INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

EXTRACTING KEYPHRASES AND RELATIONS FROM SCIENTIFIC PUBLICATIONS

SARTHAK BANSAL (2021101134)
SARTHAK.BANSAL@STUDENTS.IIIT.AC.IN
KRISHNA SINGH (2021112005)
KRISHNA.SINGH@RESEARCH.IIIT.AC.IN
PRADHUMAN TIWARI (2020115016)
PRADHUMAN.T@RESEARCH.IIIT.AC.IN
CODE LINK

MAY 2, 2024

Contents

1

Problem Statement

1.1 Introduction

The task comprises of extracting keyphrases and relations between them from scientific documents. The task is crucial for determining which papers describe which tasks and processes, use which materials and how those relate to one another . At its core , the task can be divided into 3 smaller subtasks:

- **Extracting Keyphrases**: This involves identifying and extracting descriptive phrases or terms from the document that encapsulate its main concepts and ideas. Keyphrases serve as important indicators of the content and focus of the document.
- **Classifying keyphrases**: This involves classifying the extracted keyphrases into TASK , MATERIAL AND PROCESS. For eg. named entity recognition is a TASK which is done using conditional random fields which is a PROCESS
- **Identifying semantic relations**: This includes identifying hyponyms (terms that are more specific versions of another term) and synonyms (terms that have similar meanings)

1.2 Dataset

The dataset for the task has the following:

- Paragraph of text drawn from a scientific paper (in plain text)
- Keyphrases of the text (start offset , end offset , id , surface form)
- Labels of the corresponding keyphrases

Some important findings about the dataset:

- Most of the phrases are **NOUN** phrases (around 93%)
- Long keyphrases (5 or more tokens) are present in the corpus in a decent amount
- 1/3rd of the keyphrases appear only on but an even bigger problem of false negative indicated by a very low recall value in both training and validation set. Same trend can be observed in the confusion matrices as well. once in train

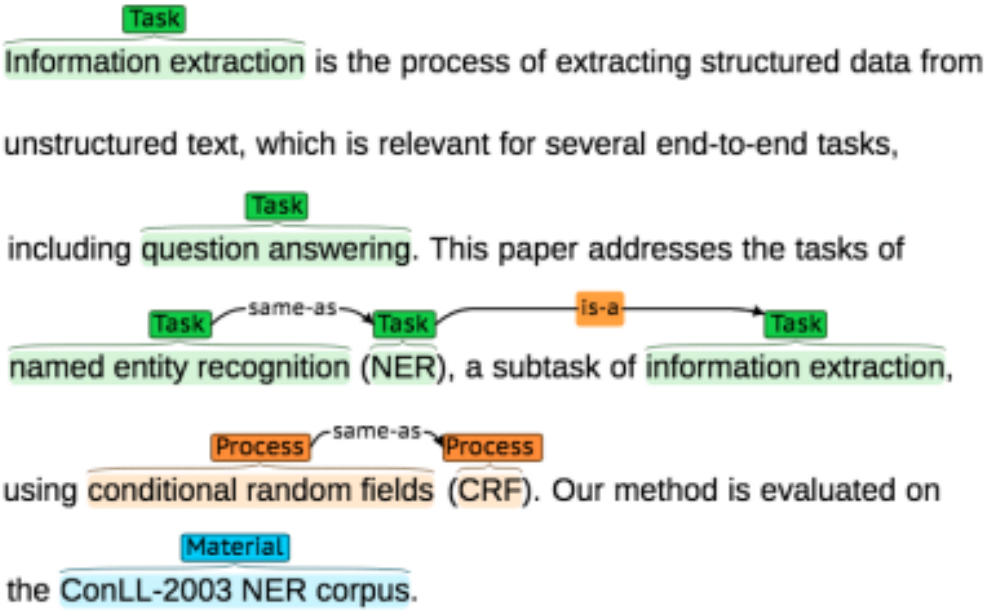


Figure 1.1: *Extracting keyphrases and classifying them*

ID	Type	Start	End
0	TASK	0	22
1	TASK	150	168
2	TASK	204	228
3	TASK	230	233
4	TASK	249	271
5	PROCESS	279	304
6	PROCESS	306	309
7	MATERIAL	343	364

ID1	ID2	Type
2	0	HYPONYM-OF
2	3	SYNONYM-OF
5	6	SYNONYM-OF

Figure 1.2: *Extraction Relations*

2

Literature Review

The paper, Anju et al., 2018 achieved state of the art results for the task of key phrase extraction from the scientific publications using the Conditional random field (CRF) approach based on 15 features including POS Tags , Named Entities , TF-IDf , Chunking etc. The sameval task of 2017, Prasad et al., 2017 which had these same objectives also used the above mentioned CRF approach for the task of Keyphrase Identification and Classification, while applied the sklearn.ensemble's random forest classifier, with syntactic similarity features for the task of keyphrase relation identification.

The paper Eger et al., 2017 uses three different models to solve the above tasks . One uses learnable embeddings and a character-level convolutional neural network (char-CNN) for classification . Other is a stacked learner which takes five base classifiers from scikit-learn and trains them repeatedly on 90% of the training data, extracting their predictions on the remaining 10% . The final model is an attention based Bidirectional Long Short-Term Memory network which uses convolutional layers to generate an attention vector which is supplied along with the pre-trained embeddings to the LSTM layer to generate softmax probabilities

The authors of the paper Garg et al., 2020, introduced a scholarly tool called SEAL, whose keyphrase extraction module consisted of two-stage neural architecture composed of Bidirectional Long Short-Term Memory cells augmented with Conditional Random Fields and the classification module consisting of a Random Forest classifier.

The paper, Saha, 2020 tested the hypothesis that the contextual information is sufficient for identifying a homonymous word. They did this by using BERT embeddings of the words to capture the context and then applying various clustering algorithms on them.

3

Keyword Extraction Module

The Keyword Extraction Module is a crucial part of the methodology. It is responsible for extracting the most relevant words or phrases from the scientific documents. This module uses two methods to extract keywords: a statistical approach using POS tagging and a deep learning approach using LSTM layers and Pre-trained embeddings

3.1 DataLoader

Tokenisation is done such that we also store the starting offset of each token . This is essential as the ground truth labels need to be a list of 1s or 0s (where 1 indicating that the token is a keyword).

```
def tokenise(self , text):
    tokens = [] # List to store tokens
    starting_offsets = [] # List to store starting offsets
    current_token = '' # Variable to store current token
    offset = 0 # Starting offset

    for char in text:
        if char == ' ':
            if current_token: # If token is not empty
                tokens.append(current_token.lower()) # Append token in lowercase
                starting_offsets.append(offset - len(current_token)) # Store
                    starting offset
                current_token = '' # Reset current token
            offset += 1 # Move offset to next character
        else:
            current_token += char # Append character to current token
            offset += 1 # Move offset to next character

    # Handling the last token if it exists after the loop ends
    if current_token:
        tokens.append(current_token.lower()) # Append token in lowercase
        starting_offsets.append(offset - len(current_token)) # Store starting
            offset
```

```
    return starting_offsets , tokens
```

Now, we iterate through the annotation file and assign the label 1 or 0 to the respective token. A collate function is also used to make the inputs in a batch of uniform length

3.2 POS Tags

Part of Speech (POS) tagging is a process of tagging a word in a text (corpus) corresponding to a particular part of speech, based on its definition and context. Using POS Tagging, we generate a cadidate set of keywords based on 6 different patterns for **Noun Phrases**. They are as follows:

- **Single Noun (NN | NNS | NNP | NNPS)**: A noun phrase can simply be a single noun. The regular expression for this is: *r'NN.?'*
- **Determiner + Noun (DT + NN | NNS | NNP | NNPS):** A determiner (DT) can precede a noun. The regular expression for this is: *r'DT NN.?'*
- **Adjective + Noun (JJ + NN | NNS | NNP | NNPS)**: An adjective (JJ) can descrive a noun. The regular expression for this is: *r'JJ NN.?'*
- **Determiner + Adjective + Noun (DT + JJ + NN | NNS | NNP | NNPS**: A determiner can precede an adjective, which describes a noun. The regular expression for this is: *r'DT JJ NN.?'*
- **Noun + Preposition + Noun (NN | NNS | NNP | NNPS + IN + NN | NNS | NNP | NNPS)**: A noun phrase can include a preposition (IN) linking two nouns. The regular expression for this is: *r'NN.?IN NN.?'*
- **Noun + Verb + Noun (NN | NNS | NNP | NNPS + VB | VBP | VBZ | VBG | VBD | VBN + NN | NNS | NNP | NNPS)**: A noun phrase can include a verb linking two nouns. The regular expressions for this are: *r'NN.?VB.?NN.?'*

```
def extract_noun_phrases(pos_tags):
    # define patterns for noun phrases
    patterns = [
        r'NN.? VB.? NN.?',
        r'NN.? IN NN.?',
        r'DT JJ NN.?',
        r'JJ NN.?',
        r'DT NN.?',
        r'NN.?'
    ]
    noun_phrases = []
    pos_string = ' '.join(tag for word, tag in pos_tags)
    for pattern in patterns:
        for match in re.finditer(pattern, pos_string):
            # storing the starting and beginning of the match
            start, end = match.span()
```

```
            start = pos_string[:start].count(' ')
            end = pos_string[:end].count(' ')
            noun_phrase = ' '.join(word for word, tag in pos_tags[start:end])
            if noun_phrase.strip():
                noun_phrases.append(noun_phrase)
    noun_phrases = [phrase for phrase in noun_phrases if phrase]
    return noun_phrases
```

This methodology is used to implement the extraction and identification of noun phrases from a given text based on POS Tagging. The pre-defined regular expression patterns are utilized to detect various noun phrase structures. By analyzing the POS Tags associated with each word in the input text, the function identifies matches to the patterns described above, extract the corresponding words, and returns a list of identified noun phrases for further processing. This process is then utilized to construct a custom dataset comprising of three fields: tokenized words, a binary indicator denoting whether the word is a keyword, and the corresponding POS tag in the third column.

```
def load_gold_keywords(ann_file):
    gold_keywords = set()
    with open(ann_file, 'r') as file:
        for line in file:
            if line.startswith('T'):
                parts = line.split()
                keyword = ' '.join(parts[4:])
                gold_keywords.add(keyword.lower())
    return gold_keywords
```

Moreover, as a means to evaluate the precision and effectiveness of the implemented method, a validation process is initiated. This validation retrieves the gold standard keywords from the annotation files, where they are in the last column, demarcated by tabs as delimiters.

```
def calculate_accuracy(extracted_keywords, gold_keywords):
    correctly_extracted = len(extracted_keywords.intersection(gold_keywords))
    total_gold_keywords = len(gold_keywords)
    accuracy = correctly_extracted / total_gold_keywords if
        total_gold_keywords > 0 else 0
    return accuracy
```

This approach involves conducting exact matches of the extracted noun phrases to calculate the accuracy of the predicted keywords. This process calculates iteratively for each individual file within the dataset. Upon culmination, the aggregate accuracy across all files is calculated, providing insight into the method's performance. This calculated average accuracy serves as a comprehensive metric to gauge the overall efficacy of the methodology when applied to the entirety of the dataset.

The flowchart of the entire methodology can be visualized as follows, where keywords

are extracted using their POS Tags.

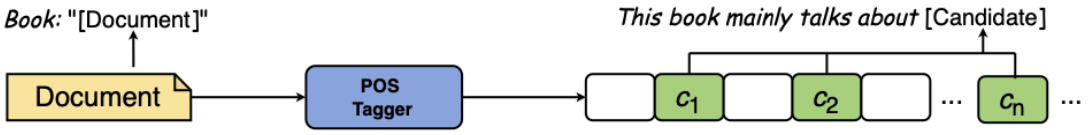


Figure 3.1: *Extracting Keywords using POS Tags*

3.2.1 Results

The results obtained on the train and the dev set are as follows. For this methodology, the train set, dev set and the test set do not make any difference as there is no training of the model being executed here, only extraction of keywords with the help of noun phrase patterns.

	Accuracy
Train	17.51
Dev	19.05

3.2.2 Observations

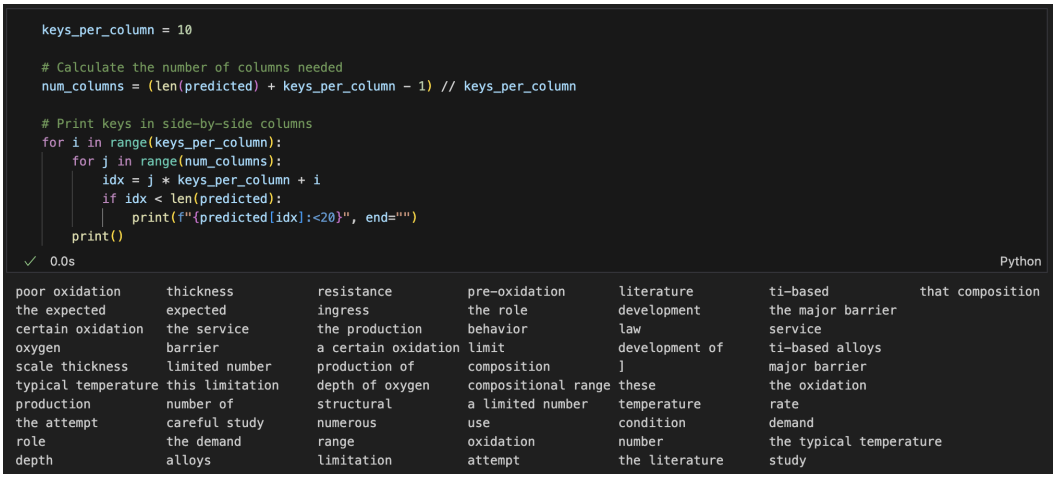
This methodology performs really well for extracting keywords of shorter length, i.e. 2 or 3. Some of the annotation files contain keywords of length longer than 5 words, which this methodology does not correctly predict. An expample of this is:

```
    for key in gold_keywords:
        print(key)
✓ 0.0s                                                    Python
development of coatings and pre-oxidation techniques
pre-oxidation techniques
oxygen
alloys
β-21s
alloys with substantially improved oxidation resistance
oxidation
understand the role that composition has on the oxidation behavior of ti-based alloys
ti-based alloys
coatings
```

Figure 3.2: *Gold Keywords for S0010938X1500195X.txt*

The precision, recall, and F1-scores do not make much sense in this scenario, therefore, are not calculated.

The accuracy, i.e. correctness of the extracted keywords for the above file is **40%**. Therefore, we can conclude that statistical method, i.e. POS Tags, performs well for extracting keywords of shorter length, typically 2 or 3 words. However, it struggles when dealing with longer keywords phrases. This limitation becomes apparent when



```
keys_per_column = 10

# Calculate the number of columns needed
num_columns = (len(predicted) + keys_per_column - 1) // keys_per_column

# Print keys in side-by-side columns
for i in range(keys_per_column):
    for j in range(num_columns):
        idx = j * keys_per_column + i
        if idx < len(predicted):
            print(f"{predicted[idx]:<20}", end="")
    print()
```

✓ 0.0s Python

```
poor oxidation      thickness          resistance         pre-oxidation      literature         ti-based           that composition
the expected        expected           ingress            the role           development        the major barrier
certain oxidation   the service        the production     behavior           law                service
oxygen              barrier            a certain oxidation limit             development of     ti-based alloys
scale thickness     limited number     production of      composition        ]                  major barrier
typical temperature this limitation    depth of oxygen    compositional range these             the oxidation
production          number of          structural         a limited number   temperature        rate
the attempt         careful study      numerous           use                condition          demand
role                the demand         range              oxidation          number             the typical temperature
depth               alloys             limitation         attempt            the literature     study
```

Figure 3.3: *Predicted Keywords for S0010938X1500195X.txt*

comparing the predicted keyword list against the gold standard annotations. Some of the reasons are as follows:

- **Length of Keyword Phrases:** The method's difficulty in correctly predicting longer keyword phrases arises from the complexity of the regular expression patterns used. These patterns are often designed to capture simpler noun phrases, making it challenging to accurately identify and extract longer sequences of words.
- **Contextual Understanding:** Another factor contributing to the discrepancy between predicted and gold standard keyword lists is the method's reliance solely on part-of-speech tagging. While this approach can be effective for identifying individual words or short phrases based on their syntactic structure, it may overlook the semantic context in which these words appear.

3.3 Deep Learning

We employ two deep learning approaches, both utilizing SciBERT embeddings, pre-trained token-level embeddings tailored to scientific text. These embeddings enhance token representation. They feed into three stacked layers of **Bidirectional Long Short-Term Memory (BiLSTM)** cells, enabling capture of long-term dependencies, bidirectional context, and scientific semantic understanding. The output is compressed via a linear layer for subsequent tasks. In the first approach, token labels are predicted with a **Conditional Random Field (CRF) Layer** layer and its **negative log-likelihood loss function**, while the second approach employs a sigmoid layer with **Binary Cross-Entropy loss** for predictions.

```
class BiLSTMCRF(nn.Module):
    def __init__(self, num_labels=1 , crf = True):
        super(BiLSTMCRF, self).__init__()
        # Load pre-trained SciBERT embeddings
        self.bert =
            BertModel.from_pretrained("allenai/scibert_scivocab_uncased")
        # freeze the embeddings
        for param in self.bert.parameters():
            param.requires_grad = False
        # BiLSTM layers
        self.bilstm = nn.LSTM(input_size=768, hidden_size=96,
            bidirectional=True, batch_first=True)
        self.bilstm2 = nn.LSTM(input_size=192, hidden_size=48,
            bidirectional=True, batch_first=True)
        self.bilstm3 = nn.LSTM(input_size=96, hidden_size=24,
            bidirectional=True, batch_first=True)
        # Linear layer for downsizing
        self.linear = nn.Linear(48, num_labels)
        if not crf:
            self.sigmoid = nn.Sigmoid()
        # CRF layer
        if crf:
            self.crf = CRF(num_labels , batch_first=True)
        self.iscrf = crf

    def forward(self, input_ids , gt_tags = None):
        # Get SciBERT embeddings
        bert_outputs = self.bert(input_ids)[0]
        # Apply BiLSTM layers
        lstm_out, _ = self.bilstm(bert_outputs)
        lstm_out, _ = self.bilstm2(lstm_out)
        lstm_out, _ = self.bilstm3(lstm_out)
        linear_out = self.linear(lstm_out)
        if not self.iscrf:
            output = self.sigmoid(linear_out)
            output = output.squeeze(2)
            return output
        # Apply CRF layer
        if self.iscrf:
            if gt_tags is not None:
                loss = -self.crf(linear_out , gt_tags)
            else:
                crf_out = self.crf.decode(linear_out)
                return crf_out
            return loss
```

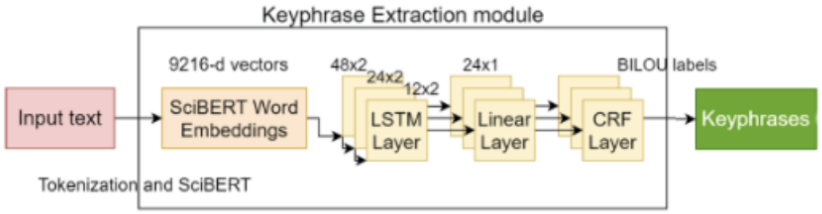


Figure 3.4: *Extracting Keywords using Deep Learning*

Results

The quantitative results for both the methods are posted below . It is observed that the CRF method works better than the BCE method in training set while the BCE method is slightly better in validation set .

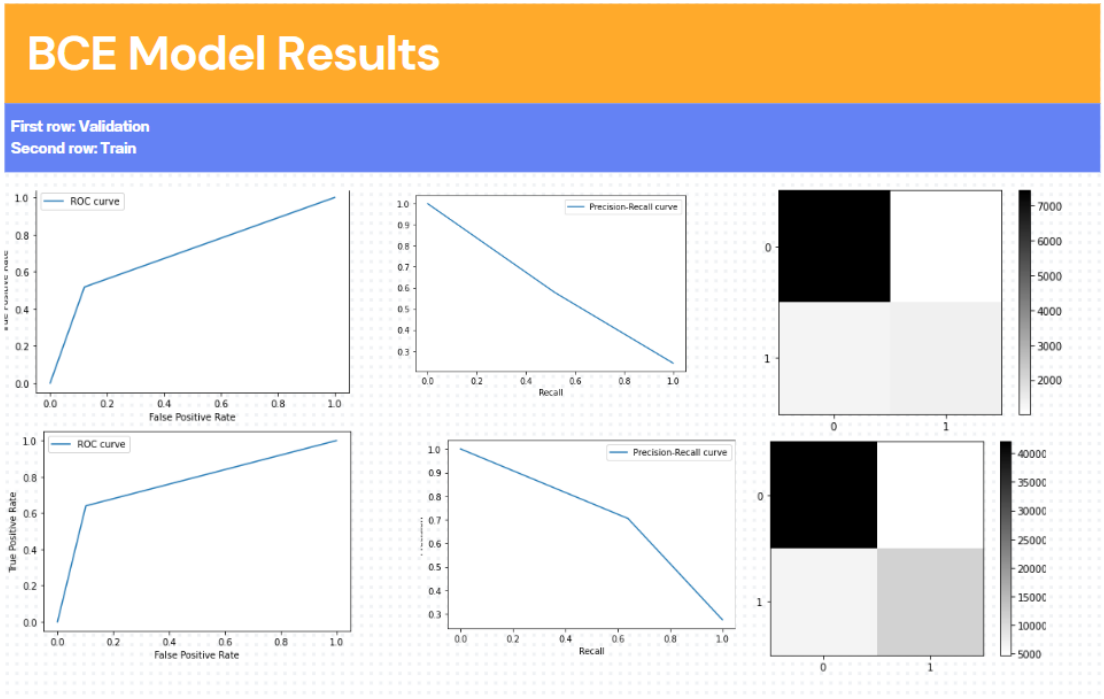


Figure 3.5: *Results of model with Binary Cross Entropy Loss*

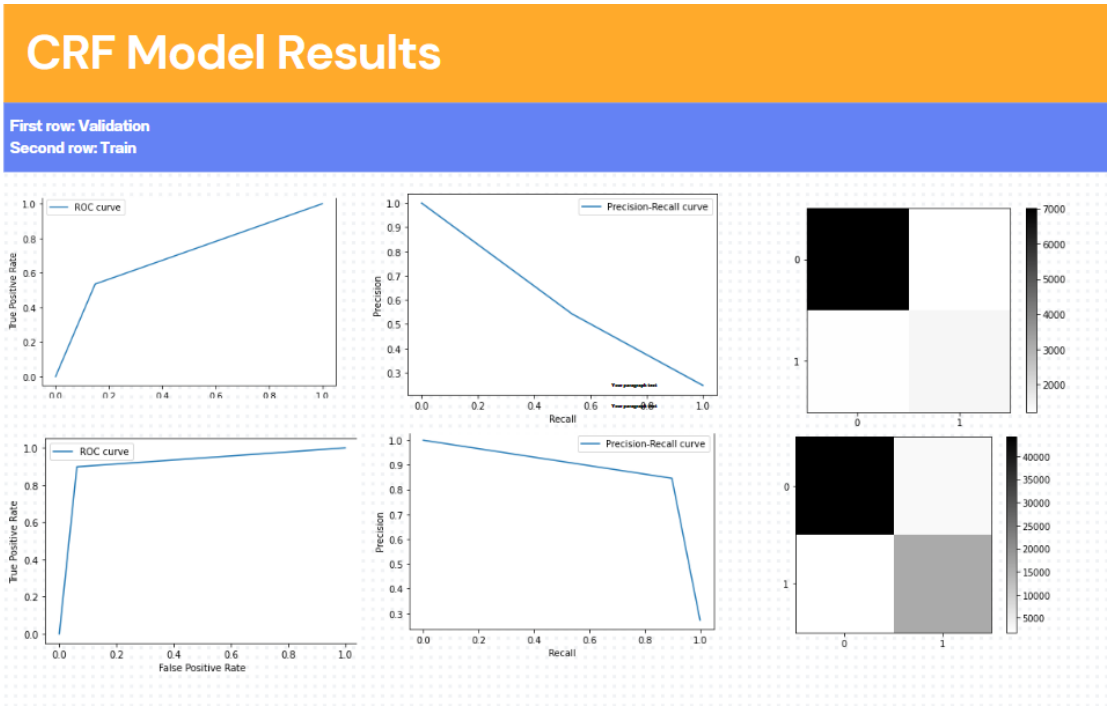


Figure 3.6: *Results of model with CRF*

	Accuracy	Precision	Recall	F1-Score
Train - CRF	92.75	84.61	89.83	87.14
Train - BCE	82.72	70.46	63.95	67.04
Val - CRF	77.35	54.20	53.52	53.86
Val - BCE	79.20	57.80	51.74	54.60

Table 3.1: *Metrics*

3.3.1 Observations

During inference, it's noted that the Conditional Random Field (CRF) tends to generate longer keyphrases, whereas the Binary Cross-Entropy (BCE) method yields shorter ones. This discrepancy likely arises from the CRF's capability to capture sequential dependencies in input data, enabling the generation of coherent token sequences for keyphrase extraction. In contrast, BCE loss, functioning at the token level, emphasizes individual tokens, potentially overlooking longer contiguous sequences unless specifically accounted for in the model or training data.

```
some_text = "Fig. 9 displays the growth of two of the main corrosion products that develop or form on the
surface of Cu40Zn with time, hydrozincite (Fig. 9a) and Cu2O (Fig. 9b). It should be remembered that both
phases were present already from start of the exposure. The data is presented in absorbance units and allows
comparisons to be made of the amounts of each species between the two Cu40Zn surfaces investigated, DP and HZ7.
The tendency is very clear that the formation rates of both hydrozincite and cuprite are quite suppressed for
Cu40Zn with preformed hydrozincite (HZ7) compared to the diamond polished surface (DP). In summary, without
being able to consider the formation of simonkolleite, it can be concluded that an increased surface coverage
of hydrozincite reduces the initial spreading ability of the NaCl-containing droplets and thereby lowers the
overall formation rate of hydrozincite and cuprite."
```

Figure 3.7: *Inference Text*

```
[('two of the main corrosion products', 30, 64),
 ('surface of Cu40Zn', 93, 110),
 ('start of the exposure. The data', 232, 263),
 ('absorbance units', 280, 296),
 ('made of the amounts of each species', 326, 361),
 ('Cu40Zn surfaces investigated, DP', 378, 410),
 ('Cu40Zn with preformed hydrozincite (HZ7)', 530, 570),
 ('diamond polished surface (DP).', 587, 617),
 ('increased surface coverage of hydrozincite reduces', 721, 771),
 ('NaCl-containing droplets', 809, 833),
 ('hydrozincite and cuprite.', 883, 908)]
```

Figure 3.8: *Inference Using CRF Method*

```
[('corrosion', 46, 55),
 ('surface of Cu40Zn', 93, 110),
 ('time, hydrozincite (Fig.', 116, 140),
 ('exposure.', 245, 254),
 ('data', 259, 263),
 ('amounts', 338, 345),
 ('each', 349, 353),
 ('Cu40Zn surfaces investigated, DP', 378, 410),
 ('both hydrozincite', 475, 492),
 ('cuprite', 497, 504),
 ('Cu40Zn', 530, 536),
 ('preformed hydrozincite', 542, 564),
 ('polished surface', 595, 611),
 ('formation', 665, 674),
 ('hydrozincite', 751, 763),
 ('NaCl-containing droplets', 809, 833),
 ('overall formation', 857, 874),
 ('hydrozincite', 883, 895)]
```

Figure 3.9: *Inference Using BCE Method*

4

Keyword Classification Module

The Keyword Classification Module is responsible for classifying the extracted keywords into one of the three categories: **Task, Process** or **Material**. This classification is crucial for understanding the role of each keyword in the context of the scientific document. The module uses two methods for classification:

4.1 DataLoader

This method uses pre-trained embeddings. For each keyphrase in the text, a context window of the tokens around the keyphrase is considered. The sequence of token IDs for the keyphrase and its context is constructed by including a specified number of tokens before and after the keyphrase. This is controlled by the **window size** parameter, which determines the number of tokens to include from both the left and right of the keyphrase. The sequence is represented with a fixed size, determined by the **maximum length** of the keyphrase. If the length of the sequence is less than the maximum length, it is padded with **zero** tokens until it reaches that length. This ensures that all sequences have the same length, which is necessary for the training of the model. Each token in the sequence is then mapped to a **d**-dimensional word embedding, where d is the dimensionality of the embeddings. The mapping is performed using a dictionary that maps each word to its corresponding embedding.

The choice for the word embeddings are:

- **Glove Embeddings:** GloVe (Global Vectors for Word Representation) embeddings are a type of word embedding that convert words into vectors in high-dimensional space. These vectors capture semantic relationships between words, such as similarity and analogy, based on their co-occurrence statistics in a corpus of text. The current implementation can use Glove Embeddings. They are loaded from a file and stored in a dictionary mapping words to their corresponding embeddings. The embeddings are used to convert the tokens in the text files into a numerical format that can be processed by the model.
- **BERT Embeddings:** We are utilizing a pre-trained BERT model (bert-base-uncased) to generate contextual embeddings of 768 dimensions for each token in the input

keyphrases. This captures rich semantic information and relationships between words. Importantly, we are freezing the BERT parameters to leverage the pre-trained knowledge while focusing on training the specific classification layers of your model. The primary rationale behind refraining from fine-tuning these embeddings is attributed to the limited size of the training dataset

Therefore, we can frame our classification as a mapping f_θ (where θ represents model parameters) from concatenated word embeddings to one of the three classes **MATERIAL, PROCESS**, and **TASK**:

$$f_\theta : \mathbb{R}^{d \times \text{max\_len\_keyphrase}} \rightarrow \{\text{Material, Process, Task}\}$$

```
class MyDataset(Dataset):
    def __init__(self, data_dir , label_dict , window_size = 2 ,
        max_len_keyphrase = 30):
        self.data_dir = data_dir
        self.txtfiles = []
        self.annfiles = []
        self.max_len_keyphrase = max_len_keyphrase

        for file in os.listdir(data_dir):
            if file.endswith(".txt"):
                self.txtfiles.append(file)
        # or use glove embeddings
        self.tokeniser = BertTokenizer.from_pretrained('bert-base-uncased')
        self.keyphrases_with_context = []
        self.labels = [] # Store labels separately
        self.label_dict = label_dict
        for i, txtfile in enumerate(self.txtfiles):
            sampleid = txtfile.split(".")[0]
            annfilename = sampleid + ".ann"
            with open(os.path.join(self.data_dir, annfilename), 'r') as file:
                ann = file.read()
            with open(os.path.join(self.data_dir, txtfile), 'r') as file:
                txt = file.read()
            offsets , tokenisedtxt = self.tokenise(txt)
            token_ids = self.tokeniser.convert_tokens_to_ids(tokenisedtxt)
            for line in ann.split('\n'):
                if line == '':
                    continue
                words = line.split()
                if words[0][0] != 'T':
                    continue
                label = self.label_dict[words[1]]
                self.labels.append(label) # Save the label
                ssofset = words[2]
```

```
            endoffset = words[3]
            start_index = 0
            for i in range(len(offsets)):
                if offsets[i] >= int(ssofset):
                    start_index = i
                    break
            end_index = 0
            for i in range(len(offsets)):
                if offsets[i] >= int(endoffset):
                    end_index = i
                    break
            mins = max(0 , start_index - window_size)
            maxs = min(len(token_ids) , end_index + window_size)
            tks = []
            for j in range(mins , maxs):
                tks.append(token_ids[j])
            if len(tks) < max_len_keyphrase:
                tks += [self.tokeniser.pad_token_id] * (max_len_keyphrase -
                    len(tks))
            self.keyphrases_with_context.append(tks)

    # other methods including tokenise (same as keyword extraction),
        load_glove_embeddigns, collate_fn, len and get_item
```

4.2 Statistical Methods

For the keyphrase classification task using Statistical methods, we employ a **stacked-learner**. A stacked learner is a machine learning model that learns to predict the output based on the predictions of other machine learning models. It is a way of combining multiple models to improve the performance of the prediction.

In this approach, we use **5 base classifiers** from scikit-learn. These include RandomForestClassification with two different parameterizations, ExtraTreesClassifier with two different parameterizations, and XGBClassifier. These are the individual models that are first trained on the data. Their predictions are used as input for the meta-classifier.

These classifiers are trained repeatedly on 90% of the training data, and their predictions on the remaining 10% are extracted. This process is iterated 10 times in a cross-validation manner, ensuring that we have a complete sample of predictions of the base classifiers on the training data.

We then use a **Multi Layer Perceptron (MLP)** as a *meta-classifier*. The role of this meta-classifier is to combine the predictions of the base classifiers into a final output prediction. The MLP is trained for 100 epochs, and the model that performs best on a 10% development set is chosen as the model to apply on unseen test data. The architecture of the Statistical Model for Keyword Classification is as follows:

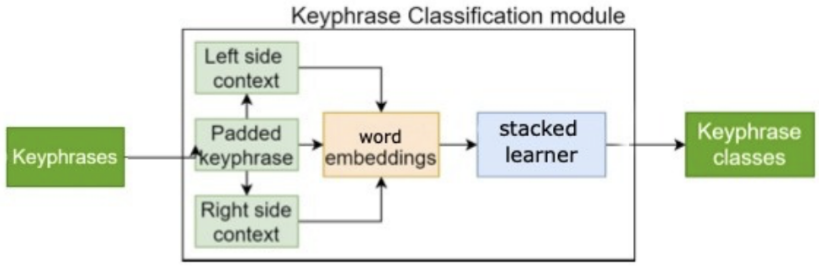


Figure 4.1: *Classification using Statistical Methods*

```
class StackedLearner:
    def __init__(self, base_classifiers, meta_classifier):
        # initialize base and meta classifiers
        self.base_classifiers = base_classifiers
        self.meta_classifier = meta_classifier

    def fit_base_classifiers(self, X, y):
        # cross-validation
        skf = StratifiedKFold(n_splits=10)
        base_predictions = []
        for clf in self.base_classifiers:
            clf_base_predictions = []
            # 10-fold cross-validation
            for train_index, val_index in skf.split(X, y):
                X_train, X_val = X[train_index], X[val_index]
                y_train, _ = y[train_index], y[val_index]
                clf.fit(X_train, y_train)
                y_val_pred = clf.predict(X_val)
                clf_base_predictions.append(y_val_pred)
                base_predictions.append(np.concatenate(clf_base_predictions))
        return np.array(base_predictions).T

    def fit_meta_classifier(self, base_predictions, y):
        self.meta_classifier.fit(base_predictions, y)

    def predict(self, X):
        base_predictions = [] # collect base predictions
        for clf in self.base_classifiers:
            base_predictions.append(clf.predict(X))
        base_predictions = np.array(base_predictions).T
        # final preds from meta classifier
        return self.meta_classifier.predict(base_predictions)

    def compute_accuracy(self, y_true, y_pred):
        return accuracy_score(y_true, y_pred)
```

4.2.1 Results

In the process of statistical training, four different types of GloVe embeddings are utilized i.e. **50-dimensional**, **100-dimensional**, **200-dimensional**, and **300-dimensional**. These embeddings capture semantic relationships between words and provide a rich, dense representation of word meanings. By experimenting with these four different versions of GloVe embeddings, we aimed to investigate the impact of the dimensionality of word embeddings on the performance of our stacked learner model. The results of these experiments contribute to our understanding of how the choice of word embeddings influences the effectiveness of the stacked learner model.

The performance of the stacked learner model is evaluated on the training, development, and test datasets. The model's predictions for each dataset are obtained using the **predict** method of the **StackedLearner** class.

The table presents the performance metrics of the Stacked Learner model trained with four different types of GloVe embeddings. The metrics include accuracy, precision, recall, and F1-score for the training, development, and test datasets.

	Accuracy	**Precision**	**Recall**	**F1-Score**
Train - 50d	0.86	0.87	0.86	0.86
Dev - 50d	0.57	0.60	0.58	0.57
Test - 50d	0.52	0.53	0.52	0.52
Train - 100d	0.86	0.87	0.86	0.86
Dev - 100d	0.57	0.59	0.58	0.57
Test - 100d	0.52	0.53	0.53	0.52
Train - 200d	0.88	0.90	0.89	0.89
Dev - 200d	0.55	0.60	0.56	0.55
Test - 200d	0.52	0.53	0.53	0.52
Train - 300d	0.85	0.87	0.86	0.86
Dev - 300d	0.55	0.59	0.56	0.55
Test - 300d	0.52	0.53	0.53	0.52

Table 4.1: *Metrics for Stacked Learner*

From the table, we can see that the model performs best on the training dataset across all types of embeddings, with the highest accuracy achieved with the 200-dimensional embeddings. This suggests that the model is able to learn effectively from the training data and the additional dimensions in the embeddings may provide more expressive representations of the words.

Alse note that the precision, recall, and F1-score are relatively consistent with the accuracy for each dataset and type of embeddings. Therefore the model's performance is balanced across different classes.

The training accuracy remains relatively stable across different dimensions of GloVe embeddings. This suggests that the model is able to learn effectively from the training data regardless of the dimensionality of the embeddings. Both the development and

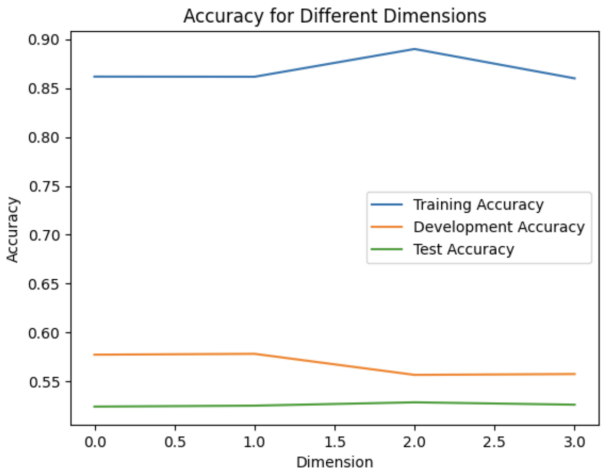


Figure 4.2: *Plot of accuracy vs Glove dimension*

test accuracies show a decline as the dimensionality of the embeddings increases.

4.3 Deep learning: Architecture

For the keyphrase classification task using deep learning we are employing a multi-layered approach, combining BERT embeddings, convolutional layers, attention mechanisms, and a Bi-LSTM network. Let's break down each step:

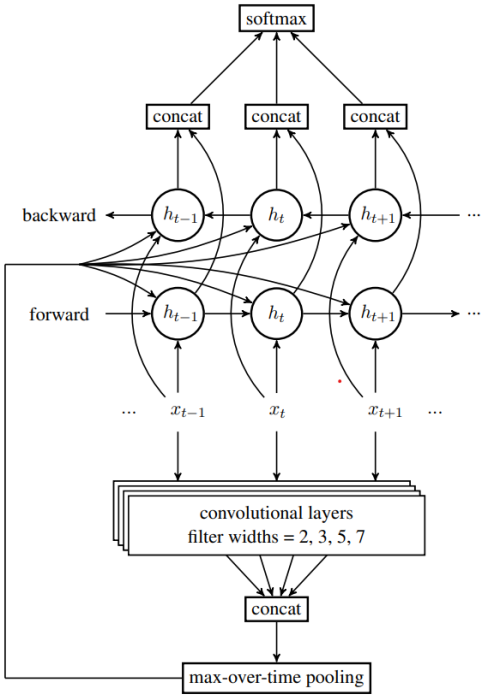


Figure 4.3: *Classifying Keywords using Deep Learning*

4.3.1 Attentional Vector Generation:

- **Convolutional Layers:** We are applying 4 parallel convolutional layers with varying kernel sizes (1, 3, 5, 7) to capture local features and patterns within the embeddings at different n-gram lengths. We are also applying ReLU at the end of each layer to introduce non-linearity and improve model expressiveness.
- **Concatenation and Pooling:** We are then concatenating the outputs of convolutional layers and passing it through a max-over-time pooling layer. This helps us find out most prominent featues across the entire keyphrase.
- **Attention weights:** Then we are passing the pooled output through a softmax layer to calculate the attention weight for each token. These weights signify the contribution each token make in the classification task.

4.3.2 Contextual encoding using Bi-LSTM:

Then to capture contextual information and long-term dependencies in a keyphrase input in both forward and backward direction we are passing its original BERT embeddings through a Bi-LSTM network.

4.3.3 Classification probability from final weighted output

- **Applying weights:** Then we are element-wise multiplying the attention weights calculated earlier with the Bi-LSTM outputs. This step effectively emphasizes the important tokens within the keyphrase representation.
- **Final representation:** Then we sum the resulting weighted vectors of each token to generate a single fixed-length vector representation of the entire keyphrase.
- **Classification layer:** This final vector is passed through a linear layer to produce probability scores for each of the keyphrases classes. Since, we are using the n-cross entropy loss function to compute the loss here, we are not applying a an explicit softmax layer at the end.

```
class KeyphraseClassificationAttentionLSTM(nn.Module):
    def __init__(self , num_labels , embedding_type = 'bert'):
        '''Model takes in B kayphrase tokens (all of length 30) and outputs
            probabilities for num_labels classes'''
        '''Input Shape : (B,30) Output Shape : (B,num_labels)'''
        super(KeyphraseClassificationAttentionLSTM , self).__init__()
        '''First generate Embeddings for the input tokens'''
        self.embedding = BertModel.from_pretrained('bert-base-uncased')
        self.embed_dim = 768
        for param in self.embedding.parameters():
            param.requires_grad = False
        '''Then Generate Attentional Vectors for the input tokens'''
        self.conv1 = nn.Conv1d(in_channels = self.embed_dim , out_channels =
            256 , kernel_size = 1 , padding = 0) #(B,768,30)->(B,256,30)
```

```
        self.conv2 = nn.Conv1d(in_channels = self.embed_dim , out_channels =
            256 , kernel_size = 3 , padding=1) #(B,768,30)->(B,256,30)
        self.conv3 = nn.Conv1d(in_channels = self.embed_dim , out_channels =
            256 , kernel_size = 5 , padding=2) #(B,768,30)->(B,256,30)
        self.conv4 = nn.Conv1d(in_channels = self.embed_dim , out_channels =
            256 , kernel_size = 7 , padding=3) #(B,768,30)->(B,256,30)
        self.max_over_time_pooling = nn.MaxPool1d(kernel_size=256*4)
            #(B,30,256*4)->(B,30)
        '''Then we use Bi-lstm to generate hidden states for the input tokens'''
        self.bilstm = nn.LSTM(input_size = self.embed_dim , hidden_size = 256 ,
            num_layers = 2 , bidirectional = True , batch_first = True)
            #(B,30,768)->(B,30,512)
        '''Output Layer'''
        self.linear = nn.Linear(512 , num_labels)

    def forward(self , input_ids , bert_tokeniser = None):
        embeddings = self.embedding(input_ids)[0] #shape:B,30,768
        #All conv outputs have shape: B,256,30
        conv1_out = self.conv1(embeddings.permute(0,2,1))
        conv2_out = self.conv2(embeddings.permute(0,2,1))
        conv3_out = self.conv3(embeddings.permute(0,2,1))
        conv4_out = self.conv4(embeddings.permute(0,2,1))
        concat_conv_outs = torch.cat([conv1_out , conv2_out , conv3_out ,
            conv4_out] , dim = 1) #shape:B,256*4,30
        concat_conv_outs = concat_conv_outs.permute(0,2,1) #shape:B,30,256*4
        max_pooled = self.max_over_time_pooling(concat_conv_outs) #shape:B,30
        max_pooled = F.softmax(max_pooled , dim = 1) #shape:B,30
        bilstm_outs , _ = self.bilstm(embeddings) #shape:B,30,512
        '''Multiply the max_pooled and bilstm_outs to weigh the importance of
            each token'''
        attentional_vector = torch.mul(max_pooled , bilstm_outs) #shape:B,30,512
        '''Now to get the final output we sum over the 30 tokens'''
        final_output = torch.sum(attentional_vector , dim = 1) #shape:B,512
        final_output = self.linear(final_output) #shape:B,num_labels
        return final_output
```

4.3.4 Deep Learning: Results and Observation

Here are the quantitative results for Deep learning approach for Keyphrase Classification .

	Accuracy	Precision	Recall	F1-Score
Train	75.68	78.09	73.08	74.67
Val	71.42	68.09	64.43	65.80

As one can see , there is not much difference between training and validation set

(a) *Training Set* **(b)** *Validation Set*

Figure 4.4: *Confusion Matrices*

depicting that the model is not over-fitted .

Comparing with the best statistical approach , we can see that the statistical method is drastically overfit and has poor performance on the dev set compared to the deep learning method .

Alse note that the precision, recall, and F1-score are relatively consistent with the accuracy for each dataset. Therefore the model's performance is balanced across different classes

5

Relationships Extraction Module

In this section, we'll discuss a new deep learning method we've developed. It's designed to identify semantic relationships, specifically focusing on spotting hyponyms and synonyms. Hyponyms are terms that are more specific versions of another term, while synonyms are words that mean the same thing. Our goal with this approach is to improve how accurately we can identify these relationships in scientific text.

5.1 Dataloader

The dataloader is constructed in such a way that for each sample text file , all the keyphrase pairs in it are appended to **keyphrase pairs** list . While iterating through the text file , all the hyponym and synoynm pairs are stored simultaneously (identified through their ids in the text file) . The method of storing keyphrases with contexts is very similar to section 4.1 with the only change being that the dataloader returns :

- **2 Keyphrases**
- **Whether they are hyponym or not (0 or 1)**
- **Whether they are synonym or not (0 or 1)**
- **Whether they belong to same class (0 or 1)**

Whether they belong to same class or not is stored so as to calculate metrics while post-processing and providing enough negative samples to the model to learn specific features corresponding to hyponyms and synonyms

```
class MyDataset(Dataset):
    def __init__(self, data_dir ,class_label_dict , window_size = 2 ,
        max_len_keyphrase = 30):
        self.data_dir = data_dir
        # iterate through the files in the data directory
        self.txtfiles = []
        self.annfiles = []
        self.max_len_keyphrase = max_len_keyphrase
        for file in os.listdir(data_dir):
            if file.endswith(".txt"):
```

```
            self.txtfiles.append(file)
        self.tokeniser = BertTokenizer.from_pretrained('bert-base-uncased')
        # self.hyponym_label_dict = hyponym_label_dict
        self.keyphrase_pairs = []
        self.ishyponym = [] # for each keyphrase pair, 1 if it is a hyponym
            pair, 0 otherwise
        self.issynonym = [] # for each keyphrase pair, 1 if it is a synonym
            pair, 0 otherwise
        self.class_label_dict = class_label_dict
        self.same_pair = []
        '''Keyphrases are stored in the ann files in the following format:'''
        for txtfile in self.txtfiles:
            sampleid = txtfile.split(".")[0]
            annfilename = sampleid + ".ann"
            with open(os.path.join(self.data_dir, annfilename), 'r') as file:
                ann = file.read()
            with open(os.path.join(self.data_dir, txtfile), 'r') as file:
                txt = file.read()
            offsets , tokenisedtxt = self.tokenise(txt)
            token_ids = self.tokeniser.convert_tokens_to_ids(tokenisedtxt)
            keyphrases = [] # contains all the keyphrases in the txtfile
            keyphrases_matching_dict = {} # contains the txtfile index(T1,T2
                etc) with index of keyphrase in keyphrases
            hyponym_pairs = [] # contains all the hyponym pairs in the txtfile
            synonym_pairs = [] # contains all the synonym pairs in the txtfile
            class_labels = [] # contains the class labels of the keyphrases
            '''First get all keyphrases'''
            for line in ann.split('\n'):
                if line == '':
                    continue
                words = line.split()
                txtindex = words[0]
                ssofset = words[2]
                endoffset = words[3]
                if txtindex[0] == 'T':
                    '''get the index of first token whose offset is greater than
                        or equal to ssofset'''
                    start_index = 0
                    for i in range(len(offsets)):
                        if offsets[i] >= int(ssofset):
                            start_index = i
                            break
                    '''get the index of first token whose offset is greater than
                        or equal to endoffset'''
                    end_index = 0
                    for i in range(len(offsets)):
                        if offsets[i] >= int(endoffset):
```

```
                end_index = i
                break
            '''get the keyphrase tokens with context tokens'''
            mins = max(0 , start_index - window_size)
            maxs = min(len(token_ids) , end_index + window_size)
            tks = []
            for j in range(mins , maxs):
                tks.append(token_ids[j])
            # pad tks with bert_padding_token to make it of length
                max_len_keyphrase
            if len(tks) < max_len_keyphrase:
                tks += [self.tokeniser.pad_token_id] * (max_len_keyphrase
                    - len(tks))
            label = self.class_label_dict[words[1]]
            keyphrases.append(tks)
            class_labels.append(label)
            keyphrases_matching_dict[txtindex] = len(keyphrases) - 1
        elif txtindex[0] == 'R':
            arg1 = words[2].split(':')[1]
            arg2 = words[3].split(':')[1]
            # there is a forward relation between arg1 and arg2
            if arg1[0] == 'T' and arg2[0] == 'T':
                keyphrase1_index = keyphrases_matching_dict[arg1]
                keyphrase2_index = keyphrases_matching_dict[arg2]
                hyponym_pairs.append((keyphrase1_index ,
                    keyphrase2_index))
        elif txtindex[0] == '*':
            '''synonym class'''
            arg1 = words[2]
            arg2 = words[3]
            if arg1[0] == 'T' and arg2[0] == 'T':
                keyphrase1_index = keyphrases_matching_dict[arg1]
                keyphrase2_index = keyphrases_matching_dict[arg2]
                synonym_pairs.append((keyphrase1_index ,
                    keyphrase2_index))
    '''Now make all keyphrase pairs and add them to
        self.keyphrase_pairs'''
    for i in range(len(keyphrases)):
        for j in range(i+1 , len(keyphrases)):
            self.keyphrase_pairs.append((keyphrases[i] , keyphrases[j]))
            if ((i , j) in hyponym_pairs) or ((j , i) in hyponym_pairs):
                self.ishyponym.append(1)
            else:
                self.ishyponym.append(0)
            if ((i , j) in synonym_pairs) or ((j , i) in synonym_pairs):
                self.issynonym.append(1)
            else:
```

```
                self.issynonym.append(0)
            if class_labels[i] == class_labels[j]:
                self.same_pair.append(1)
            else:
                self.same_pair.append(0)
# other methods including tokenise (same as keyword extraction),
    collate_fn, len and get_item
```

5.2 Model architecture

Our approach for this task closely resembles the methodology employed in the previous task. It combines the strengths of pre-trained language models with convolutional feature extraction to capture semantic relationships and local patterns within keyphrases. Below is a detailed breakdown of this approach:

5.2.1 Convolution feature extraction from concatenated sequences

- **Concatenation of Keyphrases:** Before feeding the input to the BERT model, the two keyphrase sequences (each with left context, keyphrase, and right context) are concatenated along the sequence dimension. This creates a single input sequence containing both keyphrases with their respective contexts. This allows the model to learn and compare the representations of both keyphrases simultaneously within the shared BERT embedding space.
- **Multiple Convolutions:** Four convolutional layers with varying kernel sizes (1, 3, 5, 7) are applied to the BERT embeddings. Each kernel size captures different n-gram features and local patterns within the keyphrase representations. ReLU activation is used after each convolutional layer to introduce non-linearity and improve model expressiveness.

5.2.2 Feature Aggregation

- **Concatenation:** The outputs of all four convolutional layers are concatenated along the feature dimension, combining the diverse features extracted at different n-gram levels.
- **Max-over-Time Pooling:** A max-pooling operation is applied across the time dimension (sequence length) to extract the most salient features and reduce dimensionality.

5.2.3 Dense Layers for Classification

- **Fully Connected Layers** The pooled features are passed through a series of fully connected layers with ReLU activation. This allows the model to learn complex relationships between the extracted features and perform the final classification

- **Output Layer:** The final layer outputs a 3-dimensional vector, representing the probabilities of the input keyphrases being synonyms, hypernyms, or unrelated. Now, again because we are using the n-cross entropy loss function to compute the loss here, we are not applying a an explicit softmax layer at the end.

```
class FeatureExtractor(nn.Module):
    def __init__(self , emb_size = 3):
        super(FeatureExtractor , self).__init__()
        self.embedding = BertModel.from_pretrained('bert-base-uncased')
        self.embed_dim = 768
        for param in self.embedding.parameters():
            param.requires_grad = False

        self.conv1 = nn.Sequential(nn.Conv1d(in_channels = self.embed_dim ,
            out_channels = 256 , kernel_size = 1 , padding = 0) ,
        nn.ReLU()) # (B , 768 , 60) -> (B , 256 , 60)

        self.conv2 = nn.Sequential(nn.Conv1d(in_channels = self.embed_dim ,
            out_channels = 256 , kernel_size = 3 , padding = 1) ,
        nn.ReLU())

        self.conv3 = nn.Sequential(nn.Conv1d(in_channels = self.embed_dim ,
            out_channels = 256 , kernel_size = 5 , padding = 2) ,
        nn.ReLU())

        self.conv4 = nn.Sequential(nn.Conv1d(in_channels = self.embed_dim ,
            out_channels = 256 , kernel_size = 7 , padding = 3) ,
        nn.ReLU())

        self.max_over_time_pooling = nn.MaxPool1d(kernel_size=60)
        self.fc = nn.Sequential(
            nn.Linear(256*4 , 512) , nn.ReLU() , nn.Linear(512 , 256) ,
                nn.ReLU() , nn.Linear(256 , emb_size)
        )

    def forward(self , x1 , x2):
        x = torch.cat([x1 , x2] , dim = 1) # shape : B , 60
        embeddings = self.embedding(x)[0] # shape : B , 60 , 768
        embeddings = embeddings.permute(0 , 2 , 1)

        conv1_out = self.conv1(embeddings) # shape : B , 256 , 60
        conv2_out = self.conv2(embeddings) # shape : B , 256 , 60
        conv3_out = self.conv3(embeddings) # shape : B , 256 , 60
        conv4_out = self.conv4(embeddings) # shape : B , 256 , 60

        concat_conv_outs = torch.cat([conv1_out , conv2_out , conv3_out ,
```

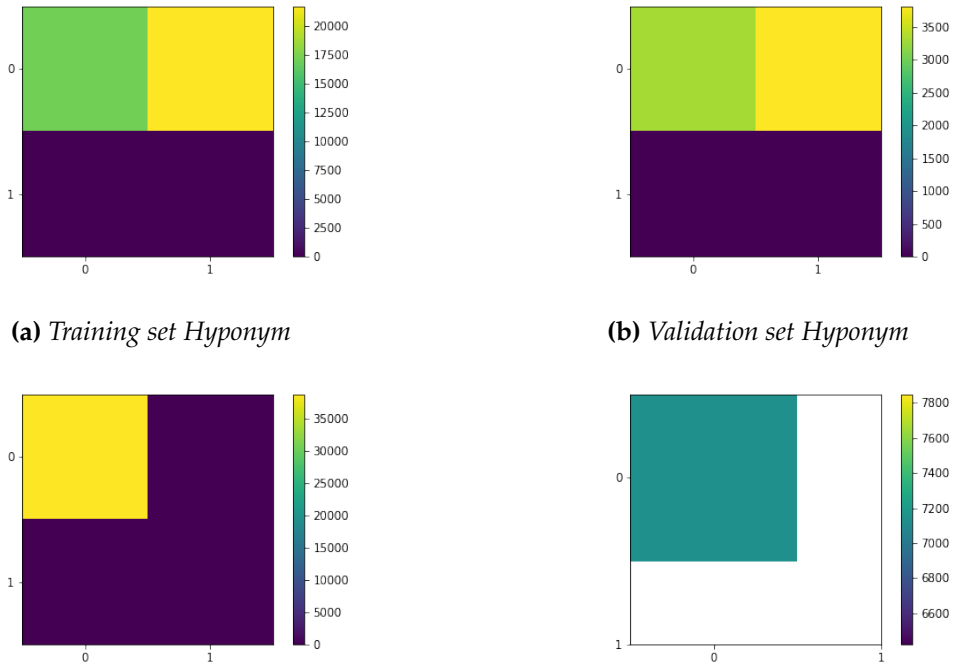
```
        conv4_out] , dim = 1) # shape : B , 256*4 , 60
    max_pooled = self.max_over_time_pooling(concat_conv_outs)
    max_pooled = max_pooled.squeeze(2) # B , 256*4
    features = self.fc(max_pooled)

    return features
```

5.3 Results and observations

Here are the quantitative results for Deep learning approach for Keyphrase Classification .

	Accuracy	Precision	Recall	F1-Score
Train - Hyponynm	44.01	50.00	22.08	30.56
Val - Hyponym	46.41	50.00	23.20	31.69
Train - Synonym	99.99	50.00	49.99	49.99
Val - Synonym	100	100	100	100



(a) *Training set Hyponym*

(b) *Validation set Hyponym*

(c) *Training set Synonym*

(d) *Validation set Synonym*

Figure 5.1: *Confusion matrices*

As one can see that for Synonym , the problem is pretty much solved.

For Hyponym , there is a problem of false positives being identified by the model indicated by low precision but an even bigger problem of false negative indicated by a

very low recall value in both training and validation set. Same trend can be observed in the confusion matrices as well.

Bibliography

Anju, R. C., Sree Harsha Ramesh, and P. C. Rafeeque (2018). "Keyphrase and Relation Extraction from Scientific Publications". In: *Advances in Machine Learning and Data Science*. Ed. by Damodar Reddy Edla, Pawan Lingras, and Venkatanareshbabu K. Singapore: Springer Singapore, pp. 113–120. ISBN: 978-981-10-8569-7.

Eger, Steffen et al. (2017). *EELECTION at SemEval-2017 Task 10: Ensemble of nEural Learners for kEyphrase ClassificaTION*. arXiv: 1704.02215 [cs.CL].

Garg, Ayush, Sammed Shantinath Kagi, and Mayank Singh (2020). "SEAL: Scientific Keyphrase Extraction and Classification". In: *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020*. JCDL '20. Virtual Event, China: Association for Computing Machinery, pp. 527–528. ISBN: 9781450375856. DOI: 10.1145/3383583.3398625. URL: https://doi.org/10.1145/3383583.3398625.

Prasad, Animesh and Min-Yen Kan (Aug. 2017). "WING-NUS at SemEval-2017 Task 10: Keyphrase Extraction and Classification as Joint Sequence Labeling". In: *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*. Ed. by Steven Bethard et al. Vancouver, Canada: Association for Computational Linguistics, pp. 973–977. DOI: 10.18653/v1/S17-2170. URL: https://aclanthology.org/S17-2170.

Saha, Rohan (2020). "Homonym Identification using BERT - Using a Clustering Approach". en. In: DOI: 10.13140/RG.2.2.29120.07681. URL: http://rgdoi.net/10.13140/RG.2.2.29120.07681.