CIS 678 Kaggle Challange 2

Predict multi-modal data by using information about relationships between features across samples in each modality

By Zack Peters

Introduction:

The multi-modal analysis known as CITE-seq, a single-cell sequencing assay, allows for the relationship between ribonucleic acid (RNA) expression and protein expression to be explored by measuring both simultaneously in single cells. Using CITE-seq, protein expression is measured using Antibody-Derived Tags (ADT) and RNA is quantified using single-cell RNA sequencing. The ability to predict what proteins will be expressed given gene expression has been a goal of the scientific community for decades. Thus, the goal of this project will be to design an algorithm to predict the protein expression profile of 25 proteins in 1000 test cells given the gene expression of 639 genes in those cells. The Ordinary Least Squares (OLS) regression method will be implemented to determine the linear regression equation which describes the relationship between protein and RNA gene expression.

Approach:

2.1 Import Statements

Before diving into the data analysis and model building, it's crucial to import the necessary Python libraries. These libraries facilitate various aspects of our project:

- **numpy**: Provides support for efficient numerical computations.
- **pandas**: Essential for data manipulation and analysis, offering data structures like DataFrames.
- **sklearn**: A comprehensive library for machine learning, including tools for model selection, evaluation, and preprocessing.
- **matplotlib**: Enables data visualization, which is key for understanding data distributions and model performance.
- **mpl_toolkits.mplot3d**: An extension of matplotlib for 3D plotting.

```
import numpy as np
import sklearn as sk
import pandas as pd
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

2.2 Splitting Data

A crucial step in preparing our dataset for machine learning models is splitting it into training and testing sets. This function, `Split_Data`, accomplishes this by taking two arrays as input: `train_rna_array` and `train_adt_array`. It then performs a randomized split, ensuring that both the RNA and ADT arrays are divided consistently, allowing for a balanced approach to training and testing the model.

The split is configured to allocate 70% of the data to training and 30% to testing. By using a random state generated at the function's call, we introduce a level of randomness to our training and testing sets, which helps validate the model's robustness and generalizability.

```
def Split_Data(train_rna_array, train_adt_array):
        random = int(np.random.rand()*100)
        # Create local Test/Train split
        rna_train, rna_test, adt_train, adt_test = train_test_split(train_rna_array.T, train_adt_array.T,
test_size=0.3, random_state=random)
        return rna_train.T, rna_test.T, adt_train.T, adt_test.T
```

2.3 Loading and Displaying Data

To begin our analysis, we must first load the data from CSV files. This involves reading the training and test datasets for both ADT (antibody-derived tags) and RNA sequences. By utilizing pandas' `read_csv` function, we can easily import these datasets into our Python environment.

Once the data is loaded, it's important to understand its structure. We do this by checking the shape of each dataset, giving us insight into the number of samples and features we're working with.

Finally, displaying the first few rows of one of these datasets, such as the RNA training set, allows us to get a glimpse of the data structure and the type of information each feature holds.

```
import pandas as pd

# Loading the data
train_adt = pd.read_csv('C:/Users/zackl/Desktop/678/training_set_adt.csv')
train_rna = pd.read_csv('C:/Users/zackl/Desktop/678/training_set_rna.csv')
test_rna = pd.read_csv('C:/Users/zackl/Desktop/678/test_set_rna.csv')

# Checking the dimensions of the datasets
len_tr_adt = train_adt.shape
len_tr_rna = train_rna.shape
len_te_rna = test_rna.shape
```

```
# Displaying the first few rows of the RNA training dataset
train_rna.head()
```

2.4 Preprocessing Data: Removal and Conversion

This involves two key steps: removing unnecessary columns and converting the data into a format suitable for machine learning models.

Removing the First Column

The first column, often an identifier or index, may not contribute to the analysis and thus is removed to ensures that only relevant features are included.

```
TRAIN_adt = train_adt.iloc[:,1:]
TRAIN_rna = train_rna.iloc[:,1:]
TEST_rna = test_rna.iloc[:,1:]
```

Converting Data into Array Matrix

To facilitate mathematical operations and model training, the data is converted into numpy arrays. This conversion is crucial for performance and compatibility with machine learning algorithms.

```
import numpy as np

train_adt_array = np.array(TRAIN_adt)
train_rna_array = np.array(TRAIN_rna)
test_rna_array = np.array(TEST_rna)
```

Printing the Dimensions

This section was for making sure the dimensions are all lined up before I go into data manipulation.

```
print('train adt shape:', train_adt_array.shape)
print('train rna shape:', train_rna_array.shape)
```

```
print('test rna shape:', test_rna_array.shape)
```

These preprocessing steps are fundamental in preparing the data, ensuring it is in the right format and only contains relevant information for building and training our models.

2.5 Implementing Least Squares for Prediction

To address the challenge of predicting ADT values based on RNA expression levels, we employ a least squares algorithm. This mathematical approach is designed to find the linear relationship between RNA expression (independent variables) and ADT values (dependent variables) by minimizing the sum of the squares of the differences between observed and predicted ADT values.

Algorithm Overview:

1. **Matrix Preparation**: Compute matrices a and b from the RNA expression data (`train_rna_array`) and the ADT data (`train_adt_array`), respectively.
2. **Solving for Weights (`w`)**: Calculate the weights `w` that minimize the difference between observed and predicted ADT values using the formula `w = solve(a, b)`.
3. **Prediction**: Apply the calculated weights `w` to predict ADT values (`y_hat`) from RNA expression data.

```
import numpy as np

# Calculating matrices a and b
a = np.dot(train_rna_array, np.transpose(train_rna_array))
b = np.dot(train_adt_array, np.transpose(train_rna_array))

# Solving for weights w
w = np.linalg.solve(a, np.transpose(b))

# Predicting ADT values for training data
y_hat_train = np.dot(np.transpose(train_rna_array), w)

# Adjusting the shape of y_hat for consistency
y_hat_train = np.transpose(y_hat_train)
```

2.6 Implementing Gradient Descent for Optimization

Gradient descent is a foundational technique in machine learning for minimizing the cost function, which in this case, is the discrepancy between the predicted ADT values (`y_hat`) and

the actual ADT values (`yy`). This method iteratively adjusts the weights (`w`) to reduce the cost function's value.

Overview:

1. **Initialization**: Start with random weights and set the learning rate and a number of epochs.
2. **Iteration**: For each epoch, update the weights based on the gradient of the cost function with respect to those weights.
3. **Gradient Calculation**: The gradient is calculated by the dot product of the error and the input features. It points in the direction of the steepest increase in the cost function.
4. **Weight Update**: Adjust the weights in the opposite direction of the gradient to reduce the error.

```
import numpy as np

XX = train_rna_array
yy = train_adt_array

num_epochs = 100
learn_rate = 0.001

# Initialize weights randomly
w = np.random.rand(XX.shape[0], 25)

for trials in range(1, num_epochs):
        for i in range(0, XX.shape[1]-1):
        y_hat2 = np.dot(XX[:,i].reshape(639,1).T, w)
        error = yy[:,i] - y_hat2
        gradient = error.T * XX[:,i]
        w = w + learn_rate * gradient.T

        pose = np.dot(XX.T, w)
        pose_long = np.reshape(pose, -1)
        yy_long = np.reshape(yy, -1)
        loss = np.corrcoef(pose_long, yy_long)[0, 1]
        print(f'Trial # {trials} Loss: {loss}')
```

Reshaping arrays and Calculating Correlation

After predictions, it's beneficial to reshape the arrays to perform statistical analyses. Reshaping `y_hat` and `train_adt_array` into one-dimensional arrays facilitates the computation of statistical measures such as the Pearson correlation coefficient.

```
import numpy as np

# Reshaping the arrays for statistical analysis
y_hat_long = np.reshape(y_hat, -1)
train_adt_array_long = np.reshape(train_adt_array, -1)

# Calculating the Pearson correlation coefficient
correlation_matrix = np.corrcoef(train_adt_array_long, y_hat_long)
```

Generating and Exporting Results

To organize and export the predicted ADT values (y_hat) for further analysis or sharing, I did these steps:

1. **Index Creation**: Generate a list of index strings for each entry in the prediction array. This aids in identifying each prediction uniquely.
2. **DataFrame Construction**: Used pandas to create a DataFrame from the y_hat array, with the generated index strings as row identifiers. Assign a column name to describe the data.
3. **Saving to CSV**: Export the DataFrame to a CSV file for easy access and sharing.

```
# Creating index strings for each prediction
index_strings = ["ID_" + str(i) for i in range(1, len(y_hat_long) + 1)]

# Constructing the DataFrame
DF = pd.DataFrame(y_hat_long, index=index_strings)
DF.columns = ['Expected']
DF.index.name = "Id"

# Exporting the DataFrame to a CSV file
DF.to_csv("C:/Users/zackl/YHat.csv")
```

Displaying a Graph

This code snippet demonstrates how to create a 3D scatter plot using matplotlib's Axes3D tool. It starts by reshaping x, y, and z arrays into one-dimensional arrays to ensure compatibility with the plotting function. A figure and a 3D subplot are then created, onto which the data points are plotted as red circles. Labels for each axis are set to 'X', 'Y', and 'Z' to clearly denote the dimensions being represented. Finally, the plot is displayed using plt.show() providing a visual representation of the data in three-dimensional space.

```
x = x.reshape(-1)
y = y.reshape(-1)
z = loss_vector.reshape(-1)
```

```
# Create figure and 3D axis
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot points
ax.scatter(x, y, z, c='r', marker='o')

# Set labels
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

# Show plot
plt.show()
```

Results:

Figure 1 is a 3 dimensional graph, with the learn rate on the X, the lambda on the y and the loss on Z. The smallest learning rate, 0.00000001 and the smallest lambda 0.00000001 lambda resulted in the lowest loss.

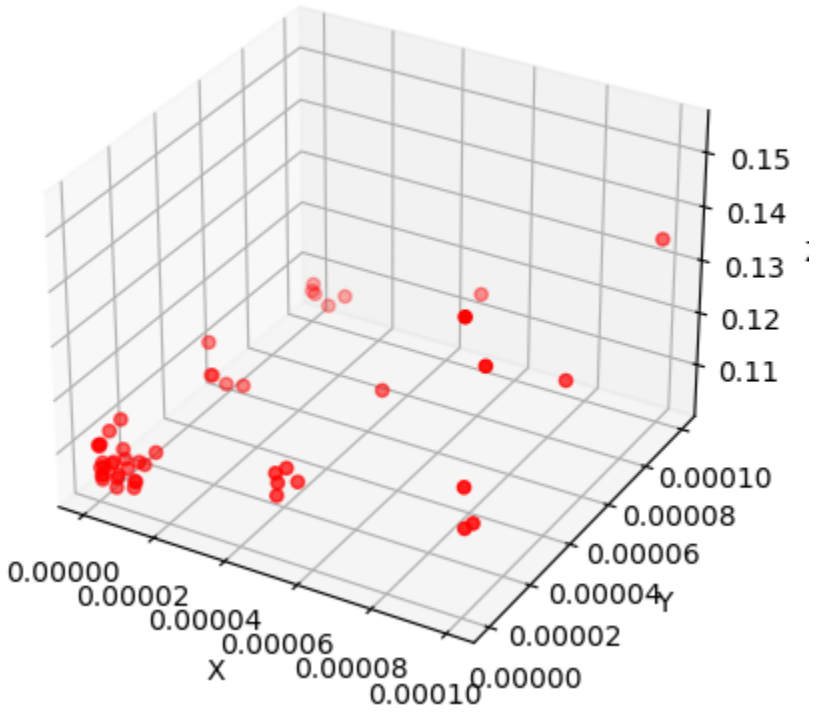

Figure 1: Learn rate, lambda and loss rate

This least squares model has its strengths and weaknesses. The strengths included the simplicity of the model, it being a good fit for linear models, and its flexibility of handling more

complex problems. Some of the weaknesses include the sensitivity to outliers, overfitting, and its dependency on assumptions.

Conclusion

Through a systematic approach that included data preprocessing, algorithm implementation, and optimization via gradient descent, I have demonstrated the feasibility and potential of computational techniques in deciphering complex biological data.

This project began with the foundational step of importing necessary libraries, followed by loading and preprocessing the data to ensure it was in the right format for analysis. We then implemented a least squares algorithm to establish a baseline for our predictions, and further refined our model through gradient descent, an optimization technique that iteratively adjusted our model's parameters to minimize prediction error.

The results were visualized through 3D scatter plots, providing a tangible sense of the model's performance and the relationship between variables. By reshaping arrays and calculating the Pearson correlation coefficient, we gained insights into the accuracy of our predictions, ultimately facilitating a deeper understanding of the underlying biological processes.