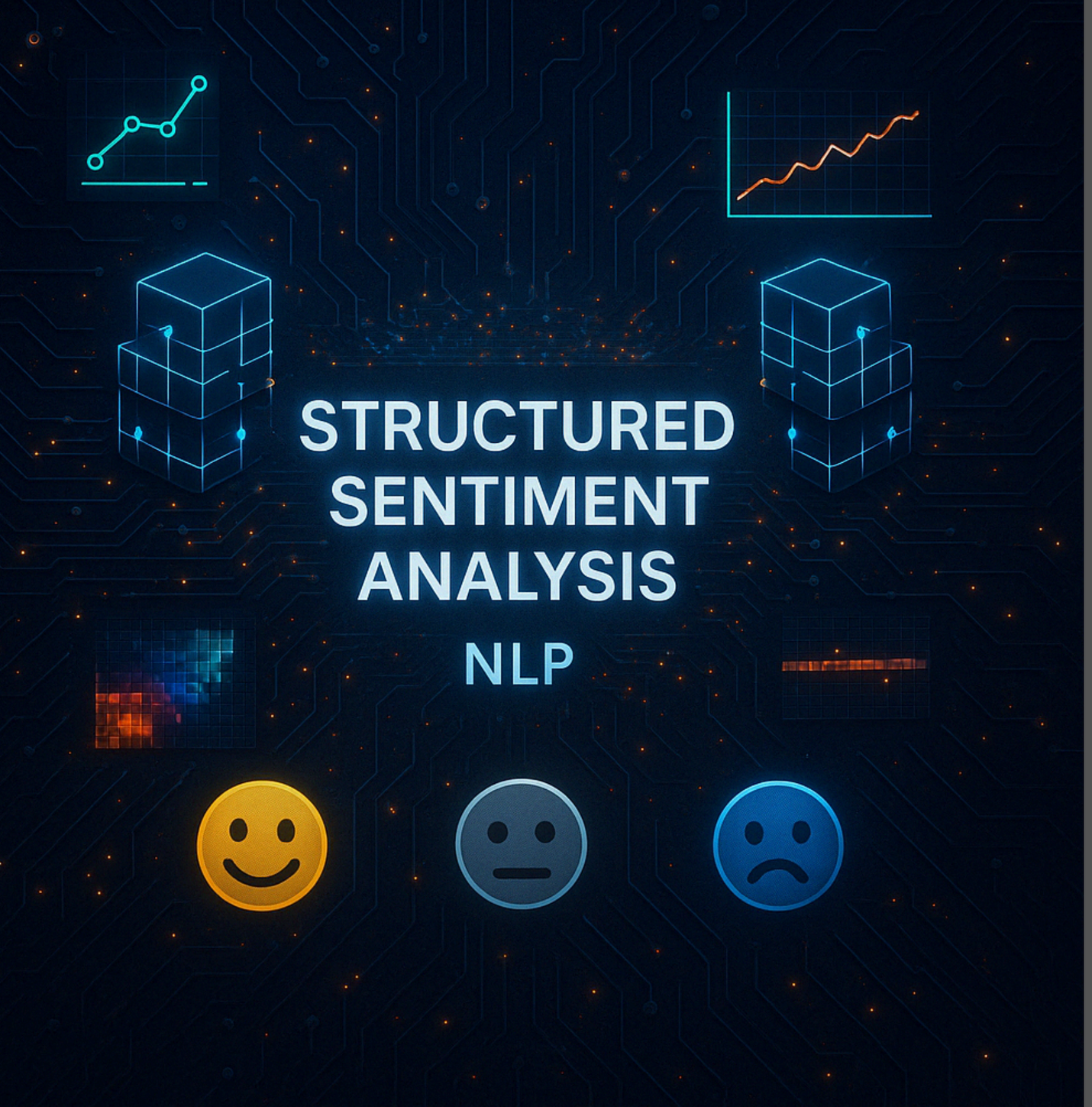


STRUCTURED SENTIMENT ANALYSIS

GROUP - 24

PROJECT

NATURAL LANGUAGE PROCESSING

VAIBHAV GUPTA - 2022553
RATNANGO GHOSH - 2022397
PRANSHU GOEL - 2022369

INTRODUCTION:

TASK DESCRIPTION:

- Structured Sentiment Analysis (SSA) extracts opinion tuples (h,t,e,p,i) from text :
  - hh: Holder (who has the opinion)
  - tt: Target (what the opinion is about)
  - ee: Expression (sentiment words)
  - pp: Polarity (Positive, Negative, Neutral, None)
  - ii: Intensity (Strong, Average, Weak)
- Task Scope :
  - Monolingual (e.g., English) and cross-lingual (e.g., Basque) subtasks.
  - Dataset: [DATASET Link] with annotated opinions.

STRUCTURE OF MODEL

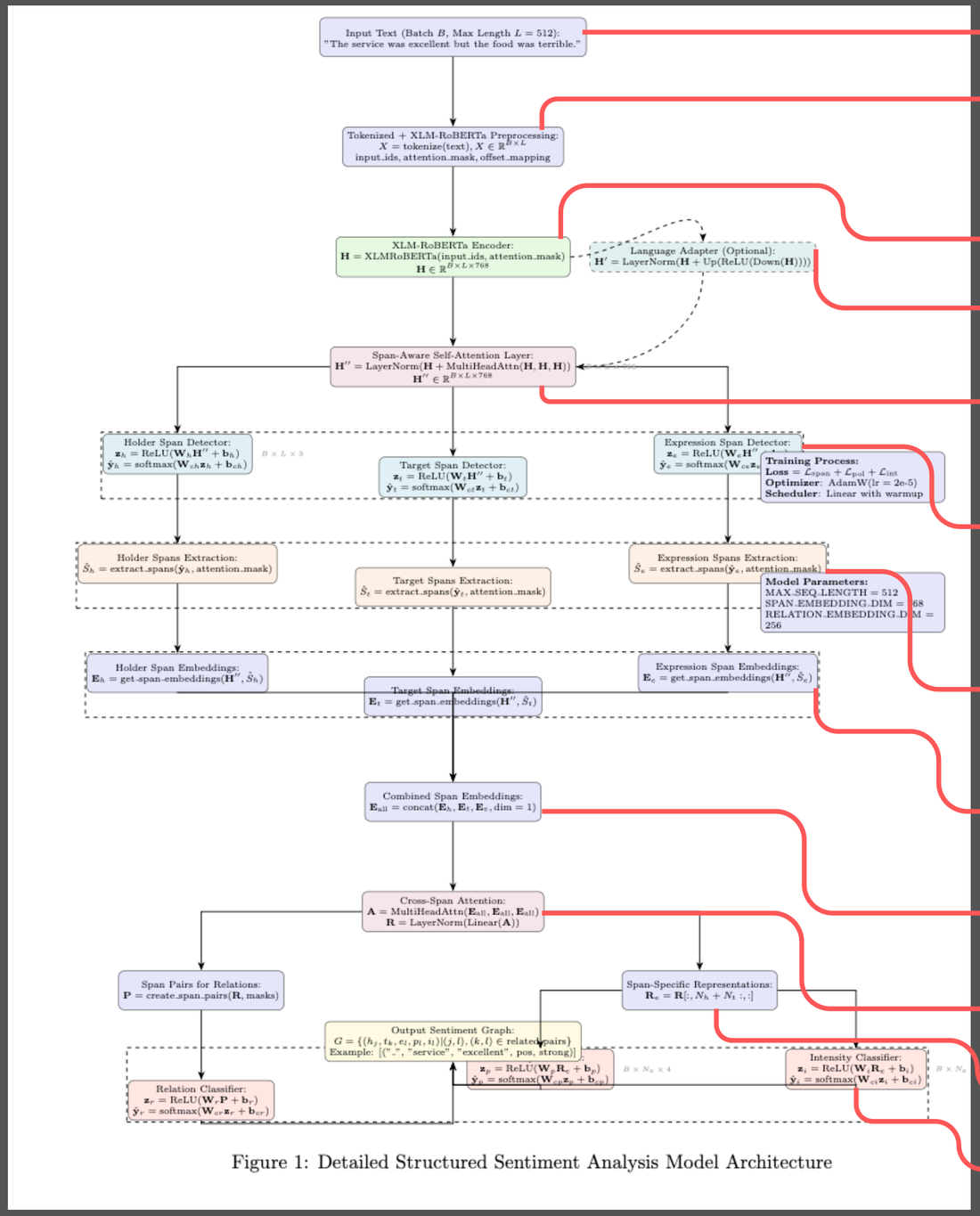


Figure 1: Detailed Structured Sentiment Analysis Model Architecture

The sentence is fed to the model, which can be of a max length of 512. (Batch Size:B, MaxLength: L)

The sentence is tokenized and encoded with the XLM-RoBERTa embeddings (multilingual trained).

A multilingual transformer encoder generates contextualized token embeddings.

A plug-in adapter layer (optional) with layer norm, dropout, and MLP layers is used for cross-lingual adaptability.

Applies multi-headed self-attention on top of encoder outputs to capture better span-specific representations.

Three parallel span detection heads predict candidate spans for:
Holder (Who expresses the sentiment)
Target (The entity the sentiment is about)
Expression (The sentiment-bearing phrase)

Uses offset mappings and attention masks to extract span positions from the predictions.

Generates span-level embeddings by aggregating token embeddings over the detected spans using self-attended features.

Concatenates the embeddings of holder, target, and expression to represent each sentiment tuple.

Applies a cross-span attention mechanism to model relationships between holders, targets, and expressions.

Combines individual span embeddings with cross-attended features to refine understanding for downstream tasks.

Optionally classifies the intensity of the sentiment (e.g., mild, strong) from the span-specific representations.

DATA LOADING AND PREPROCESSING

```python
def process_example(self, entry):
    try:
        text = entry.get('text', '')
        sent_id = entry.get('sent_id', '')
        opinions = entry.get('opinions', [])
        if not text:
            logger.warning(f"Empty text for entry with sent_id {sent_id}")
            return None

        tokenized = self.tokenizer(
            text,
            max_length=self.max_length,
            padding='max_length',
            truncation=True,
            return_offsets_mapping=True,
            return_tensors='pt'
        )
        offset_mapping = tokenized['offset_mapping'][0]
        input_ids = tokenized['input_ids'][0]
        attention_mask = tokenized['attention_mask'][0]

        holder_labels = torch.zeros(self.max_length, dtype=torch.long)
        target_labels = torch.zeros(self.max_length, dtype=torch.long)
        expression_labels = torch.zeros(self.max_length, dtype=torch.long)
```

Extracts the main sentence text, its ID sent_id, and the list of opinions (which are annotations for sentiment).

If the sentence is empty then it logs a warning and skips it.

Tokenizes the sentence using XLMRobertaTokenizerFast and returns:
input_ids, attention_mask, and offset_mapping (map between text character positions and tokens).
All tensors are returned in PyTorch (pt) format.

Unpacks the tokens for further processing.

Initializes three span label vectors:

- Each token is initially labeled 0 ("O" — outside any span).
- Labels will later be updated with 1 ("B") and 2 ("I") for beginning and inside span

```python
        opinion_data = []
        for opinion in opinions:
            if isinstance(opinion, dict):
                holder_span = self._extract_span_safely(opinion, 'Source', offset_mapping)
                target_span = self._extract_span_safely(opinion, 'Target', offset_mapping)
                expression_span = self._extract_span_safely(opinion, 'Polar_expression', offset_mapping)
                polarity = self._get_polarity_label(opinion.get('Polarity', 'None'))
                intensity = self._get_intensity_label(opinion.get('Intensity', 'Average'))

                if all(span is not None for span in [holder_span, target_span, expression_span]):
                    opinion_data.append({
                        'holder_span': holder_span,
                        'target_span': target_span,
                        'expression_span': expression_span,
                        'polarity': polarity,
                        'intensity': intensity,
                    })
                    self._mark_span(holder_labels, holder_span[0], holder_span[1])
                    self._mark_span(target_labels, target_span[0], target_span[1])
                    self._mark_span(expression_labels, expression_span[0], expression_span[1])

        return {
            'sent_id': sent_id,
            'text': text,
            'input_ids': input_ids,
            'attention_mask': attention_mask,
            'holder_labels': holder_labels,
            'target_labels': target_labels,
            'expression_labels': expression_labels,
            'opinion_data': opinion_data,
            'num_opinions': len(opinion_data)
        }
    except Exception as e:
        logger.error(f"Error processing example: {e}")
        return None
```

Loops through each opinion and safely extracts the character-level spans of:
Holder: who expresses the opinion.
Target: the subject of the opinion.
Expression: the actual phrase expressing sentiment.
Converts polarity (Positive/Negative/Neutral/None) and intensity (Strong/Average/Weak) to label indices.

If all three spans are valid, adds them to a structured opinion_data list (used later for classification tasks).

Updates the label arrays (holder_labels, etc.) with B/I labels using span indices.

- Returns the full dictionary for one example: tokenized input + all span labels + opinion metadata.
- This is used directly by the model and loss functions.

MODEL CLASS

CrossSpanAttention: allows span embeddings to interact (across holder, target, expression).

Classifiers:
relation_classifier: predicts if a span pair is related.
polarity_classifier: predicts sentiment polarity.
intensity_classifier: predicts intensity (strong, average, weak).

Loads a pre-trained xlm-roberta-base transformer.
Saves the output dimension (usually 768).

SelfAttentionLayer: enhances token embeddings using contextual self-attention.
Each SpanDetector predicts BIO labels (O, B, I) for: holder, target, and expression.

```python
class StructuredSentimentModel(nn.Module):
    def __init__(self, pretrained_model_name="xlm-roberta-base", use_adapters=False, num_languages=8):
        super(StructuredSentimentModel, self).__init__()
        self.encoder = XLMRobertaModel.from_pretrained(pretrained_model_name)
        self.hidden_size = self.encoder.config.hidden_size
        self.span_attention = SelfAttentionLayer(self.hidden_size)
        self.holder_detector = SpanDetector(self.hidden_size)
        self.target_detector = SpanDetector(self.hidden_size)
        self.expression_detector = SpanDetector(self.hidden_size)
        self.cross_span_attention = CrossSpanAttention(self.hidden_size)
        self.relation_classifier = RelationClassifier(RELATION_EMBEDDING_DIM)
        self.polarity_classifier = PolarityClassifier(RELATION_EMBEDDING_DIM)
        self.intensity_classifier = IntensityClassifier(RELATION_EMBEDDING_DIM)
        self.use_adapters = use_adapters
        if use_adapters:
            self.language_adapters = nn.ModuleList([LanguageAdapter(self.hidden_size) for _ in range(num_languages)])
        self._init_weights()
```

```python
def _init_weights(self):
    modules = [self.span_attention, self.holder_detector, self.target_detector,
               self.expression_detector, self.cross_span_attention,
               self.relation_classifier, self.polarity_classifier, self.intensity_classifier]
    for module in modules:
        for name, param in module.named_parameters():
            if 'weight' in name and len(param.shape) >= 2:
                nn.init.xavier_uniform_(param)
            elif 'bias' in name:
                nn.init.zeros_(param)
```

Initializes weights using Xavier Uniform for layers with ≥2D weights.
Biases are initialized to 0.
Lists all components that need initialization.

FORWARD FUNCTION: STRUCTURED SENTIMENT MODEL

Parameters:
input_ids: token indices from the tokenizer.
attention_mask: masks for padding tokens (1 for real tokens, 0 for padding).
language_id: optional index if language-specific adapters are used.
labels: only used during training; a tuple of:
holder_labels, target_labels, expression_labels, gold_opinions.

Feeds the input into XLM-RoBERTa.
Gets contextualized hidden states: shape (batch_size, seq_len, hidden_dim).

If adapters are enabled, applies a language-specific LanguageAdapter to the embeddings per example.

Applies a self-attention layer to enhance token representations, especially for detecting spans (Holder, Target, Expression).

Predicts token-level labels (O/B/I) for each of the three span types.

If labels are provided → training mode.

The labels are unpacked and the loss function is set

Computes span classification loss for each span type.
Applies attention mask to ignore padding.

Prepares to collect expression span embeddings and corresponding polarity/intensity labels.
Loop through each sample.
Extracts annotated spans.
Computes mean-pooled embeddings for each span type.
Combines span embeddings → applies CrossSpanAttention → extracts expression-related embeddings.
Collects expression embeddings and ground-truth labels.

Concatenates expression span embeddings from all samples.
Feeds into polarity and intensity classifiers.
Computes total training loss.
Handles edge case: no valid opinions in batch.
Returns losses and span predictions.

```python
def forward(self, input_ids, attention_mask, language_id=None, labels=None):
    batch_size = input_ids.size(0)
    encoder_outputs = self.encoder(input_ids, attention_mask=attention_mask)
    hidden_states = encoder_outputs.last_hidden_state

    if self.use_adapters and language_id is not None:
        adapted_states = torch.zeros_like(hidden_states)
        for i in range(batch_size):
            adapted_states[i] = self.language_adapters[language_id[i].item()](hidden_states[i])
        hidden_states = adapted_states

    span_aware_states = self.span_attention(hidden_states, attention_mask)
    holder_logits = self.holder_detector(span_aware_states)
    target_logits = self.target_detector(span_aware_states)
    expression_logits = self.expression_detector(span_aware_states)

    if labels is not None:
        holder_labels, target_labels, expression_labels, gold_opinions = labels
        loss_fct = nn.CrossEntropyLoss()
        holder_loss = loss_fct(holder_logits.view(-1, NUM_LABELS_SPAN)[attention_mask.view(-1) == 1], holder_labels.view(-1)[attention_mask.view(-1) == 1])
        target_loss = loss_fct(target_logits.view(-1, NUM_LABELS_SPAN)[attention_mask.view(-1) == 1], target_labels.view(-1)[attention_mask.view(-1) == 1])
        expression_loss = loss_fct(expression_logits.view(-1, NUM_LABELS_SPAN)[attention_mask.view(-1) == 1], expression_labels.view(-1)[attention_mask.view(-1) == 1])
        span_loss = holder_loss + target_loss + expression_loss

        # Compute polarity and intensity losses using gold opinions
        all_expr_emb = []
        all_polarity_labels = []
        all_intensity_labels = []
        for i in range(batch_size):
            opinions = gold_opinions[i]
            if not opinions:
                continue
            gold_holder_spans = [op['holder_span'] for op in opinions]
            gold_target_spans = [op['target_span'] for op in opinions]
            gold_expr_spans = [op['expression_span'] for op in opinions]
            holder_emb, _ = self.get_span_embeddings(span_aware_states[i:i+1], [gold_holder_spans], attention_mask[i:i+1])
            target_emb, _ = self.get_span_embeddings(span_aware_states[i:i+1], [gold_target_spans], attention_mask[i:i+1])
            expr_emb, _ = self.get_span_embeddings(span_aware_states[i:i+1], [gold_expr_spans], attention_mask[i:i+1])
            all_emb, all_mask = self._combine_spans(
                holder_emb, torch.ones_like(holder_emb[..., 0], dtype=torch.bool),
                target_emb, torch.ones_like(target_emb[..., 0], dtype=torch.bool),
                expr_emb, torch.ones_like(expr_emb[..., 0], dtype=torch.bool)
            )
            relation_aware_emb = self.cross_span_attention(all_emb, all_mask)
            expr_relation_aware = relation_aware_emb[:, -len(gold_expr_spans):, :]
            all_expr_emb.append(expr_relation_aware[0])
            all_polarity_labels.extend([op['polarity'] for op in opinions])
            all_intensity_labels.extend([op['intensity'] for op in opinions])

        if all_expr_emb:
            all_expr_emb = torch.cat(all_expr_emb, dim=0)
            polarity_logits = self.polarity_classifier(all_expr_emb)
            intensity_logits = self.intensity_classifier(all_expr_emb)
            gold_polarity = torch.tensor(all_polarity_labels, dtype=torch.long, device=hidden_states.device)
            gold_intensity = torch.tensor(all_intensity_labels, dtype=torch.long, device=hidden_states.device)
            polarity_loss = loss_fct(polarity_logits, gold_polarity)
            intensity_loss = loss_fct(intensity_logits, gold_intensity)
            total_loss = span_loss + polarity_loss + intensity_loss
        else:
            total_loss = span_loss

        return {
            'loss': total_loss,
            'holder_logits': holder_logits,
            'target_logits': target_logits,
            'expression_logits': expression_logits
        }
```

TRAIN_MODEL FUNCTION:

Initializes best_f1 to track the best F1 score achieved on the dev set.

model: an instance of StructuredSentimentModel.
train_dataloader: the training data loader.
dev_dataloader: the validation (dev) data loader.
optimizer: the optimizer (usually AdamW).
scheduler: a learning rate scheduler.
device: training device (cuda or cpu).
num_epochs: number of training epochs.
output_dir: path to save model checkpoints.

Moves all relevant tensors to the GPU/CPU.
opinion_data is a list of opinion dictionaries (not a tensor).

Feeds the batch through the model.
Since labels are passed, the model enters training mode and computes the total loss.

Extracts loss.
Clears old gradients.
Backpropagates to compute gradients.
Updates model parameters.
Adjusts learning rate using the scheduler.

Computes the average F1 score and if this epoch has the best F1 so far, save the model checkpoint.

```python
def train_model(model, train_dataloader, dev_dataloader, optimizer, scheduler, device, num_epochs, output_dir):
    best_f1 = 0.0
    for epoch in range(num_epochs):
        logger.info(f"Starting epoch {epoch+1}/{num_epochs}")
        model.train()
        train_loss = 0.0
        train_steps = 0
        for batch in tqdm(train_dataloader, desc=f"Epoch {epoch+1}/{num_epochs}"):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            holder_labels = batch['holder_labels'].to(device)
            target_labels = batch['target_labels'].to(device)
            expression_labels = batch['expression_labels'].to(device)
            gold_opinions = batch['opinion_data']  # List of lists
            outputs = model(
                input_ids=input_ids,
                attention_mask=attention_mask,
                labels=(holder_labels, target_labels, expression_labels, gold_opinions)
            )
            loss = outputs['loss']
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            scheduler.step()
            train_loss += loss.item()
            train_steps += 1
        avg_train_loss = train_loss / train_steps if train_steps > 0 else 0
        logger.info(f"Epoch {epoch+1} - Average training loss: {avg_train_loss:.4f}")

        eval_results = evaluate_model(model, dev_dataloader, device)
        avg_f1 = sum(m['f1'] for m in eval_results.values()) / len(eval_results)
        logger.info(f"Epoch {epoch+1} - Evaluation F1: {avg_f1:.4f}")
        if avg_f1 > best_f1:
            best_f1 = avg_f1
            torch.save({
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'best_f1': best_f1
            }, os.path.join(output_dir, f"best_model_f1_{best_f1:.4f}.pt"))
    return model
```

RESULTS:

MPQA (English language) model on Hindi Language

Quite promising outputs can be seen

Prediction on Hindi sentence

```
Analysis for: 'राष्ट्रपति द्रौपदी मुर्मू ने राष्ट्र का अभिनंदन किया और देशवासियों से आग्रह किया कि वे सभी के जीवन को निरंतर प्रगति, समृद्धि और खुशहाली के रंगों से भर दें।'
Holders: ['राष्ट्रपति द्रौपदी मुर्मू', 'देशवास']
Targets: ['राष्ट्र', 'देशवासियों', 'कि वे सभी के जीवन को निरंतर प्रगति, समृद्धि और खुशहाली के रंगों से भर दें']
Expressions: ['का अभिनंदन', 'से आग्रह किया']

Opinions:
  Opinion 1:
    Holder: 'देशवास'
    Target: 'कि वे सभी के जीवन को निरंतर प्रगति, समृद्धि और खुशहाली के रंगों से भर दें'
    Expression: 'का अभिनंदन'
    Polarity: Positive
    Intensity: Average
  Opinion 2:
    Holder: 'देशवास'
    Target: 'कि वे सभी के जीवन को निरंतर प्रगति, समृद्धि और खुशहाली के रंगों से भर दें'
    Expression: 'से आग्रह किया'
    Polarity: Positive
    Intensity: Average
```

BIO encoding of sentence

BIO Encodings:

Token	Holder	Target	Expression
<s>	O	O	O
_राष्ट्रपत...	I	O	O
ि	I	O	O
द्र	I	O	O
ौ		I	O	O
पद	I	O	O
ी		I	O	O
_मु	I	O	O
र्म	I	O	O
ू	I	O	O
_ने	O	O	O
_राष्ट्र	O	I	O
_का	O	O	I
_अभि	O	O	I
न	O	O	I
दन	O	O	I
_किया	O	O	O
_और	O	O	O
_देश	I	I	O
वास	I	I	O
ियों	O	I	O
_से	O	O	I
_आग्रह	O	O	I
_किया	O	O	I
_कि	O	I	O
_वे	O	I	O
_सभी	O	I	O
_के	O	I	O
_जीवन	O	I	O
_को	O	I	O
_निर	O	I	O
ं	O	I	O
तर	O	I	O
_प्रगति	O	I	O
,	O	I	O
_समृद्धि	O	I	O
_और	O	I	O
_खुश	O	I	O
हा	O	I	O
ली	O	I	O
_के	O	I	O
_रंग	O	I	O
ों	O	I	O
_से	O	I	O
_भर	O	I	O
_दें	O	I	O
।	O	O	O
</s>	O	O	O

RESULTS:

Opener_en (English language) model
on Spanish Language
Quite promising outputs can be seen

Prediction on Spanish sentence

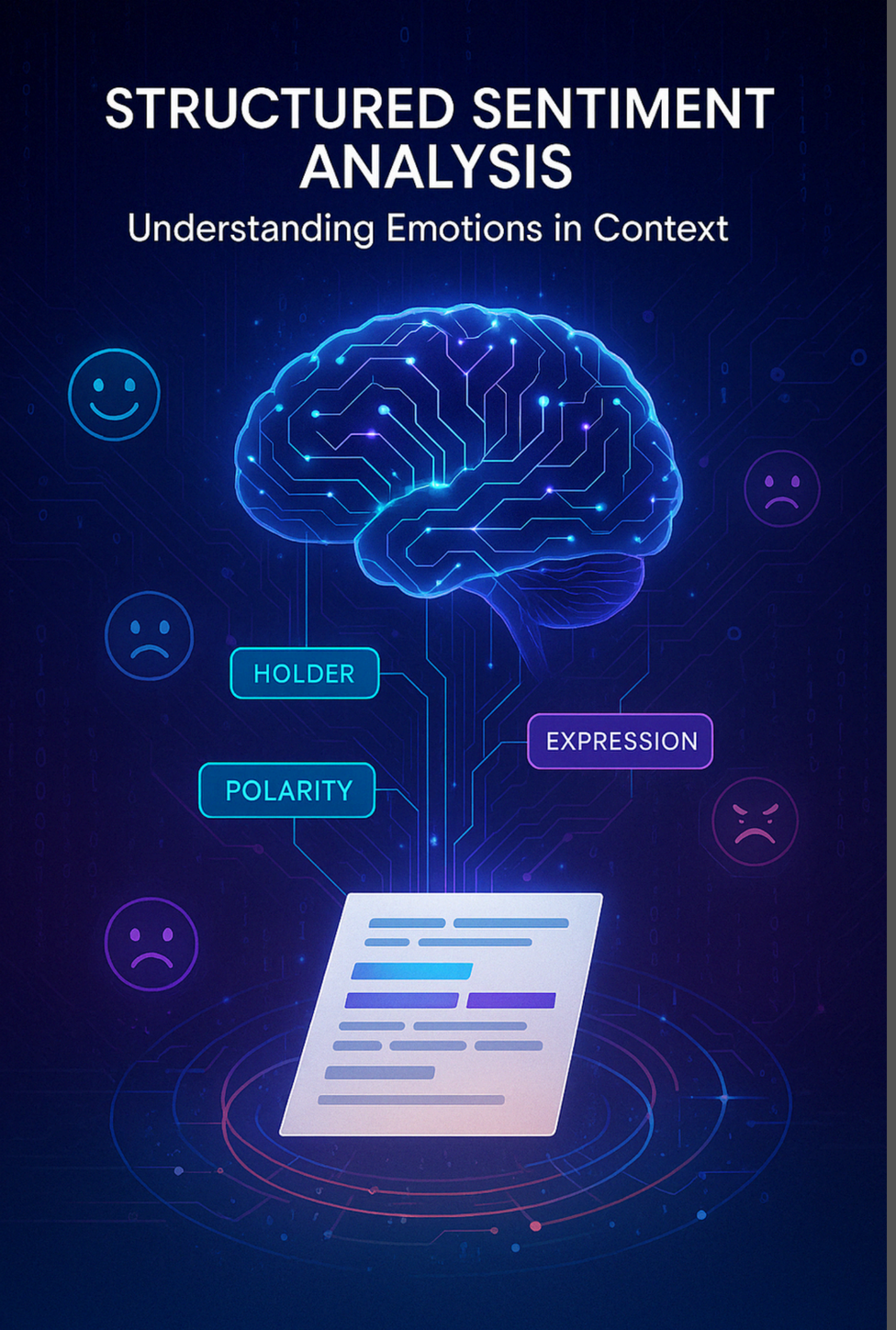
```
Analysis for: 'No es un hotel barato , por supuesto , pero si el dinero no es impedimento es sin duda de los mejores , sino el mejor , hotel de la capital '
Holders: []
Targets: ['hotel', 'hotel']
Expressions: ['No', 'barato', 'duda', 'los mejores', 'mejor']

Opinions:
  Opinion 1:
    Holder: ''
    Target: 'hotel'
    Expression: 'No'
    Polarity: Negative
    Intensity: Average
  Opinion 2:
    Holder: ''
    Target: 'hotel'
    Expression: 'barato'
    Polarity: Negative
    Intensity: Average
  Opinion 3:
    Holder: ''
    Target: 'hotel'
    Expression: 'duda'
    Polarity: Positive
    Intensity: Strong
  Opinion 4:
    Holder: ''
    Target: 'hotel'
    Expression: 'los mejores'
    Polarity: Positive
    Intensity: Strong
  Opinion 5:
    Holder: ''
    Target: 'hotel'
    Expression: 'mejor'
    Polarity: Positive
    Intensity: Strong
```

BIO encoding of sentence

BIO Encodings:

Token	Holder	Target	Expression
<s>	B	O	O
_No	O	O	B
_es	O	O	O
_un	O	O	O
_hotel	O	B	O
_barato	O	O	I
_	O	O	O
,	O	O	O
_por	O	O	O
_supuesto	O	O	O
_	O	O	O
,	O	O	O
_pero	O	O	O
_si	O	O	O
_el	O	O	O
_dinero	O	O	O
_no	O	O	O
_es	O	O	O
_impedi	O	O	O
mento	O	O	O
_es	O	O	O
_sin	O	O	O
_duda	O	O	I
_de	O	O	O
_los	O	O	I
_mejores	O	O	I
_	O	O	O
,	O	O	O
_sino	O	O	O
_el	O	O	O
_mejor	O	O	I
_	O	O	O
,	O	O	O
_hotel	O	B	O
_de	O	O	O
_la	O	O	O
_capital	O	O	O
</s>	O	O	O



THANK YOU NLP GROUP 24 SIGNING OFF !!