The first step in training an LLM to recognize tandem repeats is to show it what *is* and what *isn't* one. Our initial source for tandem repeat data comes from VNTRseek. This was supplemented with non-tandem repeat data from https://genome.ucsc.edu website, with an option selected to mask all repeat sequences as "N"s. The "N"s were removed, as used as the points to split the full DNA data into sequences. In order to avoid the model drawing conclusions based on a sequence's length, all sequences shorter than the shortest tandem repeat at 26 characters were removed. The sequences of tandem repeats were then labeled as "1", and non repeats labeled as "0".

The GROVER Tutorial served as a guide for getting started with training the LLM. The initial training was proceeding unreasonably slowly as it was running on the machine's CPU, so installing and enabling CUDA so as to utilize the GPU was necessary.

Initial attempts to modify and repurpose the GROVER tutorial code were unsuccessful, and so we attempted to set up the training from scratch. The full dataset proved too large for 32 gigabytes of RAM, so a smaller subset was used. After 21 hours the experiment was stopped early, as both the validation loss and, more surprisingly, the training loss seemed to be worsening after each epoch. The below training arguments were used:

```
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=10,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=10,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    # tokenizer missing
)

trainer.train()
```

To conserve SSD space, the working directory was changed to the computer's secondary HDD. While possibly unrelated, the project wouldn't run after this, and it took

quite some time to realize the PyTorch version needed to be updated (2.7.0+cu117 -> 2.7.0+cu118).

Running on mini dataset (<3,000 rows) with below args:

```
train_args = TrainingArguments(
                                seed = 42,
                                output_dir=".",
                                per_device_train_batch_size=16,
                                eval_strategy="epoch",
                                learning_rate=0.000001,
                                num_train_epochs=10,
                            )
trainer = transformers.Trainer(
                                model=model,
                                tokenizer=tokenizer,
                                compute_metrics=compute_metrics,
                                train_dataset=train_dataset,
                                eval_dataset=val_dataset,
                                args = train_args,
                            )
trainer.train()
```

Important differences are:

1. A hardcoded seed is provided
    a. ensures reproducibility of the training process by setting a fixed random seed for all random number generators used during training
2. per_device_eval_batch_size was removed
    a. This is the batch size used during evaluation.
    b. Can be larger because weights are not updated and gradients are not computer during eval
3. weight_decay was removed.
    a. Applies L2 regularization to the model's weight during training
    b. Helps prevent overfitting by penalizing large weights in the model
    c. During optimization, weight_decay adds a term to the loss function that is proportional to the square of the weights' magnitude
    d. This encourages the optimizer to keep the weights smaller, which can improve generalization
    e. L_new = L + (weight_decay * ||weights||^2) where ||weights||^2 is the sum of the squared weights
4. learning rate was added
    a. controls the step size at which the optimizer updates the model's weights during training
    b. determines how quickly or slowly the model learns from the training data
    c. Default value is 5e-5
    d. Can be adjusted during training by using a learning rate scheduler
5. tokenizer was added to Trainer() constructor call.

Epoch	Training Loss	Validation Loss	Accuracy	F1	Matthews Correlation	Precision	Recall
1	No log	0.041660	0.998325	0.499581	0.000000	0.499162	0.500000
2	No log	0.014446	0.998325	0.499581	0.000000	0.499162	0.500000
3	No log	0.012658	0.998325	0.499581	0.000000	0.499162	0.500000
4	No log	0.012288	0.998325	0.499581	0.000000	0.499162	0.500000
5	0.090900	0.012193	0.998325	0.499581	0.000000	0.499162	0.500000
6	0.090900	0.012181	0.998325	0.499581	0.000000	0.499162	0.500000
7	0.090900	0.012209	0.998325	0.499581	0.000000	0.499162	0.500000
8	0.090900	0.012241	0.998325	0.499581	0.000000	0.499162	0.500000
9	0.011300	0.012269	0.998325	0.499581	0.000000	0.499162	0.500000
10	0.011300	0.012275	0.998325	0.499581	0.000000	0.499162	0.500000

Validation loss seems reasonable, but the metrics are the same between each epoch, which seems incorrect. Furthermore, the lack of Training Loss data for the first 4 epochs is strange.

Per_device_train_batch_size

- Single subset of data processed together in forward and backward pass through model
- Small Batch Size: More updates per epoch, but noisier gradients.
- Large Batch Size: Smoother gradients, but requires more memory and may generalize less effectively.

While trying to fix the metrics, I noticed that I was using the parameter “eval_strategy”, but online resources were mentioning “evaluation_strategy.” Attempts to use that parameter instead through an error. Following this rabbit hole, I found that my version of transformers installed was a fork, and a later version (4.51.3) than the one from HuggingFace (4.38.2). This worked as far as the parameters go, but training attempts now caused “TypeError: Accelerator.__init__() got an unexpected keyword argument 'dispatch_batches'”. This ended up being due to a version mismatch between the transformer and accelerate packages.

At this point Visual Studio Code began showing visual bugs, possibly related to the tdqm package, where the progress bar during training was no longer being shown. After significant time was invested trying to resolve this, the choice was made to switch to using Google Colab.

Google Colab's environment was more stable, and the experiment above was repeated. Now the metrics correctly were updated between each epoch, despite no code changes:

Epoch	Training Loss	Validation Loss	Accuracy	F1	Matthews Correlation	Precision	Recall
1	No log	0.321385	0.887500	0.887246	0.793414	0.900639	0.892813
2	No log	0.210015	0.950000	0.949969	0.901651	0.950149	0.951503
3	No log	0.155432	0.962500	0.962447	0.925486	0.962135	0.963352
4	No log	0.123495	0.970000	0.969939	0.940073	0.969613	0.970461
5	No log	0.103376	0.972500	0.972438	0.944986	0.972155	0.972830
6	No log	0.093591	0.972500	0.972438	0.944986	0.972155	0.972830
7	0.190500	0.087923	0.972500	0.972438	0.944986	0.972155	0.972830
8	0.190500	0.084361	0.977500	0.977438	0.954888	0.977318	0.977570
9	0.190500	0.083070	0.977500	0.977438	0.954888	0.977318	0.977570
10	0.190500	0.082431	0.977500	0.977438	0.954888	0.977318	0.977570

```
TrainOutput(global_step=750, training_loss=0.14822403717041016, metrics={'train_runtime': 1195.8649, 'train_samples_per_second': 10.035, 'train_steps_per_second': 0.627, 'total_flos': 3157332664320000.0, 'train_loss': 0.14822403717041016, 'epoch': 10.0})
```

While these results seemed promising, working in a web browser posed its own set of issues. Without direct access to a local drive, the only way to provide data was to map a Google drive to the environment. Furthermore, leaving the webpage idle for a significant amount of time would result in the session timing out, removing all variables and requiring the Google drive to be remounted. Because of this, the decision was made to revisit the approach to the project, and move from working in Python notebooks to a standard Python project.

Most of the control of the model would be handled by an instance of the TandemTrainer class, while some of the more external code (e.g. using CUDA, building dataset, etc.) from the ipynb would exist outside of the class.

An object of the TandemTrainer class is instantiated with a model, tokenizer, and dataset. At the start, the TandemTrainer class had only two methods: preprocess, which handled splitting the dataset and tokenizing the data, and train, which would run the experiment. With a fairly lean main method, the experiment was able to run:

```
def main():
    set_cuda()
```

```
    tokenizer =
transformers.AutoTokenizer.from_pretrained("PoetschLab/GROVER")
    model =
transformers.AutoModelForSequenceClassification.from_pretrained("Poets
chLab/GROVER")

    try:
        dataset = pd.read_csv("dna_dataset.csv")
    except FileNotFoundError:
        non_repeats = get_non_repeats("output.txt")
        repeats = get_repeats("VNTRseek_NA19240_ERX283215.vcf")
        dataset = make_dataset(non_repeats, repeats, max_size=3000)

    tt = TandemTrainer(model=model, tokenizer=tokenizer,
dataset=dataset)
    tt.preprocess()
    tt.train()
```

The next step at this point was to allow for a model to be saved, reloaded, and used for predictions. To this end, the predict, save_model, and load_trained_model methods were added to the TandemTrainer class. The predict method initially failed, as the model had been trained on the GPU, while predictions were defaulting to the CPU:

> RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu! (when checking argument for argument index in method wrapper_CUDA__index_select)

Slight modifications needed to be made to ensure that predictions would be run on the same device the model was trained on. With this completed, the model could be trained once and used for predictions.