Group 66 Final Project Report

NTHU 110060011 周冠霖
NTHU 110060032 蔡翔峯
NTHU 110062209 簡晟棋

1 Introduction

This project focuses on the Automated Essay Scoring 2.0 Competition on Kaggle, aiming to design models capable of accurately evaluating student essays.

Based on our survey, competing teams generally adopt one of two main approaches: fine-tuning language models like DeBERTa to fit the competition dataset or extracting meaningful feature representations directly from the raw text. Many teams also combine these strategies using model ensemble techniques to boost performance.

Our team focused on extracting meaningful features from the data to improve model performance. After analyzing the training samples for patterns, we proposed a cluster-based approach that combines feature engineering with deep semantic features from language models, aiming to maximize the accuracy and reliability of essay scoring.

2 Related works

In this section, we will first provide a brief introduction to the methods adopted by the teams we referred to, followed by an analysis of their strengths and weaknesses.

2.1 Some Extra Features - [CV 0.83]

This team extracted new features without relying on fine-tuning, Bag of Words, Tf-idf, or sentence embeddings. Specifically, they used Textstat features to evaluate readability, complexity, and grade level. Additionally, they extracted Named Entity Recognition (NER), Part-of-Speech (POS), and TAG features using spaCy. Sentiment analysis and various length and ratio features were computed using NLTK and basic functions. Furthermore, they derived features from feedback data to assess cohesion, syntax, vocabulary, phraseology, grammar, and conventions. Their approach achieved a private leaderboard score of 0.8185.

In our view, their approach has distinct strengths and weaknesses. It effectively utilizes diverse feature engineering techniques, offering a thorough analysis of text structure and characteristics. However, its limited ability to capture deep semantic information may hinder performance on more nuanced language tasks.

2.2 Quick start + LGBM

The other team we referred to named "Quick start + LGBM" adopted the following approaches in their solution. They extracted paragraph, sentence, and word-level features directly from raw texts. They also utilized features derived from Tf-idf and CountVector representations. Additionally, they incorporated advanced features from DeBERTa-v3-large. Their approach achieved a private leaderboard score of 0.83197.

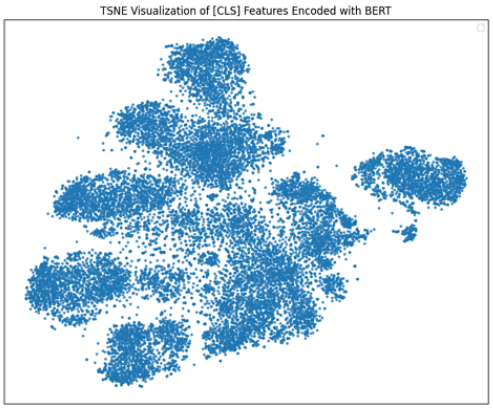
From our perspective, their approach has several strengths and drawbacks as well. It combines structural and statistical features, while also leveraging a language model to capture deep semantic and contextual insights from the text. However, a lot of the features are based on frequency or ratio, and the high-dimensional nature of these features may lead to the curse of dimensionality, potentially affecting the model's performance.

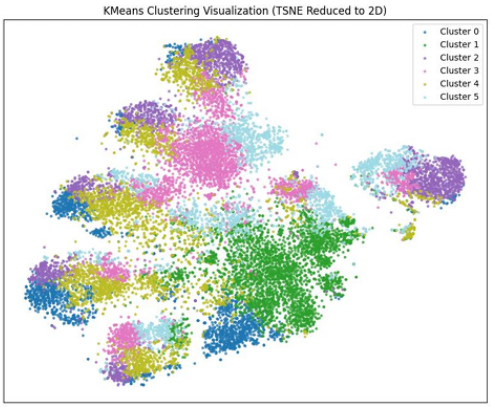
3 Method

In this section, we will first introduce our observations regarding the distribution of data features encoded by the language model, along with the hypotheses we have proposed. Next, we will present the method we proposed based on these hypotheses. Finally, we will briefly discuss how we referenced other teams' approaches to extract additional features from the text.

3.1 Our observation

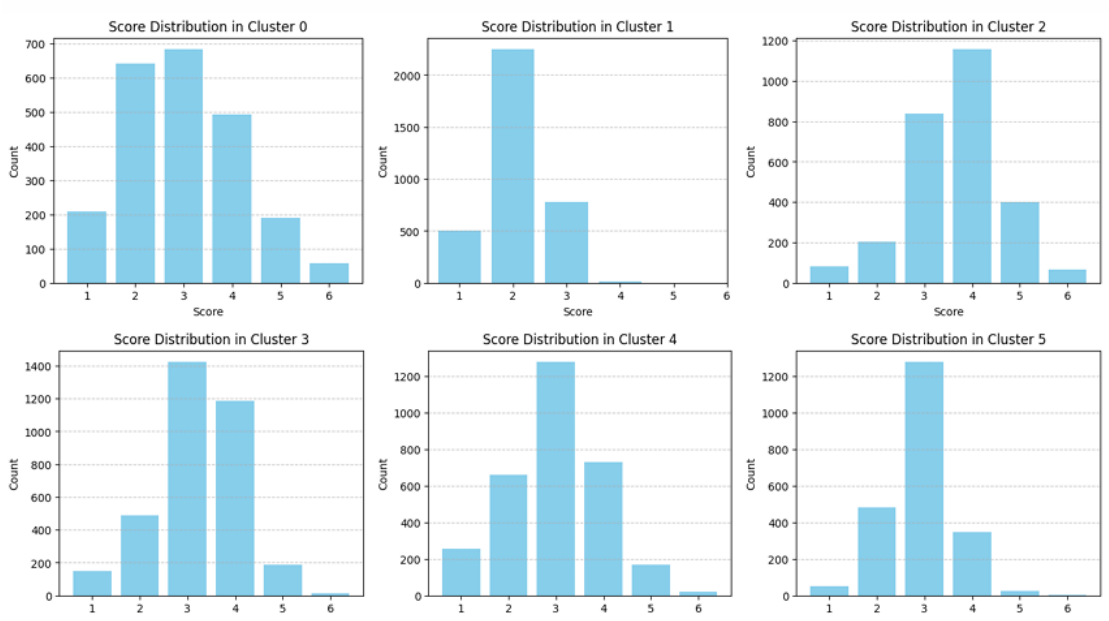
We began by observing the features of each data point after being encoded by the language model (LM). After performing dimensionality reduction and visualization using t-SNE, we noticed clear clusters in the encoded features. Based on this observation, we applied the k-means clustering algorithm to the LM-encoded features to prepare for further analysis. When visualizing the k-means clustering results with t-SNE, we again observed distinct clusters. The visualizations are shown below:





The clustering of the LLM-encoded features could be explained by two possible reasons: First, articles with similar scores tend to have similar features. Second, articles with similar content or characteristics tend to have similar features.

To verify which of these explanations best accounts for the observed clustering, we used the k-means clustering algorithm to divide all training data into 6 clusters, based on the score range of 1-6. We then observed the distribution of ground truth scores within each cluster and found that the score distribution varied across clusters. For example, cluster 1 had very few articles with scores above 4, while cluster 2 contained a noticeably higher proportion of articles with scores above 4 compared to the other clusters.



Based on our analysis and observations, we proposed two hypotheses. First, the features extracted from BERT captured deeper semantic characteristics such as themes or topics. Second, Articles with similar themes are likely to exhibit similar score distributions.

3.2 Main Methodology

Based on our hypotheses, we propose the following approach. First, we use clustering or classification to divide all the data into different groups. Then, we incorporate the information about these groups as additional features during training.

Moreover, as discussed in the 1st Place and 3rd Place Solutions, we found that some of the training samples come from the Persuade-Corpus 2.0 dataset, where each essay article is well annotated. Based on this, we propose two methods for performing clustering or classification, depending on whether the Persuade-Corpus 2.0 data is used.

Specifically:

- In k-Means-base-method, we first apply the k-means algorithm to group the training samples into 15 clusters.

```
n_clusters = 15
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
kmeans.fit(persuade_bert)
train_topic = kmeans.predict(train_bert) # train_bert: encoded features of training samples
test_topic = kmeans.predict(test_bert) # test_bert: encoded features of training samples

train_topic_one_hot = np.eye(n_clusters)[train_topic]
test_topic_one_hot = np.eye(n_clusters)[test_topic]
```

- In kNN-base-method, we first apply the kNN algorithm to annotate each training sample using the data from the corpus 2.0 dataset.

```
knn_clf = KNeighborsClassifier(n_neighbors=15)
knn_clf.fit(persuade_bert, persuade['prompt_name'])

encoder = OneHotEncoder(sparse=False)
encoded_data = encoder.fit(persuade['prompt_name'].values.reshape(-1, 1))

train_topic = knn_clf.predict(train_bert)
test_topic = knn_clf.predict(test_bert)

train_topic_one_hot = encoder.transform(train_topic.reshape(-1, 1))
test_topic_one_hot = encoder.transform(test_topic.reshape(-1, 1))
```

- Based on the clustering results, we assign each sample a label as additional feature.

Our approach enables the model to capture the score distributions of articles with similar semantic features, helping it learn to predict article scores more effectively and ultimately leading to better model performance.

3.3 Feature Engineering Based on the Teams We Referenced

In this section, we will briefly introduce how we referenced the approaches of other teams for feature engineering.

Readability Features.

Following the first team we referenced, we derive readability metrics such as Flesch Reading Ease and SMOG Index using the *textstat* library to quantify text complexity.

```
def textstat_features(text):
    features = {}
    features['flesch_reading_ease'] = textstat.flesch_reading_ease(text)
    features['flesch_kincaid_grade'] = textstat.flesch_kincaid_grade(text)
    features['smog_index'] = textstat.smog_index(text)
    features['coleman_liau_index'] = textstat.coleman_liau_index(text)
    features['automated_readability_index'] = textstat.automated_readability_index(text)
    features['dale_chall_readability_score'] = textstat.dale_chall_readability_score(text)
    features['difficult_words'] = textstat.difficult_words(text)
    features['linsear_write_formula'] = textstat.linsear_write_formula(text)
    features['gunning_fog'] = textstat.gunning_fog(text)
    features['text_standard'] = textstat.text_standard(text, float_output=True)
    features['spache_readability'] = textstat.spache_readability(text)
    features['mcalpine_eflaw'] = textstat.mcalpine_eflaw(text)
    features['reading_time'] = textstat.reading_time(text)
    features['syllable_count'] = textstat.syllable_count(text)
    features['lexicon_count'] = textstat.lexicon_count(text)
    features['monosyllabcount'] = textstat.monosyllabcount(text)

    return features
```

Linguistic Features.

Following the first team we referenced, we utilize spaCy to extract linguistic properties such as Named Entity Recognition (NER), Parts of Speech (POS) counts, tag counts, Sentence-level syntactic properties and other indicators.

```
def extract_linguistic_features(text):
    doc = nlp(text)
    features = {}

    # NER Features
    entity_counts = {"GPE": 0, "PERCENT": 0, "NORP": 0, "ORG": 0, "CARDINAL": 0, "MONEY": 0, "DATE": 0, "LOC": 0, "PERSON": 0,
                     "QUANTITY": 0, "EVENT": 0, "ORDINAL": 0, "WORK_OF_ART": 0, "LAW": 0, "PRODUCT": 0, "TIME": 0, "FAC": 0, "LANGUAGE": 0}

    for entity in doc.ents:
        entity_counts[entity.label_] += 1 if entity.label_ in entity_counts else 0
    features['NER_Features'] = entity_counts

    # POS Features
    pos_counts = {"ADJ": 0, "NOUN": 0, "VERB": 0, "SCONJ": 0, "PRON": 0, "PUNCT": 0, "DET": 0, "AUX": 0, "PART": 0, "ADP": 0, "SPACE": 0,
                  "CCONJ": 0, "PROPN": 0, "NUM": 0, "ADV": 0, "SYM": 0, "INTJ": 0, "X": 0}
    for token in doc:
        pos_counts[token.pos_] += 1 if token.pos_ in pos_counts else 0
    features['POS_Features'] = pos_counts

    # tag Features
    tags = {"RB": 0, "-RRB-": 0, "PRP$": 0, "JJ": 0, "TO": 0, "VBP": 0, "JJS": 0, "DT": 0, "''": 0, "UH": 0, "RBS": 0, "WRB": 0, ".": 0,
            "HYPH": 0, "XX": 0, "``": 0, "SYM": 0, "VB": 0, "VBN": 0, "WP": 0, "CC": 0, "LS": 0, "POS": 0, "NN": 0, ",": 0, "NNPS": 0,
            "RP": 0, ":": 0, "$": 0, "PDT": 0, "VBZ": 0, "VBD": 0, "JJR": 0, "-LRB-": 0, "IN": 0, "RBR": 0, "WDT": 0, "EX": 0, "MD": 0,
            "_SP": 0, "NNP": 0, "CD": 0, "VBG": 0, "NNS": 0, "PRP": 0}

    for token in doc:
        tags[token.tag_] += 1 if token.tag_ in tags else 0
    features['tag_Features'] = tags

    # tense features
    tenses = [i.morph.get("Tense") for i in doc]
    tenses = [i[0] for i in tenses if i]
    tense_counts = Counter(tenses)
    features['past_tense_ratio'] = tense_counts.get("Past", 0) / (tense_counts.get("Pres", 0) + tense_counts.get("Past", 0) + 1e-5)
    features['present_tense_ratio'] = tense_counts.get("Pres", 0) / (tense_counts.get("Pres", 0) + tense_counts.get("Past", 0) + 1e-5)

    # len features
    features['word_count'] = len(doc)
    features['sentence_count'] = len([sentence for sentence in doc.sents])
    features['words_per_sentence'] = features['word_count'] / features['sentence_count']
    features['std_words_per_sentence'] = np.std([len(sentence) for sentence in doc.sents])
    features['unique_words'] = len(set([token.text for token in doc]))
    features['lexical_diversity'] = features['unique_words'] / features['word_count']

    paragraph = text.split('\n\n')

    features['paragraph_count'] = len(paragraph)
    features['avg_chars_by_paragraph'] = np.mean([len(paragraph) for paragraph in paragraph])
    features['avg_words_by_paragraph'] = np.mean([len(nltk.word_tokenize(paragraph)) for paragraph in paragraph])
    features['avg_sentences_by_paragraph'] = np.mean([len(nltk.sent_tokenize(paragraph)) for paragraph in paragraph])

    # sentiment features
    analyzer = SentimentIntensityAnalyzer()
    sentences = nltk.sent_tokenize(text)

    compound_scores, negative_scores, positive_scores, neutral_scores = [], [], [], []
    for sentence in sentences:
        scores = analyzer.polarity_scores(sentence)
        compound_scores.append(scores['compound'])
        negative_scores.append(scores['neg'])
        positive_scores.append(scores['pos'])
        neutral_scores.append(scores['neu'])

    features["mean_compound"], features["std_compound"] = np.mean(compound_scores), np.std(compound_scores)
    features["mean_negative"], features["std_negative"] = np.mean(negative_scores), np.std(negative_scores)
    features["mean_positive"], features["std_positive"] = np.mean(positive_scores), np.std(positive_scores)
    features["mean_neutral"], features["std_neutral"] = np.mean(neutral_scores), np.std(neutral_scores)

    return features
```

Spellcheck.

Following the first team we referenced, we use SpellChecker to identify any spelling errors in the article.

```
def spell_check(text):
    spell = SpellChecker()
    words = nltk.word_tokenize(text)
    misspelled = spell.unknown(words)

    mispelled_count = len(misspelled)
    misspelled_ratio = mispelled_count / len(words)

    return mispelled_count, misspelled_ratio
```

Feedback Features.

Following the first team we referenced, we encode each article in the feedback dataset using a pretrained DeBERTa small model to generate embeddings. These embeddings are then utilized to train a new feedback model, which is subsequently applied to predict feedback pseudo-labels across the entire dataset.

```
feedback_path = '/kaggle/input/feedback-data'
sentpath = '/kaggle/input/sent-debsmall'
feedback_df = pd.read_csv(feedback_path+'/feedback_data.csv')

feed_embeds = []
merged_embeds = []
test_embeds = []

for i in range(5):
    model_path = sentpath + f'/deberta_small_trained/temp_fold{i}_checkpoints'
    word_embedding_model = models.Transformer(model_path, max_seq_length=1024)
    pooling_model = models.Pooling(word_embedding_model.get_word_embedding_dimension())
    model = SentenceTransformer(modules=[word_embedding_model, pooling_model])
    model.half()
    model = model.to(device)

    feed_custom_embeddings_train = model.encode(feedback_df.loc[:, 'full_text'].values, device=device,
                                                show_progress_bar=True, normalize_embeddings=True)
    feed_embeds.append(feed_custom_embeddings_train)
    merged_custom_embeddings = model.encode(train.loc[:, 'full_text'].values, device=device,
                                            show_progress_bar=True, normalize_embeddings=True)
    merged_embeds.append(merged_custom_embeddings)
    test_custom_embeddings = model.encode(test.loc[:, 'full_text'].values, device=device,
                                          show_progress_bar=True, normalize_embeddings=True)
    test_embeds.append(test_custom_embeddings)

feed_embeds = np.mean(feed_embeds, axis=0)
merged_embeds = np.mean(merged_embeds, axis=0)
test_embeds = np.mean(test_embeds, axis=0)

targets = ['cohesion', 'syntax', 'vocabulary', 'phraseology', 'grammar', 'conventions']

ridge = Ridge(alpha=1.0)
multioutputregressor = MultiOutputRegressor(ridge)
multioutputregressor.fit(feed_embeds, feedback_df.loc[:, targets])

feedback_predictions = multioutputregressor.predict(merged_embeds)
test_feedback_predictions = multioutputregressor.predict(test_embeds)
```

Paragraph Features.

Following the second referenced team, we calculate paragraph lengths, sentence counts, and word counts, then count paragraphs exceeding or falling below specific thresholds and extract global features like the overall distribution of these metrics.

```
def Paragraph_Preprocess(tmp):
    tmp = tmp.explode('paragraph')
    tmp = tmp.with_columns(pl.col('paragraph').map_elements(dataPreprocessing))
    tmp = tmp.with_columns(pl.col('paragraph').map_elements(lambda x: len(x)).alias("paragraph_len"))
    tmp = tmp.with_columns(pl.col('paragraph').map_elements(lambda x: len(x.split('.'))).alias("paragraph_sentence_cnt"),
                    pl.col('paragraph').map_elements(lambda x: len(x.split(' '))).alias("paragraph_word_cnt"),)
    return tmp

paragraph_fea = ['paragraph_len','paragraph_sentence_cnt','paragraph_word_cnt']
def Paragraph_Eng(tmp):
    tmp = Paragraph_Preprocess(tmp)
    aggs = [
        *[pl.col('paragraph').filter(pl.col('paragraph_len') >= i).count().alias(f"paragraph_{i}_cnt") for i in \
                [50,75,100,125,150,175,200,250,300,350,400,500,600,700] ],
        *[pl.col('paragraph').filter(pl.col('paragraph_len') <= i).count().alias(f"paragraph_{i}_cnt") for i in [25,49]],
        *[pl.col(fea).max().alias(f"{fea}_max") for fea in paragraph_fea],
        *[pl.col(fea).mean().alias(f"{fea}_mean") for fea in paragraph_fea],
        *[pl.col(fea).min().alias(f"{fea}_min") for fea in paragraph_fea],
        *[pl.col(fea).first().alias(f"{fea}_first") for fea in paragraph_fea],
        *[pl.col(fea).last().alias(f"{fea}_last") for fea in paragraph_fea],
        ]
    df = train_tmp.group_by(['essay_id'], maintain_order=True).agg(aggs).sort("essay_id")
    df = df.to_pandas()
    return df
```

Sentence Features.

Following the second referenced team, we calculate sentence lengths, filter out data with sentences longer than 15 words, and count the number of words in each sentence. We then count sentences exceeding or falling below specific thresholds and extract global features, such as the overall distribution of these metrics.

```
def Sentence_Preprocess(tmp):
    tmp = tmp.with_columns(pl.col('full_text').map_elements(dataPreprocessing).str.split(by=".").alias("sentence"))
    tmp = tmp.explode('sentence')
    tmp = tmp.with_columns(pl.col('sentence').map_elements(lambda x: len(x)).alias("sentence_len"))
    tmp = tmp.filter(pl.col('sentence_len')>=15)
    tmp = tmp.with_columns(pl.col('sentence').map_elements(lambda x: len(x.split(' '))).alias("sentence_word_cnt"))
    return tmp

sentence_fea = ['sentence_len','sentence_word_cnt']
def Sentence_Eng(tmp):
    tmp = Sentence_Preprocess(tmp)
    aggs = [
        *[pl.col('sentence').filter(pl.col('sentence_len') >= i).count().alias(f"sentence_{i}_cnt") for i in [15,50,100,150,200,250,300] ],
        *[pl.col(fea).max().alias(f"{fea}_max") for fea in sentence_fea],
        *[pl.col(fea).mean().alias(f"{fea}_mean") for fea in sentence_fea],
        *[pl.col(fea).min().alias(f"{fea}_min") for fea in sentence_fea],
        *[pl.col(fea).first().alias(f"{fea}_first") for fea in sentence_fea],
        *[pl.col(fea).last().alias(f"{fea}_last") for fea in sentence_fea],
        ]
    df = train_tmp.group_by(['essay_id'], maintain_order=True).agg(aggs).sort("essay_id")
    df = df.to_pandas()
    return df
```

Word Features.

Following the second referenced team, we calculate the length of each word and remove data with a word length of 0. We then count the number of words exceeding specific length thresholds and extract global features, such as the overall distribution of these metrics.

```
def Word_Preprocess(tmp):
    tmp = tmp.with_columns(pl.col('full_text').map_elements(dataPreprocessing).str.split(by=" ").alias("word"))
    tmp = tmp.explode('word')
    tmp = tmp.with_columns(pl.col('word').map_elements(lambda x: len(x)).alias("word_len"))
    tmp = tmp.filter(pl.col('word_len')!=0)
    return tmp

def Word_Eng(train_tmp):
    tmp = Word_Preprocess(tmp)
    aggs = [
        *[pl.col('word').filter(pl.col('word_len') >= i+1).count().alias(f"word_{i+1}_cnt") for i in range(15) ],
        pl.col('word_len').max().alias(f"word_len_max"),
        pl.col('word_len').mean().alias(f"word_len_mean"),
        pl.col('word_len').std().alias(f"word_len_std"),
        pl.col('word_len').quantile(0.25).alias(f"word_len_q1"),
        pl.col('word_len').quantile(0.50).alias(f"word_len_q2"),
        pl.col('word_len').quantile(0.75).alias(f"word_len_q3"),
        ]
    df = train_tmp.group_by(['essay_id'], maintain_order=True).agg(aggs).sort("essay_id")
    df = df.to_pandas()
    return df
```

Prediction from Language Models.

Following the second referenced team, we incorporate the average prediction scores obtained from their fine-tuned language models as one of our input features.

```
predictions = []
for model in models:
    model = AutoModelForSequenceClassification.from_pretrained(model)
    trainer = Trainer(
        model=model,
        args=args,
        data_collator=DataCollatorWithPadding(tokenizer),
        tokenizer=tokenizer
    )
    preds = trainer.predict(ds).predictions
    predictions.append(softmax(preds, axis=-1))

predicted_score = 0.
for p in predictions:
    predicted_score += p
```

3.3 Training and Inference Settings

We used LightGBM with a learning rate of 0.01 and 10,000 estimators as the primary regression model, training with stratified 15-fold validation for robust performance. During inference, predicted scores were converted into discrete bins using optimized thresholds to align with the competition's scoring criteria.

4 Result

4-1. Main Results

We conducted experiments using two different pretrained models: DeBERTa-v3-large and BERT-base-uncased. The table below records the performance of our method.

Method	Private score
Some Extra Features -[CV 0.83]	0.8185
Quick start + LGBM	0.8319
Ours-kNN w/ DeBERTa-v3-large	0.8322
Ours-kNN w/ BERT-base-uncased	0.8330
Ours-kMeans w/ DeBERTa-v3-large	0.8319
Ours-kMeans w/ BERT-base-uncased	**0.8333**

Our method consistently matches or outperforms the referenced team's approach, regardless of the chosen backbone or algorithm. Notably, kMeans with BERT-base-uncased achieves the highest private score (0.8333), slightly surpassing other combinations. kNN with DeBERTa-v3-large follows closely with a score of 0.8322. These results highlight the robustness of our approach across pretrained models and clustering methods, outperforming baselines like Some Extra Features and Quick start + LGBM. This demonstrates the adaptability and strong potential of our method for real-world applications.

4-2. Ablation Study on k Value in KNN Algorithm

We also conducted ablation experiments on the k value in the KNN algorithm. The table below presents the performance results of our method.

Method	k	Private Score
Ours-kNNw/ DeBERTa-v3-large	5	0.8322
Ours-kMeans w/ DeBERTa-v3-large	15	0.8307
Ours-kMeans w/ BERT-base-uncased	5	0.8326
Ours-kMeans w/ BERT-base-uncased	**15**	**0.8330**

As shown in the table, we found that larger k provides stronger performance. Specifically, kMeans with BERT-base-uncased (k = 15) achieves the highest private score of 0.8330, outperforming the smaller k value of 5 (score: 0.8326). A similar trend is observed with DeBERTa-v3-large, where k = 15 yields competitive results. We believe this is because larger k can mitigate the effects of noise and outliers by averaging over a broader set of neighbors, leading to more stable and accurate predictions. This highlights the importance of tuning k for optimal performance and further underscores the robustness of our method across different configurations.

4-3. Analysis

In this section, we will compare and analyze our method with those used by the reference teams.

As previously mentioned, we believe that the approach taken by the first team (Some Extra Features -[CV 0.83]) is unable to capture deeper semantic information. In contrast, our method integrates the features used by the second team (Quick start + LGBM) and incorporates information processed through embeddings encoded by an LLM. This allows the model to grasp the overall semantic content of the text.

Regarding the second team, while their method cannot capture structural and grammatical information as effectively as the first team, we previously pointed out that their biggest issue lies in the potential occurrence of the curse of dimensionality. We believe that Tf-idf and CountVector can implicitly capture thematic and content-related information from the text. However, our approach allows the model to extract such information with lower-dimensional features, effectively addressing the curse of dimensionality. Additionally, we have incorporated the features extracted by the first team to compensate for the limitations of the second team's method, which struggles to capture textual structure, grammar, and related aspects.

5 Conclusion and Limitation

Conclusion.

In this project, our team applied a cluster-based approach that combines advanced feature engineering with deep semantic features from language models. By extracting meaningful patterns from the training data, we improved model performance and enhanced the accuracy of automated essay scoring. This hybrid method leveraged the strengths of both traditional feature extraction and LM embeddings, achieving a competitive private score of 0.8333. The results demonstrate the effectiveness of clustering and feature engineering, highlighting potential for further advancements in automated text evaluation.

Limitation.

Although we achieved a competitive score in this Kaggle competition, our method has several limitations. First, our approach heavily depends on clustering results, which can lead to poor performance if the clustering is ineffective. Additionally, we assume that the test samples follow the same clustering patterns as the training data. If new test cases involve articles not represented in the training set, our approach may fail.

For future work, exploring hybrid models and alternative clustering techniques for unseen topics could help address these limitations.