REPORT

DNA CLASSIFICATION

Understanding Dataset Structure and Behavior

I took some time to really dig into our dataset, trying to understand how it's set up and how it behaves. I looked at all the different pieces of data we have, trying to see how they fit together and if there are any interesting patterns or things that could affect our analysis.

Experimenting with Various Models, Including LLM and Machine Learning, and Crafting a Deep Learning Model

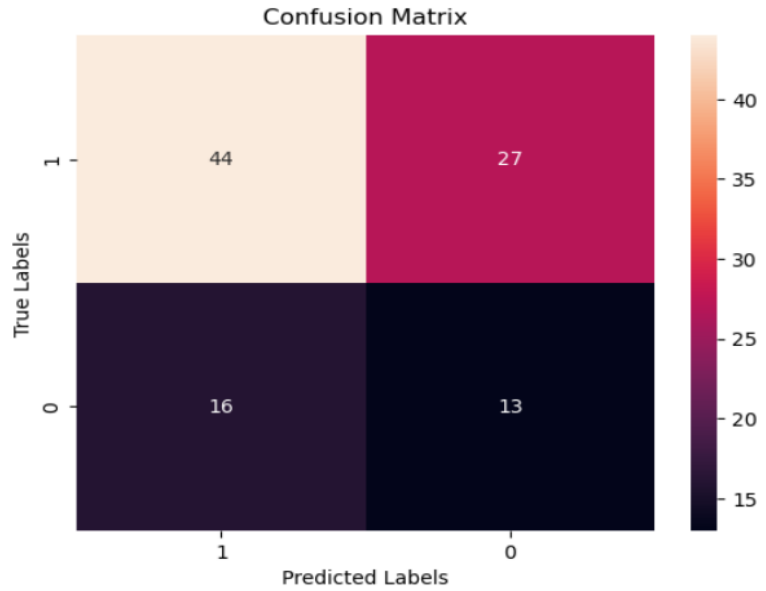
I tried out different models to see which one works best. I paid special attention to Large Language Models (LLM). These models are really good at handling complex connections between different pieces of data. I wanted to see if they could give us more detailed insights for our project.

Implementation

When implementing our model, the first step was sequence vectorization.

1. **CountVectorizer**
2. **TfidfVectorizer**
3. **Text Vectorizer**: When using the text vectorizer, we needed to specify two important parameters: the maximum number of tokens and the input length.
4. **Pre-trained Tokenizer**: I utilized the DNABert tokenizer and applied it on a deep learning model.

Results




```
accuracy = 0.570
precision = 0.615
recall = 0.570
f1 = 0.586
```

```
score :0.7
Prediction:[1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1]
```

```
{'accuracy': 62.0,
 'precision': 0.6276858345021037,
 'recall': 0.62,
 'f1': 0.6236190476190475}
```

```
[30]: loaded_model.evaluate(test_dataset)

      4/4 [==============================] - 11s 3s/step - loss: 0.6588 - accuracy: 0.6400
[30]: [0.6588319540023804, 0.6399999856948853]
```

Exploration of Dataset Structures and Model Training

Experimenting with different Dataset Structures

After receiving the updated training and testing dataset, which was already smaller and preprocessed, I experimented with additional preprocessing techniques to further reduce the size of the sequences and optimize memory computation during training.

This summarizes the preprocessing steps I undertook:

- **Use full data:** Initially, I worked with all sequences without making any alterations to store the results comprehensively.
- **Take random sequences:** I selected random sequences for each individual to reduce their number. For example, if these are the original sequences of a specific DNA: "**ATCG ATCG ATCG ATCG TTCG TTCG TTCG GGGG GGGG CCCC CCCC AAAA …**", after selecting 5 random sequences, the new DNA would be: "**ATCG TTCG CCCC GGGG TTCG**".
- **Take the most frequent sequences:** In this technique, I exclusively considered sequences that were more frequent for each DNA. Using the previous example, if we only select sequences repeated more than twice, the new DNA sequences would be: "**ATCG TTCG**". Alternatively, sequences could be selected while maintaining their frequencies, such as: "**ATCG ATCG ATCG ATCG TTCG TTCG TTCG** ".
- **Reduce sequence number in each DNA:** let's keep the previous example, this would be our new DNA: "**ATCG TTCG GGGG CCCC AAAA …**"
- **Caracter embedding:** Here I transformed my data to look like this "**A T C G T T C G G G G G C C C C A A A A …**" in this technique I reduced the token numbers in this case it's only 4.

Model Creation

After creating all the data sets, I have fitted each one of them with different models, but am getting only two results:

```
Epoch 1/5
26/26 [==============================] - 272s 10s/step - loss: 0.6840 - accuracy: 0.6034 - val_loss: 0.6394 - val_accuracy: 0.7100
Epoch 2/5
26/26 [==============================] - 264s 10s/step - loss: 0.6555 - accuracy: 0.6406 - val_loss: 0.6212 - val_accuracy: 0.7100
Epoch 3/5
26/26 [==============================] - 263s 10s/step - loss: 0.6517 - accuracy: 0.6406 - val_loss: 0.6312 - val_accuracy: 0.7100
Epoch 4/5
26/26 [==============================] - 263s 10s/step - loss: 0.6470 - accuracy: 0.6406 - val_loss: 0.6240 - val_accuracy: 0.7100
Epoch 5/5
26/26 [==============================] - 263s 10s/step - loss: 0.6431 - accuracy: 0.6406 - val_loss: 0.6202 - val_accuracy: 0.7100
```

As we can see we have an under fitting

```
[63]: history_model_1 = model_1.fit(train_dataset,
                                    epochs=15,
                                    validation_data=(test_dataset))
```

```
Epoch 1/15
104/104 [==============================] - 26s 237ms/step - loss: 0.6640 - accuracy: 0.6412 - val_loss: 0.6351 - val_accuracy: 0.7100
Epoch 2/15
104/104 [==============================] - 24s 231ms/step - loss: 0.6524 - accuracy: 0.6412 - val_loss: 0.6098 - val_accuracy: 0.7100
Epoch 3/15
104/104 [==============================] - 24s 230ms/step - loss: 0.6400 - accuracy: 0.6418 - val_loss: 0.6120 - val_accuracy: 0.7100
Epoch 4/15
104/104 [==============================] - 24s 232ms/step - loss: 0.5894 - accuracy: 0.6833 - val_loss: 0.5713 - val_accuracy: 0.6800
Epoch 5/15
104/104 [==============================] - 24s 230ms/step - loss: 0.4845 - accuracy: 0.7758 - val_loss: 0.6065 - val_accuracy: 0.6700
Epoch 6/15
104/104 [==============================] - 24s 230ms/step - loss: 0.3661 - accuracy: 0.8528 - val_loss: 0.6984 - val_accuracy: 0.6600
Epoch 7/15
104/104 [==============================] - 24s 229ms/step - loss: 0.2555 - accuracy: 0.9117 - val_loss: 0.8075 - val_accuracy: 0.6000
Epoch 8/15
104/104 [==============================] - 24s 230ms/step - loss: 0.1667 - accuracy: 0.9447 - val_loss: 0.9363 - val_accuracy: 0.6400
Epoch 9/15
104/104 [==============================] - 24s 229ms/step - loss: 0.1107 - accuracy: 0.9730 - val_loss: 1.0958 - val_accuracy: 0.6300
Epoch 10/15
104/104 [==============================] - 24s 229ms/step - loss: 0.0733 - accuracy: 0.9838 - val_loss: 1.1947 - val_accuracy: 0.6300
Epoch 11/15
104/104 [==============================] - 24s 230ms/step - loss: 0.0499 - accuracy: 0.9874 - val_loss: 1.3045 - val_accuracy: 0.6400
Epoch 12/15
104/104 [==============================] - 24s 231ms/step - loss: 0.0316 - accuracy: 0.9928 - val_loss: 1.4196 - val_accuracy: 0.6300
Epoch 13/15
104/104 [==============================] - 24s 229ms/step - loss: 0.0273 - accuracy: 0.9928 - val_loss: 1.4774 - val_accuracy: 0.6600
Epoch 14/15
104/104 [==============================] - 24s 229ms/step - loss: 0.0177 - accuracy: 0.9958 - val_loss: 1.5663 - val_accuracy: 0.6600
Epoch 15/15
104/104 [==============================] - 24s 230ms/step - loss: 0.0137 - accuracy: 0.9976 - val_loss: 1.6637 - val_accuracy: 0.5900
```

And here we have an overfitting

Exploration of OpenAI API and Model Selection

At first, I wanted to try ChatGPT-4, but it costs money for every query. So, I searched for free APIs and found ChatGPT-3, but it wasn't made for binary classification. Then, I spent time reading OpenAI's docs to see how they do classification. Turns out, they use a pre-trained model for encoding text, but you need a paid subscription to use it without limits.

Addressing Overfitting Issues

In our last talk, I mentioned we had a problem with overfitting. So, I did some tests to find out why. Turns out, the issue is with the data itself. The DNA samples with different labels are too similar, causing our problem.

This figure shows the 10 most common sequences in DNAs with labels 1:

```python
label_1 = []
for sentence in label_1_:
    label_1.extend(sentence.split())  # Tokenize and convert to lowercase

label_1_counts = Counter(label_1)

first_ten_label_1 = dict(label_1_counts.most_common(10))

print("First ten items in label_1_counts:")
for key, value in first_ten_label_1.items():
    print(f"{key}: {value}")
```

```
First ten items in label_1_counts:
TAGCAGCACGTAAATATTGGCG: 748729
TGAGGTAGTAGGTTGTATAGTT: 636401
TGTAGTGTTTCCTACTTTATGGA: 577481
TTCAAGTAATTCAGGATAGGTT: 509619
TTCAAGTAATCCAGGATAGGCT: 460932
TGAGGTAGTAGATTGTATAGTT: 412932
TGAGGTAGTAGTTTGTACAGTT: 383488
TTAAGTGACGATAGCCTA: 362591
CAACGGAATCCCAAAAGCAGCT: 305426
TGTCAGTTTGTCAAATACCCCA: 285740
```

And this one shows the most 10 with labels 0:

```python
label_0 = []
for sentence in label_0_:
    label_0.extend(sentence.split())  # Tokenize and convert to lowercase

label_0_counts = Counter(label_0)

first_ten_label_0 = dict(label_0_counts.most_common(10))

print("First ten items in label_1_counts:")
for key, value in first_ten_label_0.items():
    print(f"{key}: {value}")
```

```
First ten items in label_1_counts:
TAGCAGCACGTAAATATTGGCG: 532464
TGTAGTGTTTCCTACTTTATGGA: 510965
TGAGGTAGTAGGTTGTATAGTT: 485987
TTAAGTGACGATAGCCTA: 475881
TTCAAGTAATTCAGGATAGGTT: 343890
TTCAAGTAATCCAGGATAGGCT: 318597
ACTCGATAGTGTGTTTGTTTGTT: 303750
GTTTGGCGGCCATAGCGAGT: 281569
TGAGGTAGTAGTTTGTACAGTT: 280542
TGAGGTAGTAGATTGTATAGTT: 274243
```

Upon reviewing these results, I discerned that the most prevalent sequences found in the DNAs labeled as 1 are also the most frequent in the DNAs labeled as 0. So, I had the idea to remove those sequences that are similar in both labels and keep only the different ones, and this was the result:

```
First ten items in label_1_counts:
GTAGCGTTGTTATTTTT: 4455
GAAGTTGTCGGGTACT: 3284
TCAGTATGAAAAGGCTCGA: 3026
AATGTCTAGAGAATGTAGC: 2450
AAGAAGAAGAAGTAAC: 2274
CAGTCATTGGAAGAACA: 2263
GTAGCGTTGTTATTTT: 2258
TAGGTAAAGCGTGTGGTA: 2191
GATGTTGGAAAGTCGCC: 2065
AGAAAGAAGGAAGACTCCGT: 1971
GTGTAGTAGGAGTAGACT: 1933
GTTAGGAAGTTGGATT: 1851
AACAACAAGAAGAAGAAGTAAC: 1591
TCAGTATGAAAAGGCTCG: 1558
AAAGAAGATTATGCTCGAT: 1538
AACAAGAAGAAGAAGTAAC: 1317
AAGGAATCGGAGTTCACA: 1263
AAATGAATCGTTTAATTGGTCGTA: 1226
TTTGAACGTAACGTGCAGATA: 1221
TCGACGGAGAAAGGCTATG: 1221
```

```
gctgagagtataggatt: 5160
agaaagaggtaagagttaatta: 4085
tatattcaagaaaagcgactatc: 2973
ttctggcggtggatgca: 2929
ataaatgggcttgtcgtgagaca: 2470
atgcaggggcctcgtgggtta: 2419
agaaagaggtaagagtta: 2380
tatattcaagaaaagcgact: 2091
tatattcaagaaaagcgactat: 1997
agaagaatatgcaacc: 1949
aagaagaggtctggtgcattca: 1906
taggaagtttcagcaaataaaac: 1892
atgcaggggcctcgtgggtt: 1781
ttggattgcgaaaact: 1775
attctggcggtggatgca: 1708
taggaagtttcagcaaataaaacg: 1692
tatattcaagaaaagcgactatcg: 1576
aaaatctgcggagtgc: 1531
tatattcaagaaaagcgacta: 1473
agaaagaggtaagagttaattatg: 1458
```

Addressing Label Similarity

After getting the new data, I used it to improve my model. But I faced overfitting again. So, I looked into the problem and found it was due to the data. The issue is that the DNAs with the same label don't have much in common. Each DNA has its own unique sequences without any shared patterns. So, I worked to fix this and make sure DNAs with the same label are more similar.

```
AGGAAG: 71812
AAAGAG: 65841
AAGAAG: 64550
AGAAGA: 64203
AAGAAA: 64187
GAAGAA: 63526
AAGAGG: 61997
GAAGAG: 60396
AAGAGT: 58285
AGAGGT: 58133
GAAAAG: 57727
AGAGTT: 57519
GAAAGA: 56406
GGAAGA: 53379
GGAAGT: 52645
TAAGAG: 52363
GAAGGT: 50718
TGAAGA: 49136
AAGAGA: 48244
GAAGTT: 48103
```

```
GAAGAA: 105188
AAGAAG: 99085
AGAAGA: 91497
GGAAGA: 70902
TGAAGA: 67655
AAAGAG: 66139
GAAGAT: 61752
GAAGAG: 60951
AGGAAG: 60938
TGGAAG: 56651
GAAAGA: 56349
AAAGAA: 55775
GTTGTT: 54598
AAGAGA: 54240
AAGGAA: 54165
AAAAGA: 52856
AAGAGT: 52237
AAGAAA: 51993
GAAGTT: 51649
AGAAGT: 50218
```

I observed that there are similar sequences present in both labels, but with varying frequencies. Specifically, the most frequent sequences in DNAs labeled as 0 are the least frequent in DNAs labeled as 1, and vice versa. However, directly applying these new data still resulted in overfitting. To address this, I needed to

find a way to effectively utilize the obtained results. That's when I came up with the idea of retaining only the frequent sequences shared between both labels.

The issue with this technique is that it can't be applied to the test data without prior knowledge of which samples are labeled as 1 and which ones are labeled as 0. I attempted to make the technique more general, but it didn't yield any results. Therefore, I decided to explore a new technique for now and revisit the label similarity approach later.

Exploration of Custom Transformer Creation

For the new approach, I figured it's better to make our own transformer instead of using trained LLM models because they have limits. So, I checked out some articles to learn how to make a custom transformer.

This part took from me too much time, so I will explain it in resumed way:

1. Learned how to create a Transformer and how it works.
2. Searched for new DNA dataset that is much smaller than our dataset to work with.
3. Create the encoder and fit it with the demo data.
4. Increase the accuracy of the model.
5. Create a new model that can fit with our own data.

In conclusion, the encoder performed well with the demo dataset. However, when attempting to apply it to our dataset, the recurring issue of kernel crashing emerged. Unfortunately, I can't get an instance capable of training the encoder with our dataset due to its high memory requirements.

So, I had to go further and look for new ideas to work on and go back to the encoder later.