National College of Ireland

Configuration Manual

MSc Research Project

MSc Data Analytics

Mandeep Kaur Taneja

Student ID: 21123837

School of Computing

National College of Ireland

Supervisor: Christian Horn

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Mandeep Kaur Taneja		
Student ID:	21123837		
Programme:	Msc in Data Analytics	**Year:**	2022-2023
Module:	Research Project		
Lecturer:	Christian Horn		
Submission Due Date:	15/12/2022		
Project Title:	Bounded Memory Coreference Resolution Using SpanBERT on Litbank Dataset		
Word Count:	811 **Page Count:** 14		

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Mandeep Kaur Taneja

Date: 15/12/2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Mandeep Kaur Taneja
Student ID: 21123837

1 Introduction

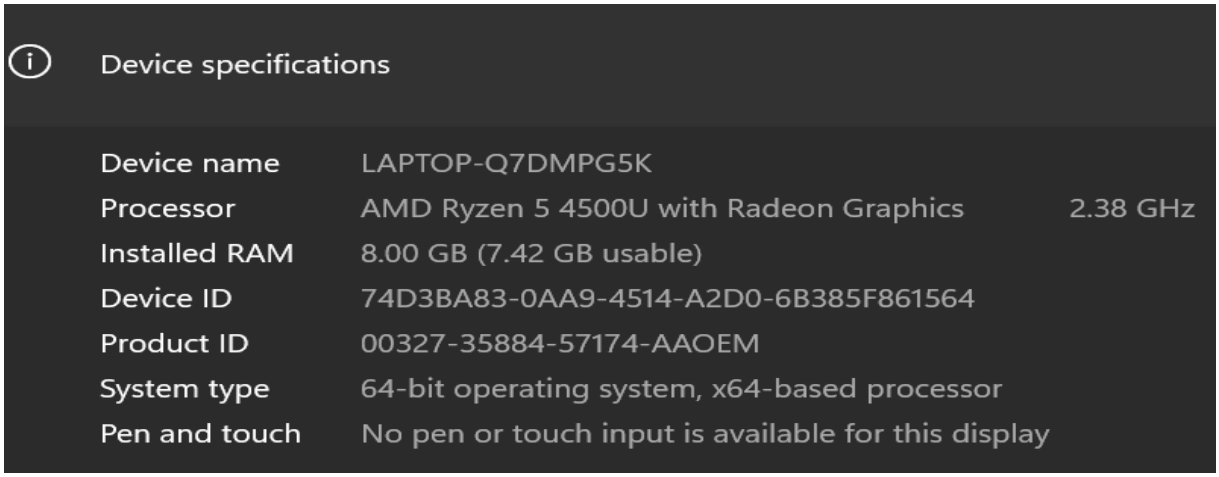
The objective of the research is to implement coreference resolution using Google Colab. For the given research we have used the pretrained SpanBERT base model from hugging Face. We have applied Learned Bounded Memory architecture that reduces the memory requirement for the model. We have also fine-tuned the hyperparameters like memory cells of the model to make it work in google Colab.

2 System Requirements

To execute this project Google Colab was used with the following configurations.

2.1 Hardware Configuration

Due to enormous training data Google Collaboratory's cloud machine is used for training the models. The configuration of the host device are –



Also Google Colab provides 12GB of RAM with the free version. In the Fig 2, the specification of Google colab is mentioned.

Parameter	Google Colab
GPU	Nvidia K80 / T4
GPU Memory	12GB / 16GB
GPU Memory Clock	0.82GHz / 1.59GHz
Performance	4.1 TFLOPS / 8.1 TFLOPS

2.2 Software Configuration

The whole project is implemented on Google Colab using Python 3.8. In order to access Google Colab, a Gmail account is required. The major libraries used in the research are listed in the table below.

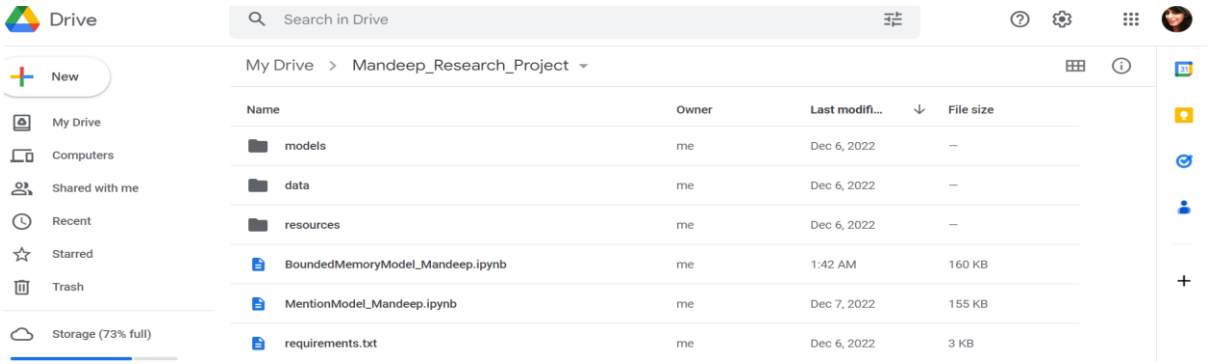
IDE	Google Collaboratory
Programming Language	Python 3.8
Framework	Pytorch
Modelling Libraries	Scipy, Transformers, HuggingFace,

3 Environment Setup

First, we will mount the entire project directory that is present in our drive to Google colab by using the following command.

```
from google.colab import drive
import os
drive.mount('/content/drive/')
os.chdir('/content/drive/My Drive/Mandeep_Research_Project/')
```

To setup the environment we will be using the requirements.txt file in the folder package to install the required libraries for the project to run successfully.



Also, we will be installing the main libraries.

```
%%capture
! pip install -q torch==1.6.0
! pip install -q transformers==4.2.2
! pip install -q scipy==1.4.1
! pip install -r requirements.txt
```

Here is a small snippet of the requirement.txt file.

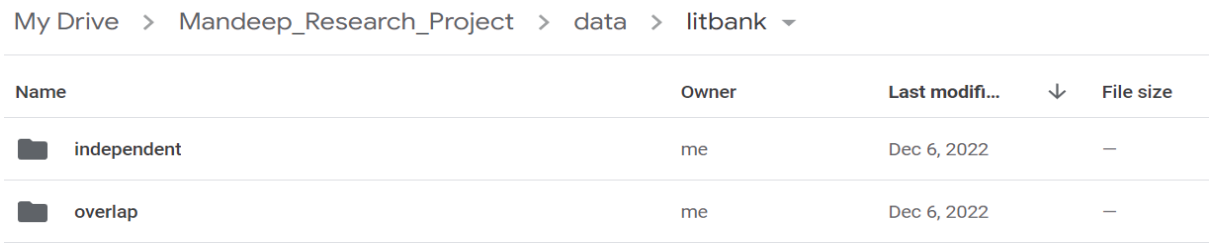
```
jupyter-core==4.6.3
Keras-Applications==1.0.8
Keras-Preprocessing==1.1.1
kiwisolver==1.1.0
lazy-object-proxy==1.4.3
Markdown==3.2.1
MarkupSafe==1.1.1
matplotlib==3.2.0
mccabe==0.6.1

mkl-fft==1.3.0
mkl-random==1.2.2
mkl-service==2.4.0
mmh3==2.5.1
murmurhash==1.0.2


nbformat==5.0.4
neovim==0.3.1
nltk==3.4.5
notebook==5.7.16
numpy==1.22.3
oauthlib==3.1.0
olefile==0.46
```

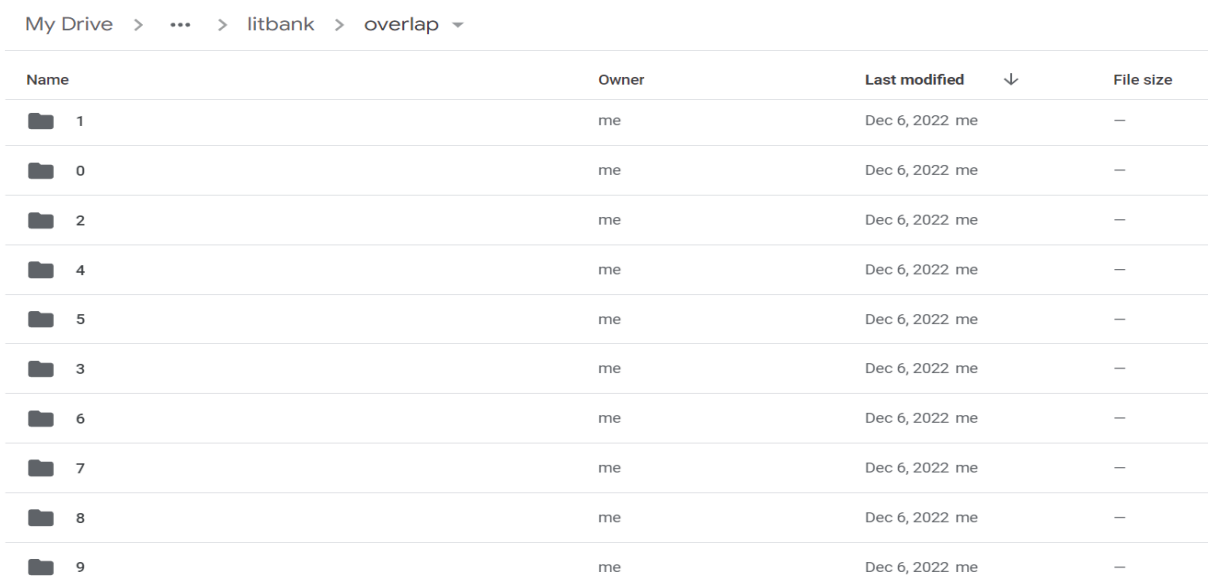
4 Implementation

4.1 Data Collection and Processing

The LitBank dataset for the research is collected from the github repository https://github.com/dbamman/litbank.

For the given research we have used only 25% of the Litbank dataset and we divided the data into 2 directories overlap and independent. For this research we have focused on overlap data and created 10-fold cross-validation points with 80/10/10 splits.





4.2 Mention Model Building

After installing all the required libraries. We first train the Mention model. For this we have used pretrained SpanBERT base model from hugging face.

```
class BaseDocEncoder(nn.Module):
    def __init__(self, model_size='base', pretrained_bert_dir=None, max_training_segments=4, device="cuda", **kwargs):
        super(BaseDocEncoder, self).__init__()
        self.device = device

        self.max_training_segments = max_training_segments

        # Check if the pre-trained model already exists
        if pretrained_bert_dir and path.exists(path.join(pretrained_bert_dir, "spanbert_{}".format(model_size))):
            self.bert = BertModel.from_pretrained(
                path.join(pretrained_bert_dir, "spanbert_{}".format(model_size)), output_hidden_states=False)
        else:
            # Use the model from huggingface
            self.bert = AutoModel.from_pretrained(f"shtoshni/spanbert_coreference_{model_size}")

        self.tokenizer = BertTokenizer.from_pretrained('bert-base-cased', output_hidden_states=False)
        self.pad_token = 0

        for param in self.bert.parameters():
            # Don't update BERT params
            param.requires_grad = False

        bert_hidden_size = self.bert.config.hidden_size
        self.h_size = bert_hidden_size

    def forward(self, example):
```

Then we have used the pretrained document encoder which encode each segment of data independently.

```
class OverlapDocEncoder(BaseDocEncoder):
    def __init__(self, **kwargs):
        super(OverlapDocEncoder, self).__init__(**kwargs)

    def encode_doc(self, example):

        if self.training and self.max_training_segments is not None:
            example = self.truncate_document(example)
        sentences = example["real_sentences"]
        start_indices = example["start_indices"]
        end_indices = example["end_indices"]

        sentences = [(['[CLS]'] + sent + ['[SEP]']) for sent in sentences]
        sent_len_list = [len(sent) for sent in sentences]
        max_sent_len = max(sent_len_list)
        padded_sent = [self.tokenizer.convert_tokens_to_ids(sent)
                       + [self.pad_token] * (max_sent_len - len(sent))
                       for sent in sentences]
        doc_tens = torch.tensor(padded_sent).to(self.device)

        num_chunks = len(sent_len_list)
        attn_mask = get_seq_mask(torch.tensor(sent_len_list).to(self.device)).to(self.device).float()

        with torch.no_grad():
            outputs = self.bert(doc_tens, attention_mask=attn_mask)  # C x L x E

        encoded_repr = outputs[0]

        encoded_out_unpadded = []
```

Then we define the MLP module

```
[ ]  import torch.nn as nn
     class MLP(nn.Module):
         def __init__(self, input_size, hidden_size, output_size,
                      num_hidden_layers=1, bias=False, drop_module=None):
             super(MLP, self).__init__()
             self.layer_list = []

             self.activation = nn.ReLU()
             self.drop_module = drop_module
             self.num_hidden_layers = num_hidden_layers

             cur_output_size = input_size
             for i in range(num_hidden_layers):
                 self.layer_list.append(nn.Linear(cur_output_size, hidden_size, bias=bias))
                 self.layer_list.append(self.activation)
                 if self.drop_module is not None:
                     self.layer_list.append(self.drop_module)
                 cur_output_size = hidden_size

             self.layer_list.append(nn.Linear(cur_output_size, output_size, bias=bias))
             self.fc_layers = nn.Sequential(*self.layer_list)

         def forward(self, mlp_input):
             return self.fc_layers(mlp_input)
```

Load the Litbank data for training , validation and testing.

```
[ ]  import json
     def data_load(data_dir, seg_max_length, dataset='litbank'):
         all_splits = []
         for split in ["train", "dev", "test"]:
             jsonl_file = path.join(data_dir, "{}.{}.jsonlines".format(split, seg_max_length))
             with open(jsonl_file) as f:
                 split_data = []
                 for line in f:
                     split_data.append(json.loads(line.strip()))
             all_splits.append(split_data)

         data_training, data_devp, data_testing = all_splits

         if dataset == 'litbank':
             assert(len(data_training) == 80)
             assert(len(data_devp) == 10)
             assert(len(data_testing) == 10)
         '''
         elif dataset == 'ontonotes':
             assert (len(data_training) == 2802)
             assert (len(data_devp) == 343)
             assert (len(data_testing) == 348)
             '''

         return data_training, data_devp, data_testing
```

Base Controller model to extract Gold Mentions in the data.

```
import torch
import torch.nn as nn

class Controller(BaseController):
    def __init__(self, size_mlp=1024, depth_mlp=1, max_span_width=30, top_span_ratio=0.4,
                 **kwargs):
        super(Controller, self).__init__(**kwargs)
        self.max_span_width = max_span_width
        self.size_mlp = size_mlp
        self.depth_mlp = depth_mlp
        self.top_span_ratio = top_span_ratio

        self.embeddings_span_width = nn.Embedding(self.max_span_width, 20)
        self.span_width_prior_embeddings = nn.Embedding(self.max_span_width, 20)
        self.mention_mlp = MLP(input_size=self.ment_emb_to_size_factor[self.ment_emb] * self.h_size + 20,
                               hidden_size=self.size_mlp,
                               output_size=1, num_hidden_layers=self.depth_mlp, bias=True,
                               drop_module=self.drop_module)
        self.span_width_mlp = MLP(input_size=20, hidden_size=self.size_mlp,
                                  output_size=1, num_hidden_layers=1, bias=True,
                                  drop_module=self.drop_module)
        self.mention_loss_fn = nn.BCEWithLogitsLoss(reduction='sum')

    def get_men_width_scores(self, cand_starts, cand_ends):
        idx_span_width = cand_ends - cand_starts
        embs_span_width = self.span_width_prior_embeddings(idx_span_width)
        width_scores = torch.squeeze(self.span_width_mlp(embs_span_width), dim=-1)

        return width_scores
```

Testing the model for F1 score and getting the best model after 25 epochs.

```
            # Get model paths
            self.model_dir = model_dir
            self.data_dir = data_dir
            self.model_path = path.join(model_dir, 'model.pth')
            self.best_model_path = path.join(best_model_dir, 'model.pth')

            #metadata
            self.model = Controller(**kwargs)
            self.model = self.model.cuda()

            self.initialize_setup(init_lr=init_lr)
            self.model = self.model.cuda()
            modinfo_print(self.model)

            if not eval:
                if self.pretrained_model is not None:
                    model_state_dict = torch.load(self.pretrained_model)
                    print(model_state_dict.keys())
                    self.model.load_state_dict(model_state_dict, strict=False)
                    self.eval_model(split='valid')
                    # self.eval_model(split='test')
                    sys.exit()
                else:
                    self.train(max_epochs=max_epochs,
                               max_gradient_norm=max_gradient_norm)
            # Model evaluation
            self.final_eval(model_dir)

        def initialize_setup(self, init_lr, lr_decay=10):
```

Update the best model if for new epoch F1 score increases.

```
    #update model if performance increases
    if fscore > self.train_info['val_perf']:
        self.train_info['val_perf'] = fscore
        self.train_info['threshold'] = threshold
        logging.info('Saving best model')
        self.save_model(self.best_model_path)

    # Save model
    self.save_model(self.model_path)


    elapsed_time = time.time() - start_time
    print("Epoch: %d, Time: %.2f, F-score: %.3f"
                  % (epoch + 1, elapsed_time, fscore))
```

The main module where hyperparameters are passed and modelling is initiated.

```
logging.basicConfig(format='%(asctime)s - %(message)s', level=logging.INFO)

def main():

    parser = argparse.ArgumentParser()

    # Add arguments to parser
    parser.add_argument(
        '-base_data_dir', default='data',
        help='Root directory of data', type=str)
    parser.add_argument(
        '-dataset', default='litbank', choices=['litbank'] , type=str)
    parser.add_argument('-base_model_dir',
                        default='models',
                        help='Root folder storing model runs', type=str)
    parser.add_argument('-model_size', default='large', type=str,
                        help='BERT model type')
    parser.add_argument('-doc_enc', default='overlap', type=str,
                        choices=['independent', 'overlap'], help='BERT model type')
    parser.add_argument('-pretrained_bert_dir', default='resources', type=str,
                        help='SpanBERT model location')
    parser.add_argument('-seg_max_length', default=512, type=int,
                        help='Max segment length of BERT segments.')
    parser.add_argument('-top_span_ratio', default=0.3, type=float,
                        help='Ratio of top spans proposed as ments.')

    parser.add_argument('-ment_emb', default='endpoint', choices=['attn', 'max', 'endpoint'],
                        type=str)
```

4.3 Bounded Memory Model Building

For the Bounded Memory model, we have used the Learned Bounded Memory architecture that use heuristic approach to ignore an already tracked entity in order to keep an untracked entity in the memory.

```
class LearnedFixedMemory(BaseFixedMemory):
    def __init__(self, **kwargs):
        super(LearnedFixedMemory, self).__init__(**kwargs)

    def get_overwrite_ign_mask(self, ent_counter):
        free_cell_mask = (ent_counter == 0.0).float().to(device=self.device)
        if torch.max(free_cell_mask) > 0:
            free_cell_mask = free_cell_mask * torch.arange(self.num_cells + 1, 1, -1).float().to(device=self.device)
            free_cell_idx = torch.max(free_cell_mask, 0)[1]
            last_unused_cell = free_cell_idx.item()
            mask = torch.zeros(self.num_cells + 2).to(device=self.device)
            mask[last_unused_cell] = 1.0
            mask[-1] = 1.0  # Not a mention
            return mask
        else:
            return torch.ones(self.num_cells + 2).to(device=self.device)

    def predict_new_or_ignore(self, vect_query, ment_score, vect_mem,
                              ent_counter, feature_embs, ment_feature_embs):
        # Fertility Score
        mem_fert_input = torch.cat([vect_mem, feature_embs], dim=-1)
        ment_fert_input = torch.unsqueeze(torch.cat([vect_query, ment_feature_embs], dim=-1), dim=0)
        fert_input = torch.cat([mem_fert_input, ment_fert_input], dim=0)

        # del mem_fert_input
```

Pass all the training parameters

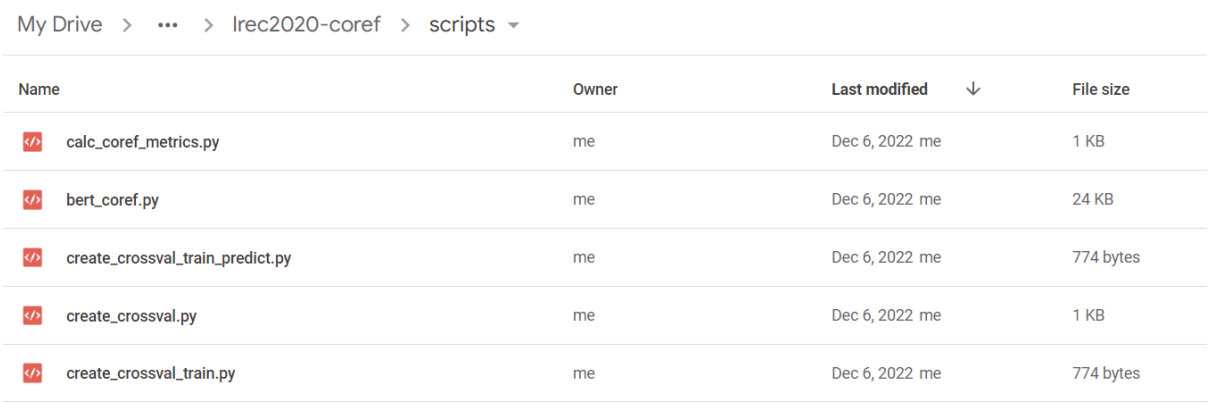
```
# Training params
parser.add_argument('-cross_val_split', default=0, type=int,
                    help='Cross validation split to be used.')
parser.add_argument('-new_ent_wt', help='Weight of new entity term in coref loss',
                    default=1.0, type=float)
parser.add_argument('-num_train_docs', default=None, type=int,
                    help='Number of training docs.')
parser.add_argument('-max_training_segments', default=None, type=int,
                    help='Maximum number of BERT segments in a document.')
parser.add_argument('-sample_invalid', help='Sample prob. of invalid mentions during training',
                    default=0.2, type=float)
parser.add_argument('-dropout_rate', default=0.3, type=float,
                    help='Dropout rate')
parser.add_argument('-label_smoothing_wt', default=0.0, type=float,
                    help='Label Smoothing')
parser.add_argument('-max_epochs',
                    help='Maximum number of epochs', default=30, type=int)
parser.add_argument('-seed', default=0,
                    help='Random seed to get different runs', type=int)
parser.add_argument('-init_lr', help="Initial learning rate",
                    default=2e-4, type=float)
parser.add_argument('-no_singletons', help="No singletons.",
                    default=False, action="store_true")
parser.add_argument('-eval', help="Evaluate model",
                    default=False, action="store_true")
parser.add_argument('-slurm_id', help="Slurm ID",
                    default=None, type=str)
parser.add_argument('-f')
```

The coreference model use the best mention model from the previous file which is already stored in the same directory in the file model.pth.

```
    # Get mention model name
    args.pretrained_mention_model = path.join(
        path.join(args.base_model_dir, menmod_name_get(args)), "best_models/model.pth")
    print(args.pretrained_mention_model)

    # Log directory for Tensorflow Summary
    log_dir = path.join(model_dir, "logs")
    if not path.exists(log_dir):
        os.makedirs(log_dir)

    config_file = path.join(model_dir, 'config')
    with open(config_file, 'w') as f:
        for key, val in opt_dict.items():
            logging.info('%s: %s' % (key, val))
            f.write('%s: %s\n' % (key, val))

    Experiment(args, **vars(args))


if __name__ == "__main__":
    main()
```

5 Evaluation

To evaluate our model, we have used Kenton lee's coreference metrics (calc_coref_metrics.py) implementation which is present in the following location.



After the model is trained, we have used the CoNLL Perl script (scorer.pl) for final evaluation. it is used when both ground truth data and official conll script is available.