5 Methods

As mentioned in section 2, the methods for this project will include text pre-processing, feature vector generation using the Word2Vec technique, training the deep learning classifiers, and evaluating the performance metrics to determine the best classifier for ADD classification in mailing lists. In addition to the mentioned steps, it is necessary to tune the hyper-parameters, such as learning rate and batch size, so an additional step of hyper-parameter optimization is necessary before training the classifiers. The high-level overview of the process is shown in figure 9.

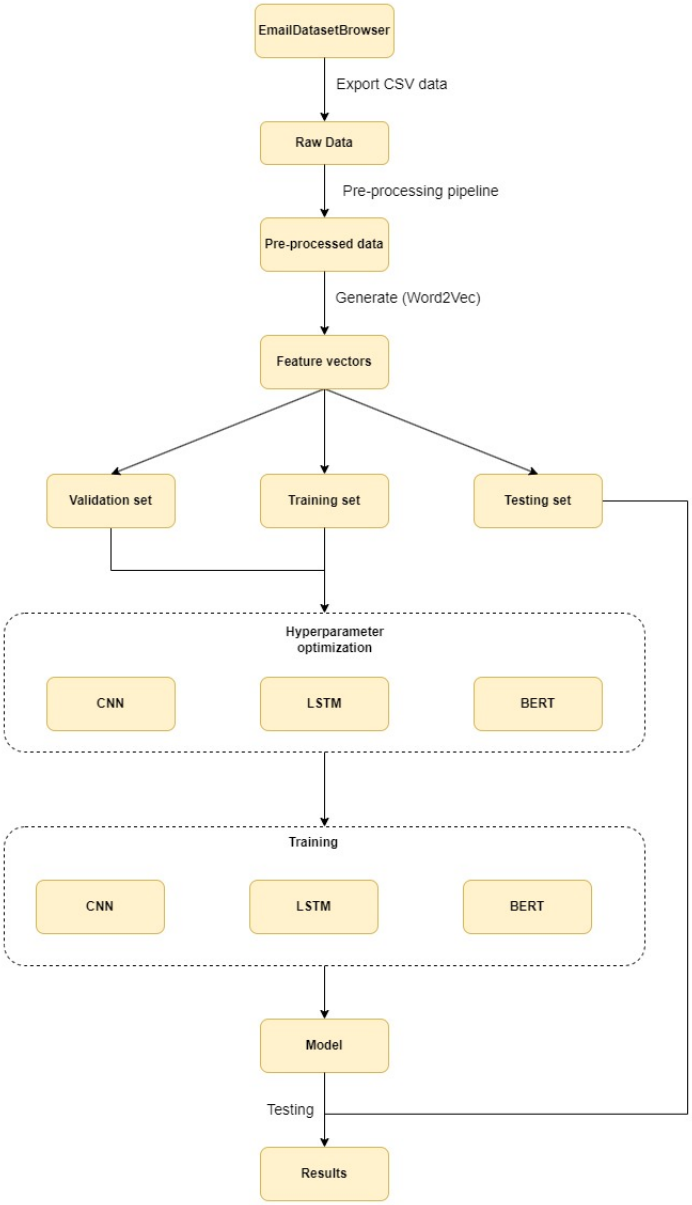

Figure 9: Process overview

First, the raw dataset is exported from the EmailDatasetBrowser [7] in CSV format, and the pre-processing pipeline is applied to it. Then, feature vectors are generated and split into training, testing, and validation sets. The training and validation sets are then used to optimize hyperparameters and

train the classifiers. Finally, the testing set is used to evaluate the performance of the classifiers. The following sections explain each step in more detail.

5.1 Pre-processing

Before training the classifiers, it is important to remove any irrelevant information from the email texts. The following figure shows the pre-processing pipeline:

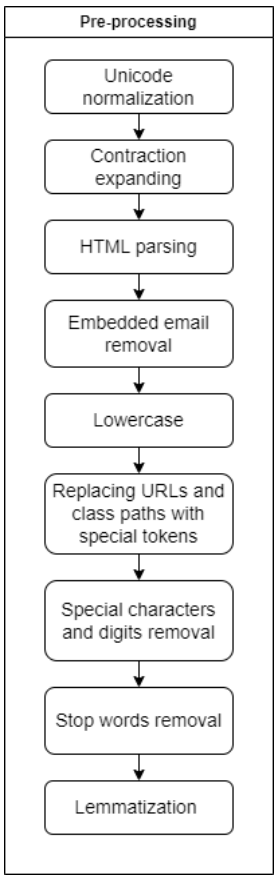

Figure 10: Text pre-processing pipeline

5.1.1 Unicode normalization

Unicode normalization is performed to normalize unicode data. This will, for instance, remove umlauts and accents from characters (e.g., *café* will become *cafe*). To achieve this, *normalize_unicode* function from *text-preprocessing*[1] library for *Python* was applied to email texts.

5.1.2 Contraction expanding

This step removes contractions, which will later help in stopword removal. For instance, if a sentence contains *'I've'*, it will be expanded to *'I have'*, and both words will be later removed, since both of them are stopwords. This was achieved by using the *expand_contraction* function from the same *text-preprocessing* library as the one used in unicode normalization.

[1] https://pypi.org/project/text-preprocessing/

5.1.3 HTML parsing

This step is necessary since some emails contained text inside HTML tags when exported from EmailDatasetBrowser [7]. Parsing HTML ensures that the tags are removed, but the text inside these tags is kept. This is achieved by using the HTML parser from the *Beautiful Soup*[2] library.

5.1.4 Embedded email removal

Some emails contained their parent emails when exported from EmailDatasetBrowser [7]. To avoid having duplicates in the dataset, embedded emails are removed by using several regular expressions. Whenever a parent email is embedded inside its child email, it starts with one of the common patterns that can be found using a regular expression. If such a pattern is found, all the text following it is erased. The following regular expressions were used:

```
re.compile('On .+ wrote:'),
re.compile(".+ hat am \\d+\\.\\d+\\.\\d+ \\d+:\\d+ geschrieben:"),
re.compile("From:[\\S\\s]*Date:[\\S\\s]*To:[\\S\\s]*Subject:", re.MULTILINE)
```

5.1.5 Lowercase

This step ensures that all the text is in lowercase by using the *lower* method of *Python's* built-in string class.

5.1.6 Replacing URLs and class paths with special tokens

Since the dataset contains emails between software developers, they often contain URLs and class paths (i.e., *java.util.concurrent.FutureTask*). URLs and class paths do not contribute significant information to the classification task, and can introduce noise to the data. Therefore, they are replaced with special tokens <*url*> and <*classpath*>. These tokens are found inside email texts using the following regular expressions:

```
url_pattern = r'(https?://\S+)'
classpath_pattern = r'\b(org\.|class\.|com\.)[\w.]+'
```

5.1.7 Special characters and digits removal

Special characters, such as $ and %, as well as digits, do not contribute information that is beneficial for classification, and are therefore removed from emails. These special characters can be found using the following regular expression:

```
r'[^a-zA-Z0-9\s]
```

This expression matches characters that are not a letter (uppercase or lowercase), not a digit, and not a whitespace character.

[2] https://pypi.org/project/beautifulsoup4/

5.1.8 Stop words removal

Removing stop words is a common technique in natural language pre-processing. Stop words are very common words in a specific language that do not carry significant meaning on their own. In English, stop word examples would include words like *'the', 'is'*, and *'are'*. To get a list of stopwords, the *Natural Language Toolkit (NLTK)*[3] package was used, and more specifically it's *corpus* module. Each email was tokenized using *NLTK's word_tokenize* function, and individual words were filtered based on whether the word is in the stopword list.

5.1.9 Lemmatization

The last step in the pre-processing pipeline is lemmatization, which is a process of reducing inflected words to their root word (e.g., *better* $\rightarrow$ *good*). Lemmatization helps in reducing the complexity of text data, while aiding in focusing on the meaning of the word. Lemmatization was implemented using *NLTK's WordNetLemmatizer* class.

5.2 Word embedding

To train deep learning networks such as CNN and LSTM, it is first necessary to transform words into numerical representations that such networks are able to understand. One way of achieving that is creating a word embedding for each word in the input text.

Word embeddings are generated by training a neural network model on a large body of text. The model learns to predict word occurrences within specific contexts, resulting in similar numerical vectors generated for similar words.

For this project, word embeddings of length 200 were generated by training a model on the whole email dataset, including emails that were not labeled. *Gensim*[4] library for *Python* was used to train the model. The dataset underwent the pre-processing pipeline from Figure 10 before training. It was necessary to choose between skip-gram (SG) and continuous bag of words (CBOW) algorithms when generating word embeddings. The difference between the two algorithms is that CBOW predicts the word given its surrounding words, while SG works by predicting surrounding words given the current word [13]. It was decided to use SG, since it achieves a higher semantic accuracy than CBOW (55% and 24%), while its syntactic accuracy is only slightly less (59% and 64%) [13]. Moreover, the results produced by SG seem intuitively better than results produced by CBOW. Figures 11a and 11b show 10 most similar words to *'python'* and *'java'* after the training process.

(a) CBOW result

(b) Skip-gram result

Figure 11: CBOW and Skip-Gram results

[3]https://pypi.org/project/nltk/
[4]https://pypi.org/project/gensim/

In addition to the generated word embeddings, a Word2Vec model that was pre-trained on 15GB of *Stack Overflow* posts was used [14]. This model contains word embeddings of length 200, which is the reason why the generated embeddings are of the same length, since it ensures a fair comparison between the two. From this point, the word embeddings generated from the email dataset will be referred to as 'custom', while the ones obtained from the pre-trained Word2Vec model will be referred to as 'pre-trained'. The differences in the classifier performance based on whether the custom or pre-trained word embeddings were used are included in section 7.

5.3 Classifiers

This section will describe the 3 classifiers that were trained for this project. When descibing the architecture of the classifiers, the optimized hyperparameters are used, which were obtained during the hyperparameter optimization described in section 6.5.

The output of each classifier has 4 dimensions corresponding to the 4 ADD types described in section 3.1. If all 4 outputs are below a certain threshold, the email is classified as non-architectural. This has 2 advantages over using 5 outputs, where the 5^{th} output would correspond to a *not-ak* label:

- The classifier would not be able to classify an email as architectural and non-architectural at the same time.
- In case of 5 outputs, all outputs can be below a certain threshold, in which case an email would not be classified as either architectural or non-architectural. Using 4 outputs eliminates this issue.

It was decided to use sequences of 500 words as inputs to the classifiers. By looking at figure 8, it can be seen that by choosing the length to be 500, the majority of the emails would be included in their entirety. Moreover, it is always clear what types of ADDs an email contains within the first 500 words. Since some emails are shorter than 500 words, they are padded with padding vectors. To ensure that the model does not learn anything from the padding vectors, an embedding layer in *TensorFlow*[5] provides a *mask_zero* argument, which is set to true for this project, thus ensuring that padding vectors are ignored. The DistilBERT classifier uses a different mechanism to ignore padding vectors, which is described in section 5.3.3

5.3.1 CNN

Convolutional neural networks typically consist of convolutional, pooling, and dense layers. The convolutional layer is responsible for performing the convolution operation, which is simply a multiplication of a part of the input with the filter. A filter is a matrix initialized with certain weights, with the size controlled by a parameter called the kernel size. Figure 12 demonstrates how a 1-dimensional convolution works. In this case, the yellow matrix is the input, the red matrix is the filter, the green matrix is the output, and the kernel size is 3. The second element of the output is achieved by multiplying [0, 1, 0] with [1, 0, 1], which results in 0. Each time the filter is shifted to the right by the amount called the stride , which is equal to 1 in this example, and that way the output matrix is obtained.

[5] https://www.tensorflow.org/

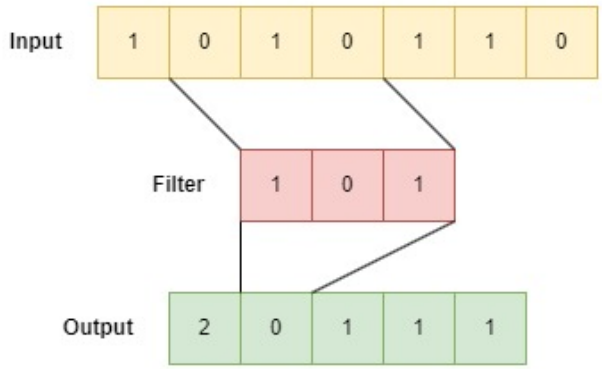

Figure 12: 1D convolution

The pooling layer is used to reduce the dimensionality of the output of a convolution layer. Since the output of a 1-dimensional convolution is simply a vector, the output of the pooling layer will be the maximum value of its input vector. In the example demonstrated in figure 12, this would be the value 2.

The CNN used for this project consists of 3 convolutional layers, each one followed by a pooling layer, as demonstrated in figure 13. The input to the embedding layer consists of 500 indices into the embedding matrix, which contains the word embeddings for each word in the dataset. The input to the convolution layers is of shape (32, 500, 200), since the batch size is 32 (as determined by the hyperparameter optimization), the sequence length was chosen to be 500, as mentioned earlier, and 200 is the dimensionality of word embeddings, as described section 5.2. The output of the convolution layers is of shape (32, 498, 64), since 64 filters are used, and for each filter, the output size of the convolution can be found using the following formula:

$$output\ dim = [(input\ dim - kernel\ size + 2 * padding)\ /\ stride] + 1 \tag{1}$$

In this case, this is equal to $[(500-3+2*0) \div 1]+1 = 498$. The output of the pooling layers is of length 64, since the pooling layer takes the maximum of each of the 64 convolution outputs. The next layer concatenates the outputs of the 3 pooling layers, resulting in an output of shape (32, 192), which is then fed to the dense layer.

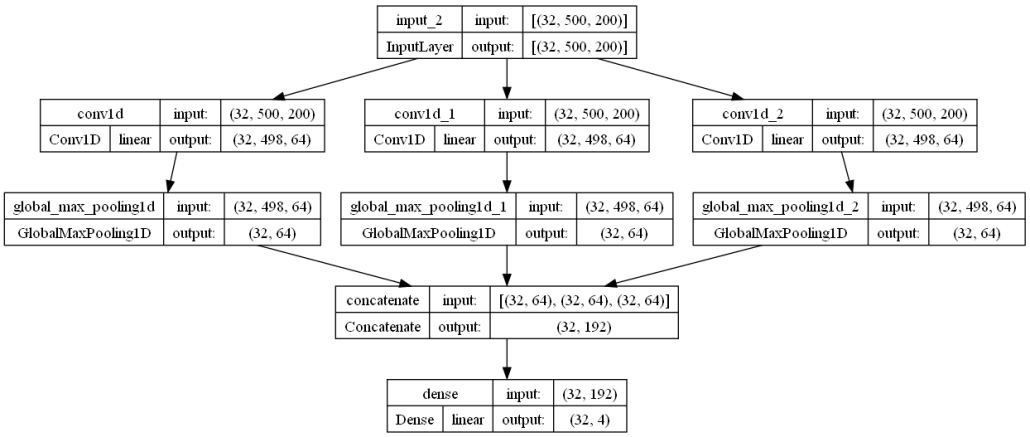

Figure 13: CNN classifier architecture

5.3.2 LSTM

LSTM stands for long short-term memory, and is a type of a recurrent neural network (RNN) that was designed to address the vanishing gradient problem of traditional RNNs, and can handle long time-series data. This makes it useful for natural language processing (NLP) tasks, since natural language is a type of sequential data where order is crucial. In a bidirectional LSTM, the input flows in both forward and backward directions, and the outputs of both directions are combined before passing to the next layer.

An example of a repeating module used in LSTM is shown in figure 14. The blue circles at the bottom represent the input (individual words represented as word embeddings for this project), while the purple circles at the top are the output, also called the hidden state. The horizontal line at the top of a repeating module represents the cell state, which is designed to enable an LSTM to remember information over long sequences. This is achieved by making careful modifications to the cell state by point-wise multiplication and addition (red circles in the diagram) with the input vectors, which are additionally passed through activation layers, in particular sigmoid and tanh (yellow boxes in the diagram). Both hidden state and cell state are then passed to the next repeating module. In a bidirectional LSTM, the LSTM in figure 14 would be the forward layer, while the backward layer would be similar, with the only difference being that the first input to the backward layer would be the last word of the input sentence. The outputs of a bidirectional LSTM are the concatenated last hidden states of the forward and backward layers.

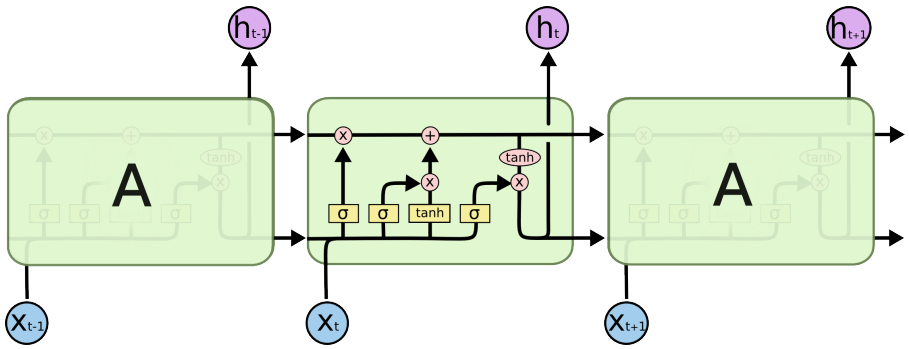

Figure 14: Repeating module of an LSTM [6]

For this project, a bidirectional LSTM was used. Figure 15 shows the layers the LSTM classifier. The input to the embedding layer again consists of 500 indices into the embedding matrix, just like the input to the CNN classifier. Since the length of the embeddings is 200, the input shape to the bidirectional LSTM layer becomes of shape (32, 500, 200). During the hyperparameter optimization, the optimal dimensionality of the output space was determined to be 512, and since the outputs of the forward and backward LSTMs are concatenated, the final output shape becomes (32, 1024). This output is then fed into a dense layer of size 128 with a ReLU (rectified linear unit) activation. The amount and the size of the dense layers between the LSTM output and the final dense layer were again determined during the hyperparameter optimization process. The output of this dense layer needs to be mapped to the number of possible labels, which is 4 for this project. This is achieved by adding one more dense layer with output shape of (32, 4).

[6] Image taken from `https://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png`

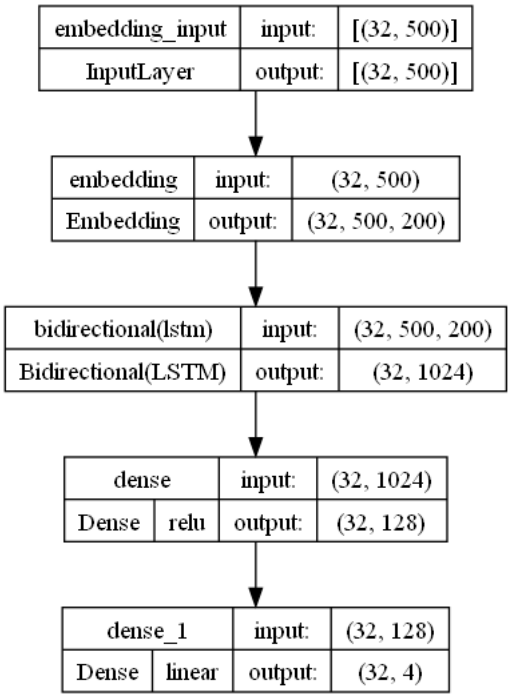

Figure 15: LSTM classifier architecture

5.3.3 BERT

BERT stands for Bidirectional Encoder Representations from Transformers, and it is a language representation model that was developed by researchers at Google in 2018, and which achieved new state-of-the-art results in several NLP tasks [15]. To create a BERT classifier, a pre-trained model provided by the *transformers*[7] library was used. More specifically, for this project, DistilBERT is used. The reason for choosing DistilBERT is that it is 40% smaller and 60% faster than BERT, while it maintains 97% of BERT's language understanding capabilities [16].

DistilBERT does not need the word embeddings, so the input to the classifier is different from the CNN and LSTM. Input consists of two parts - *input_ids* and *input_attention*, as shown in Figure 16. *Input_ids* are essentially indices into a token vocabulary created by a tokenizer. For this project, a pre-trained *DistilBertTokenizerFast* from the *transformers* library was used to obtain the *input_ids*. In addition, DistilBERT needs the attention mask to distinguish between the relevant and padding tokens. This is achieved by a binary *input_attention* tensor, in which 1 refers to a relevant token that should be used in the training process, while 0 refers to a padding token. The attention mask is also obtained from the tokenizer. Both *input_ids* and *input_attention* are of length 500, which is a maximum sequence length for this project, and the input to the DistilBERT layer is of shape (32, 500), since batch size is 32.

DistilBERT layer produces an output of shape (32, 500, 768). The dimensionality of the output being 758 is simply a design choice of the DistilBERT architecture [15]. Before passing this output to the next layer, it is necessary to select the final hidden state corresponding to a special [CLS] token that is used as the sequence representation for classification tasks [15]. Thus, the output od the DistilBERT layer is transformed into the shape of (32, 768), which is then mapped to the shape (32, 4) by adding

[7] https://huggingface.co/docs/transformers/index

a dense layer, just like in the LSTM and CNN classifiers.

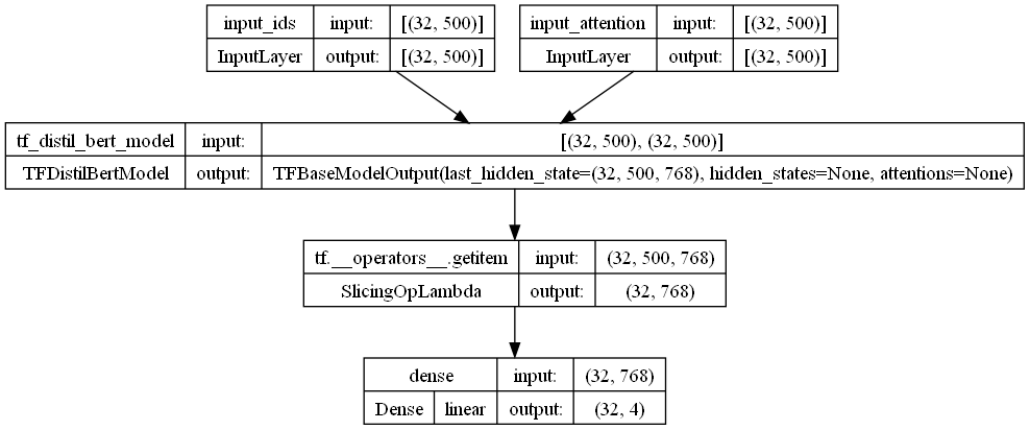

Figure 16: BERT classifier architecture

6 Experimental Setup

6.1 Tools and Technologies

For designing and training the classifiers, it was decided to use the *Tensorflow* library, since it offers a beginner-friendly high-level API called *Keras*[8] for defining and training classifiers, as well as other functionality, such as metric monitoring and useful callbacks (i.e., early stopping). *Keras* allows creating models by simply defining the necessary layers in a sequential manner.

For creating a DistilBERT classifier, Transformers library by Hugging Face was used, since it contains different pre-trained BERT models, which can be used as a starting point and can be further fine-tuned during the training process.

Classifier training and hyperparameter optimization were performed on RUG's Habrok cluster, which allows users to leverage hardware, such as GPUs, thus significantly improving the training time. To execute jobs on the cluster, the user needs to create a bash script. The bash script specifies the necessary resources, such as expected running time, memory, and the number of GPUs required. In addition to resources, the necessary modules are specified and loaded, which in case for this project included *Python* and different packages, such as *TensorFlow* with CUDA support for GPU utilization. After that, the remaining packages that are not included in the loaded modules are installed using *pip*, and the necessary *Python* script is executed.

6.2 Loss function

Since each email can have multiple labels, a classifier must output 4 independent probabilities, and the loss function must be suitable for multi-label classification. For this project, binary cross-entropy loss was used, since it treats each label independently of the others. The formula for binary cross-entropy loss (if calculated from logits) is

$$loss = z \cdot (-log(sigmoid(x))) + (1 - z) \cdot (-log(1 - sigmoid(x))) \tag{2}$$

, where z are the ground truth labels, and x are the logits. The loss function in this project is calculated from logits, which is why the last dense layers in the classifier architectures do not have sigmoid activation.

6.3 Optimizer

Choosing an optimizer is an important step in training classifiers, since it is responsible for updating model's weights. The 2 common optimizers used in deep learning are stochastic gradient descent (SGD) and adaptive moment estimation (Adam). The advantage of Adam is fast initial progress in the early stages of the training [17]. However, it generalizes worse than SGD, and it was therefore decided to use the AdamW optimizer, which improves the generalization of Adam by decoupling weight decay from the optimization steps, while maintaining the benefit of fast initial progress [17, 18].

[8] https://www.tensorflow.org/guide/keras

6.4 Class weights

Since the dataset is unbalanced, it is necessary to apply class weights while training the classifiers to account for that. Class weights are used to weigh the loss function and make the model pay more attention to the under-represented classes. For this project, the following formula was used to create the class weights:

$$weight = \frac{1}{count} * \frac{num\_labels}{2} \tag{3}$$

In the equation, *count* refers to the number of positive samples of a certain label. The less positive samples a label has associated with it, the higher is its class weight.

Since the output dimensionality of the classifiers is 4, it is impossible to specify the class weight for the non-architectural emails. To deal with the fact that such emails are over-represented in the training set, they are under-sampled by randomly selecting 250 non-architectural emails.

6.5 Hyperparameter Optimization

Before obtaining and comparing the results of the classifiers, it is necessary to tune the hyperparameters, such as the learning rate, batch size, and the number of hidden layers. *Keras* offers an implementation of a HyperBand algorithm, which was used to fine-tune the model hyperparameters for this project. The algorithm trains the model with different hyperparameter configurations supplied to it, while monitoring the provided metric, which for this project was the validation loss. It then discards a specified amount of configurations that perform poorly, and repeats this process iteratively, until the best configuration is left [19]. This best configuration is then used for training the classifiers and obtaining the results of n-fold cross-validation. To fine-tune the hyperparameters using the API that Keras provides, it is necessary to inherit from the *HyperModel* class of the *keras-tuner*[9] framework.

6.6 Performance Criteria

To evaluate the performance of a model, three main metrics were used - precision, accuracy, and f1-score. Moreover, for each metric, four different values are recorded that are obtained by using four averaging methods - micro, macro, weighted, and samples averaging. The difference between the averaging methods is the following:

- **Micro**: calculates metrics globally by counting the total true positives, false negatives and false positives.
- **Macro**: calculates metrics for each label, and finds their unweighted mean.
- **Weighted**: calculates metrics for each label, and finds their average weighted by the number of true instances for each label.
- **Samples**: calculates metrics for each sample of the testing set, and finds the mean.

[9] https://keras.io/keras_tuner/

In addition, a matrix is created for each label combination that occurs in the dataset, where rows represent ground truth labels, while columns represent predicted labels. This matrix is useful since it is possible to see the exact predicted label for each sample. Both the performance metrics and the matrices can be found in section 7. To produce the performance metrics, the *classification_report* function from *scikit-learn*[10] was used.

Since the dataset only contains roughly 3000 labeled emails, a 10-fold cross validation is performed, and results of each fold are averaged. The dataset is split in 10 unique folds, such that the sample percentage is preserved for each label. This is achieved by using a *MultilabelStratifiedKFold* utility from the *iterstrat*[11] library, which is specifically designed for splitting multi-label datasets. The utility returns 10 folds, each containing indices into a *pandas*[12] dataframe, that can then be used to create training, validation, ans testing sets.

[10] https://scikit-learn.org/stable/index.html
[11] https://pypi.org/project/iterative-stratification/
[12] https://pandas.pydata.org/

7 Results

This section displays the results obtained from performing a 10-fold cross validation for CNN, LSTM and BERT classifiers. For CNN and LSTM classifiers, the performance metrics of classifiers trained with custom generated word embeddings and pre-trained word embeddings obtained from Stack Overflow posts are provided. Section 7.1 displays matrices, where row labels represent the ground truth labels, while the column labels represent the predicted labels. These matrices were obtained by combining the predictions for every fold in the 10-fold cross-validation. By looking at the matrices, it can be clearly seen that the classifiers sometimes confuse architectural emails with non-architectural ones. Non-architectural emails are especially often classified as 'process', which could be due to the fact that process ADDs do not directly describe the decisions made with regards to a software system, but rather processes surrounding the development of the system, which is why they can be more easily confused with non-architectural emails during the manual classification.

A somewhat surprising observation is that the majority of property emails are misclassified as non-architectural, although they were the easiest to identify during the manual classification process due to keywords such as 'performance' and 'security'. This can be explained by the fact that there are less emails containing property ADDs than the rest of ADD types in the training set, and although it is assigned a higher weight, it is still not enough for a classifier to learn to predict it correctly. Moreover, a significant part of emails containing property ADDs also contain some other type of ADDs, and it is evident from the matrices that classifiers have a hard time predicting emails with multiple ADD types correctly. These kinds of emails are more often predicted as non-architectural, or only one of several ADD types is classified correctly. For instance, in figure 21, the label 'existence, property' is only correctly classified 3 times, while it is classified 22 times as non-architectural, and 17 times as only 'existence'. This is likely due to the fact that there are more emails in the training set that contain a single ADD type compared to emails that contain multiple ADD types. By looking at figure 7b, it can be seen that there are only 231 emails that contain multiple ADD types. Since $\frac{2}{10}$ of the dataset are reserved for validation and testing sets, roughly $231 \cdot \frac{8}{10} =\approx 185$ emails containing multiple ADD types are available during training, which is not a very large number.

The matrices also suggest that the best performance among ADD types is achieved by existence and process ADDs, which is indeed the case. Table 1 demonstrates the precision, recall, and f1-score metrics for each individual label, as well as averaged, as described in section 6.6. For each classifier, existence and process ADDs achieve higher precision, recall, and f1-score metrics than property and technology ADDs. However, all the ADD types are outperformed by the non-architectural emails, which achieve an f1-score of up to 0.75 for DistilBERT classifier. Such a difference in performance might be due to the fact that non-architectural emails cannot be classified with multiple labels, as opposed to emails containing ADDs, which makes it easier for classifiers to predict non-architectural emails.

A useful finding is that CNN and LSTM classifiers with custom word embeddings outperform classifiers that used pre-trained word embeddings. Although posts on Stack Overflow discuss issues from software engineering domain, just like the email dataset for this project, it seems that generating word embeddings for a specific dataset achieves better results, even though there are only 40604 emails in the dataset. It is therefore reasonable to assume that if the dataset was to be expanded in the future, the performance gap would only increase, and if it is attempted to improve the classifiers in the future, custom word embeddings must be used.

As for the best classifier, we can look at the blue cells in table 1, which indicate the highest individual and averaged f1-scores. DistilBERT outperforms other classifiers in terms of f1-scores in most cases, although by a very small margin. Since f1-score is a measure of accuracy, it can be claimed that DistilBERT is slightly more accurate than CNN and LSTM when it comes to classifying ADDs in mailing lists.

7.1 Classification matrices

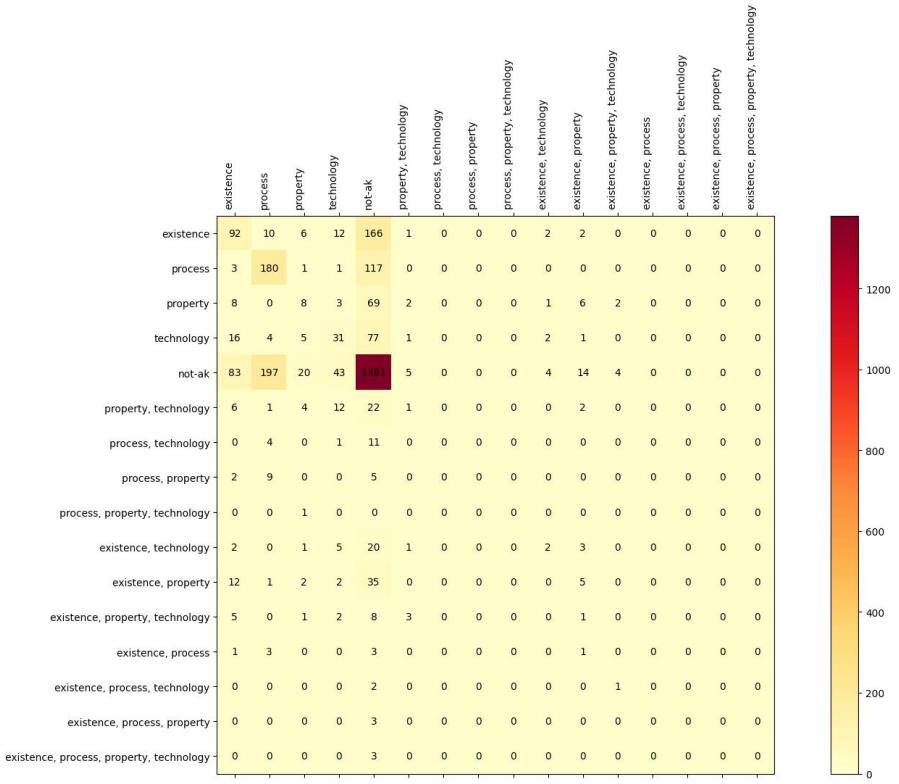

Figure 17: CNN matrix (custom word embeddings)

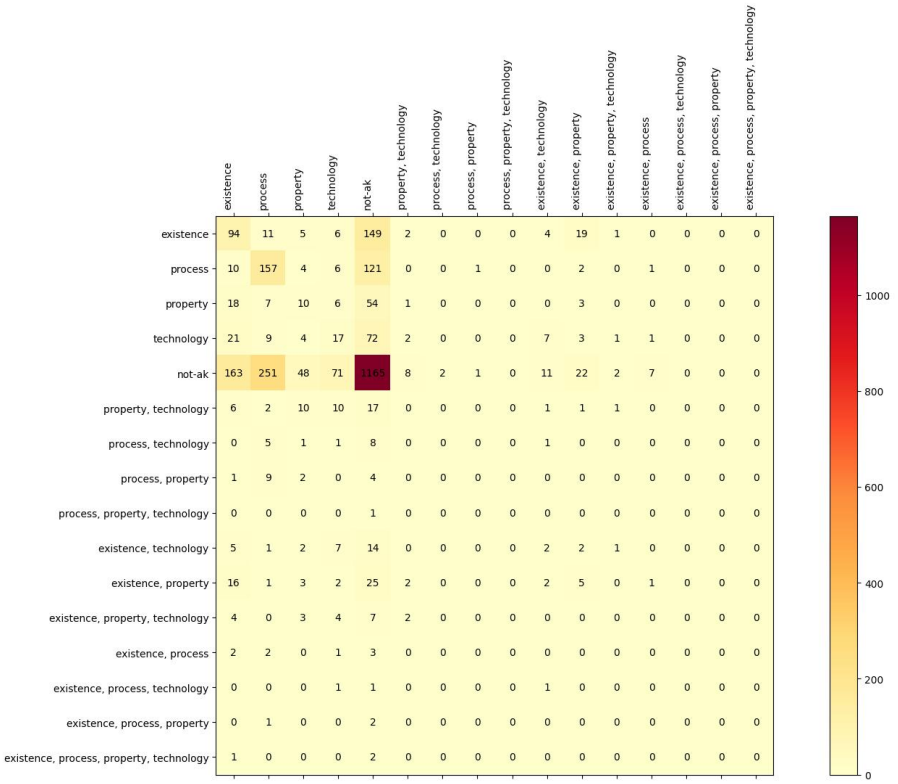

Figure 18: CNN matrix (pre-trained word embeddings)

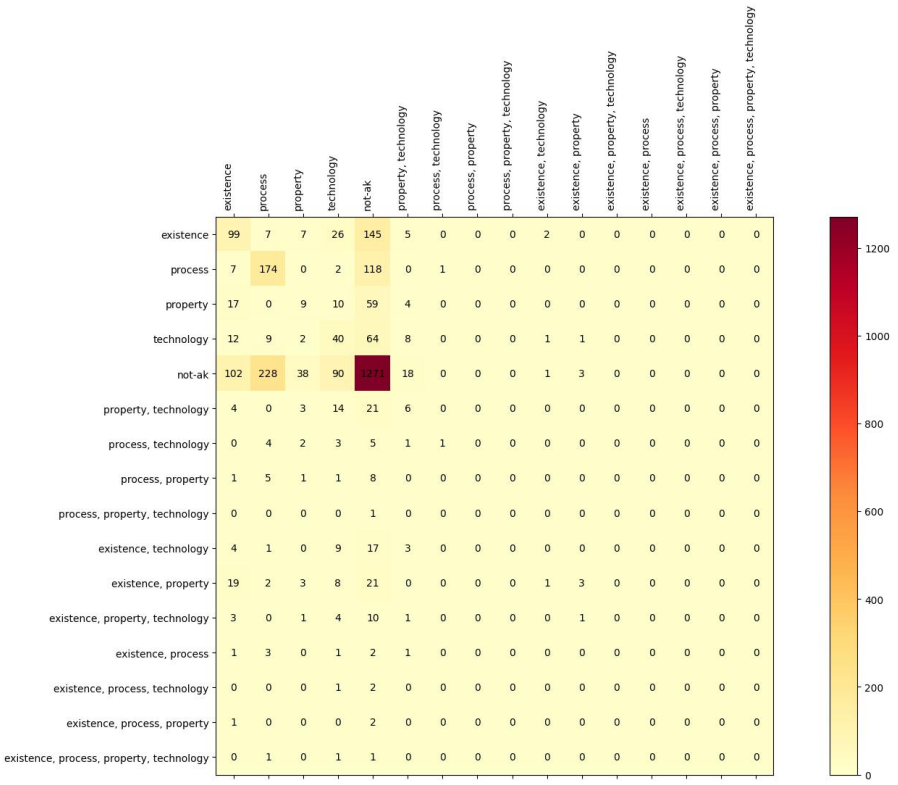

Figure 19: LSTM matrix (custom word embeddings)

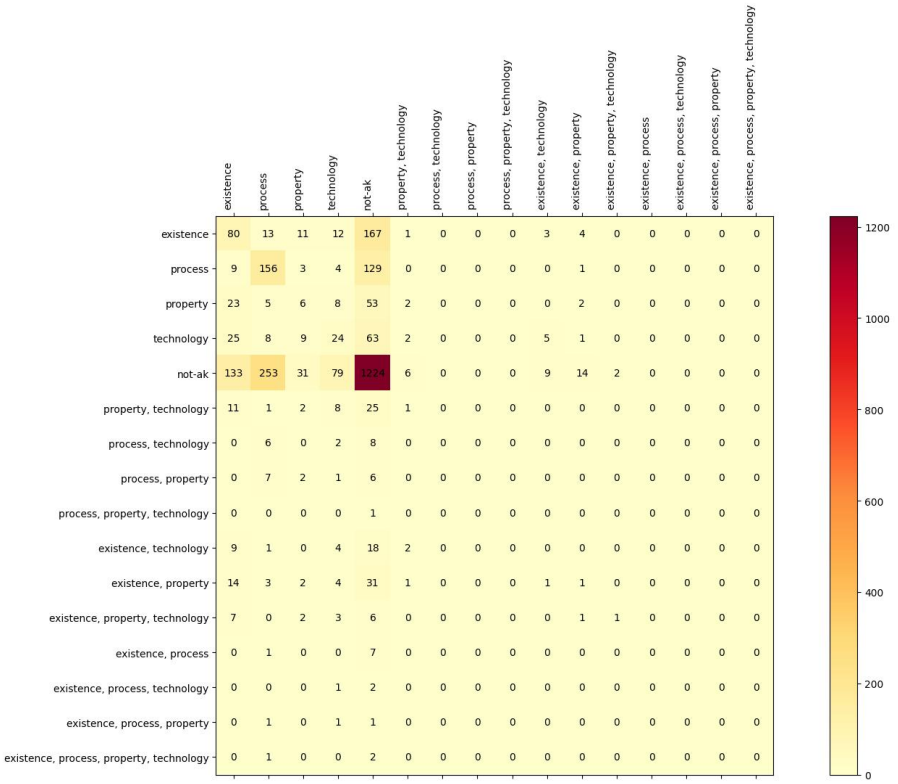

Figure 20: LSTM matrix (pre-trained word embeddings)

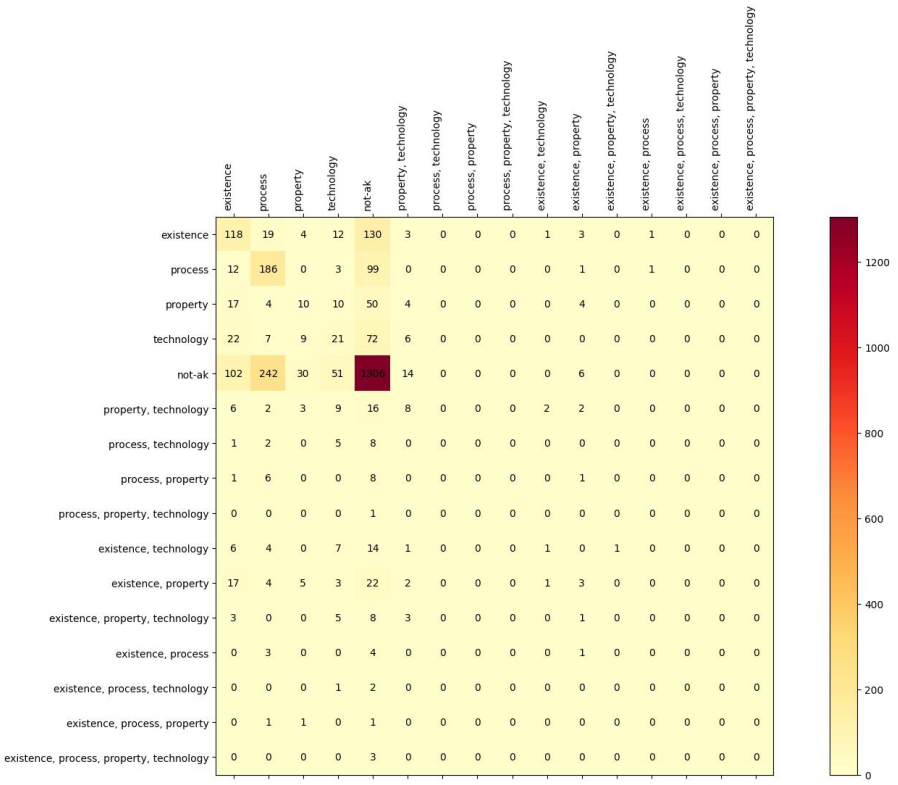

Figure 21: DistilBERT matrix.

Classifier		Precision	Recall	F1-score
CNN (custom word embeddings)	existence	0.46	0.31	0.37
	process	0.48	0.56	0.52
	property	0.36	0.15	0.22
	technology	0.43	0.24	0.31
	not-ak	0.72	0.76	0.74
	micro	0.63	0.60	0.61
	macro	0.49	0.41	0.43
	weighted	0.60	0.60	0.59
	samples	0.62	0.63	0.62
CNN (pre-trained word embeddings)	existence	0.36	0.38	0.37
	process	0.37	0.50	0.43
	property	0.25	0.17	0.20
	technology	0.32	0.23	0.26
	not-ak	0.71	0.67	0.69
	micro	0.55	0.53	0.54
	macro	0.40	0.39	0.39
	weighted	0.55	0.53	0.54
	samples	0.56	0.55	0.55
LSTM (custom word embeddings)	existence	0.47	0.32	0.38
	process	0.43	0.54	0.48
	property	0.26	0.13	0.17
	technology	0.35	0.35	0.35
	not-ak	0.73	0.73	0.73
	micro	0.60	0.57	0.58
	macro	0.45	0.41	0.42
	weighted	0.59	0.57	0.57
	samples	0.61	0.60	0.60
LSTM (pre-trained word embeddings)	existence	0.34	0.29	0.31
	process	0.38	0.49	0.43
	property	0.21	0.09	0.13
	technology	0.28	0.20	0.24
	not-ak	0.70	0.70	0.70
	micro	0.56	0.53	0.54
	macro	0.38	0.35	0.36
	weighted	0.54	0.53	0.53
	samples	0.57	0.55	0.56
DistilBERT	existence	0.47	0.37	0.42
	process	0.41	0.57	0.48
	property	0.37	0.19	0.25
	technology	0.40	0.27	0.32
	not-ak	0.75	0.75	0.75
	micro	0.62	0.59	0.60
	macro	0.48	0.43	0.44
	weighted	0.61	0.59	0.60
	samples	0.63	0.63	0.63

Table 1: Classifier performance metrics