the model, L2 regularization, Dropout Regularization, and learning rate. We will measure the success of the model with F1 accompanied by confusion matrices. Finally, we are aiming to achieve a 65% accuracy for our model's performance with this task.

# 2 Data

The dataset that we used for our project, "Book Genre Prediction" was found on Kaggle, posted by Atharva Inamdar. The dataset has had multiple updates since its initial posting. We have the version from 30th of May 2023, which can be found on our Github.

The dataset consists of 4,657 books, with information about their title, genre, and summaries. Our model only makes use of genre and summary information. The genres the books are classified are "thriller", "fantasy", "science", "history", "horror", "crime", "romance", "psychology", "sports" and "travel". Their distribution of summaries per genre can be seen in Figure 1. Each summary of a book has on average 360 words, and the frequency distribution of word count per summary can be found in Figure 2.

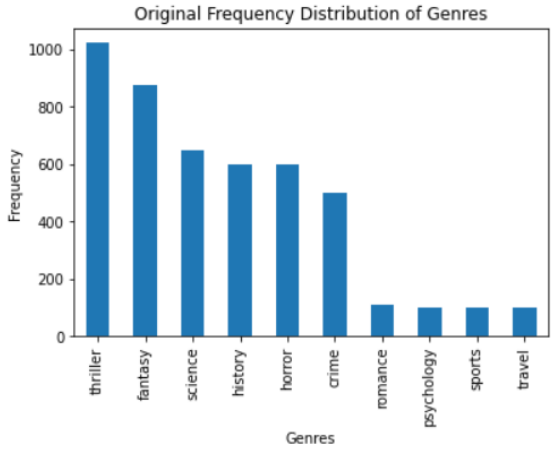


Figure 1: Histogram displaying the distribution of summaries per genre.

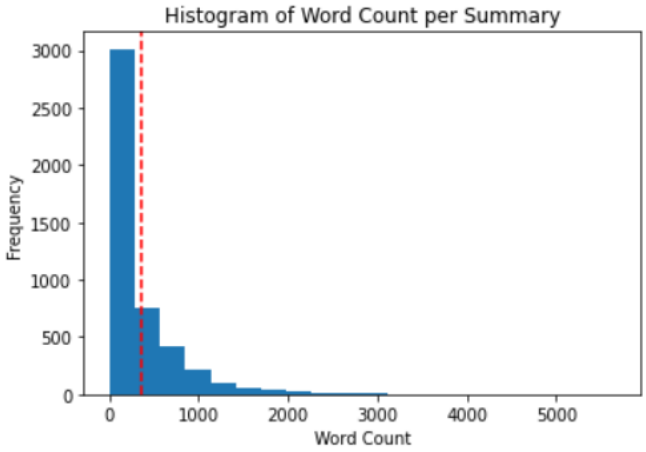


Figure 2: Histogram depicting the number of words and the relevant number of summaries for each word count. The red line represents the average word count of a summary from the dataset.

# 3 Methods and Experiment

## 3.1 Cleaning Procedure

We first removed duplicate summaries from our data. We iterate through each summary, removing non-alphabetic and stop words (the standard list of stop words from the `nltk` python library). We store the clean lowercase summaries along with the vocabulary frequency information. We will use them to calculate the Term Frequency and Inverse Document Frequency, which will give us a two-dimensional numpy array with a shape of number of documents by size of the vocabulary, where each element represents the TF-IDF value for a token in a specific doc-

ument. We split our training and testing data with an 80%-20% ratio.

### 3.2 TF-IDF for the summary feature

TF-IDF (introduced by Jones (1972)) is a popular statistical method used to determine the relevance of terms in certain documents based on their Term Frequency (TF) and Inverse Document Frequency (IDF).

The term frequency (TF) relates to the intuition that if a term is often mentioned, it must be relevant to the subject. The inverse document frequency (IDF) measures the uniqueness of the word in the given corpus (collection of all documents provided). The main idea behind TF-IDF is to highlight terms frequent in a specific document (high TF) but relatively rare in the corpus (high IDF). We use the TF-IDF score to reduce the noise that more common but less informative words would cause. By incorporating TF-IDF, we enhance the discriminatory power of the summary feature.

The TF-IDF score is calculated per term using the formulas below:

$$\mathrm{TF}(t,d) = \frac{\text{times term } t \text{ appears in document } d}{\#\text{ terms in document } d}, \quad (1)$$

$$\mathrm{IDF}(t,D) = \log\left(\frac{\#\text{ documents in corpus D}}{\#\text{ documents with term } t \text{ in D}}\right), \quad (2)$$

and incorporates its results as described in the following formula:

$$\text{TF-IDF}(t,d,D) = \mathrm{TF}(t,d) \times \mathrm{IDF}(t,D). \quad (3)$$

This means that the TF-IDF score is the cross product of the term frequency and the inverse document frequency. For our model, we only consider the first 1500 most frequent words from every resulted corpus.

### 3.3 Our MLP

For the training of a neural network that is able to predict the genre of the book from a number of features, we chose to use a simple multi-layer perceptron. We considered the use of an MLP the most suitable for the size and structure of our chosen dataset. Its inherent characteristics and flexibility allows us to strive for a higher accuracy ratio, comparable to the one we would obtain if we had a more uniform dataset.

The layers of a multi-layer perceptron can be divided in three sections: input, hidden and output layers. The hidden layers enable the model to learn hierarchical representations of the data, extracting higher-level features from the input. Having more hidden layers and neurons can increase the model's capacity to capture and represent intricate patterns, enhancing its accuracy.

Another useful characteristic is its ability for nonlinear mapping, or in other words, the use of nonlinear activation functions. This helps to capture higher-level data patterns that increase the accuracy (PyCodeMates, 2023).

We concluded that MLP has tools that can be used in order to infer the complex data patterns in our book genre identification dataset, even for the classes that do not have as many examples.

However, the main advantage of the MLP resides in its flexibility. The size can vary, and the layer activation functions can change. Besides, regularization, cross-validation and data augmentation can be used alongside for the better training of the model.

We have an input layer with the same number of neurons as the length of the vocabulary, a changing number of hidden layers and an output layer with 10 neurons, which is the length of the book genre types. The hidden layers use activation function `tanh` (hyperbolic tangent), which maps the input values to a value in the range between -1 and 1, causing the layer to be centered 0, which is advantageous for the MLP backpropagation algorithm (LeCun et al., 2002).

For the output layer, we use a one-dimensional softmax function (Jaeger, 2023). This function extends the idea of logistic regression into the multi-class world, transforming an n-dimensional vector into a probability distribution of n possible outcomes. This allows datasets with small number of labels, such as ours, to have as output the probability of each input of belonging to each class.

The hyperparameters to consider are the model size, the criterion (loss function), the learning rate, the regularization term L2 and the dropout rate. We will maintain the same loss function, but the other hyperparameters will be changing along with each combination we train.

### 3.4 Cross-Entropy Loss Function

For the loss function we use cross-entropy, very suitable as our output is given in a probability distribution due to the softmax layer.

Entropy can be thought of as the amount of uncertainty of an event or the difference between two probability distributions. More formally, is the average number of bits required to represent or transmit an event drawn from the probability distribution for the random variable (Shannon, 1948). High entropy means we have a balanced probability, as events have equal probability of happening. Entropy ($H$) can be calculated for each class

$x$ with the following formula

$$H(X) = -\sum_{x \in X} P(x) \cdot \log_2(P(x)). \tag{4}$$

Cross entropy is the average number of bits needed to encode data coming from a source with distribution $P$ when we use model $q$ (Murphy, 2012), taking into account the model $q$ has the predicted distribution $Q$ we are trying to match to the true distribution $P$.

$$H(P, Q) = -\sum_{x \in X} P(x) \cdot \log(Q(x)). \tag{5}$$

The cross-entropy function measures the error or variance from the true distribution, and it is often used for categorical multi-class classification problems. This loss will be minimized by the chosen optimizer.

### 3.5 Adam Optimizer

For the optimizer, we will use Adam (Adaptive Moment Estimation), an extended version of stochastic gradient descent. The power of momentum gradient descent to hold the history of updates and the adaptive learning rate makes Adam optimizer a powerful method. Adam computes individual adaptable learning rates for different parameters by estimating a running average of first and second moment of the gradient (Kingma and Ba, 2014). The algorithm can be found in the appendix, in subsection 7.2. The "moments" we will be talking about are the statistical moments of the gradient.

We define a moment $m$ of a variable $X$ as the expected value $E$ of said variable to the power of $n$. Here $n$ is the order of the moment.

$$m_n = E[X^n] \tag{6}$$

The first moment of a variable is the expected value of that variable itself (i.e., the mean). In the case of Adam, the first moment is the running average of the gradients.

The second moment of a variable is the expected value of that variable squared. In the case of Adam, the second moment is the running average of the element-wise squared gradients. To estimate these, Adam uses exponentially moving averages, computed on the gradient evaluated on a current batch, shown in functions 7 and 8:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t, \tag{7}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2, \tag{8}$$

where $m$,$v$ are moving averages in iteration of time steps $t$, $g$ is the gradient on the batch and the betas $(\beta_1, \beta_2)$ are hyperparameters of the algorithm. They are already efficient in their default values of 0.9 and 0.999, so we won't change them.

We can rewrite functions 7 and 8 as

$$m_t = (1 - \beta_1) \sum_{i=0}^{t} \beta_1^{t-i} \, g_i. \tag{9}$$

To further counter the bias of the estimates towards zero, Adam applies bias correction such that we have equations 10 and 11:

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t}, \tag{10}$$

$$\hat{v_t} = \frac{v_t}{1 - \beta_2^t}. \tag{11}$$

To perform the update we therefore use the formula 12, using learning rate $\alpha$ and a small value added for numerical stability $\epsilon$, such that we avoid division by 0.

$$\Delta_t = \alpha \cdot \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon} \tag{12}$$

### 3.6 Regularization: Dropout

In the context of neural networks, dropout is a regularization technique used during training to prevent overfitting. It involves temporarily dropping out (i.e., setting to zero) a random subset of the units (neurons) in a layer during each training step (Srivastava et al., 2014). This "dropping out" process is applied independently to each unit with a certain probability, and it's quite useful for small datasets (Hinton et al., 2012).

The dropout technique helps prevent overfitting by reducing the interdependencies between units, forcing the network to learn more redundant representations, and preventing the co-adaptation of neurons.

We insert dropout layers between the linear layers to improve the generalization performance of the model and make it more robust to noise or variations in the input data. The dropout parameter determines the probability of dropping out of each unit. For this project, we will experiment with a dropout rate value between 0 and 0.6, inclusive. By setting a dropout probability, the model incorporates dropout regularization during training, which can lead to better generalization performance on unseen data.

By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections.

### 3.7 Regularization: L2

L2 regularization, also known as weight decay, is a technique commonly used in machine learning to prevent overfitting and improve the generalization ability of a model.

Regularization is achieved by adding a regularization term to the loss function during training. L2 regularization encourages the model's weights to be small by penalizing large weight values. It is calculated as the sum of squares of all the weights ($w_i^2$) in the model multiplied by a regularization parameter L2 ($\lambda$), as seen in the formula 13.

$$\hat{f} = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{n} L(\hat{y}_i, y_i) + \lambda \sum_{i=1}^{N} w_i^2 \tag{13}$$

The value of L2 ($\lambda$) determines the strength of the L2 regularization. By tuning L2, you can control the trade-off between fitting the training data well and keeping the model weights small to prevent overfitting. In our project, we will experiment with values between $1e^{-5}$ and $1e^{-2}$, inclusive. We will sample with a logarithmic uniform distribution.

The L2 regularization term helps in preventing the model from becoming too complex or relying heavily on specific features by adding a penalty for large weights. This regularization technique can help improve the generalization performance of the model on unseen data.

### 3.8 Data Augmentation

Given the skewed distribution of genre frequency in our dataset, we decided to experiment with augmenting our data, using the Python library NLPAug (Ma, 2019). According to Wei and Zou (2019), textual data augmentation is most beneficial with small datasets. Wei and Zou's work showed that models trained on a dataset of size 500 showed, on average, a three percent performance improvement when the dataset was augmented. With a size of 2000, there was an average improvement of 0.8%. As described in the Data section, our most frequent genre occurred 1023, and our least frequent, combined "Other" category occurs 411 times. As a result, our category sizes are near the range that show improvements in performance when augmenting the dataset.

Data augmentation is mainly defined by two parameters: $\alpha$, the probability of each word in the sequence being changed, and $n_{aug}$, the number of augmented sentences that should be generated per original sentence (Wei and Zou, 2019). There are multiple operations by which text can be augmented, of which we compared two: Random Deletion (RD) and Synonym Replacement (SR). RD deletes words in a text with a probability of $\alpha$. On the other hand, SR replaces $n$ words in the original text with randomly chosen synonym. The number of replaced words $n$ is determined by the formula

$$n = \alpha l, \tag{14}$$

where $l$ is the length of the original text in terms of the number of words in the text (Wei and Zou, 2019).

While the source of the synonyms depends on the database used, we opted to use WordNet. WordNet is a network of linked "synsets", which are sets of synonyms for a sense of a word (Brown and Anderson, 2006). Synsets are distinguished by their parts-of-speech (POS) and are connected by various relationships (the main relationship being synonymy, but other relationships include antonymy, hyponymy, and semantic relations) (Brown and Anderson, 2006). One word may belong to multiple synsets depending on the sense that is meant (for example, the word "bank" could refer to the financial establishment or the side of a river, each of which would be its own synset). To select a synonym for data augmentation, NLPAug first uses a POS-tagger to identify the POS of the word to be replaced, $w$. It selects a random synset containing $w$, which has the same POS as $w$. From this synset, it randomly selects a synonym to replace $w$ with, generating a new sentence.

We are interested and comparing RD and SR due to the difference in how they affect the model's vocabulary. RD decreases the vocabulary size of our model as it removes words, while SR increases the vocabulary size by adding synonyms. Importantly, data augmentation was conducted after our dataset was cleaned. Otherwise, there is the risk of deleting or replacing words that would have been removed by the cleaning process regardless. This would result in two identical data points, nullifying the effect of the augmentation. This could cause problems if there are identical data points in training and validation or testing data.

### 3.9 K-fold cross-validation

In order to assess the risks of our model and avoid overfitting, we opt for using cross-validation. This is the technique that involves splitting the training data into a training and validation subset. Specifically, we will be doing k-fold cross-validation.

$K$-fold cross-validation splits the data into $k$ disjoint subsets known as folds (Jaeger, 2023). Then, we consider one fold the test set and train the classifier on the remaining $k-1$ data folds. This sampling process is repeated $k$ times and, at each iteration, a different fold is chosen as the test set. Additionally, at every iteration, the error rate is computed on the test set in order to reduce bias and to have a more reliable performance estimation. If for example $k$ was chosen to be 10, then the classifier would be trained on 10 different models (where we consider 90% of our data), test the model 10 times and average the results.

However, the only disadvantage for using cross-validation is that we cannot examine any data from the corpus, be-

cause we use the entirety of it for testing. This could lead to overestimating the performance of the model.

Luckily, there is a way to make use of the results that the cross-validation gives by predefining and isolating a fixed test set and then perform a $k$-fold cross-validation. This way, information leaking would be avoided during the training process.

It should be noted that in the experiments in which we augment data, the entire training data is augmented. The validation set is split from the original training data, and the training data is then merged with the augmented data when training occurs. Therefore, when cross-validation occurs, the model may be trained on an augmented sentence, of which the original version is then used for validation (Jurafsky and Martin, 2009).

### 3.10 F1 Score

Due to the skewed distribution of summaries in each genre, an appropriate evaluation metric was necessary to determine our model's performance. Measures of success can make use of the frequency of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN).

Accuracy such as calculated in equation 15 is not an appropriate measure of success due to the genres with low frequencies (romance, psychology, sports, travel). The model could only predict the other six categories perfectly and still have an accuracy of 91%.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}} \tag{15}$$

Precision is a metric which measures the ratio of true positives (correctly classified items) to the sum of true and false positives (Olson and Delen, 2008). In other words, the items that were incorrectly classified to a genre. It can be calculated as shown in equation 16.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{16}$$

Recall is a metric which measures the ratio of true positives to the sum of true positives and false negatives (items that were incorrectly not classified to the true genre).

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{17}$$

The F-Score may use any value $0 \leq \beta \leq +\infty$. When $\beta = 1$, F1 comes to be equivalent to the harmonic mean of P and R, if it's higher, then the measure is more recall-oriented, and if lower, then it's more precision-oriented (Sasaki et al., 2007). We used the F1 score as our evaluation metric because its balance between precision and recall can help compensate for our classes with low frequency.

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{18}$$

### 3.11 Random Search (RS)

We performed a hyperparameter search to determine the best-performing architecture within our hyperparameter space. The first search algorithm we used is the Random Search algorithm. This algorithm randomly selects hyperparameter values from the hyperparameter space to train the model on.

### 3.12 Tree-structured Parzen Estimator Search Algorithm (TPE)

TPE is a search algorithm that combines the ideas of Bayesian optimization and Parzen estimators by making a probabilistic model of the hyperparameter space.

As explained in a question of Stack Overflow, this search algorithm allows us to explore a hyperparameter search space in a tree-like fashion, such that the value of the previously assigned hyperparameters constraints the values to be chosen from the later assignments.

With such a search space $X$, and an objective function $f : X \to \mathbb{R}$, we consider the problem of finding $x^* \in \text{argmin}_{x \in X} f(x)$. TPE focuses on modeling the conditional probabilities of hyperparameters $p(x|y)$ using two probability functions or densities $l(x)$ and $g(x)$ (Bergstra et al., 2011). A more detailed explanation with pseudocode can be read in the appendix subsection 7.1.

### 3.13 Experimental Setup

To attempt to find the best-performing architecture, we used Random Search and TPE to search through our hyperparameter space, defined below. For each search algorithm, 200 hyperparameter combinations were trialled. Note that the Hidden Layer Sizes denotes the number of layers and their respective sizes. For example, the model [512, 32] has two hidden layers, the first with 512 neurons and the second with 32 neurons.

Our search space is not strictly discrete, as it uses distributions and continuous ranges for some hyperparameters:

- `hp.loguniform`, used for the learning rate and the L2 regularization, it creates a logarithmic scale distribution for the learning rate, having its range

from 1e-5 to 1e-2, as seen from table 1, meaning it spans from 0.00001 to 0.01.

- `hp.uniform`, used to determining the dropout rate, it creates a uniform distribution between 0 and 0.6 in order to determine the probability of randomly dropping out units during training.
- `hp.choice`, used for the size or architecture of the model, which offers the possibility of choosing between different model sizes as represented in table 1.
- `hp.quniform`, used for both the number of time the synonym and deletion operations are performed, which creates a discrete uniform distribution ranging from 0 to 2.

| Hyperparameter | Values |
|---|---|
| Learning Rate | $[e^{-5}, e^{-2}]$ |
| Dropout Rate | [0, 0.5] |
| L2 Regularization | $[e^{-6}, e^{-2}]$ |
| Model Size | [512, 32], [256, 256, 32], [128], [64, 64], [32] |
| RD $n_{aug}$ | 0, 1, 2 |
| SR $n_{aug}$ | 0, 1, 2 |

Table 1: Search space for hyperparameter values

We used $\alpha = 0.1$ for both RD and SR augmentation. Our models were trained for a maximum of 400 epochs using 5-fold cross-validation, with a batch size of 512 and early stopping on training loss with patience of 20.

# 4 Results

## 4.1 Comparing Search Algorithms

An overall comparison of the RS and TPE search algorithms show that the TPE search resulted in a higher median F1 score (taken from validation data in the last fold of cross-validation) for the models trialled. The median is used as a measure of central tendency due to the skewed frequency distributions, seen in Figures 3 and 4. The median F1 score for the RS models was 0.48, while the median for the TPE models was 0.65. It can be seen in Figure 4 that there is a higher frequency of models with F1 scores between 0.6 and 0.8, compared to lower F1 scores. On the other hand, the RS models show high frequency both at the 0.0-0.1 F1 score range and the 0.6-0.7 range.

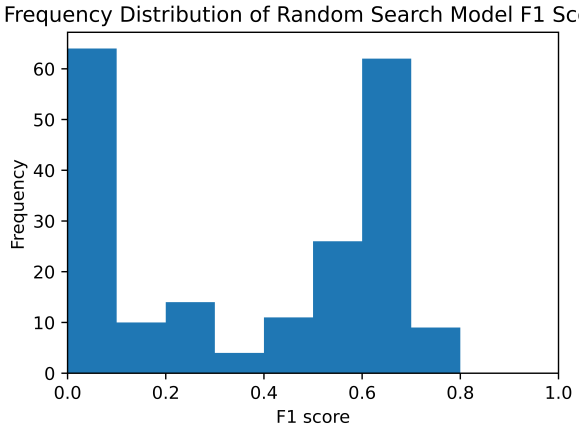


Figure 3: Frequency distribution of F1 scores of models trialled during RS algorithm.

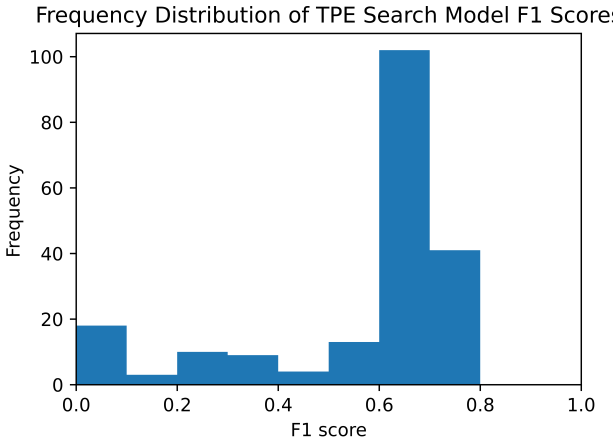


Figure 4: Frequency distribution of F1 scores of models trialled during TPE algorithm.

Due to the difference in these distributions, we wanted to determine if there was a significant effect of the type of search used on the performance of the models produced by the search over time. Ordinary least squares regression was used to determine the presence of such a relationship. The RS algorithm regression, shown in Figure 5, had a regression coefficient of $-6e^{-4}$, and did not significantly affect the F1 scores over time, $R^2 = .016$, $F(1, 198) = 3.143$, $p = .078$.

On the other hand, the TPE search algorithm, shown in Figure 6, had a regression coefficient of $7e^{-4}$. It showed significant effect on F1 scores over time, $R^2 = .043$, $F(1, 198) = 8.966$, $p = .003$.

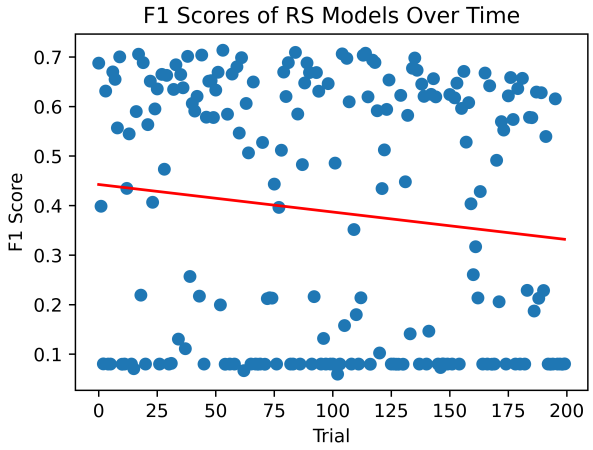


Figure 5: Graph showing the F1 scores of models trialled during the RS algorithm over time, with the linear regression line plotted in red.

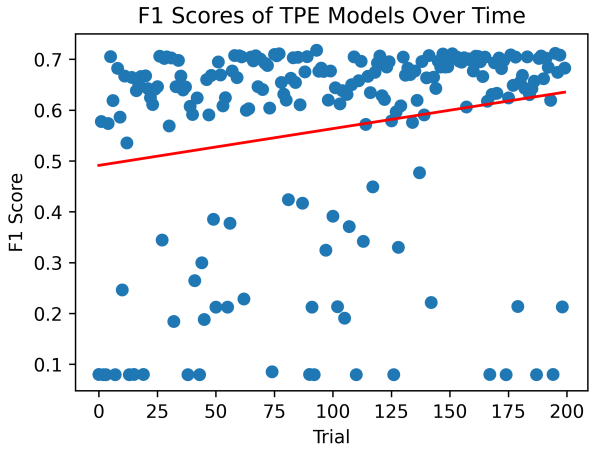


Figure 6: Graph showing the F1 scores of models trialled during the TPE search algorithm over time, with the linear regression line plotted in red.

## 4.2 Comparing Best Models from Search Algorithms

We compared the best model found in the RS to the best model from the TPE search, referred to as $\text{RS}_{best}$ and $\text{TPE}_{best}$, respectively. The "best" model refers to the model in each search which resulted in the highest F1 score on the validation data of the last fold of cross-validation. In this case, the $\text{RS}_{best}$ and $\text{TPE}_{best}$ The hyperparameter values of these models and their F1 scores can be found in Table 2. It can be seen that both models achieve similar F1 scores.

| **Parameters** | $\text{TPE}_{best}$ | $\text{RS}_{best}$ |
|---|---|---|
| Learning Rate | $3.24e^{-4}$ | $3.56e^{-4}$ |
| Dropout | 0.315 | 0.110 |
| L2 | $5.77e^{-6}$ | $6.34e^{-6}$ |
| Model Size | [128] | [32] |
| RD $n_{aug}$ | 0.0 | 0.0 |
| SR $n_{aug}$ | 1.0 | 1.0 |
| F1 | 0.718 | 0.714 |

Table 2: Hyperparameter values and F1 score of the best models found during RS and TPE search.

To compare the two models, two new models with the same hyperparameters as the best models were trained on the entire training data. They will be referred to as $\text{RS}_{final}$ and $\text{TPE}_{final}$ to represent the best model from the RS and TPE search, respectively. The training procedure is the same as described in section 3.13, except that cross-validation was no longer used. After training, the model was tested on our testing data, for which the final accuracy and F1 score was recorded, which can be seen in Table 3. It can be seen that the F1 scores achieved on the testing data is much lower than the results from $\text{RS}_{best}$ and $\text{TPE}_{best}$, of which the F1 scores were based on the validation data. However, this is not to say that the final models did not learn, as it can be seen in Figure 7 that training loss does decrease over training epochs.

| | $\text{TPE}_{final}$ | $\text{RS}_{final}$ |
|---|---|---|
| Accuracy | 0.177 | 0.176 |
| F1 | 0.163 | 0.152 |

Table 3: Final F1 score of models trained on best hyperparameter combinations found from RS and TPE search.

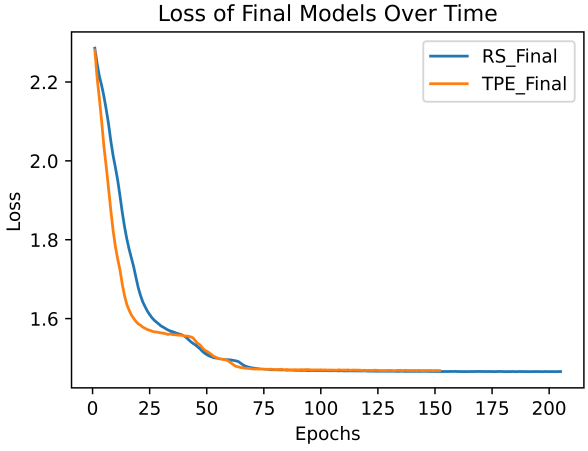


Figure 7: Comparison of training loss over time of $\text{RS}_{final}$ and $\text{TPE}_{final}$.

To further analyse the differences between the *best* and *final* models, confusion matrices were constructed for each.

The confusion matrices for the *final* models are based on results from testing data, whereas the matrices for the *best* models are based on the validation from the last fold of cross-validation. It can be seen from Figures 8 and 9 that both the *final* models $\text{RS}_{final}$ and $\text{TPE}_{final}$ predict the fantasy genre most frequently, regardless of actual genre labels. Interestingly, the psychology, romance, sports, and travel genres are almost never predicted, despite thriller being the most common genre in the original dataset.

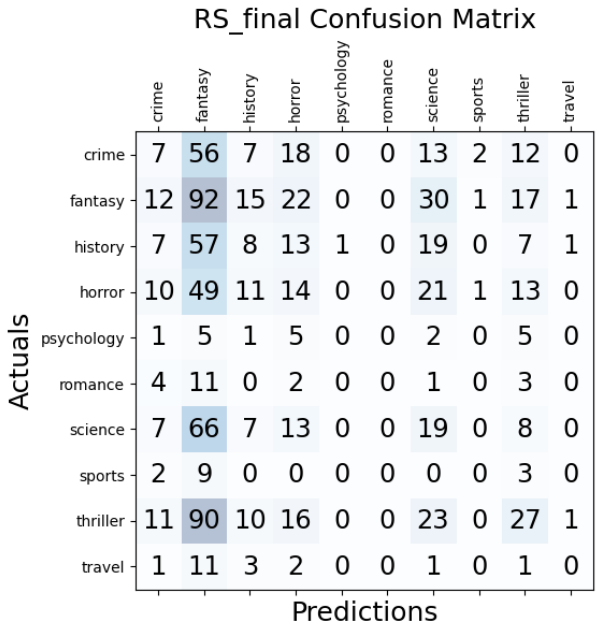


Figure 8: Confusion matrix of $\text{RS}_{final}$.

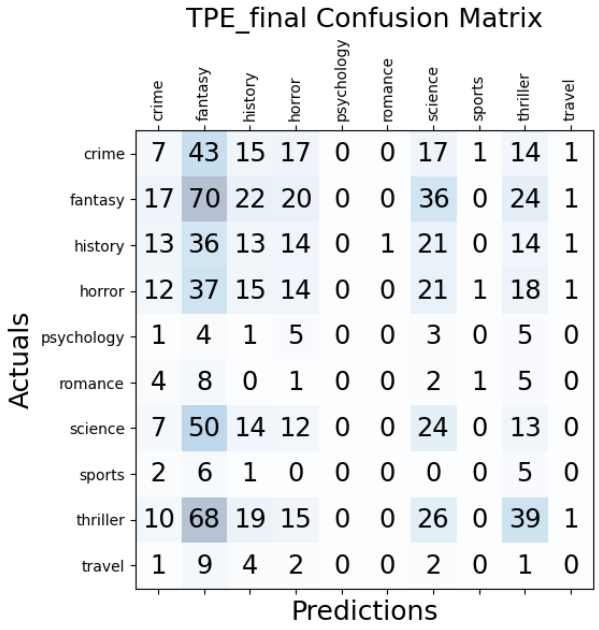


Figure 9: Confusion Matrix for $\text{TPE}_{final}$.

In comparison, the confusion matrices for $\text{RS}_{best}$ and $\text{TPE}_{best}$ much more coherent predictions, seen in Figures 10 and 11, respectively. For almost all genres, the most frequently predicted genre match the true label. Similar to the confusion matrices for the *final* models, the genres "psychology", "romance", "sports", and "travel" are the least often predicted.

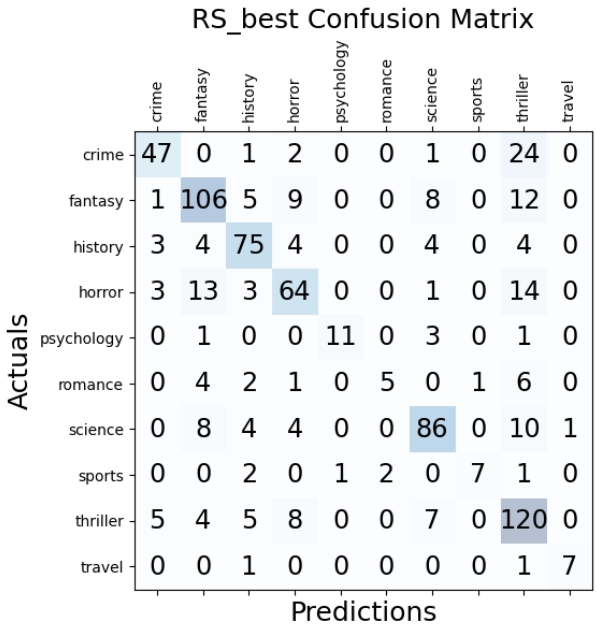


Figure 10: Confusion matrix for $\text{RS}_{best}$.

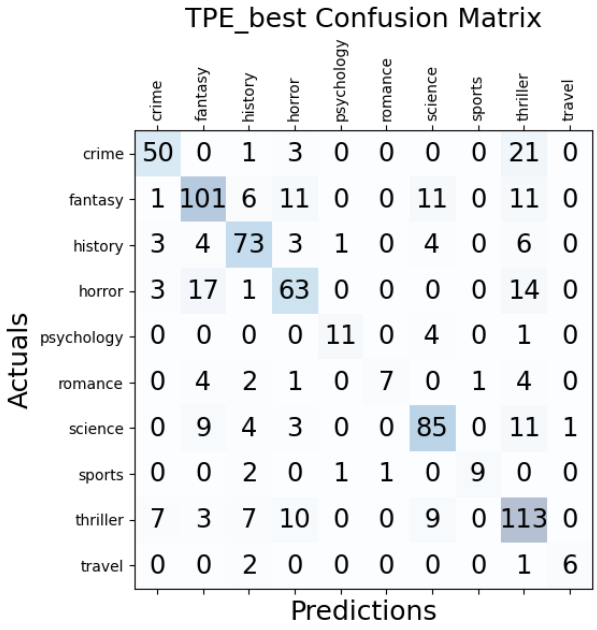


Figure 11: Confusion matrix for $\text{TPE}_{best}$.

# 5 Discussion

The aim of this project was to create a neural network model that can accurately classify books into 10 different genres based on a fraction of their story, i.e. a summary. We tested two hyperparameter optimizers for our model, RS and TPE. As seen from the results, while the TPE search algorithm on average found more models of high performance compared to the RS algorithm, their respective best models performed similarly. The best models not only were similar in their F1 scores during search, but also in their final F1 score training on the full training set. Therefore, in the case of this research, there was no advantage to using the TPE search algorithm over the RS algorithm. We originally aimed to produce a model with an F1 score of 0.65, but our best model only achieved 0.18. What follows is a discussion aiming to highlight possible limitations that lead to this outcome.

One of the intriguing outcomes of this research is the difference in performance of the *final* models compared to the *best* models. In particular, we raise the question: Why is the performance on testing data so low compared to the *best* models' performance on validation data? We believe this relates to the fact that both models make use of SR data augmentation. The data augmentation was performed on the entirety of the original training set. We then split our original training set into validation and training data, and combined the augmented data with the remaining training data. This means that there were augmented sentences in the training data which were simply augmented versions of sentences in the validation data. Therefore, there was an overlap (albeit not identical) between the training and validation data. We believe that this data leak caused the model to overfit. This also raises a problem during the search algorithms, since when augmented data was used, the model's performance was evaluated on data it had already partially seen, leading to a high F1 score. Models which did not use any augmented data performed worse on validation data (as this is truly unseen data), and were therefore labelled as "worse" models. Eventually, when the *final* models were trained on the full training set and evaluated on the testing data, they could not make use of the shortcuts it did with the validation data, leading to such a drastically lower F1 score. The model is not capable of distinguishing genres, it only memorised them before. Instead of distinguishing genres, it learnt to minimize loss by predicting the "fantasy" genre, the second most frequent genre in the training data, as opposed to the most frequent one being "thriller".

This leads to our second question: Why were our models not able to distinguish genres? To answer this question, we point out that the overlap in words between the training and testing data was 10323 words, or 68.8% of the 15000 most frequent words. That is, almost a third of the words in the testing data are unseen. This may have prevented our model from generalizing, as it was not able to determine the context due to out-of-vocabulary (OOV) words in testing. It should be noted that we ran an experiment in which we combined the 15000 most frequent words of both the training and testing data, and filtered our data with this combined vocabulary. We hoped that this would help the model deal with OOV words, but after training and testing, performance did not significantly improve compared to the *final* versions presented in our results. Therefore, it seems that the vocabulary overall was too small, or there was not enough data on the vocabulary for the model to learn to generalize properly. Consider genres like "psychology" and "science" that have disciplinary overlap, or "thriller" and "horror" or "crime" that have contextual similarities. Due to the limited vocabulary, the model may not have been able to distinguish between these similar types of genres, as the vocabularies associated to them may have overlapped.

It should also be noted that there is an issue with our SR method of data augmentation: the semantic meaning of the synonym. As described in Section 3.8, this method first selects a *random sense* of the augmented word $w$, and then selects a random synonym of that sense. The issue arises from this first round of selection: the algorithm has no understanding of the semantic meaning of $w$. As a result, $w$ may have been replaced with a word that is completely unrelated to the context, or potentially even changes the context. This introduces more confusion for our model to deal with, which may have prevented it from generalizing properly.

A potential solution to this problem is to use Contextual Word Embeddings to augment text. This method, provided by the `nlpaug` library, consists of using large, pre-trained language models to find synonyms for a word (Ma, 2019). Examples of such models are BERT, RoBERTa, or DistilBERT. As the models are pre-trained they already have a base understanding of the semantic meaning of a word, and can make use of the context (in this case, the summaries) to improve their understanding for the task at hand. This would have been a better method of SR, as the new words are more likely to be contextually coherent, therefore preventing further confusion. It was our original plan to use this method, but due to our limited computational power and time, we were not able to make use of it. In the future, it would be interesting to experiment with this method to see if it improves model performance.

# 6 References


Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, & K. Weinberger (Eds.), *Advances in neural information processing systems*. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf

Brown, E. K., & Anderson, A. 1. (2006). The encyclopedia of language & linguistics (2nd ed.). http://www.bibliothek.uni-regensburg.de/dbinfo/frontdoor.phtml?titel_id=6277

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Jaeger, H. (2023). NN Lecture Notes. https://www.ai.rug.nl/minds/uploads/LN_NN_RUG.pdf

Jurafsky, D. 1., & Martin, J. H. 1. (2009). *Speech and language processing : An introduction to natural language processing, computational linguistics, and speech recognition* (Second edition.). Pearson Prentice Hall. https://toc.library.ethz.ch/objects/pdf/z01_978-0-13-187321-6_01.pdf

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

LeCun, Y., Bottou, L., Orr, G. B., & Müller, K.-R. (2002). Efficient backprop. In *Neural networks: Tricks of the trade* (pp. 9–50). Springer.

Ma, E. (2019). NLP Augmentation. https://github.com/makcedward/nlpaug

Murphy, K. P. (2012). *Machine learning: A probabilistic perspective*. MIT press.

Olson, D. L., & Delen, D. (2008). *Advanced data mining techniques*. Springer Science & Business Media.

Ozaki, Y., Tanigaki, Y., Watanabe, S., Nomura, M., & Onishi, M. (2022). Multiobjective tree-structured parzen estimator. *Journal of Artificial Intelligence Research*, *73*, 1209–1250.

PyCodeMates. (2023). Multi-layer perceptron explained: A beginner's guide. https://www.pycodemates.com/2023/01/multi-layer-perceptron-a-complete-overview.html

Sasaki, Y., et al. (2007). The truth of the f-measure. *Teach tutor mater*, *1*(5), 1–5.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell system technical journal*, *27*(3), 379–423.

Sparck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, *28*(1), 11–21.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929–1958.

Wei, J. W., & Zou, K. (2019). EDA: easy data augmentation techniques for boosting performance on text classification tasks. *CoRR*, *abs/1901.11196*. http://arxiv.org/abs/1901.11196

# 7 Appendix

## 7.1 TPE Pseudocode

Pseudocode taken from Multiobjective tree-structured Parzen estimator (Ozaki et al., 2022). TPE models $p(x_i|y)$ for each parameter in the search space make use of a threshold $y^*$ and two probability density functions $l(x_i)$ and $g(x_i)$.

**Algorithm 1** Tree-structured Parzen Estimator

**Require:**
$D = \{(x^{(1)}, y^{(1)}), ..., (x^{(k)}, y^{(k)})\}$ : observations
$n_t \in \mathbb{N}$ : number of iterations
$n_c \in \mathbb{N}$ : number of candidates
$\gamma \in (0, 1)$ : quantile

**for** $t \leftarrow 1, ..., n_t$ **do**
  $D_l \leftarrow \{(x, y) \in D | y$ is included in the best-$[\gamma|D|]$ objective values in $D\}$
  $D_g \leftarrow \frac{D}{D_l}$
  **repeat**
    $i \leftarrow i \in [1, n]$ such that $x_i$ is active and $x_i^*$ has not been sampled
    construct $l(x_i)$ with $\{x_i|(x, y) \in D_t\}$ and $g(x_i)$ with $\{x_i|(x, y) \in D_g\}$
    $C_i \leftarrow \{x_i^{(j)} l(x_i) | j = 1, ..., n_c\}$ ▷ sample $n_c$ candidates for $x_i^*$
    $x_i^*$ argmax$_{x_i \in C_i} \frac{l(x_i)}{g(x_i)}$ ▷ approximate argmax$_{x_i \in C_i} \frac{l(x_i)}{g(x_i)}$
  **until** all active parameters have been sampled
  $D \leftarrow D\{(x^*, f(x^*)\}$ ▷ $x^*$ is the vector composed of all sampled $x_i^*$
**end for**
**return x** with the minimum $y$ value in $D$

$l(x_i)$ is created using a subset of observed $x_i$ values $\{x_i^{(j)} \in X_i | y^{(j)} (= f(x^{(j)})) < y^*, j = 1, ..., k\}$, while $g(x_i)$ is created using the remaining observed $x_i$ values. The value $y^*$ is chosen as a quantile $\gamma \in (0, 1)$ of the observed y values, satisfying the condition $p(y < y^*) = \gamma$. Essentially, $l(x_i)$ represents the density of favorable $x_i$ values, while $g(x_i)$ represents the density of unfavorable $x_i$ values.

In each iteration, we approximate the optimal value of each active hyperparameter by selecting the candidate with the maximum $l(x_i)$ to $g(x_i)$ ratio. We update this value until we finish iterating.

### 7.2 Adam Optimizer Pseudocode

**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 $\beta_1^t$ and $\beta_2^t$ we denote $\beta_1$ and $\beta_2$ 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 $1^{\text{st}}$ moment vector)
$v_0 \leftarrow 0$ (Initialize $2^{\text{nd}}$ 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 12: Pseudocode of the Adam Optimizer algorithm. Taken from Kingma and Ba (2014)