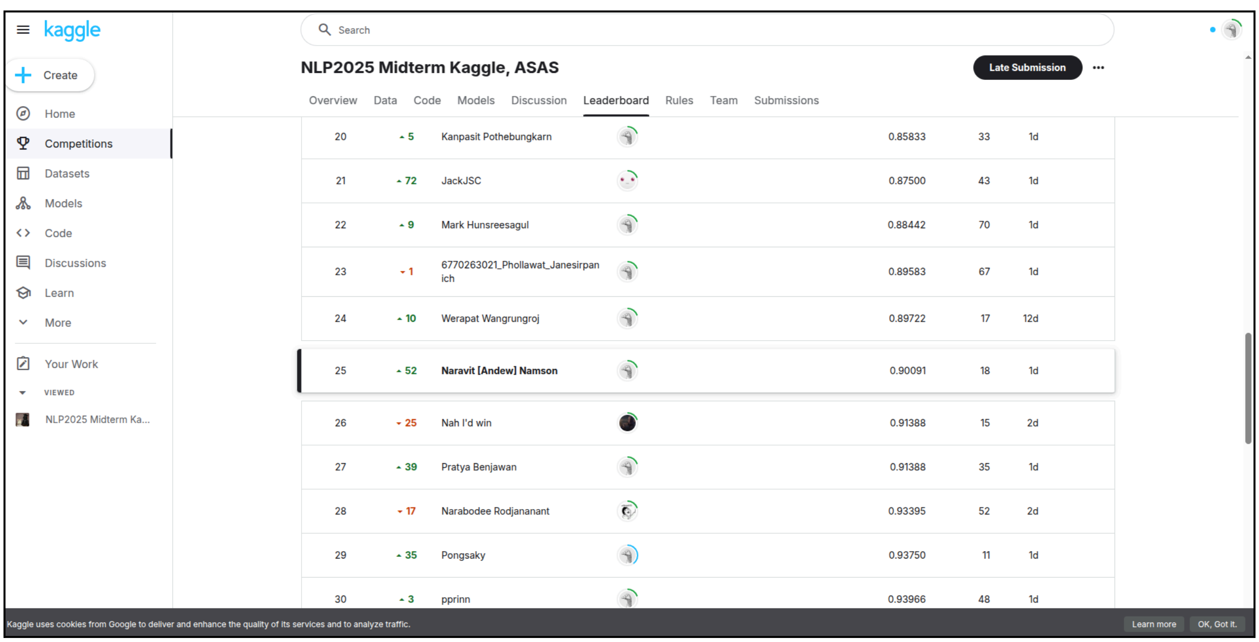
NLP Midterm Report

- I use **WangchanBERTa** as base model (model_BERT.ipynb)
- **Fine-tune model separately** for each question set
- **Input as tokenized answer** of student
- **Output as single real number** as predicted score
- Use **cosine** learning rate
- Use manual **cyclical learning rate** scheduling
- Use **all train data** as **training** set
- **Sampling each score equally** as **validation** set



Full Description

1. Model description

```
model_name = "airesearch/wangchanberta-base-att-spm-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
✓ 0.9s
```

```
model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=1)
model = model.to("cuda")
✓ 0.9s
```

I use **WangchanBERTa** as the base model for fine-tuning. My model takes an input (details will be discussed in the next section) that is **tokenized using its tokenizer**, ensuring the token length does not exceed **512 tokens**. The model then outputs a **single real number**, representing the predicted score for that input.

2. Methodology

- For each question set (Q1-Q4), I **fine-tune models separately**. Given the constraint that BERT only accepts a maximum token length of 512, I assume that the model only needs to learn patterns in each question to achieve specific scores, rather than understanding the question itself. Since some answers exceed 512 tokens, excluding the question from the input helps save space for the answer, allowing the model to focus on learning answer patterns effectively.

Select Question Set

```
question_set = 4

train_texts, train_labels, val_texts, val_labels, test_texts, test_ids = read_dataset(question_set)

train_dataset = Dataset.from_dict({"text": train_texts, "label": train_labels})
val_dataset = Dataset.from_dict({"text": val_texts, "label": val_labels})
test_dataset = Dataset.from_dict({"text": test_texts, "ID": test_ids})

train_dataset = train_dataset.map(tokenize_function, batched=True)
val_dataset = val_dataset.map(tokenize_function, batched=True)
test_dataset = test_dataset.map(tokenize_function, batched=True)
```

- I use a **cosine learning rate** because, based on my experiments, it leads to a more stable minima compared to other learning rate schedulers.
- I use a **cyclical learning rate** to help the model escape local minima and reach better minima. From my experiments, a long cycle tends to get stuck in a local minimum, but frequent high learning rate pulses push the model toward better minima.

```
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    save_strategy="no",
    learning_rate=1e-4,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    num_train_epochs=20, # 10 -> 20 -> 20
    run_name="run",
    metric_for_best_model="eval_loss",
    lr_scheduler_type="cosine",
)
```

- I **use all of the training data** to fine-tune the model and **sample each score equally for validation**, rather than splitting the data into separate training and testing sets. The reason is that I assume the dataset is too small, and through experimentation, I found that using all available data for training achieves better results.

```
def read_dataset(qset):
    train_df = pd.read_csv(f"./dataset/processed/train_Q{qset}.csv")
    test_df = pd.read_csv(f"./dataset/processed/test_Q{qset}.csv")
    val_df = pd.read_csv(f"./dataset/processed/valid_Q{qset}.csv")

    train_texts = train_df['answer'].apply(clean_text)
    train_labels = train_df['score'].astype(np.float32)
    val_texts = val_df['answer'].apply(clean_text)
    val_labels = val_df['score'].astype(np.float32)
    test_texts = test_df['answer'].apply(clean_text)
    test_ids = test_df['ID']

    train_texts = pd.concat([train_texts, val_texts])
    train_labels = pd.concat([train_labels, val_labels])

    return train_texts, train_labels, val_texts, val_labels, test_texts, test_ids
```

- To **achieve lower loss on the test set**, I train a separate model for each **Qx** and submit its predictions to the test set. If the new result for **Qx** is better than the previous one, I keep it; otherwise, I restore the previous prediction using a dedicated function. Additionally, I have a function to **merge predictions from all question sets** into the final output.

Export Prediction

```python
output_path = f"./output_tf/Q{question_set}.csv"
output = []
coll = ['ID', 'Score']
for i in range(len(test_dataset['ID'])):
    # print(f"{test_dataset['ID'][i]},{predictions.predictions[i][0]}")
    output.append([test_dataset['ID'][i], min(max(0, predictions.predictions[i][0]), 5)])

df = pd.DataFrame(output, columns=coll)
df.to_csv(output_path, index=False)
```

✓ 0.0s

```python
sets = [1, 2, 3, 4]
all_out = []
coll = ['ID', 'Score']
for s in sets:
    output_path = f"./output_tf/Q{s}.csv"
    df = pd.read_csv(output_path)
    sc, idd = df['Score'].tolist(), df['ID'].tolist()
    for i in range(len(sc)):
        all_out.append([idd[i], sc[i]])

all_out = sorted(all_out, key=lambda x: x[0])

df = pd.DataFrame(all_out, columns=coll)
df.to_csv('./output_tf/all.csv', index=False)
print(df)
```

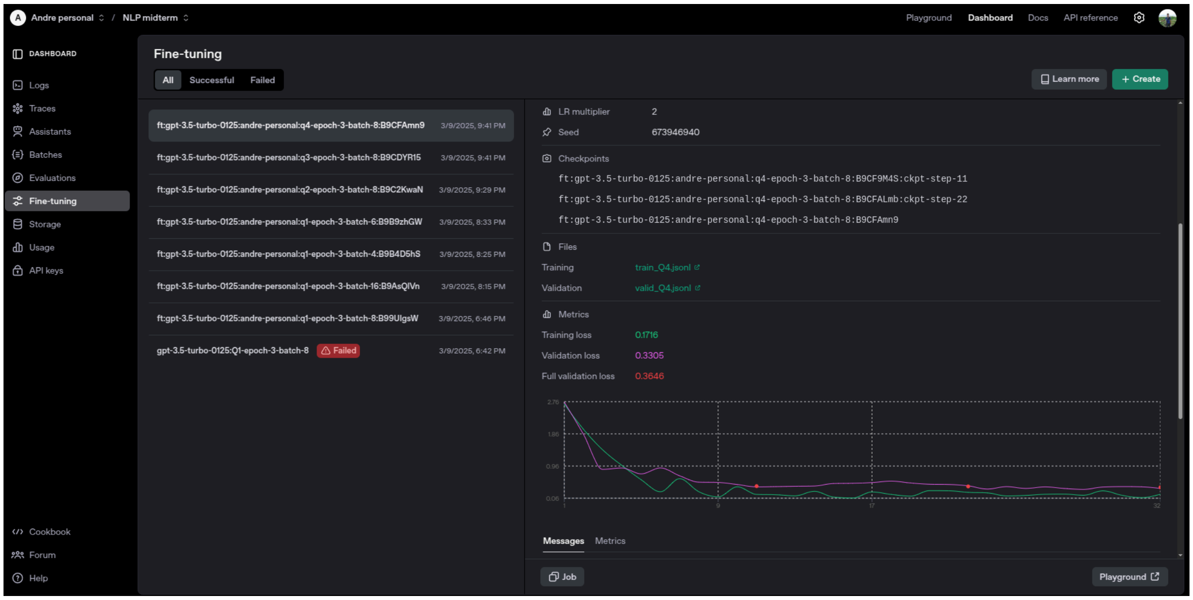
Restore

in case of new prediction for any question set worser than the previous one, we can restore the previous one.

```python
restore_file = "./output_tf/Q1.csv"
from_file = "./output_tf/all_old.csv"

df = pd.read_csv(restore_file)
df2 = pd.read_csv(from_file)

#iterate over ID
idx = 0
for i in range(len(df)):
    print(df['ID'][i])
    while df['ID'][i] != df2['ID'][idx]:
        idx += 1
    print(df2['ID'][idx])
    print(df['Score'][i], df2['Score'][idx])
    print()
    df.loc[i, 'Score'] = df2['Score'][idx]

df.to_csv(restore_file, index=False)
```

3. Previous method

1.) Fine-tune ChatGPT 3.5 Turbo (model_GPT.ipynb)




```
"messages": [
    {"role": "system", "content": "You are an AI grader that evaluates student answers on a scale from 0 to 5 in steps of 0.25."},
    {"role": "user", "content": f"Question: {row['question']}\nAnswer: {row['answer']}"},
    {"role": "assistant", "content": f"Score: {row['score']}"}
]
```

predictions_all_2.csv Complete · 8d ago	3.29166	4.73194	☐

2.) Fine-tune ConGen-XLMR (model_ConGen.ipynb)

```
class myModel(torch.nn.Module):
    def __init__(self, input_dim, output_dim, model_name, tokenizer):
        super(myModel, self).__init__()
        self.encoder = SentenceTransformer(model_name)
        self.linear0 = torch.nn.Linear(768, 768)
        self.linear = torch.nn.Linear(768, 256)
        self.linear2 = torch.nn.Linear(256, output_dim)
        self.relu = torch.nn.ReLU()
        self.sigmoid = torch.nn.Sigmoid()
        self.dropout = torch.nn.Dropout(0.1)
        self.tokenizer = tokenizer

    def forward(self, x):
        x = self.encoder.encode(tokenizer.batch_decode(x), convert_to_tensor=True, device='cuda')
        x = self.linear0(x)
        x = self.relu(x)
        x = self.linear(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.linear2(x)
        return x
```

all.csv Complete · 2d ago	3.28501	2.04828	☐