Prepared by Lloyd W Jones III

Protein Disorder and Per-Residue Embeddings for Simple Neural Network Classification Project

Theoretical Downstream Process

April, 2025

Background Information

Current scenario:

- This Project Uses a premade embedded vector from **ESM-2**
  - the model behind **ESMfold**
- This vector contains what the model was **“thinking about”** while trying to predict the next amino acid in the sequence

Why?

- This project was done as a test case for the use of embedding vectors as an easier way to train and implement **“Downstream Tasks”**

Background Information

Implementation

- Generate a simple multi-layer neural network using the **tensorflow** library
    - Simple Prediction of structured vs disordered regions using the Neural Network as a simpler classifier
- Iterate and improve the model while keeping a history notating what was done
- Present the final/best model generated

Template Workflow

1. Import Your Libraries

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score
import tensorflow as tf
```

- No further Libraries should be necessary to run a project similar to this

2. Import Your Data

```
data = np.load('disprot_esm_embed_10188.npz')
# feature data
x_data = data['X']
# ground truth
y_data = data['y']
```

- This data was saved as a numpy array and as such has a different procedure for proper importing

3. Splitting the data into Training and Testing Data

```
x_train, x_test, y_train, y_test = train_test_split(
                                   x_data, y_data, test_size = 0.20,
                                   random_state = 42)
```

- This is a single line of code, formatting and readability prefernces is the reason for the unique formatting

Empty markdown cell, double-click or press enter to edit

Template Workflow

4. Setting up and Designing the Neural Network model

`nameOfModel = tf.keras.sequential()`

- This states the name of the model being designed
- The model is a simple linear neural network model by which the layers are sequentially ordered
    - This type of model contains no branching or skipped connections

`nameOfModel.add(tf.keras.layers.Dense(units = n, activation = 'type'))`

- Adds the initial and all following layers of the Neural Network

units = the number of nodes in a particalur layer of the nueral network

- Determines the capacity of the layer and should be relative to the number of features in the data set

activation = type of equation used to determine the output of a neuron

- Very important and should be chosen based on your desired output as each type has its own strengths and weaknesses

`nameOfModel.compile(loss = 'type')`

- Compiles the model to then be ran

The **"loss type"** you select is important:

- Changes what your final layer output should be
- How the model is being graded
- How you should Interpret the results for future alterations

Template Workflow

5. Testing the Model

1a. Fitting the model

```
nameOfModel.fit(x_train, ytrain)
```

- Fits the model to the training data
- Outputs a fitted model with a model loss metric
    - Was used during hyperparameter testing as a precheck for model quality

loss value > 1 indicates the model should be redesigned

- usually indicative of underfitting and not a complex enough model

loss value < 0.6 is ideal the closer to zero the better

- Be aware of over fitting

2a. Prediction Testing

```
predictionRaw = nameOfModel.predict(x_test)
```

- runs the prediction model on the test data

```
predictionProbability = predictionRaw.flatten()
```

- flattens matrix into a single dimension while perserving the values

```
prediction = predictionProbability.round()
```

- rounds the prediction to the nearest whole integer

Template Workflow

6. Prepping the Results for Crosstabulation testing

```
test_y = pd.DataFrame(y_test) # Reformating for input into DataFrame

performanceDataFrame = pd.DataFrame(
                       data = {
                               'Prediction':prediction,
                               'Prediction_Probability':predictionProbability,
                               'Ground_Truth':test_y[0]
                               })

performanceDataFrame['90_Probability'] = (performanceDataFrame['Prediction_Probability'] >= 0.90).astype(int)
performanceDataFrame['10_Probability'] = (performanceDataFrame['Prediction_Probability'] >= 0.10).astype(int)
```

7. Crosstabulation Table generation

```
crosstabpred = pd.crosstab(performanceDataFrame['Prediction'],
                           performanceDataFrame['Ground_Truth'], margins= False) # Base Prediction

crosstab10 = pd.crosstab(performanceDataFrame['10_Probability'],
                         performanceDataFrame['Ground_Truth'], margins= False) # Predictions with 10% Certainity from the model

crosstab90 = pd.crosstab(performanceDataFrame['90_Probability'],
                         performanceDataFrame['Ground_Truth'], margins= False) # Predictions with 90% Certainity from the model
```

First Complete Model

Failed initial Iteration

```
ogModel = tf.keras.Sequential()
ogModel.add(tf.keras.layers.Dense(units=25, input_shape=(1280,), activation='relu'))
```

- failure point over compression of features
    - makes the model inconsistent as depending on how it compresses 1280 features into 25 features it could have a model loss of 4.0 or 0.6

```
ogModel.add(tf.keras.layers.Dense(units=15, activation='sigmoid'))
ogModel.add(tf.keras.layers.Dense(units=15, activation='relu'))
ogModel.add(tf.keras.layers.Dense(units=1, activation='relu'))
```

- short comming of the model with the over compression at the first step having only 4 layers isn't enough to accurately capture the realtionships for prediction

First Complete Model Results

✓ 0.0s Python

Ground_Truth	0.0	1.0
Prediction		
0.0	1365	154
1.0	91	428

Ground_Truth	0.0	1.0
10_Probability		
0	697	10
1	759	572

Ground_Truth	0.0	1.0
90_Probability		
0	1456	383
1	0	199

```python
# AUC & Percision Calculations
falsePos, truePos, thresh = roc_curve(y_test, prediction)
areaUnderCurve = roc_auc_score(y_test, prediction)
precision = precision_score(y_test, prediction)
print(f'False Positive rate of {falsePos[1]:.4f} \nTrue Positive rate {truePos[1]:.4f}')
print(f'\nPrecision Score of {precision:.4f} \nArea Under Curve {areaUnderCurve:.4f}')
```

✓ 0.0s Python

```
False Positive rate of 0.0625
True Positive rate 0.7354

Precision Score of 0.8247
Area Under Curve 0.8364
```

Spaces: 8 Cell 17 of 30

[78] ✓ 0.0s Python

Ground_Truth	0.0	1.0
Prediction		
0.0	1456	582

Ground_Truth	0.0	1.0
10_Probability		
0	1456	582

Ground_Truth	0.0	1.0
90_Probability		
0	1456	582

```python
# AUC & Percision Calculations
falsePos, truePos, thresh = roc_curve(y_test, prediction)
areaUnderCurve = roc_auc_score(y_test, prediction)
precision = precision_score(y_test, prediction)
print(f'False Positive rate of {falsePos[1]:.4f} \nTrue Positive rate {truePos[1]:.4f}')
print(f'\nPrecision Score of {precision:.4f} \nArea Under Curve {areaUnderCurve:.4f}')
```

[79] ✓ 0.0s Python

```
False Positive rate of 1.0000
True Positive rate 1.0000

Precision Score of 0.0000
Area Under Curve 0.5000
/opt/anaconda3/lib/python3.12/site-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 c
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

Changes

- Increased The number of layers
- Increased the number of nodes in the introductory layers

Why?

- Fix the variablity in model quality and loss
- capture more relationships in the embedded vector to allow for better prediction statistics

Intermediate Complete Model

```
# initial number of features: 1280
myModel = tf.keras.Sequential()
# 3/4 compression
myModel.add(tf.keras.layers.Dense(units = 640, activation = 'relu'))
myModel.add(tf.keras.layers.Dense(units = 480, activation = 'relu'))
myModel.add(tf.keras.layers.Dense(units = 360, activation = 'relu'))
myModel.add(tf.keras.layers.Dense(units = 270, activation = 'sigmoid'))
# 2/3 compression
myModel.add(tf.keras.layers.Dense(units = 180, activation = 'relu'))
# 1/3 compression
myModel.add(tf.keras.layers.Dense(units = 60, activation = 'relu'))
myModel.add(tf.keras.layers.Dense(units = 20, activation = 'sigmoid'))
# 1/4 compression
myModel.add(tf.keras.layers.Dense(units = 5, activation = 'relu'))
# Output layer
myModel.add(tf.keras.layers.Dense(units = 1, activation = 'sigmoid'))
```

```
myModel.compile(loss='binary_crossentropy')
```

```
myModel.fit(x_train, y_train)
```

```
255/255 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 0.5326

<keras.src.callbacks.history.History at 0x168a47d70>
```

Intermediate Model Results

Ground_Truth Prediction	0.0	1.0
0.0	1426	205
1.0	30	377

Ground_Truth 10_Probability	0.0	1.0
0	1388	142
1	68	440

Ground_Truth 90_Probability	0.0	1.0
0	1453	478
1	3	104

General prediction has an improved false positive rate but has a increaseed false negative rate that

- could be improved with tuning or feature engineering of the data
  - the model seems to struggle with positive cases as it doesn't have a lot of positive cases to train on
- due to the source of the data being a embeded vector, Adjusting the weights would have to be done instead of feature engineering

10% probability has low false negatives but very high false positive rate

- too sensitive not enough specificity

90% probability has next to no false positives but a sizeable amount of false negatives

- need to increase sensitivity even at the cost of specificity

Ground_Truth Prediction	0.0	1.0
0.0	1330	148
1.0	126	434

Ground_Truth 10_Probability	0.0	1.0
0	596	42
1	860	540

Ground_Truth 90_Probability	0.0	1.0
0	1456	582

```
# AUC & Percision Calculations
falsePos, truePos, thresh = roc_curve(y_test, prediction)
areaUnderCurve = roc_auc_score(y_test, prediction)
precision = precision_score(y_test, prediction)
print(f'False Positive rate of {falsePos[1]:.4f} \nTrue Positive rate {truePos[1]:.4f}')
print(f'\nPrecision Score of {precision:.4f} \nArea Under Curve {areaUnderCurve:.4f}')
```

[103] ✓ 0.0s Python

```
False Positive rate of 0.0865
True Positive rate 0.7457

Precision Score of 0.7750
Area Under Curve 0.8296
```

Changes

- Density of nodes Decreased
- Decrease number of Layers
- Re-weight positive instance

Why?

- Fix the positive prediction struggle without altering the data
- More consistent loss and prediction statistics

Final Complete Model

```
# initial number of features: 1280
myModel = tf.keras.Sequential()
# 1/2 compression
myModel.add(tf.keras.layers.Dense(units = 640, activation = 'relu'))
# 200 units per compression
myModel.add(tf.keras.layers.Dense(units = 440, activation = 'relu'))
myModel.add(tf.keras.layers.Dense(units = 240, activation = 'sigmoid'))
myModel.add(tf.keras.layers.Dense(units = 40, activation = 'relu'))
# 1/2 compression rounded
myModel.add(tf.keras.layers.Dense(units = 20, activation = 'relu'))
myModel.add(tf.keras.layers.Dense(units = 10, activation = 'relu'))
# Output layer
myModel.add(tf.keras.layers.Dense(units = 1, activation = 'sigmoid'))
```

✓ 0.0s

```
myModel.compile(loss='binary_crossentropy')
```

✓ 0.0s

```
myModel.fit(x_train, y_train, class_weight={0 : 1.0, 1 : 1.55})
```

✓ 1.6s

```
255/255 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.5739

<keras.src.callbacks.history.History at 0x17ec24d10>
```

Final Model Results

Ground_Truth Prediction	0.0	1.0
0.0	1390	76
1.0	66	506

Ground_Truth 10_Probability	0.0	1.0
0	932	10
1	524	572

Ground_Truth 90_Probability	0.0	1.0
0	1446	250
1	10	332

General prediction has an improved false positive rate and a improved false negative rate

- This is the final model but further improvements in the output could be made with a larger vector or better tuning

10% probability has a very low false negatives but a very high false positive rate

- too sensitive not enough specificity

90% probability has next to no false positives but has too many false negatives to be useable

- need to increase sensitivity even at the cost of specificity

Ground_Truth Prediction	0.0	1.0
0.0	1387	100
1.0	69	482

Ground_Truth 10_Probability	0.0	1.0
0	1039	14
1	417	568

Ground_Truth 90_Probability	0.0	1.0
0	1440	225
1	16	357

```
# ROC Curve & AUC
falsePos, truePos, thresh = roc_curve(y_test, prediction)
areaUnderCurve = roc_auc_score(y_test, prediction)
precision = precision_score(y_test, prediction)
print(f'False Positive rate of {falsePos[1]:.4f} \nTrue Positive rate {truePos[1]:.4f}')
print(f'\nPrecision Score of {precision:.4f} \nArea Under Curve {areaUnderCurve:.4f}')
```

6] ✓ 0.0s Python

```
False Positive rate of 0.0474
True Positive rate 0.8282

Precision Score of 0.8748
Area Under Curve 0.8904
```

Conclusion

By Carefully Designing and iterating on a Neural Network Model it can be used in a basic classification downstream task using an embedded vector

Even with tuning and skewing positive cases for the weights the model still seemed to struggle with positive cases as is common with unbalanced datasets