Coding Project

Professor: D. Katsaros

Armpounioti Maria-Eleni 03183
Patakioutis Sofoklis 03310

Neuro-fuzzy Computing 2023-2024

Department of Electrical & Computer Engineering
University of Thessaly, Volos

{marmpounioti, spatakioutis}@e-ce.uth.gr

Table of Contents

Abstract. Text classification is one of the primary applications of machine learning models and especially neural networks. Various model architectures have been implemented and studied to determine the most efficient and accurate model in this task. In this project we were given a dataset of articles that belong in 2 levels of categories. We were asked to both manipulate the dataset to be ready for feeding the model and to train a neural network model with an architecture of our choosing. This report presents the steps and decisions we made during the whole process as well as the results we reached eventually.

Keywords: machine learning, neural networks, text classification

1 Introduction

Due to the inherent large size of the project we divided it into steps. In the *first step* we closely examined the dataset, made some statistical analysis and noted observations about the nature of the set, the information we thought of as useful and the balance between the articles of various categories. The *second step* was about manipulating and preprocessing the dataset to prepare the input that was going to feed the model. In the *third step* we designed our model architectures after extensive search and performed numerous trainings until the model metrics reached sufficient numbers. The *fourth and final step* included the testing metrics to evaluate the model's generalization degree and the collection and processing of all the results we got.

Throughout the whole project steps we used Python and run everything on a Google Colab notebook (more technical details will be listed in a separate section).

2 Dataset analysis

2.1 Nature of the dataset

The dataset we were given ('news classification.csv') contains 10917 articles that belong in 17 primary and 109 secondary categories. Each article tuple has the following columns of information: *(data_id, id, date, source, title, content, author, url, published, published_utc, collection_utc, category_level_1, category_level_2).*

Out of these columns, we decided that the useful ones are the 'date', 'source', 'title', 'content' and of course the target output columns 'category_level_1' and 'category_level_2'.

The other ones in our opinion were redundant, repeating the same information or not contributing at all to each article's category classification. For example the 'id' attribute repeats the 'date', 'source' and 'title' column information while the 'data_id' one is just an indexing of the articles in the csv file and contains no information about the article's theme.

2.2 Distribution of articles over the classes

Data imbalance is a common issue when attempting to train neural network models. It occurs when one or more classes have significantly more training samples than the rest of the classes. If not taken into account this usually leads to the model training and fitting well only on the classes that have a lot of samples and not fitting on the rest of the classes, resulting in poor accuracy metrics and the model lacking generalization.

With that in mind, it is extremely important to take into account the article distribution over the classes. Thus, we run some statistic metrics on the dataset to observe how many articles each level 1 and level 2 class has available to train on. The analysis produced the following results:

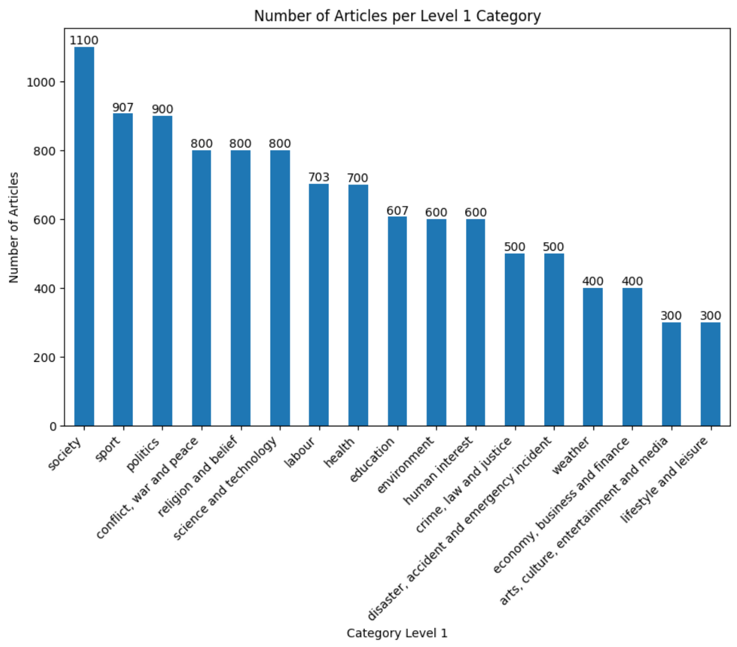


Fig. 1. Article distribution on Level 1 classes

We observed **major** data imbalance between articles over the first level of classes as shown in the figure above. We noted this fact and took it into account when manipulating the dataset and when figuring out the model's training settings (more details about this in the next sections).

On the contrary, we observed that in the second level of classes the articles distribution was almost perfect across the categories, so we do not need to care for data imbalance when designing the second level classifier of our model. Here are some examples of distributions, for some of the level 2 classes:

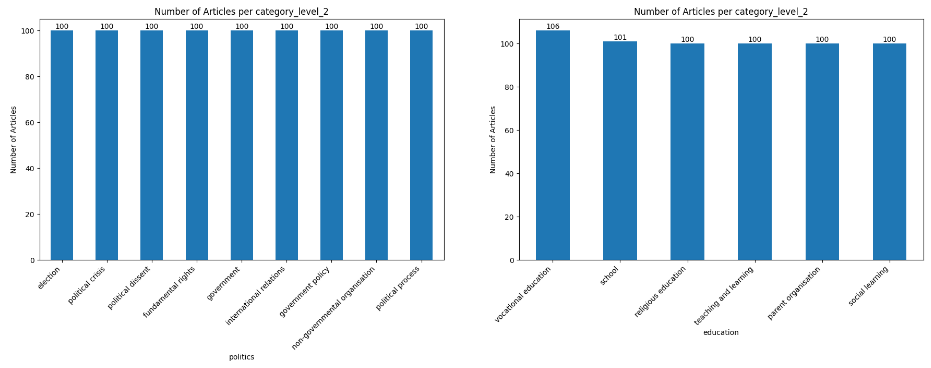


Fig. 2. Article distribution on level 2 classes examples

2.3 Article Length Distribution

Naturally, the length of the articles content on its own and when combined with the other input columns is varying. However, the model requires a constant input length to function. As a result, inputs larger than this length need to be truncated and inputs shorter that this length need padding (more details on this process later). We need to decide this input length in a balanced way. If the number is too large, most articles will need to add padding to meet the requirements leading to a lot of useless information in each training sample. Similarly, if the number is too short, most articles will be truncated leading to a massive loss of information in each training sample. With that in mind, we decided to produce the following figure showing the article length distribution over all articles:

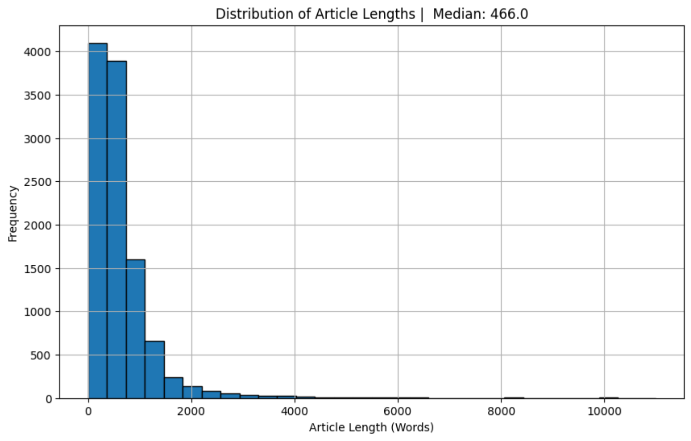


Fig. 3. Article length distribution

We observed from the figure that most articles have relatively short sizes, and we calculated the median of all article lengths to be approximately 466 words. We decided to use this value as the optimal input length.

2.4 Project implementation

For this process we created a simple set of functions:

- **visualize_dataset** (dataset, column, main_category)
- **visualize_article_length_distribution** (dataset)

3 Dataset manipulation

3.1 Splitting process

The dataset files were uploaded to cloud (Google Drive) to be easily accessible by the Google Colab notebook where the project was developed. The process we followed is illustrated in the figure below:

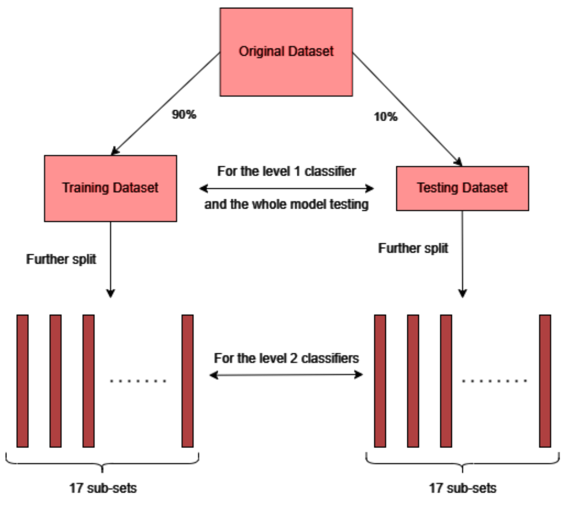


Fig. 4. Dataset split visualization

We decided to keep *10%* of the dataset for testing. To achieve this, we separated the original file in two parts, the training set and the testing set. This was enough for the level 1 training and testing but we needed to further split the dataset for the level 2 training process. The need for this extra split was based the approach we followed in designing the model architecture (which we will describe thoroughly in a later section). We split both the training and testing dataset into 17 sub-datasets so that each sub-set has only one level 1 class and stored them in separate files. While splitting the dataset into training and testing parts we made sure that the proportions of input samples between the classes remained the same. We used the function *'train_test_split'* from *'sklearn'* library for this process. The use of the parameter *'stratify'* ensured the ratio consistency we mentioned above.

3.2 Project implementation

For this process we created a simple set of wrapper functions as a small interface:

- **load_dataset** (filepath, columns)
- **save_dataset** (dataset, filepath)
- **dataset_split_train_test** (dataset, test_size, stratify)
- **dataset_split_into_categories** (dataset)

4 Dataset preprocessing

4.1 Processing steps

The given dataset is in a 'csv' format, which is not in a format ready to be used as input for the model. The columns need to be passed through a sequence of processing steps and then be merged together to form a training sample input. Also, the target output of the model also needs some encoding to fit in a classification model. Through extensive search we decided to pass the dataset into the following processing steps:

- **Stopword removal**: This is a process for removing stopwords (e.g. 'and', 'a', 'the', 'is') and punctuation marks (e.g. '**.**' , '**,**' , '**!**' , '**?**') , which are generally not useful for the text analysis as they do not contribute to a *Natural Language Processing* task. By removing these we decrease the size of the sentences. We created a set of these from *nltk* library and excluded them from tokenization.

- **Tokenization**: This is the process of cutting the sentence into tokens (words). For example if we tokenize the sentence *'I love machine learning'*, the result of the tokenization process will be the array [*'I', 'love',* *'machine', 'learning'*]. Its goal is to facilitate text analysis by transforming text into manageable pieces as a block of language or a vocabulary. We used *'word_tokenize'* function from *nltk* library for this procedure.

- **Lemmatization**: This is a technique which transforms words into their base or root forms, known as lemmas. For example it turns the adjective 'better' to 'good' or the verb 'is' to 'be'. Lemmatization aims to map different forms of a word to a single form. We used *'WordNetLemmatizer'* object from *'nltk'* library for this procedure.

  Note: We initially also used* ***stemming* *which is a technique which reduces words to their root form by removing suffixes and prefixes. For example, the words 'programming', 'programmer', and "programs" can all be reduced down to the common word stem 'program'. However, we observed slightly worse model performance and decided to eventually remove it from the process*

- **Column merging**: Up until this step, the columns being processed were *'title'* and *'content'* separately. In this step, those two columns are merged along with *'date'* and *'source'* to form sentences. We concatinated the *'title'*, *'date'* and *'source'* at the beginning of the sentence because they are short in size and we did not want them to be truncated during the padding process.

- **Vectorization**: Vectorization is the process of converting data into numerical vectors that represent essential features. It allows data to be processed in parallel, significantly speeding up computation. We used the *'Tokenizer'* object from *'tensorflow.keras'* library. The *'Tokenizer'* fits on the specific dataset words so we needed to save it along with the model after the training, in order to use the same one in the testing dataset preprocessing. After the fitting, the *'Tokenizer'* has a vocabulary to match each word to a unique integer index. We performed this transformation of words to integer indexes with the *'texts_to_sequences* function that the *'Tokenizer'* provides.

- **Padding**: This is the process which ensures that all sequences of the input will have the same length (in this case the input length we decided earlier). If a sequence has length less than the decided length we add padding after the end of the sequence, and if a sequence is longer it gets truncated. We used *'pad_sequences'* function from *'tensorflow.keras'* library to perform this procedure.

- **Output label encoding**: This is a technique that is used to convert categorical columns into numerical ones. It aids in handling the categorial data and in efficient memory management. We used the *'LabelEncoder'* object from *'sklearn'* library for this process. The *'LabelEncoder'* fits on the specific dataset categories so we need to save it along with the trained model and the tokenizer to be used again in the testing sets. After fitting, we mapped from every category to an integer using the *'transform'* function of the *'LabelEncoder'* and then represented each integer as an one-hot encoded vector using the *'to_categorical'* method of the *'LabelEncoder'*.

A summary figure of the above steps is shown below:

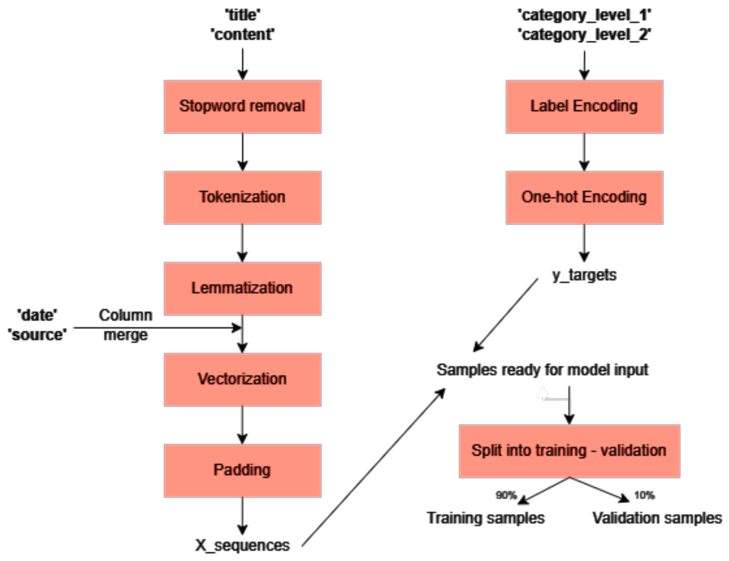


Fig. 5. Dataset preprocessing visualization

The last step before the training can begin is to split it into a 90% training set and a 10% validation set to monitor generalization and overfitting during the training process. For this step we used the function *'train_test_split'* from *'sklearn'* library. The use of the parameter *'stratify'* ensured the ratio consistency between classes that we mentioned earlier.

4.2 Project implementation

For this procedure we created a simple set of wrapper functions and a larger function that handles the whole procedure of preprocessing for us as a small interface:

- **preprocess_text** (text)
- **vectorize_data** (X_text, tokenizer)
- **add_padding** (sequences, max_len)
- **encode_target_outputs** (encoder, y_labels_encoded, num_classes)
- **dataset_split_train_validate** (X_sequences, y_targets, validation_size, stratify_category)
- **prepare_dataset_for_model** (dataset, max_words, category_level, num_classes, tokenizer, encoder)

5 Model Architecture - Training

5.1 General approach - Model Structure

Having in mind the nature of our task (***Hierarchical Multi-Label Text Classification***) we did some research on articles about the various suggested approaches to design our model. We decided to use Local Classifiers and specifically ***Local Classifier per Parent Node (LCPN)***.

The idea is that we have a bigger model for level 1 classification and then 17 models (one for each category) for level 2 classification. Each classifier is trained individually with its respective train dataset. Afterwards, when all trainings reach sufficient metrics, we connect the 2 levels and evaluate the model's performance (accuracy - inference time) on the test dataset. The idea is to take advantage of the information provided by the level 1 classifier (the level 1 category that the model predicts and trigger only the corresponding level 2 classifier.

Advantages - Drawbacks The main advantage of this architecture is its simplicity and clear structure. It is also easier to perform the level 2 trainings, as each model focuses on one class only. On the other hand, this method conveys a major disadvantage: the model's prediction depends heavily on predicting correctly the level 1 category. If the model makes a mistake there, the prediction on the level 2 category will be wrong 100% because the input will be sent to the wrong level 2 classifier.

Connecting the levels To evaluate the model's performance on the testing set, we load the whole model in a structure. Then for each sample of the testing set we perform the following steps:

1. Preprocess input sample for the level 1 classifier.
2. Predict level 1 category using the model.
3. Check if the prediction was correct. If not, we do not move forward to level 2 classification (no point, prediction will be wrong).
4. Assuming level 1 prediction was correct, take the category predicted by the level 1 classifier and preprocess the original input sample for the corresponding level 2 classifier.
5. Predict level 2 category using the model.
6. Check if the prediction was correct. If it was, increase correct predictions count by 1.

When all samples are passed through this process, we calculate the following metrics:

$$accuracy = \frac{correct\_predictions}{total\_predictions}$$

$$total\_inference\_time = \sum inference\_time\_per\_sample$$

$$average\_inference\_time = \frac{\sum inference\_time\_per\_sample}{total\_samples}$$

Here is a figure that better illustrates the way our model works:

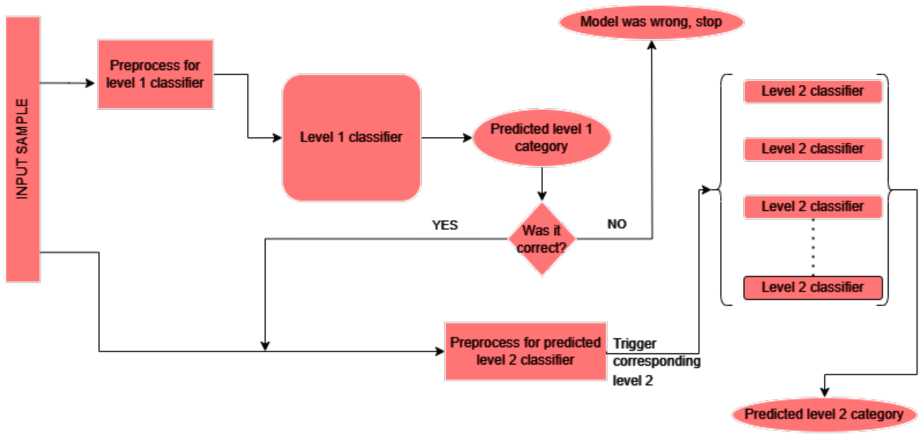

INPUT SAMPLE
Preprocess for level 1 classifier
Level 1 classifier
Predicted level 1 category
Model was wrong, stop
Was it correct?
YES
NO
Level 2 classifier
Level 2 classifier
Level 2 classifier
Level 2 classifier
Preprocess for predicted level 2 classifier
Trigger corresponding level 2
Predicted level 2 category

Fig. 6. Model Architecture

5.2 Level 1 Classifier

Architecture The first level classifier gets an input sample and produces the probability of belonging in each of the 17 level 1 categories. We firstly experimented to implement the first level classifier using CNN's. We tried one CNN layer combined with a dense layer before the dense/output layer, then two and so on. As the results were not satisfying, we added a bidirectional LSTM layer. Also we inspected a diversity of dropout degrees before and after the CNN or LSTM layers, we tried max and average pooling, we examined a variety of activation functions such as ReLu, Leaky ReLU, Softmax, Sigmoid, Linear etc and we added more than one LSTM layers. After a lot of trials, we observed that CNN layers were not conducing to better performance and by using only bidirectional LSTM layers the accuracy degree was improved. That's why we decided to remove CNN layer and keep just one Bidirectional LSTM layer.

Analytically, we came to the final structure implementation below:

1. **Embedding layer** : We used the pre-trained *'en_core_web_md'* embedding layer by *'spacy'* library which is a Word2Vec pre-trained variant, which maps every word to a vector, and we set trainable to *'true'* so that the layer further fits on our data.
2. **Spatial Dropout** (0.4) : It prevents overfitting by 'numbing' some of the feature maps. The higher its value, the more probable the 'numbing' is on each feature map.
3. **Bidirectional LSTM layer** : It is consisted of 256 neurons and it captures dependencies in both forward and backward directions.
4. **Spatial Dropout** (0.5)
5. **Global Max-pooling** : It downsamples the input representation by taking the maximum value from every feature map.
6. **Dense-Output layer** : It is consisted of 17 neurons and it classifies the input into one of the 17 first level categories based on the probabilities produced by 'Softmax' activation function.

We used the following training parameters:

1. **Optimizer - Learning rate**: Our model uses Adam optimizer with starting learning rate value equal to 0.001. According to internet research, Adam optimizer is the most popular and widely used optimizer and combines the advantages of AdaGrad and RMSProp optimizers. Adam adjusts the learning rate for each parameter by itself, requires less memory than other optimizers and is computationally efficient.

2. **Class weights**: In imbalanced datasets some classes are underrepresented, so the network does not fit sufficiently and uniformly. As we have already mentioned, there are imbalances between the number of articles of level 1 categories so to prevent the above issue we used *'compute_class_weight'* by *'sklearn'* library during the training procedure. By using this function, classes with rarer instances are assigned with higher weights and classes with more frequent instances are assigned with lower weights. As a result, the model's performance is improved in imbalanced dataset.

3. **Epochs**: Maximum number of epochs is set to 100.
4. **Early Stopping**: During the training process, we have used *'EarlyStopping'* by *'keras.callbacks'* library. Early stopping prevents overfitting by stopping the training process when the validation loss does not improve. We have set a patience of 10 epochs and the parameter restore_best_weights to *'true'*, which restores the weights which provoked the least loss.

5. **Batch size**: After trial and error procedure we have set the batch size equal to 120.

Training We kept track of the model's performance and on the training and validation accuracy and loss. Here are the final performance metrics and a figure of the accuracy and loss history thought the epochs.

$training\ \ accuracy = 94.8\%$
$validation\ \ accuracy = 80.08\%$
$validation\ \ loss = 0.66$

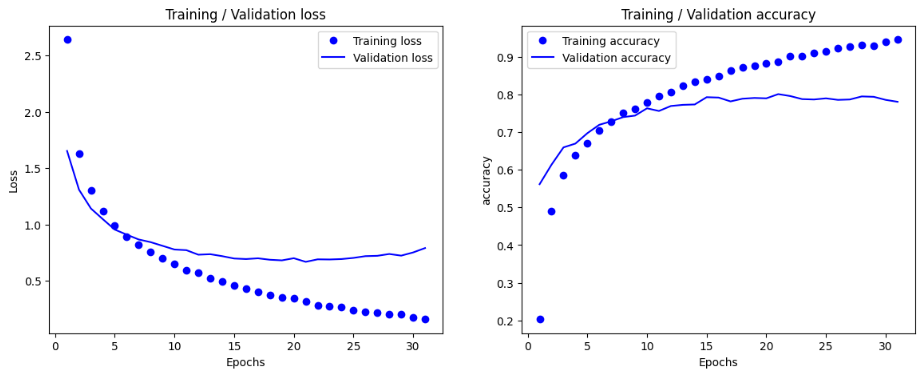


Fig. 7. Level 1 accuracy / loss

We also tested the level 1 model for generalization on the testing set and it produced the following results:

$total\ inference\ time = 1\ second$
$accuracy\ = 77.2\%$

5.3 Level 2 Classifiers

The second level classifiers get an input sample and produce the probability of belonging in one of the level 2 subcategories of the main category that the subclassifier represents.

Architecture Across all the level 2 classifiers we used a similar architecture. We generally know that CNN's are usually *'data hungry'* networks and since the first level of the model did not perform well when using them, we decided that there was no point in attempting to use them in the sub classifiers which have significantly smalller datasets to train on. As a result, our approach from the beginning was to make use of LSTM's again which proved to perform better with text sequences and smaller sets.

We used the same structure for all 17 models except the parameters *'lstm_size'*, *'batch_size'* and *'spatial_dropout'*. Here is the architecture we followed in the level 2 classifiers:

1. **Embedding layer** : We again used the pre-trained *'en_core_web_md'* embedding layer by *'spacy'* library, which maps every word to a vector, and we set trainable to *'true'* so that the layer further fits on our data.
2. **Bidirectional LSTM layer** : It is consisted of *'lstm_size'* neurons and it captures dependencies in both forward and backward directions.
3. **Spatial Dropout** (*'spatial_dropout'*): It prevents overfitting by 'numbing' some of the feature maps.
4. **Global Max-pooling** : It downsamples the input representation by taking the maximum value by every feature map.
5. **Dense-Output layer** : It consists of a number of neurons equal to the number of subcategories that each main class has and it classifies the input into one of these subcategories based on the probabilities produced by 'Softmax' activation function.

We also used the following training parameters:

1. **Optimizer:** Similarly to the level 1 training, we decided to use the Adam optimizer with a learning rate of 0.001. We experimented with some other values (0.0015, 0.002) but we did not notice any improvements in the performance of any model so we sticked to 0.001.

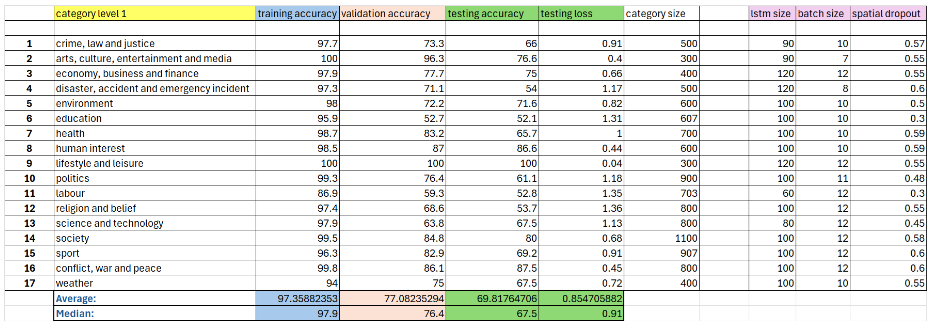
2. **Epochs:** We trained each model with a maximum of 35 epochs.

3. **Early Stopping:** We again used the early stopping setting with a patience of 7 epochs if the validation loss did not improve and the parameter restore_best_weights to *'true'*, which restores the weights which provoked the least loss.

4. **Batch Size:** We experimented for each model with various batch sizes of small size (6 to 15) due to the small sizes of the datasets. Each model has its own batch size as we mentioned earlier.

In these trainings class weights were not necessary because as we mentioned in the section *'Distribution of articles over the classes'* the articles distribution in each category over the subcategories is almost perfectly balanced.

Training We kept track of every model's performance and saved the training and validation performances on the epoch that had the smallest validation loss. We also tested each model for generalization on its own sub-set of the testing set. Here are the tabulated metrics we saved as well as the parameters that differ for each model:



	category level 1	training accuracy	validation accuracy	testing accuracy	testing loss	category size		lstm size	batch size	spatial dropout
1	crime, law and justice	97.7	73.3	66	0.91	500		90	10	0.57
2	arts, culture, entertainment and media	100	96.3	76.6	0.4	300		90	7	0.55
3	economy, business and finance	97.9	77.7	75	0.66	400		120	12	0.55
4	disaster, accident and emergency incident	97.3	71.1	54	1.17	500		120	8	0.6
5	environment	98	72.2	71.6	0.82	600		100	10	0.5
6	education	95.9	52.7	52.1	1.31	607		100	10	0.3
7	health	98.7	83.2	65.7	1	700		100	10	0.59
8	human interest	98.5	87	86.6	0.44	600		100	10	0.59
9	lifestyle and leisure	100	100	100	0.04	300		120	12	0.55
10	politics	99.3	76.4	61.1	1.18	900		100	11	0.48
11	labour	86.9	59.3	52.8	1.35	703		60	12	0.3
12	religion and belief	97.4	68.6	53.7	1.36	800		100	12	0.55
13	science and technology	97.9	63.8	67.5	1.13	800		80	12	0.45
14	society	99.5	84.8	80	0.68	1100		100	12	0.58
15	sport	96.3	82.9	69.2	0.91	907		100	12	0.6
16	conflict, war and peace	99.8	86.1	87.5	0.45	800		100	12	0.6
17	weather	94	75	67.5	0.72	400		100	10	0.55
	Average:	97.35882353	77.08235294	69.81764706	0.854705882					
	Median:	97.9	76.4	67.5	0.91					

Fig. 8. Level 2 training / testing data

The models trained with an average training accuracy of **97.36%** and average validation accuracy of **77.08%**. The individual testings produced an average accuracy of **69.82%** and average loss of **0.85**. We observed that some classifiers performed really well on their training while others either overfitted or did not even fit sufficiently but we were not able to improve the performance even with a lot of parameter tuning.

Here are some figures that better visualize the tabulated data we presented above:

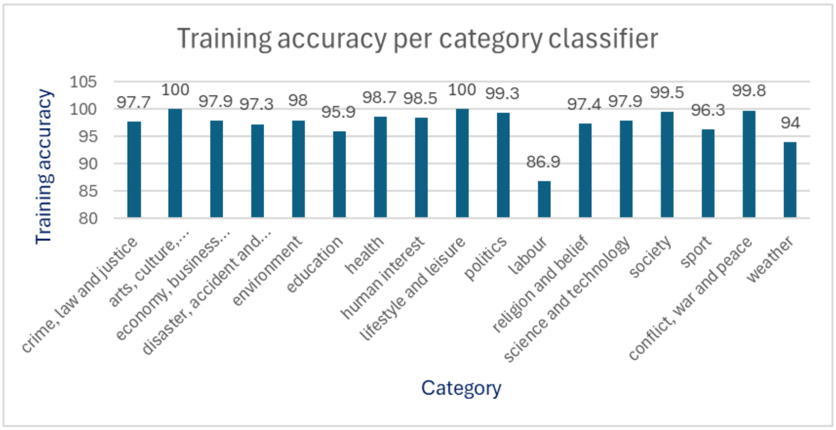


Fig. 9. Level 2 training accuracy per classifier

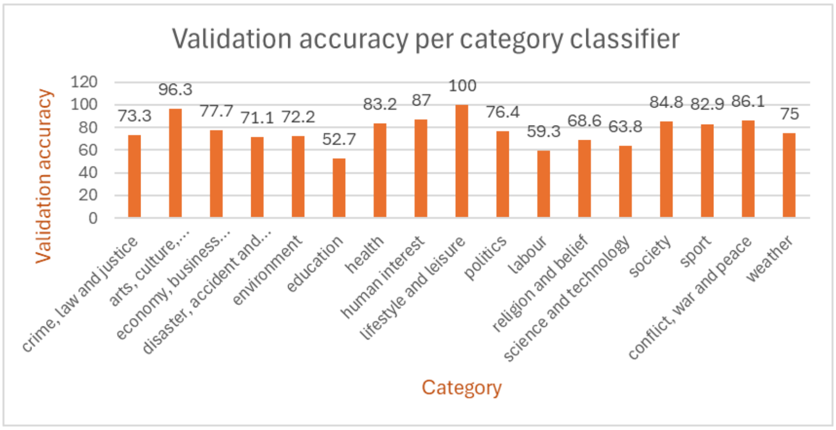


Fig. 10. Level 2 validation accuracy per classifier

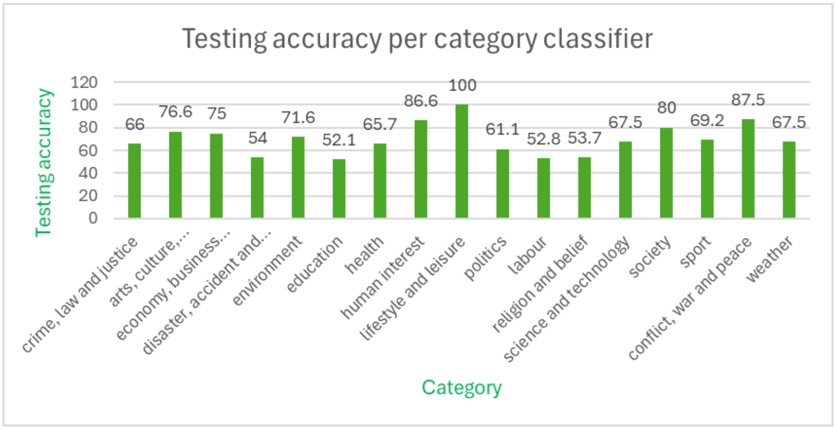


Fig. 11. Level 2 testing accuracy per classifier

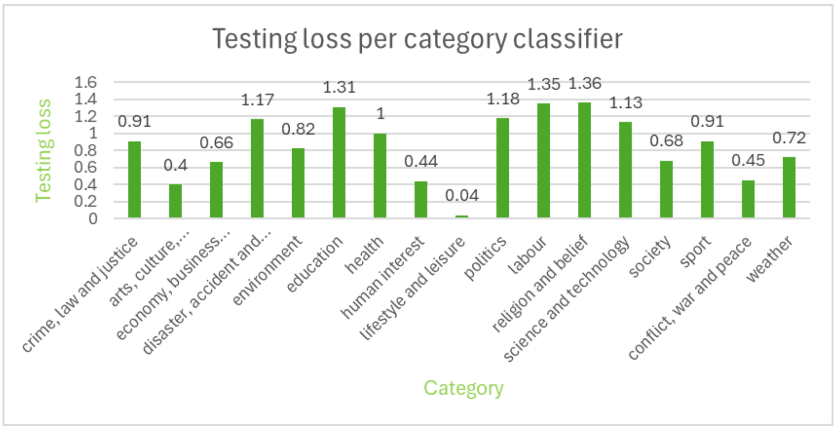


Fig. 12. Level 2 testing loss per classifier

5.4 Project implementation

Trainings We have created multiple sections in the notebook of the project that perform the the trainings we described. The section has the title *'Separate Model Trainings and Testings'*. Inside this section the cells are split into the sections *'Category Level 1 Classifier'* and *'Category Level 2 Classifiers'*.

Inside the *'Category Level 1 Classifier'* you can find titled cells for loading and preprocessing the training set, defining the model and train it, saving / loading the model, loading and preprocessing the testing set and testing the level 1 model on this set.

Inside the *'Category Level 2 Classifiers'* you can first choose the training parameters for each class and the model general architecture and then either train and save / test all the sub classifiers at once (in the sections *'Train /Test all sub classifiers at once'*) or train and save / test one classifier at a time (in the section *'Train / Test single subclassifier (for trials)'*. In both sections you will find titled cells for loading and preprocessing the training set, defining the model and train it, saving / loading the model, loading and preprocessing the testing set and testing the level 2 model on this set.

Model Connection We have created a section at the end of the project that has the title *'Model levels connection / Testing both levels'*. There we firstly have a cell for loading the entire model into a python dictionary called *'complete_model'*. The dictionary contains a label *'main_classifier'* and a label *'subclassifiers'*. Inside the subclassifiers, another nested dictionary exists with labels matching the level 1 categories. Each label of this dictionary and the *'main_classifier'* label have as a value a dictionary each, containing the model, tokenizer, label encoder and configuration settings of the specific classifier. This structure makes it simple to access the models various parts. After this, we load the test set and we have the loop to iterate through each sample and test the model.

6 Model Testing - Results

Using the code we described above, we iterate through each test sample. Each sample is passed through the process we described in the *General approach - Model Structure'* section, measuring the inference time of the model and counting the correct predictions and then we produce the metrics according to these measurements. Here are the model's performance metrics on the test dataset:

$accuracy = 54.12\%$
$total\ inference\ time = 164.72\ seconds$
$average\ inference\ time = 0.15\ seconds$

7 Technical Details

The whole project was developed in **Python**. We used a large number of libraries. Here is a list:

- **Pandas** for dealing with the dataset files.
- **Matplotlib** for dealing with plots.
- **Scikit-learn** for multiple steps of the dataset preprocessing and model trainings.
- **Tensorflow / Keras** for multiple steps of the dataset preprocessing as well as defining, training and testing the models.
- **nltk** for text preprocessing
- **os, pickle, json** for saving / loading model valuable information such as tokenizers, encoders etc.
- **Spacy** for obtaining the pretrained Word2Vec embedding model.
- **NumPy** for various math operations.
- **time** for time measurements.

The project is in the form of a **Jupiter Notebook**. We chose this form to be able to add useful titles to different sections of the code. In order to run the project we used the cloud environment **Google Colab** which offers cloud runtimes in Google's backend. We chose this platform instead of our local machines because it offers powerful GPU accelerators that drastically improved the training speed of all models. Specifically, we performed all trainings and testings using the **T4 GPU** as an accelerator.

Due to their large size, all the models and their information as well as the dataset files and the project source code are uploaded to Google Drive inside the folder *'Project Neural Networks'*. You can access it **here**.

The folder structure inside this Google Drive folder is illustrated in the figure below:

Some clarifications:

- All model folders inside the *'Models'* folder are named after their subclassifier's category. The level 1 classifier is named *'main_classifier'*.
- The same naming conventions are followed throughout all the training and testing dataset files.
- Inside each model folder you will find the model architecture and weights in the file *'model.h5'*, the tokenizer in the file *'tokenizer.pickle'*, the output label encoder in the file *'label_encoder.pickle'* and some configuration settings (model input length and some hyperparameters) in the file *'config.json'*.

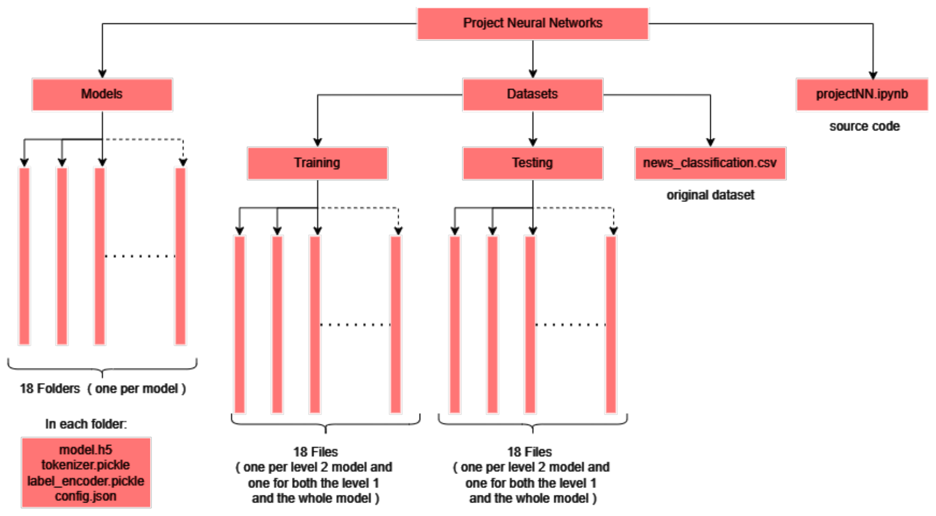


Fig. 13. Google Drive project folder structure

8 Bibliography

- **Tokenization**:
  `https://www.datacamp.com/blog/what-is-tokenization`

- **Lemmatization**:
  `https://www.geeksforgeeks.org/python-lemmatization-with-nltk/`

  `https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html`

- **Stemming**:
  `https://www.datacamp.com/tutorial/stemming-lemmatization-python`

- **Data vectorization**:
  `https://medium.com/@kaushikvikas/understanding-vectorization-applications-benefits-and-future-trends-d45b8798fa1e#:~:text=Vectorization%20converts%20data%20into%20numerical,scales%20well%20with%20large%20datasets.`

- **Label encoding**:
  `https://medium.com/@sunnykumar1516/what-is-label-encoding-application-of-label-encoder-in-machine-learning-and-deep-learning-models-c593669483ed`

  `https://www.geeksforgeeks.org/ml-label-encoding-of-datasets-in-python/`

  `https://www.mygreatlearning.com/blog/label-encoding-in-python/#:~:text=Label%20encoding%20is%20a%20simple,input%20into%20machine%20learning%20algorithms.`

- **Natural Language Processing (NLP)**:
  `https://medium.com/swlh/step-by-step-building-a-multi-class-text-classification-model-with-keras-f78a0209a61a`

- **Local Classifiers**:
  `https://towardsdatascience.com/hierarchical-classification-with-local-classifiers-down-the-rabbit-hole-21cdf3bd2382`

- **Classification with CNNs**:
  `https://medium.com/voice-tech-podcast/text-classification-using-cnn-9ade8155dfb9`

  `https://towardsdatascience.com/how-to-do-text-classification-using-tensorflow-word-embeddings-and-cnn-edae13b3e575`

- **Data imbalance**:
  `https://www.linkedin.com/advice/0/how-can-you-handle-class-imbalance-text-classification-y6vue`

- **LSTM Models**:
  `https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/`

  `https://www.analyticsvidhya.com/blog/2021/06/lstm-for-text-classification/`

  `https://nzlul.medium.com/the-classification-of-text-messages-using-lstm-bi-lstm-and-gru-f79b207f90ad`

https://medium.com/@CallMeTwitch/building-a-neural-network-zoo-from-scratch-the-long-short-term-memory-network-1cec5cf31b7

https://www.youtube.com/watch?v=YCzL96nL7j0

https://www.youtube.com/watch?v=b61DPVFX03I&t=130s

– **Bidirectional LSTM**:
https://medium.com/@anishnama20/understanding-bidirectional-lstm-for-sequential-data-processing-b83d6283befc

https://www.geeksforgeeks.org/bidirectional-lstm-in-nlp/