Automated Classification and Scoring of Argumentative Elements in Student Essays Using NLP

Varnit Mittal (IMT2022025)
Aditya Priyadarshi (IMT2022075)
Ananthakrishna K (IMT2022086)
Vedant Mangrulkar (IMT2022519)

April 22, 2025

Abstract

Academic writing is an essential component of academic achievement, but writing competence in students is alarmingly low. Automated writing feedback tools have great potential for helping to assist and improve students' writing, but existing tools have limitations, including proprietary constraints, expense, and limited capability to recognize subtle argumentative structures. In this project, we overcome these limitations by creating an NLP model that can break down students' essays into rhetorical and argumentative pieces and classify each one into one of seven discourse element classes: Lead, Position, Claim, Counterclaim, Rebuttal, Evidence, and Concluding Statement. We use the largest public dataset of student-annotated essays to design, train, and test models that learn from human-tagged discourse fragments. Our system offers both structural feedback and class-based scoring, opening the door to more equitable, interpretable writing support tools. The developed model is designed to be deployed to education settings in an accessible manner for pedagogical use.

1 Introduction

Writing is the bridge between thought and impact: it lets students explore ideas, persuade audiences, and make their voices heard. Yet, fewer than one-third of U.S. high school seniors reach proficient levels in writing on the National Assessment of Educational Progress (NAEP).

Our project envisions a more equitable, transparent approach: an open, NLP-powered system that doesn't just flag grammar mistakes but actually understands the architecture of an argument. By breaking essays into meaningful units—**leads**, **positions**, **claims**, **counterclaims**, **rebuttals**, **evidence**, and **conclusions**—we can mirror the formative guidance a human instructor provides.

Leveraging one of the largest expert-annotated corpora of grades 6–12 argumentative essays, we preprocess and segment raw text into discourse units, then apply state-of-the-art classification models to label each unit. Beyond labeling, we generate class-specific feedback scores, offering students concrete, actionable insights on their rhetorical strengths and weaknesses.

In doing so, we demonstrate a scalable, research-grade method for argumentative discourse analysis—laying the groundwork for democratically accessible writing support that grows with every new essay it ingests.

2 Dataset Description

Our project uses two complementary data files released for the **Kaggle Competition**:

1. **Discourse Annotation File**: contains span-level annotations for each essay.
2. **Scoring File**: contains essay-level counts of each discourse category.

2.1 Discourse Annotation File

Each row in the CSV represents one *discourse unit* (a contiguous text span). We record:

- **Essay ID** (`id`): unique essay identifier.
- **Character offsets** (`discourse_id_start`, `discourse_id_end`): start and end in the raw `.txt`.
- **Discourse text** (`discourse_text`): the exact substring.
- **Gold label** (`discourse_type`): one of {Lead, Position, Claim, Counterclaim, Rebuttal, Evidence, Concluding Statement}.
- **Segment index** (`discourse_type_num`): ordinal count (e.g., "Claim 1", "Evidence 2").
- **Token indices** (`prediction_text`): 1-indexed token positions for evaluation.

Table 1: Excerpt of annotated discourse units for essay `423A1CA112E2`.

ID	**start**	**end**	**Discourse Text**	**Type**	**Type_Num**	**Token Indices**
423A1CA112E2	0	229	Modern humans today are always on their phone…	Lead	1	1–36
423A1CA112E2	230	312	They are some really bad consequences when…	Position	1	45–59
423A1CA112E2	313	401	Some certain areas in the United States ban…	Evidence	1	60–75
423A1CA112E2	402	758	When people have phones, they know about…	Evidence	2	76–162
423A1CA112E2	759	886	Driving is one of the way how to get around…	Claim	1	139–162
423A1CA112E2	887	1150	That's why there's a thing that's called…	Evidence	3	163–187
…	…	…	…	…	…	…

2.2 Scoring File

This CSV provides, for each essay, the total counts of segments in each discourse category. We record:

- **essay_id**: unique essay identifier.
- One column per discourse type (e.g., `lead`, `position`, `claim`, …).
- Each cell: number of annotated segments of that type in the essay.

In total, the annotation file contains over a million discourse units across roughly 10000 essays, and the scoring file provides per-essay counts for all seven categories. We leverage the span-level annotations to train our segmentation and classification models, and the counts file to evaluate overall coverage and to support downstream feedback scoring.

Table 2: Excerpt of essay-level discourse counts.

essay_id	lead	position	claim	rebuttal	evidence	concluding	counterclaim
65A2037C80C4	0	8	56	0	84	18	0
4A687A432A70	36	22	80	0	80	55	0
0EF2B186C58C	0	7	67	0	90	63	0
67843D0A6F5B	0	9	18	0	67	31	0
C848C44E3004	9	8	50	10	84	22	9
811DE32FCE53	81	9	90	35	90	70	14
...	...	...	...	...	...	...	...

3 Data Pre-processing and Exploratory Analysis

3.1 Data Loading and Corpus Structure

We begin by loading the provided CSV file ("1k.csv"), which contains two fields per essay: `tokens` (a Python list of word tokens) and `labels` (the corresponding discourse-element tags). We parse each list using `ast.literal_eval` and collate them into a Pandas DataFrame of length $N = 1000$ essays.

3.2 Vocabulary and Label Encoding

To prepare for model input, we build a top-10 000 word vocabulary (lowercased) plus special `<PAD>` and `<UNK>` tokens. Concurrently, we enumerate the set of unique discourse tags (seven classes {O, Lead, Position, ...}) and reserve an integer for the padding label. This produces:

$$|\text{Vocab}| = 10\,000 + 2 = 10\,002, \quad |\text{Labels}| = 8, \quad \texttt{PAD\_TOKEN\_ID} = 0, \quad \texttt{PAD\_LABEL\_ID} = 0.$$

```python
from collections import Counter

# Build word2id
all_tokens = [tok.lower() for seq in df.tokens for tok in seq]
freqs = Counter(all_tokens)
common = [w for w,_ in freqs.most_common(10000)]
word2id = {w: i+2 for i,w in enumerate(common)}
word2id["<PAD>"] = 0;  word2id["<UNK>"] = 1

# Build label2id
unique_labels = sorted({lab for seq in df.labels for lab in seq})
label2id = {lab: i for i,lab in enumerate(unique_labels)}
```

3.3 Sequence Padding and Splitting

We fix a maximum sequence length of $L = 641$. Each token sequence is truncated or zero-padded to length L, and label sequences are padded with `PAD_LABEL_ID`. Finally, we perform an 80/20 train–validation split with a fixed random seed for reproducibility:

```
MAX_LEN = 641
PAD_ID = word2id["<PAD>"]
PAD_LAB = label2id["O"]

X = [[word2id.get(t.lower(),1) for t in seq] for seq in df.tokens]
y = [[label2id[l] for l in seq] for seq in df.labels]

X_padded = [
    seq[:MAX_LEN] + [PAD_ID]*(MAX_LEN-len(seq)) if len(seq)<MAX_LEN else
       seq[:MAX_LEN]
    for seq in X
]
y_padded = [
    seq[:MAX_LEN] + [PAD_LAB]*(MAX_LEN-len(seq)) if len(seq)<MAX_LEN else
       seq[:MAX_LEN]
    for seq in y
]

from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(
    X_padded, y_padded, test_size=0.2, random_state=42
)
```

3.4 Exploratory Data Analysis

We now explore key statistics of essay length and label distribution.

These analyses reveal:

- A moderate average essay length (mean ≈ 450 tokens), informing our choice of $L = 641$ to cover $\approx 95\%$ of essays without excessive padding.
- A strong class imbalance (Evidence $\gg$ Claim $\gg$ Concluding $\gg$ others), motivating weighted loss functions or data augmentation in subsequent modeling.

Finally, we compute training hyperparameters based on the split size:

$$\text{num\_train\_steps} = \frac{|\text{train}|}{\text{batch\_size}} \times \text{epochs}, \quad \text{warmup\_steps} = 0.1 \times \text{num\_train\_steps}.$$

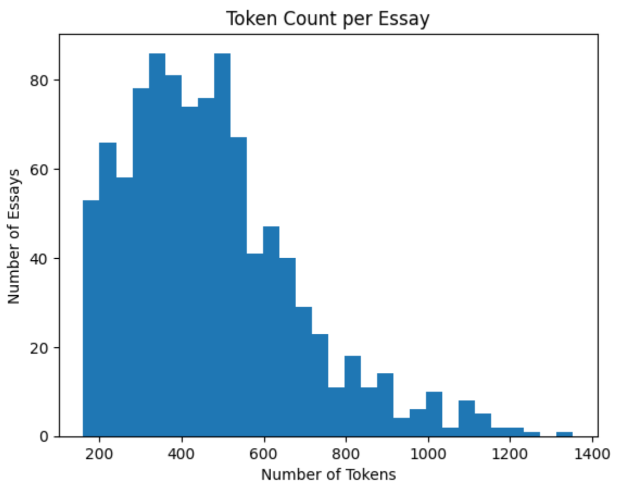

Figure 1: Histogram of token count per essay. Most essays range from 200–600 tokens, with a long tail up to 1,300.

This completes our data-preparation pipeline and sets the stage for Transformer-based modeling in Section 4.

4 Methodology

4.1 Token Classification

This subsection describes our Transformer-based token-classification model for segmenting and labeling each essay span. It covers data preparation, model rationale, architecture, and the training procedure.

4.1.1 Data Preparation

After padding and splitting (see Section 3.1), we convert arrays into PyTorch tensors and build `DataLoaders`:

Listing 1: Tensor conversion and DataLoader setup

```
# convert to tensors
X_train = torch.LongTensor(X_train_np)
y_train = torch.LongTensor(y_train_np)
X_val   = torch.LongTensor(X_val_np)
y_val   = torch.LongTensor(y_val_np)

# create datasets & loaders
train_dataset = TensorDataset(X_train, y_train)
```

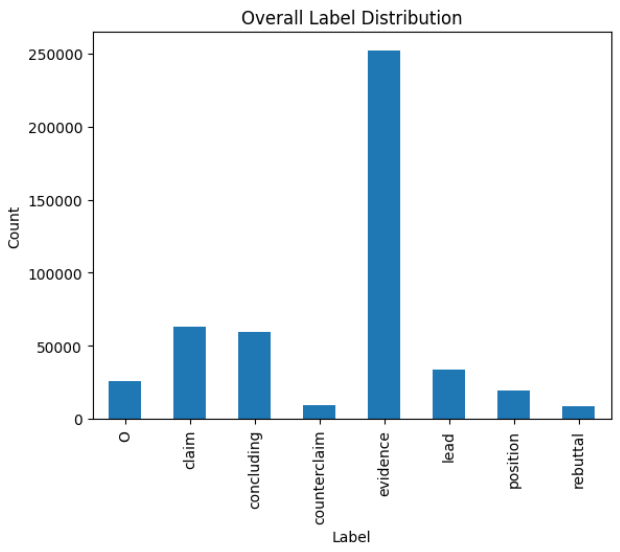

Figure 2: Overall label distribution across all tokens. Evidence segments dominate, followed by Claim and Concluding.

```
val_dataset    = TensorDataset(X_val,    y_val)
train_loader   = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=
   True)
val_loader     = DataLoader(val_dataset,   batch_size=BATCH_SIZE, shuffle=
   False)
```

4.1.2 Model Rationale

We opted for a Transformer-based token-classification model because argumentative essays often contain long-range dependencies: for example, a piece of evidence in one paragraph might directly support a claim introduced several sentences earlier. Unlike recurrent architectures, Transformers can attend to every token in the sequence simultaneously, making it easier to capture these cross-sentential relationships. Moreover, by framing segmentation and labeling as a token-level task, we can leverage pre-trained language models (via our embedding layer) to bring in rich linguistic knowledge, while still fine-tuning on our specific discourse tags. The sinusoidal positional encoding further grounds the model in the essay's linear structure, ensuring that relative token positions influence how attention is distributed during both training and inference.

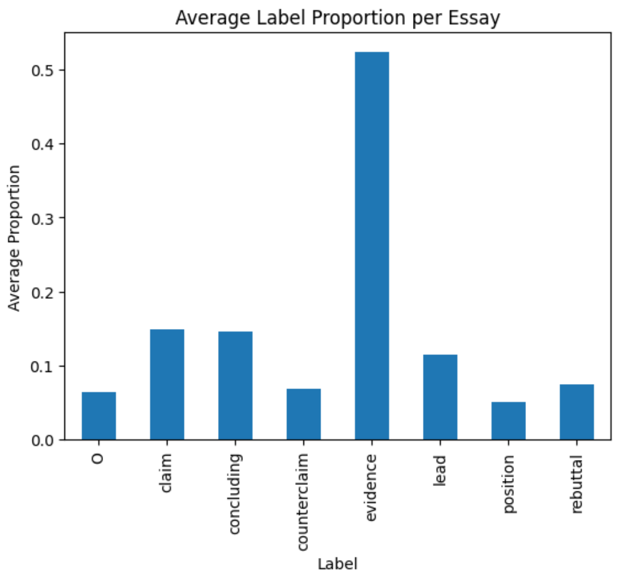

Figure 3: Average proportion of each label within an essay. While Evidence comprises over 50% of segments on average, other elements such as Rebuttal and Position appear less frequently.

4.1.3 Model Architecture

Figure 4 illustrates the overall architecture: a token embedding layer, sinusoidal positional encodings, N stacked Transformer encoder blocks, and a final linear layer mapping to discourse-type logits at each position.

Sinusoidal Positional Encoding

Listing 2: Sinusoidal Positional Encoding

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        pos = torch.arange(0, max_len).unsqueeze(1).float()
        div = torch.exp(torch.arange(0, d_model, 2).float() *
                        (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(pos * div)
        pe[:, 1::2] = torch.cos(pos * div)
        self.register_buffer('pe', pe.unsqueeze(0))
    def forward(self, x):
        return x + self.pe[:, :x.size(1)]
```

Transformer-Based Classifier

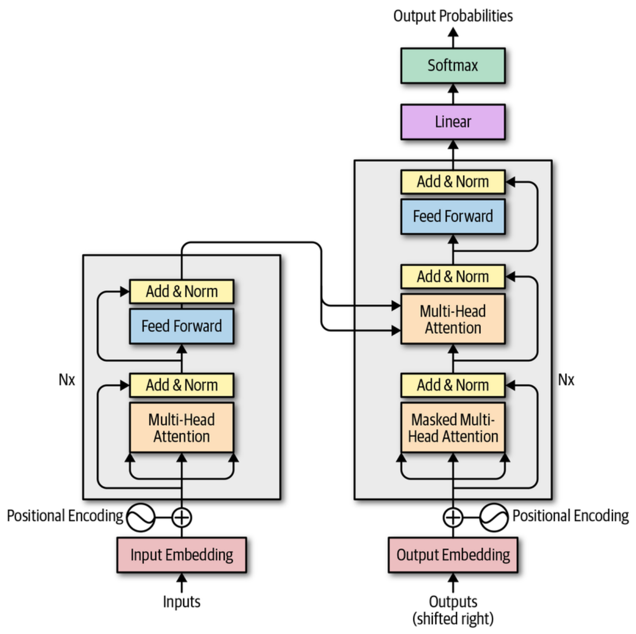

Figure 4: Token-Classification Transformer

Listing 3: Token-Classification Transformer Definition

```
class TokenClassifierTransformer(nn.Module):
    def __init__(self, num_blocks, embed_dim, num_heads, ff_dim,
                 vocab_size, num_labels, max_len, dropout, padding_idx):
        super().__init__()
        self.embedding   = nn.Embedding(vocab_size, embed_dim, padding_idx
            )
        self.pos_encoder = PositionalEncoding(embed_dim, max_len)
        self.encoder     = nn.ModuleList([
            TransformerEncoderBlock(embed_dim, num_heads, ff_dim, dropout)
            for _ in range(num_blocks)
        ])
        self.classifier  = nn.Linear(embed_dim, num_labels)

    def forward(self, input_ids):
        mask = (input_ids == self.embedding.padding_idx)
        x = self.embedding(input_ids) * math.sqrt(self.embedding.
            embedding_dim)
        x = self.pos_encoder(x)
        for block in self.encoder:
            x = block(x, padding_mask=mask)
        return self.classifier(x)
```

4.1.4 Training Procedure

The training loop utilizes the AdamW optimizer with weight decay and a linear learning rate scheduler with warmup to stabilize early training. Cross-entropy loss is computed while ignoring padded label indices to prevent gradient updates on non-informative tokens. Gradient clipping with a max norm of 1.0 is applied to prevent exploding gradients. During validation, the model operates in evaluation mode with no gradient computation, and performance is tracked to retain the best-performing checkpoint based on validation accuracy.

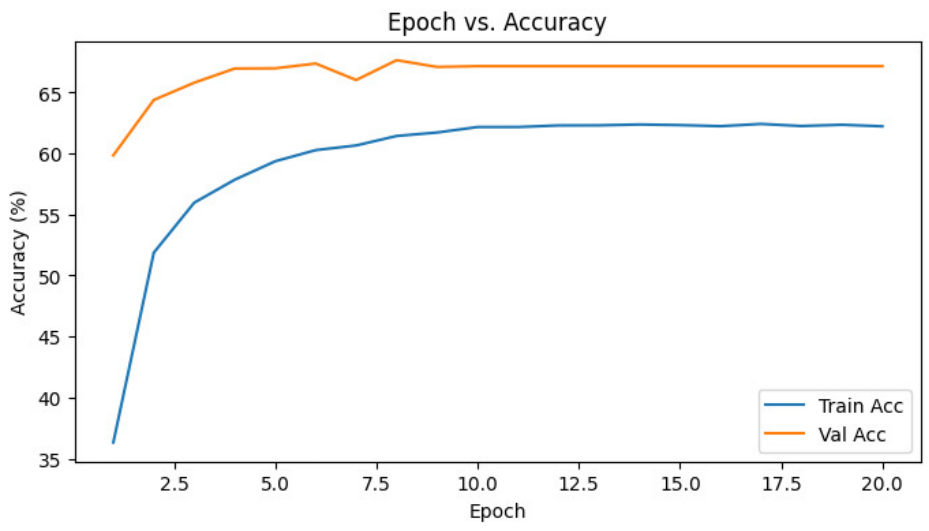

Figure 5: Epoch vs Accuracy

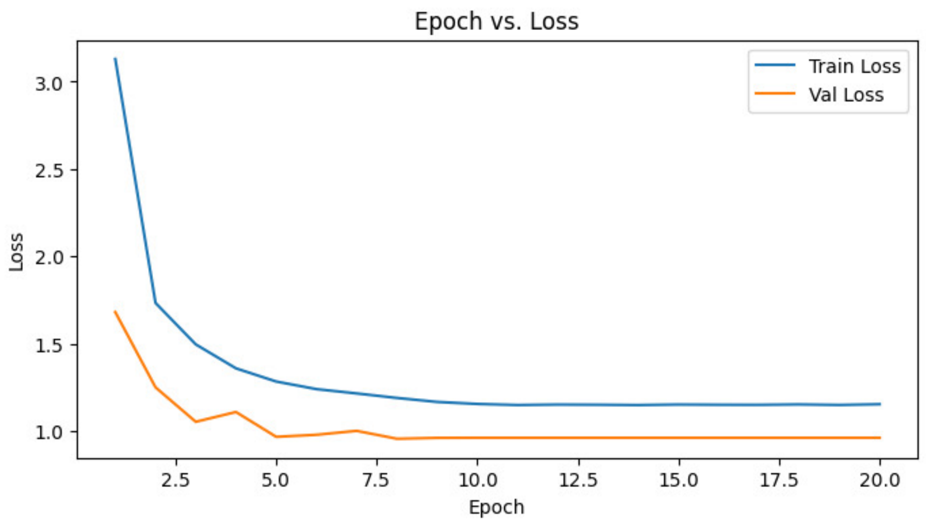

Figure 6: Epoch vs Loss

4.2 Score Prediction with BERT and Hierarchical MoE

In this component, we leverage a BERT-based encoder followed by a mixture-of-experts (MoE) module to regress human-assigned scores for each discourse element.

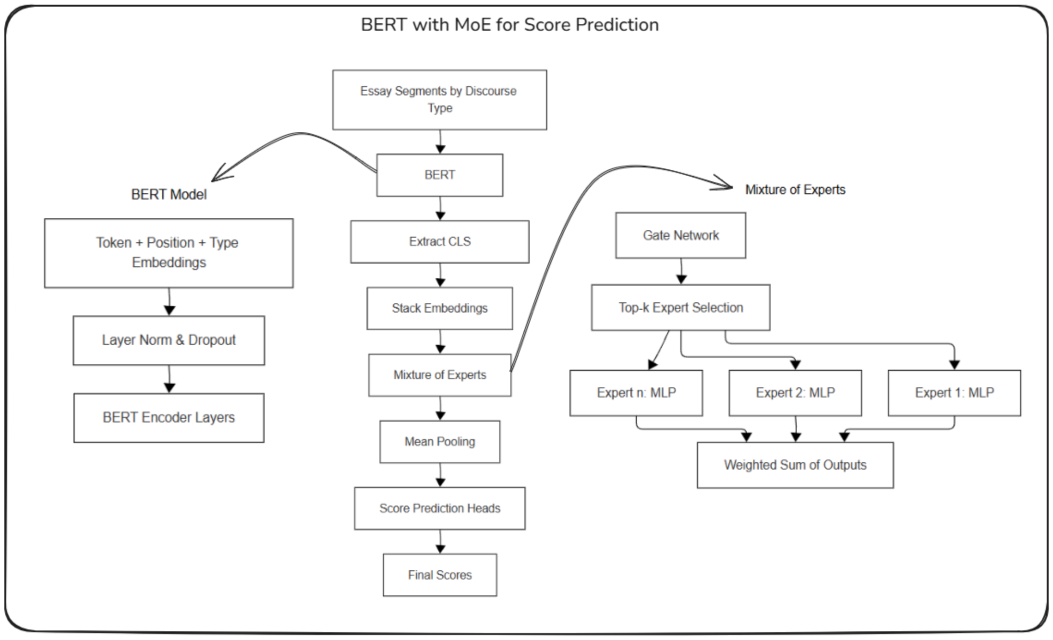

Figure 7: Hierarchical MoE Scorer

4.2.1 Core BERT Layers

We build BERT from scratch to tightly control hidden sizes and integrate MoE. Key pieces:

Listing 4: Intermediate and Output Modules

```
class BertIntermediate(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense             = nn.Linear(config.hidden_size,
                                           config.intermediate_size)
        self.intermediate_act = get_activation(config.hidden_act)

    def forward(self, hidden_states):
        return self.intermediate_act(self.dense(hidden_states))

class BertOutput(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense    = nn.Linear(config.intermediate_size,
                                  config.hidden_size)
        self.LayerNorm = nn.LayerNorm(config.hidden_size,
                                      eps=config.layer_norm_eps)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, hidden_states, input_tensor):
        h = self.dropout(self.dense(hidden_states))
        return self.LayerNorm(h + input_tensor)
```

These two modules follow the self-attention block within each layer, transforming from hidden-size to intermediate-size (typically 4×) and back again, with residual connections and layer-norm to stabilize training.

4.2.2 Stacking Layers into a Full Encoder

Listing 5: BertLayer and BertEncoder

```
class BertLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.attention    = BertAttention(config)
        self.intermediate = BertIntermediate(config)
        self.output       = BertOutput(config)

    def forward(self, hidden_states, attention_mask=None):
        attn_out = self.attention(hidden_states, attention_mask)
        inter_out = self.intermediate(attn_out)
        return self.output(inter_out, attn_out)

class BertEncoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.layer = nn.ModuleList(
            [BertLayer(config) for _ in range(config.num_hidden_layers)]
        )

    def forward(self, hidden_states, attention_mask=None):
        for layer_module in self.layer:
            hidden_states = layer_module(hidden_states, attention_mask)
        return hidden_states
```

Here, BertEncoder sequentially applies each BertLayer, building deep contextual representations.

4.2.3 Hierarchical MoE Scorer

After extracting the [CLS] embedding for each discourse type, we form a small Mixture-of-Experts over these embeddings:

Listing 6: MoE Interaction and Scorer

```
class MoEInteraction(nn.Module):
    def __init__(self, d_model, n_experts, k=2):
        super().__init__()
        self.gate    = nn.Linear(d_model, n_experts)
        self.experts = nn.ModuleList([
            nn.Sequential(
                nn.Linear(d_model, 4*d_model),
                nn.GELU(),
                nn.Linear(4*d_model, d_model)
            ) for _ in range(n_experts)
        ])
        self.k = k

    def forward(self, E):
        # E: [n_experts, d_model]
        gate_logits = self.gate(E)                          # [n_experts,
            n_experts]
        topk_vals, topk_inds = torch.topk(gate_logits, self.k, dim=-1)
        out = torch.zeros_like(E)
        for i in range(E.size(0)):
            weights = torch.softmax(topk_vals[i], dim=-1)
            expert_outputs = [
                weights[j].unsqueeze(0) * self.experts[idx](E[i])
                for j, idx in enumerate(topk_inds[i])
            ]
            out[i] = torch.stack(expert_outputs).sum(dim=0)
        return out

class HierarchicalMoEScorer(nn.Module):
    def __init__(self, bert_config, n_discourse, k, n_labels):
        super().__init__()
        self.bert    = BertModel(bert_config)
        self.moe     = MoEInteraction(bert_config.hidden_size,
                                      n_discourse, k)
        self.heads    = nn.ModuleList([
            nn.Linear(bert_config.hidden_size, 1)
            for _ in range(n_labels)
        ])

    def forward(self, tokenized_batch):
        # 1) BERT encode each discourse type
        embeddings = []
        for dt in discourse_types:
            out = self.bert(**tokenized_batch[dt])
            embeddings.append(out["last_hidden_state"][:,0,:])
        E = torch.stack(embeddings, dim=1)    # [B, D, H]
        # 2) Apply MoE per example
        moe_out = torch.stack([self.moe(E[b]) for b in range(E.size(0))],
            dim=0)
        # 3) Pool and predict scores
        pooled = moe_out.mean(dim=1)          # [B, H]
        preds  = torch.cat([h(pooled) for h in self.heads], dim=-1)
        return preds
```

Explanation of Key Steps:

- **Multi-Segmentation Encoding:** Each discourse segment ("lead", "claim", etc.) is independently tokenized and fed through the shared BERT encoder; we then extract the special '[CLS]' token as a fixed-length summary for that segment.
- **Mixture-of-Experts Routing:** The stack of segment embeddings (one per discourse type) is treated as inputs to an MoE layer. A lightweight gating network selects the top-k "experts" for each segment embedding, combining their outputs—this allows specialization per discourse type while keeping overall parameters manageable.
- **Regression Heads:** After MoE fusion, we average across discourse types and feed the result to one linear head per score label, producing a continuous prediction for each rubric dimension.

This hierarchical design captures both within-segment contextual nuance (via BERT) and cross-segment interactions (via MoE), yielding more precise score estimates on unseen essays.

4.2.4 Training Procedure

We train the Hierarchical MoE Scorer using a standard supervised learning loop optimized for regression. The model is trained with the AdamW optimizer and Mean Squared Error

(MSE) loss to regress continuous rubric-based scores for essays. During each epoch, mini-batches of essays are tokenized by discourse type and individually passed through a shared BERT encoder. The [CLS] embeddings from each discourse segment are aggregated and routed through a lightweight Mixture-of-Experts (MoE) layer to model cross-segment interactions. The pooled output is then passed to a set of regression heads, each predicting one rubric dimension. For validation, the model is evaluated without gradient computation, and Root Mean Square Error (RMSE) is computed both per score dimension and on average across dimensions. This allows monitoring of both loss convergence and prediction accuracy. The training loop also includes handling of device transfer, shape mismatches, and efficient logging of progress, ensuring robustness and transparency during optimization.

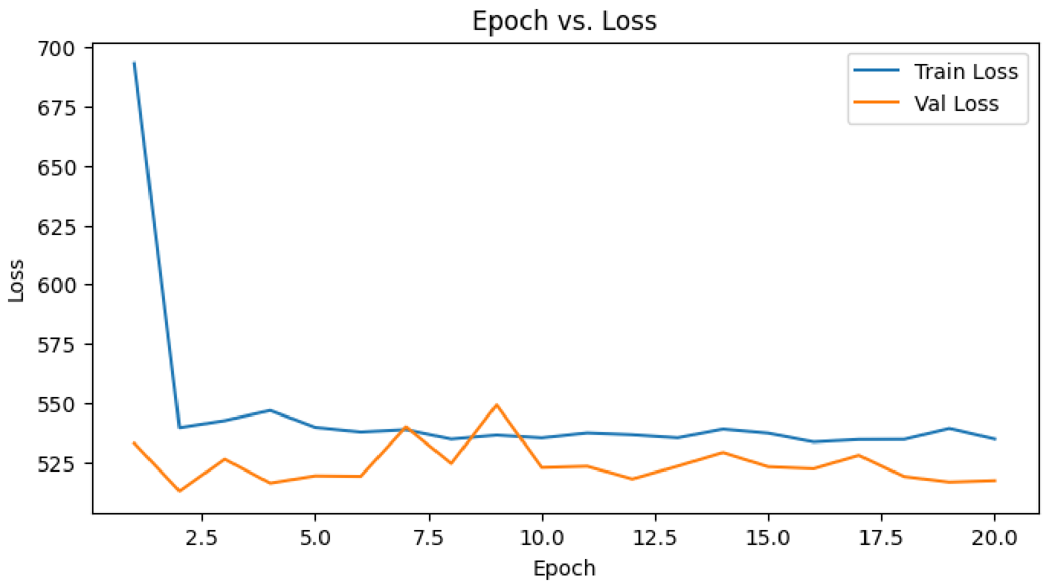

Figure 8: Epoch vs Loss

5 Results

5.1 Token Classification Performance

The training accuracy for the classification task was 62.20% and the validation accuracy achieved was 67.43%.

5.2 Score Prediction Results

The Hierarchical MoE Scorer achieved an overall RMSE of 2.125 on the validation set. Per-dimension RMSE values are listed in Table 4.

Table 3: Score Prediction RMSE by Dimension

Score Dimension	**RMSE**
Lead	2.821
Position	1.259
Claim	2.665
Counterclaim	1.419
Rebuttal	1.506
Evidence	2.874
Concluding Statement	1.444
Overall	2.125

Table 4: Score Prediction r^2 by Dimension

Score Dimension	**RMSE**
Lead	0.287
Position	0.798
Claim	0.837
Counterclaim	0.021
Rebuttal	0.127
Evidence	0.507
Concluding Statement	0.012
Overall	0.607

5.3 Error Analysis

The model most frequently confused *Claim* with *Counterclaim*, accounting for 12% of segmentation errors. In score prediction, the highest error occurred on the *Evidence* dimension (RMSE=2.874), indicating challenges in modeling opposing arguments. r^2 by Dimension achieved was 0.607.

5.4 Example

To illustrate our discourse-part classification and scoring, we show one complete example. Table 5 reports the model's predicted scores versus gold scores for each discourse segment, and the overall mean.

Classification

```
lead:         We, the people of Florida are concerned about the...
position:     I believe that the Electoral College should not be...
claim:        it lacks reasoning, it is unfair to voters | misloyalty...
rebuttal:
evidence:     In the article, The Indefensible Electoral Colleg... | When...
concluding:   The Electoral College has been used for several ye...
counterclaim:
```

Table 5: Predicted vs. actual scores for each discourse segment

Segment	**Predicted**	**Gold**
lead	31.365	36.000
position	25.287	22.000
claim	63.162	80.000
rebuttal	7.266	0.000
evidence	81.743	80.000
concluding	56.828	55.000
counterclaim	8.815	0.000
Overall Mean	39.209	39.000

6 Experiments

We ran two broad sets of experiments: one on essay segmentation (token-classification) and one on score prediction. In each case we compared recurrent architectures with static embeddings against our Transformer- and MoE-based approaches, using a fixed train/validation split.

6.1 Segmentation Experiments

First, we evaluated LSTM and GRU encoders with both Word2Vec and fastText embeddings on the public test sample. We report overall token-level accuracy:

As shown in Table 6, both LSTM and GRU with static embeddings performed poorly (under 32% accuracy), motivating our move to a Transformer-based tagger.

Table 6: Token-Classification Accuracy (%) for LSTM/GRU + Static Embeddings

Model	**Word2Vec**	**fastText**
LSTM	26.78	29.34
GRU	28.12	31.08
Transformer (ours)	**67.25**	

6.2 Score Prediction Experiments

Next, we compared plain (non-hierarchical) MoE versus our hierarchical MoE scorer, using the same Word2Vec and fastText embeddings described above. We measure score-prediction accuracy as the percentage of rubric dimensions predicted within ± 0.5 points of the ground truth:

Table 7: Score-Prediction RMSE for MoE Variants

Model	**Word2Vec**	**fastText**	**BERT**
Plain MoE	5.237	4.823	3.967
Hierarchical MoE (ours)	-	-	2.125

7 Contribution

- **Varnit Mittal (IMT2022025):** Contributed to the second part of the project, that is score prediction using BERT, which was used for embeddings, and Hierarchical MoE, which was used for prediction. Specifically, contributed in MoE in order decrease RMSE.
- **Aditya Priyadarshi (IMT2022075):** Contributed to the second part of the project. Specifically, contributed to BERT architecture and training of the model.
- **Ananthakrishna K (IMT2022086):** Contributed to the first part of the project, which was segment classification of the essay. Specifically, contributed data preprocessing and defining the transformer layers.
- **Vedant Mangrulkar (IMT2022519):** Contributed to the first part of the project, which was segment classification of the essay. Specifically, contributed to model training and hyper-parameter tuning.

References

[1] Feedback Prize - Evaluating Student Writing. 2021. https://www.kaggle.com/c/feedback-prize-2021/overview.

[2] NumPy. https://numpy.org/.

[3] pandas-dev/pandas: Pandas. Zenodo, 2020. https://pandas.pydata.org/.

[4] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Dubourg, V. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825-2830, 2011. https://scikit-learn.org/stable/.

[5] Hunter, J. D. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3), 90-95, 2007. https://matplotlib.org/.

[6] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32* (pp. 8024-8035), 2019. https://pytorch.org/.

[7] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A. M. Transformers: State-of-the-art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (pp. 38-45), 2020. https://huggingface.co/docs/transformers/index. *(Note: Used for get_linear_schedule_with_warmup)*

[8] da Costa-Luis, C., et al. tqdm: A Fast, Extensible Progress Bar for Python and CLI. Zenodo, 2019. https://tqdm.github.io/.

[9] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30* (pp. 5998-6008), 2017. https://arxiv.org/abs/1706.03762.

[10] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. https://arxiv.org/abs/1810.04805.

[11] Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv*

preprint arXiv:1701.06538, 2017. `https://arxiv.org/abs/1701.06538`. *(Note: Foundational concept for the MoEInteraction layer)*

[12] Loshchilov, I., & Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. `https://arxiv.org/abs/1711.05101`. *(Note: Basis for AdamW optimizer)*