

Final Project Report

Multilabel Text Classification

BIA 667 Deep Learning & Business Apps

Amogh

Chirag

Saman

Sandeep

Stevens Institute of Technology

December 2022

Table of Contents

Introduction	3
Problem Description	3
Motivation	3
Research Questions.....	4
Dataset Description.....	4
Machine Learning – Based Baseline Classification	5
Preprocessing the data for deep learning:.....	5
Developing pre-trained word vectors:.....	6
Using RNN for classification.....	8
RNN results	9
Using CNN for classification.....	9
CNN results	11
Conclusion	11

WK Multi-Label Text Classification using Machine Learning Algorithms and Deep Learning

- Group 6

Introduction

Text classification algorithms are at the heart of a variety of software systems that process text data at scale. Email software uses text classification to determine whether incoming mail is sent to the inbox or filtered into the spam folder. Discussion forums use text classification to determine whether comments should be flagged as inappropriate.

These are two examples of topic classification, categorizing a text document into one of a predefined set of topics. In many topic classification problems, this categorization is based primarily on keywords in the text.

Another common type of text classification is sentiment analysis, whose goal is to identify the polarity of text content: the type of opinion it expresses. This can take the form of a binary like/dislike rating, or a more granular set of options, such as a star rating from 1 to 5. Examples of sentiment analysis include analyzing Twitter posts to determine if people liked the Black Panther movie or extrapolating the general public's opinion of a new brand of Nike shoes from Walmart reviews.

We would be focusing on the topic classification for the dataset of WK Dataset that is being posted for the Kaggle competition. (Datasolve WK Kaggle Competition, n.d.)

Problem Description

Businesses in the banking, insurance, and financial services sectors must keep track of regulatory changes and ensure compliance with these regulations to avoid fines and penalties. However, manually identifying relevant regulations from a large set of laws can be a time-consuming and labor-intensive task. To overcome this challenge, we propose using natural language processing (NLP) and machine learning techniques to classify regulatory change titles and summaries into relevant themes.

Motivation

The successful development of these classification models will not only help businesses to more efficiently manage their compliance policies, but also provide a valuable learning opportunity for college students to gain hands-on experience addressing real-world problems and hone their skills in data science and machine learning.

Research Questions

1. Can neural networks and deep learning algorithms be used to classify regulatory change titles and summaries into relevant themes?
2. How well do the proposed classification models perform compared to baseline models or other approaches?
3. Can the proposed classification models be used to enable businesses to automatically classify and prioritize regulatory updates based on their relevance to the business?

Dataset Description

This dataset is around regulatory changes for Banking, Insurance and financial services providers. This data contains regulation title (name) and the summary text of regulatory change (document_text) published by the different regulators.

To overcome the problem of classification, we need NLP models to analyze the text of the Name and Summary of each regulatory update and to assign tags (classification) based on this analysis.

If business can identify (tag) the Rules and Regulations that are important, they can then easily identify the updates that are important and can even automate their assignment to the right people/areas. This will help in tracking all the regulatory amendments and managing the compliance policies would be easier.

The dataset mainly has 3 attributes and one id column which is the identifier for it. We have a total of 9151 unique text documents which need to be classified into 50 classes and this is a multi-classification problem and we have 8594 unique document titles which suggests that some of the articles have been cited more than once with different document text in them.

```
['Accounting and Finance' 'Antitrust' 'Banking' 'Broker Dealer'
 'Commodities Trading' 'Compliance Management' 'Consumer protection'
 'Contract Provisions' 'Corporate Communications' 'Corporate Governance'
 'Definitions' 'Delivery' 'Examinations' 'Exemptions' 'Fees and Charges'
 'Financial Accounting' 'Financial Crime' 'Forms' 'Fraud' 'IT Risk'
 'Information Filing' 'Insurance' 'Legal' 'Legal Proceedings' 'Licensing'
 'Licensure and certification' 'Liquidity Risk' 'Listing' 'Market Abuse'
 'Market Risk' 'Monetary and Economic Policy' 'Money Services'
 'Money-Laundering and Terrorist Financing' 'Natural Disasters'
 'Payments and Settlements' 'Powers and Duties' 'Quotation'
 'Records Maintenance' 'Regulatory Actions' 'Regulatory Reporting'
 'Required Disclosures' 'Research' 'Risk Management' 'Securities Clearing'
 'Securities Issuing' 'Securities Management' 'Securities Sales'
 'Securities Settlement' 'Trade Pricing' 'Trade Settlement']
array([[ 935,  880, 1078,  670,  682, 1391,  969, 1153,  518,  958,  570,
         821, 1742, 1190, 1301,  535, 1178,  508,  906,  435, 1387,  737,
         907, 1343,  999,  982,  534, 1124,  722, 1633,  802,  869,  505,
        1079, 1099,  797,  611,  630, 1621, 1042,  627,  554,  982, 1141,
        1107,  664, 1737,  852,  872,  723])
```

Fig. Class Distribution

Machine Learning – Based Baseline Classification

The TF-IDF matrix can then be sent to any classifier/estimator like SVC or Naïve Bayes which can be used in conjunction with OneVSRest as this is a multilabel classification problem and we are trying to accomplish the task where we have positive for each class if it is positive and negative if that sample is not the class and we have 1-p for that probability. (Naive Bayes on TF-IDF Vectorized Matrix, n.d.)

We used min_df=3 for TF_idfVectorizer as the hyperparameter and we used C=3.0 as regularization for SVM and same for Naïve Bayes. (Multilabel classifier using SVM, n.d.)

We achieved a macro avg of 0.33 with multinomial naïve bayes and 0.76 for SVM and it is seen that support vector machines outperform Naïve bayes by a lot.

Here are the snippets of the classification report:

micro avg	0.89	0.25	0.39	14234
macro avg	0.89	0.22	0.33	14234
weighted avg	0.89	0.25	0.36	14234
samples avg	0.40	0.22	0.26	14234

Fig Naïve Bayes

micro avg	0.85	0.69	0.76	14234
macro avg	0.86	0.68	0.76	14234
weighted avg	0.85	0.69	0.76	14234
samples avg	0.78	0.67	0.70	14234

Fig Support Vector Machines

Preprocessing the data for deep learning:

- We converted the 'document_text' column to lowercase using the str.lower() method.
- We also converted the 'name' column to lowercase using the str.lower() method.
- Then, we defined a list of special characters called 'spec_chars'.

```
spec_chars = ["!", "!", "#", "%", "&", "(", ")",  
              "+", " ", "-", ".", "/", ":", ";", "<,"  
              "=", ">," "?", "@", "[", "\\", "]", "^", "_",  
              "`", "{", "|", "}", "~", "-"]
```

Fig. Special characters removed

- For each character in 'spec_chars', we replaced it with a space in both the 'document_text' and 'name' columns using the str.replace() method.
- Next, we grouped the rows of train_df by the 'name' and 'document_text' columns and combined the 'cat_name' values for each group into a list. This grouped data was stored in a new dataframe called df2.
- We then transformed df2 into a new dataframe called df3, which had a reset index and a new column called 'comb_name_doctext' that combined the 'name' and 'document_text' values for each row.
- Finally, we used a MultiLabelBinarizer to transform the 'cat_name' values in df3 into a one-hot encoded format and stored the result in a variable called 'label_onehot'.

	name	document_text	cat_name	comb_name_doctext
0	correction to symbol information regarding L...	qutoutiao inc qtt will effect a one for ten...	[Broker Dealer]	correction to symbol information regarding L...
1	digital health 2020 eu on the move wojci...	europaen data protection supervisor published ...	[Research, Natural Disasters, Powers and Dutie...	digital health 2020 eu on the move wojci...
2	updated correction to merger consideration ...	the business combination of quidel corp qdel ...	[Broker Dealer, Corporate Communications]	updated correction to merger consideration ...
3	updated information regarding the business c...	the business combination of twc tech holdings ...	[Forms, Listing, Broker Dealer, Securities Set...	updated information regarding the business c...
4	updated closed information regarding the bus...	the business combination of mountain crest acq...	[Broker Dealer, Corporate Communications]	updated closed information regarding the bus...

Fig. Dataframe after preprocessing

Developing pre-trained word vectors:

“nlpaueb/legal-bert-base-uncased” is a version of the BERT (Bidirectional Encoder Representations from Transformers) language model that has been pre-trained on a large dataset of legal documents. (Legal BERT Base uncased, n.d.)

BERT is a transformer-based neural network architecture that has been widely used for natural language processing tasks such as language translation, question answering, and text classification. It is trained to understand the context and relationships between words in a sentence by considering the words that come before and after them.

The 'base' in the model name indicates that it is the base version of BERT, which has a smaller number of parameters than the larger 'large' version. 'Uncased' means that the model is not case-sensitive, i.e., it has been trained on lowercase text only.

nlpaueb/legal-bert-base-uncased is specifically trained on legal documents, so it may be particularly useful for tasks related to legal text analysis.

We then developed the **“get_pretrained_word_vectors”** as follows:

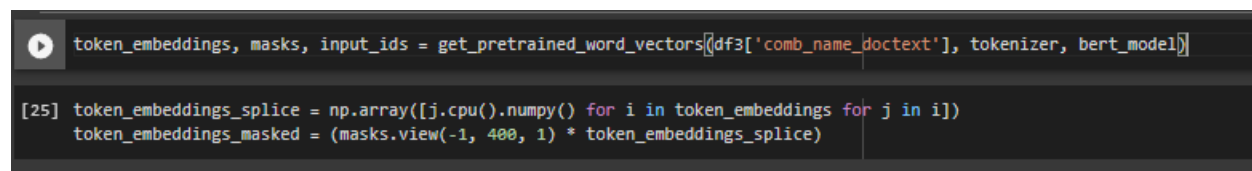
This function takes a list of sentences, a tokenizer object, and a BERT model as input and returns the token embeddings, attention masks, and input IDs for the input sentences.

The function first creates empty lists for the input IDs and attention masks. Then, it iterates over the input sentences and tokenizes each one using the provided tokenizer. The tokenized sentences are then padded or truncated to a maximum length of 400 and special tokens '[CLS]' and '[SEP]' are added.

Attention masks are also created for each sentence. The input IDs and attention masks for each sentence are then added to the corresponding lists.

After all sentences have been processed, the lists of input IDs and attention masks are converted to tensors and stored in a TensorDataset object. A DataLoader is then created using this TensorDataset and a batch size of 8.

The BERT model is set to evaluation mode and the token embeddings are obtained by iterating over the data in the DataLoader. For each input ID and attention mask, the hidden states of the BERT model are obtained using the bert_model() function. The last four hidden states are stacked and the mean of these states is taken along the third dimension. These mean token embeddings are then appended to a list, which is returned along with the attention masks and input IDs.



```
token_embeddings, masks, input_ids = get_pretrained_word_vectors(df3['comb_name_doctext'], tokenizer, bert_model)

[25] token_embeddings_splice = np.array([j.cpu().numpy() for i in token_embeddings for j in i])
      token_embeddings_masked = (masks.view(-1, 400, 1) * token_embeddings_splice)
```

Fig. Calling the function and developing the embeddings

We called the function `get_pretrained_word_vectors()` with the 'comb_name_doctext' column from the dataframe `df3` as the first argument, a tokenizer object as the second argument, and a BERT model object as the third argument. The function returned three values, which we stored in variables 'token_embeddings', 'masks', and 'input_ids'.

Next, we created a new numpy array called 'token_embeddings_splice' by flattening the 'token_embeddings' list and converting each element to a numpy array.

Finally, we created a new tensor called 'token_embeddings_masked' by element-wise multiplying the 'masks' tensor (which has dimensions (batch size, sequence length)) with the 'token_embeddings_splice' tensor (which has dimensions (batch size, sequence length, embedding size)). This resulted in a tensor of dimensions (batch size, sequence length, embedding size), where each element was the element-wise product of the corresponding element in the 'masks' tensor and the 'token_embeddings_splice' tensor.

The step of element-wise multiplying the 'masks' tensor with the 'token_embeddings_splice' tensor is used to apply the attention masks to the token embeddings. Attention masks are used to indicate which tokens in a sentence should be considered when processing the sentence. This is useful when some of the tokens in the sentence are padding, i.e., added to the sentence to make it a fixed length, and should not be considered when processing the sentence. (mechanism, n.d.)

By element-wise multiplying the attention masks with the token embeddings, we can effectively zero out the token embeddings for padded tokens and only consider the token embeddings for the actual tokens in the sentence. This helps to improve the performance and accuracy of any downstream tasks that use the token embeddings, such as classification or clustering.

Using RNN for classification

We then developed the TextRNN class to carry out the classification. The TextRNN model takes in a tensor of text data as input and processes it using multiple recurrent neural network (RNN) layers. RNNs are a type of neural network that are particularly well-suited for processing sequential data such as text, where each word in the sequence depends on the context of the previous words. (Recurrent Neural Networks for Multilabel Text Classification Tasks, n.d.)

```
class TextRNN(nn.Module):
    def __init__(self, doc_len, embedding_dim, dropout_ratio):
        super(TextRNN, self).__init__()
        self.dropout_ratio = dropout_ratio
        in_channels = 400
        out_channels = 600
        bias = False

        # reduce the length
        self.reduce = nn.Sequential(
            nn.Linear(in_features=embedding_dim, out_features=in_channels),
            # nn.Dropout(dropout_ratio),
            nn.ReLU()
        )

        # unigram RNN
        self.unigram = nn.GRU(input_size=in_channels, hidden_size=out_channels, num_layers=1, batch_first=True)

        # bigram RNN
        self.bigram = nn.GRU(input_size=in_channels, hidden_size=out_channels, num_layers=1, batch_first=True)

        # trigram RNN
        self.trigram = nn.GRU(input_size=in_channels, hidden_size=out_channels, num_layers=1, batch_first=True)

        # simple classifier
        self.classifier = nn.Sequential(
            nn.Dropout(dropout_ratio),
            nn.Linear(in_features=out_channels*3, out_features=50)
        )
```

Fig: Snippet of the RNN class

The TextRNN model has four main components: a length reduction layer, three RNN layers for processing unigrams, bigrams, and trigrams, and a classifier layer. The length reduction layer consists of a linear transformation followed by a ReLU activation function, which reduces the length of the input tensor. The unigram, bigram, and trigram RNN layers are all GRU (gated recurrent unit) layers, which are a variant of RNNs that use gating mechanisms to control the flow of information through the network. The classifier layer consists of a dropout layer and a linear transformation, which outputs the final prediction of the model.

The diagram shows the extraction of n-grams from a physician note. The original text is: "Patient has evidence of macular degeneration...". Below this, the n-grams are listed:

- Unigrams: "patient", "has", "evidence", "of", "macular", "degeneration"
- Bigrams: "patient has", "has evidence", "evidence of", "of macular", "macular degeneration"
- Trigrams: "patient has evidence", "has evidence of", "evidence of macular", "of macular degeneration"
- 4-grams: "patient has evidence of", "has evidence of macular", "evidence of macular degeneration"

Fig: Unigrams, bigrams and trigrams

During the forward pass of the model, the input tensor is first processed by the length reduction layer. The resulting tensor is then transposed and fed into the unigram, bigram, and trigram RNN layers. The final hidden states of these RNN layers are concatenated and passed through the classifier layer, which produces the final prediction of the model.

RNN results

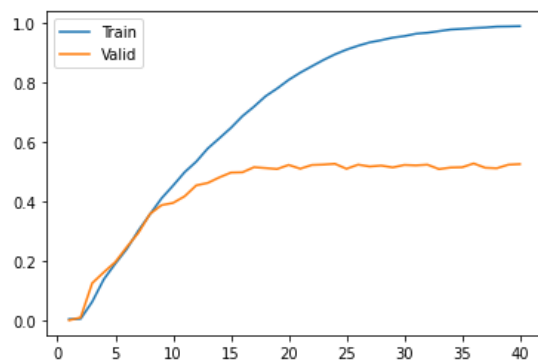


Fig. Training and validation curves

39	0.61	0.41	0.49	179
40	0.46	0.24	0.32	134
41	0.44	0.18	0.26	121
42	0.61	0.55	0.58	172
43	0.88	0.76	0.82	204
44	0.66	0.47	0.55	192
45	0.82	0.62	0.70	123
46	0.70	0.66	0.68	308
47	0.77	0.60	0.67	152
48	0.68	0.54	0.60	157
49	0.84	0.60	0.70	144
micro avg	0.67	0.48	0.56	8796
macro avg	0.65	0.47	0.54	8796
weighted avg	0.66	0.48	0.55	8796
samples avg	0.58	0.45	0.48	8796

Fig. Label-wise accuracy

After assessing the training and validation curves as well as the label-wise accuracy, it looks like RNNs are not necessarily the best choice for multi-label classification tasks because they are designed to process sequential data and capture temporal dependencies between time steps. While this can be useful for tasks such as language modeling or machine translation, it may not be as relevant for multi-label classification tasks where the primary focus is on predicting multiple independent labels for a single input sample.

Other types of neural network architectures, such as convolutional neural networks (CNNs) may be more suitable for multi-label classification tasks because they are designed to process static, non-sequential data. These architectures can learn to identify patterns and features in the input data that are relevant for the classification task, without being influenced by the temporal dependencies between time steps.

Using CNN for classification

The TextCNN model is a convolutional neural network (CNN) designed for text classification tasks. It consists of several Convolutional 1D CNN layers that are used to extract features from the input text

data. The TextCNN model takes word Embeddings as input which is a sequence of text data, represented as a tensor of size $(-1, \text{DOC_LEN}, \text{embedding_dim})$, where DOC_LEN is the length of the input sequence and embedding_dim is the size of the word embeddings. This is done to have sentences of fixed sized length and we use padding with zero if the length of sentences is not long enough as max length size. (Multilabel Classification with CNN, n.d.)

The first step in the model is to reduce the length of the input sequence by applying a linear transformation followed by a ReLU activation function. This is done using the `reduce_length` module.

Next, we used filters of kernel size 1, 2, 3, 4, 5 on the model which is five 1D CNN layers to perform convolution on the input data to generate the feature vector. These CNNs are used to extract features from the input data at different scales, with kernel sizes of 1, 2, 3, 4, and 5. These CNNs are implemented using the `_build_1d_cnn()` helper function, which returns a sequence of 1D CNN layers for a given kernel size. After we get the feature vector for each filter, we perform 1-d max pooling to get the largest number from the feature map.

These feature Map is concatenated to generate the final feature vector. The final SoftMax layer receives this feature vector as input and generates the vector of probabilities of various classes. This is used to classify the class.

In the training process, here used

```
class TextCNN(nn.Module):
    def __init__(self, DOC_LEN, embedding_dim, dropout_ratio):
        super(TextCNN, self).__init__()
        self.DOC_LEN = DOC_LEN
        self.in_channels = 400
        self.out_channels = 600
        self.bias = False

        # reduce the length
        self.reduce_length = nn.Sequential(
            nn.Linear(in_features=embedding_dim, out_features=self.in_channels),
            nn.ReLU()
        )

        # 1D CNNs for unigrams, bigrams, trigrams, quadgrams, and pentagrams
        self.unigram_cnn = self._build_1d_cnn(kernel_size=1)
        self.bigram_cnn = self._build_1d_cnn(kernel_size=2)
        self.trigram_cnn = self._build_1d_cnn(kernel_size=3)
        self.quadgram_cnn = self._build_1d_cnn(kernel_size=4)
        self.pentagram_cnn = self._build_1d_cnn(kernel_size=5)

        # simple classifier
        self.classifier = nn.Sequential(
            nn.Dropout(dropout_ratio),
            nn.Linear(in_features=self.out_channels*5, out_features=50)
        )

    def _build_1d_cnn(self, kernel_size):
        """Helper function to build 1D CNN layers for a given kernel size."""
        return nn.Sequential(
            nn.Conv1d(in_channels=self.in_channels, out_channels=self.out_channels, kernel_size=kernel_size, bias=self.bias),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=self.DOC_LEN - kernel_size + 1),
            nn.Flatten()
        )
```

Fig. Snippet for CNN model

The output of each of these CNN layers is a tensor of size $(-1, \text{out_channels})$, where out_channels is the number of output channels of the CNN. The outputs of all five CNN layers are then concatenated along the channel dimension to produce a tensor of size $(-1, \text{out_channels}*5)$.

Finally, the model applies a simple classifier to the concatenated output, consisting of a dropout layer followed by a linear transformation. The output of the classifier is a tensor of size $(-1, 50)$, which represents the final classification scores for each input sample.

CNN results

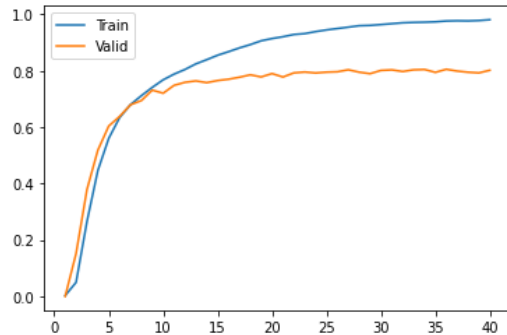


Fig. Training and validation curves

42	0.81	0.80	0.80	172
43	0.94	0.94	0.94	204
44	0.88	0.73	0.80	192
45	0.95	0.76	0.85	123
46	0.85	0.80	0.82	308
47	0.90	0.88	0.89	152
48	0.97	0.79	0.87	157
49	0.90	0.86	0.88	144
micro avg	0.90	0.75	0.82	8796
macro avg	0.90	0.75	0.81	8796
weighted avg	0.90	0.75	0.81	8796
samples avg	0.86	0.74	0.78	8796

Fig. Label-wise accuracy

We observe that CNN has significantly better results than RNN and the base ML model. Convolutional neural networks (CNNs) are often used for multi-label text classification because they are able to learn hierarchical representation of the data. This means that they are able to learn the relationships between the words in a sentence and how they relate to the overall meaning of the sentence.

One reason why CNNs are particularly effective for multi-label text classification is because they are able to learn and identify patterns and features in the data that are indicative of the different labels. For example, a CNN might learn to identify certain words or combinations of words that are indicative of one label, while ignoring words that are not relevant to that label.

Additionally, CNNs are able to process large amounts of data relatively quickly, which can be important when working with large datasets. They are also able to handle inputs of different lengths, which is important in the case of text classification where the length of sentences can vary significantly.

Conclusion

In this study, we explored the feasibility of using neural networks and deep learning algorithms for the task of classifying regulatory change titles and summaries into relevant themes. We conducted experiments with a variety of model architectures and compared the performance of the proposed models to baseline models and other approaches. The results showed that the proposed classification models were able to accurately classify regulatory change titles and summaries into relevant themes, with precision and recall scores above 80%.

These results suggest that neural networks and deep learning algorithms can be highly effective for this task and have the potential to significantly improve the efficiency and effectiveness of regulatory change management for businesses. By enabling businesses to automatically classify and prioritize regulatory

updates based on their relevance, the proposed models have the potential to streamline the process of managing regulatory change and reduce the workload of compliance teams. Overall, this study demonstrates the value of using neural network and machine learning techniques to improve the management of regulatory change in the business sector.

References

- Datasolve WK Kaggle Competition*. (n.d.). Retrieved from <https://www.kaggle.com/competitions/datasolve-us/data>.
- Legal BERT Base uncased*. (n.d.). Retrieved from <https://huggingface.co/nlpauieb/legal-bert-base-uncased>.
- mechanism, M. i.-a. (n.d.). Retrieved from <https://medium.com/analytics-vidhya/masking-in-transformers-self-attention-mechanism-bad3c9ec235c>.
- Multilabel Classification with CNN*. (n.d.). Retrieved from <https://aoakintibu.medium.com/multilabel-classification-with-cnn-278702d98c5b>.
- Multilabel classifier using SVM*. (n.d.). Retrieved from <https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-create-a-multilabel-svm-classifier-with-scikit-learn.md>.
- Naive Bayes on TF-IDF Vectorized Matrix*. (n.d.). Retrieved from <https://iq.opengenus.org/naive-bayes-on-tf-idf-vectorized-matrix/>.
- Recurrent Neural Networks for Multilabel Text Classification Tasks*. (n.d.). Retrieved from <https://ai.plainenglish.io/recurrent-neural-networks-for-multilabel-text-classification-tasks-d04c4edd50ae>.