Identifying Highly Expressed Genes in Follicular lymphoma B Cells

1. Introduction:

Follicular lymphoma (FL) is a type of cancer that originates in B cells, a class of white blood cells essential for immune function. FL is believed to arise from germinal center (GC) B cells, which are normal B cells that undergo differentiation and selection processes in specialized regions of lymph nodes to enhance immune responses. In FL, however, these cells become malignant, leading to unchecked proliferation and resistance to cell death. This project aims to analyze single-cell RNA sequencing (scRNA-seq) data to identify and examine the top 10 genes that are most highly expressed in FL B cells. By understanding how these genes play a role in the malignant transformation of B cells, the research could provide valuable insights into the underlying molecular mechanisms of FL and identify potential targets for cancer treatment.

2. Dataset Description

- **Dataset Source**
  https://www.kaggle.com/datasets/alexandervc/scrnaseq-b-cells-nature-immunology-2018-gse115795/data
- **Dataset Overview**

The dataset represents the results of single-cell RNA sequencing, where rows correspond to cells analyzed in the sample, and columns to genes. The number of rows and columns are 703 and 14968 respectively. The value in the matrix shows how strong the "expression" of the corresponding gene in the corresponding cell is, where 0 means no expression. The greater the value is, the stronger the expression of the gene in the cell. Every value in the matrix is of the same type of data: float. The dataset from kaggle is already pretty neat.

- **Data Cleaning Process:**

First, to ensure we have a clean dataset, I removed all rows containing NULL values from the original DataFrame. I also removed the “UniqueCellID” column which includes non-numeric values to manipulate the dataset further and perform calculations.

3. Research Question

This research aims to unfold which specific genes contribute to the malignant transformation of FL B cells. By doing so, it aims to provide critical insights into the underlying molecular mechanisms of FL and to identify promising targets for therapeutic intervention.

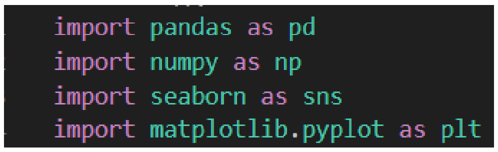
Key Questions:

What are the top 10 genes expressed in follicular lymphoma B cells?

a. Which genes rank highest based on their total expression (sum)?
b. Which genes rank highest based on their average expression (mean)?
c. Are the expressions of these top genes consistent across cells, or do the top genes identified by sum and mean show uniformly high expression?

4. Code Explanation

- **Imports**



```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

Pandas is used to import and load the dataset (pd.read_csv()), clean the data (remove rows with missing values using df.dropna()), etc).

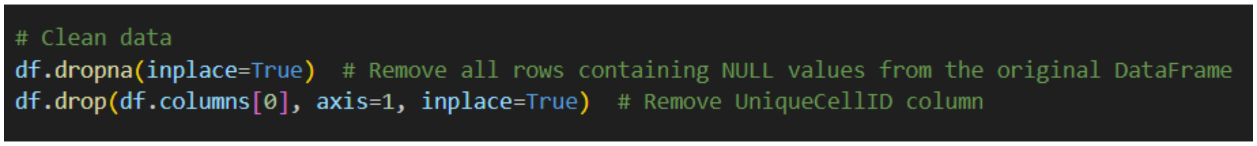
NumPy is used to handle and perform mathematical operations on the gene expression data (convert the DataFrame columns into arrays (df.columns.to_numpy(), functions (np.sum(), np.mean(), np.median())).

Seaborn creates visualizations of the gene expression data (sns.barplot, sns.violinplot).

Matplotlib customizes the visualizations.

- **Data Cleaning**

First, to ensure we have a clean dataset, I removed all rows containing NULL values from the original DataFrame.



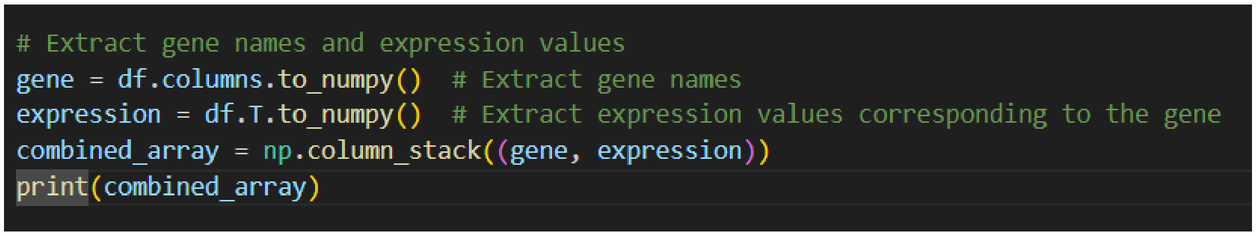
```python
# Clean data
df.dropna(inplace=True)  # Remove all rows containing NULL values from the original DataFrame
df.drop(df.columns[0], axis=1, inplace=True)  # Remove UniqueCellID column
```

I also removed the “UniqueCellID” column which includes non-numeric values since they are irrelevant for my analysis and also for easy mathematical manipulation.

df.columns[0] refers to the first column in the DataFrame. axis parameter (axis=1) is required to perform the operation along the column because pandas operations default to rows (axis=0). inplace=True ensures that the operation modifies the original DataFrame instead of returning a new one.

- **Preprocessing data**

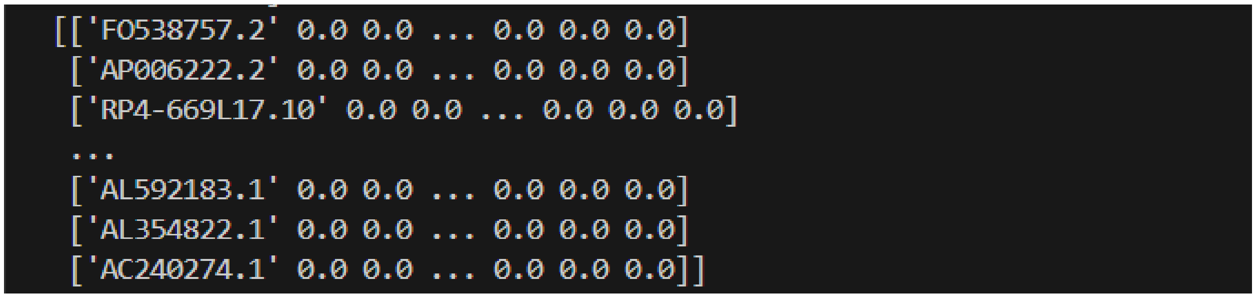
Since I would do calculations within the gene column, it is easier for visualization later by making a combined array extracting gene names and expression values.



```
# Extract gene names and expression values
gene = df.columns.to_numpy()  # Extract gene names
expression = df.T.to_numpy()  # Extract expression values corresponding to the gene
combined_array = np.column_stack((gene, expression))
print(combined_array)
```

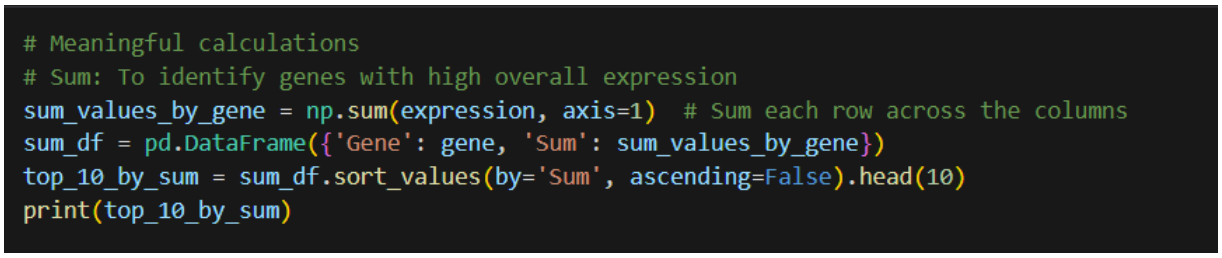
gene = df.columns.to_numpy()

This converts the column names into a NumPy array. As a result, we have a 1D NumPy array where each element is the name of a gene (['Gene1', 'Gene2', 'Gene3', ...]). Since it's a 1D array, when I later used the stack operation, it would stack as columns. Therefore, when extracting expression values, before stacking, I transposed so that in the output, each column would be each cell. I also convert df to numpy for calculations. These preprocessing steps ensure a more complete look of my dataset that I would perform mathematical operations on. The output in the terminal confirms my revised dataset for easy manipulation.



```
[['FO538757.2' 0.0 0.0 ... 0.0 0.0 0.0]
 ['AP006222.2' 0.0 0.0 ... 0.0 0.0 0.0]
 ['RP4-669L17.10' 0.0 0.0 ... 0.0 0.0 0.0]
 ...
 ['AL592183.1' 0.0 0.0 ... 0.0 0.0 0.0]
 ['AL354822.1' 0.0 0.0 ... 0.0 0.0 0.0]
 ['AC240274.1' 0.0 0.0 ... 0.0 0.0 0.0]]
```

- **Calculations**
  1. **Sum**



```
# Meaningful calculations
# Sum: To identify genes with high overall expression
sum_values_by_gene = np.sum(expression, axis=1)  # Sum each row across the columns
sum_df = pd.DataFrame({'Gene': gene, 'Sum': sum_values_by_gene})
top_10_by_sum = sum_df.sort_values(by='Sum', ascending=False).head(10)
print(top_10_by_sum)
```

sum_df = pd.DataFrame({'Gene': gene, 'Sum': sum_values_by_gene})

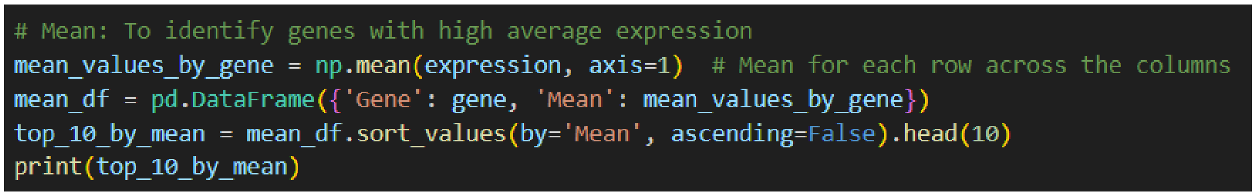
creates a DataFrame that pairs each gene with its corresponding sum of expression values.

top_10_by_sum = sum_df.sort_values(by='Sum', ascending=False).head(10)

First, I sorted the rows of sum_df by the 'Sum' column in descending order and selected the first 10 rows from the sorted DataFrame. By printing, it's easier for interpretation.

2. Mean

Similarly, I calculated the mean, created a dataframe, sorted, selected top 10 genes, and printed.

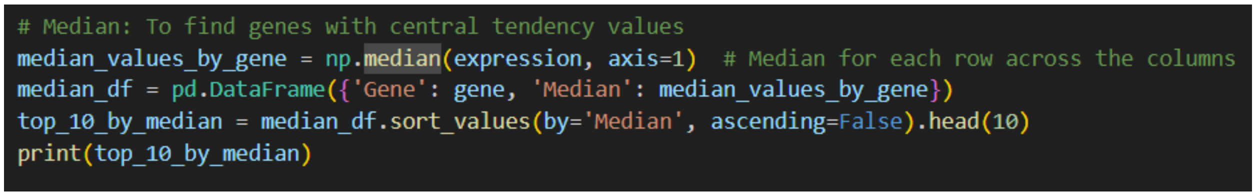


```python
# Mean: To identify genes with high average expression
mean_values_by_gene = np.mean(expression, axis=1)  # Mean for each row across the columns
mean_df = pd.DataFrame({'Gene': gene, 'Mean': mean_values_by_gene})
top_10_by_mean = mean_df.sort_values(by='Mean', ascending=False).head(10)
print(top_10_by_mean)
```

At this point, based on both printed data frames, I was able to identify the top 10 genes expressed based on sum and mean, and they happened to be the same 10 genes.

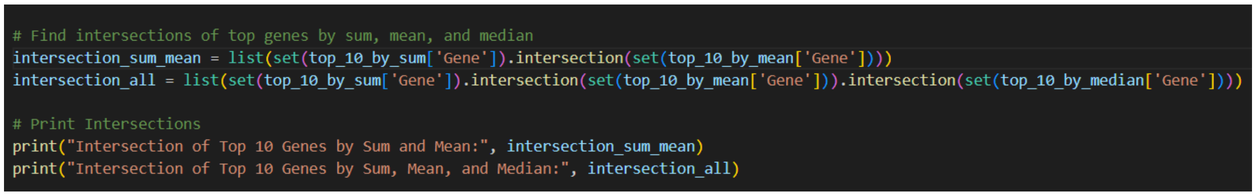
3. Median

Similarly, I calculated the mean, created a dataframe, sorted, selected top 10 genes, and printed.



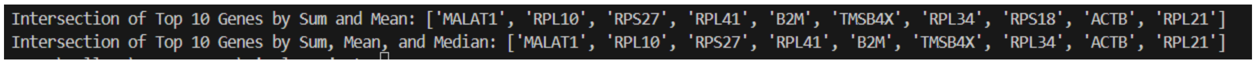
```python
# Median: To find genes with central tendency values
median_values_by_gene = np.median(expression, axis=1)  # Median for each row across the columns
median_df = pd.DataFrame({'Gene': gene, 'Median': median_values_by_gene})
top_10_by_median = median_df.sort_values(by='Median', ascending=False).head(10)
print(top_10_by_median)
```

Subsequently, I found the intersection of top 10 genes by sum and by mean, and found the intersection with the top 10 genes by median, which resulted in only 9 genes instead of 10.



```python
# Find intersections of top genes by sum, mean, and median
intersection_sum_mean = list(set(top_10_by_sum['Gene']).intersection(set(top_10_by_mean['Gene'])))
intersection_all = list(set(top_10_by_sum['Gene']).intersection(set(top_10_by_mean['Gene'])).intersection(set(top_10_by_median['Gene'])))

# Print Intersections
print("Intersection of Top 10 Genes by Sum and Mean:", intersection_sum_mean)
print("Intersection of Top 10 Genes by Sum, Mean, and Median:", intersection_all)
```

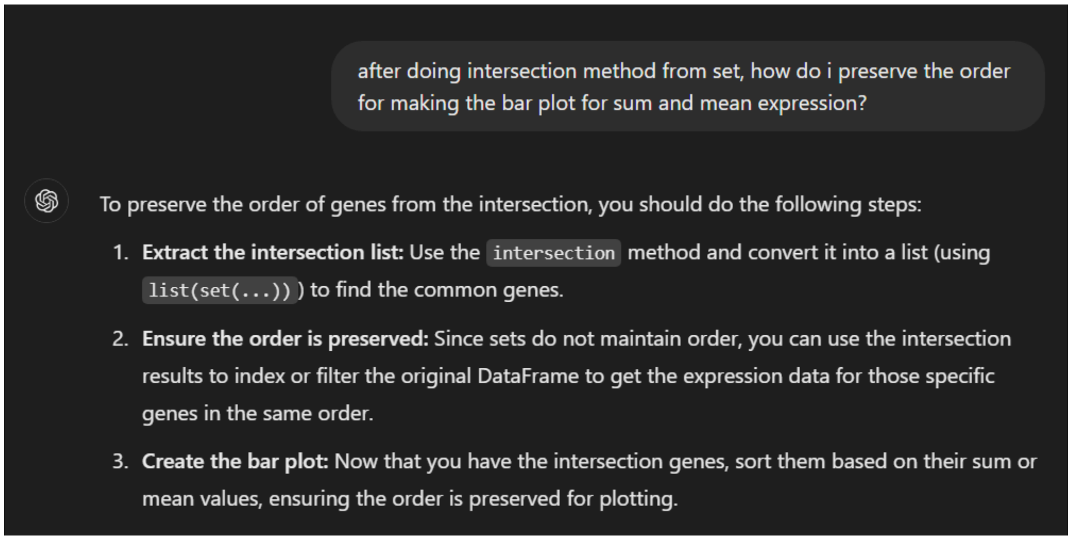


```
Intersection of Top 10 Genes by Sum and Mean: ['MALAT1', 'RPL10', 'RPS27', 'RPL41', 'B2M', 'TMSB4X', 'RPL34', 'RPS18', 'ACTB', 'RPL21']
Intersection of Top 10 Genes by Sum, Mean, and Median: ['MALAT1', 'RPL10', 'RPS27', 'RPL41', 'B2M', 'TMSB4X', 'RPL34', 'ACTB', 'RPL21']
```

- **Challenges**
  1. **When finding intersection**

Since I wanted to ensure our 10 genes are unique by sum and by mean to later find intersection, I first converted to a set. By converting a list to set, it removed all the duplicates. Then we could do a set method which is intersection.

However, I still wanted to preserve the order for the unique items for visualization of sum of expression and mean expression, so I converted back to list using list(set(...)) with the help from AI.

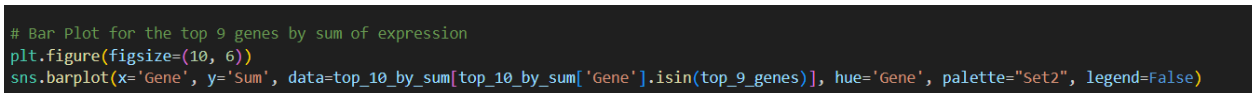

after doing intersection method from set, how do i preserve the order for making the bar plot for sum and mean expression?

To preserve the order of genes from the intersection, you should do the following steps:

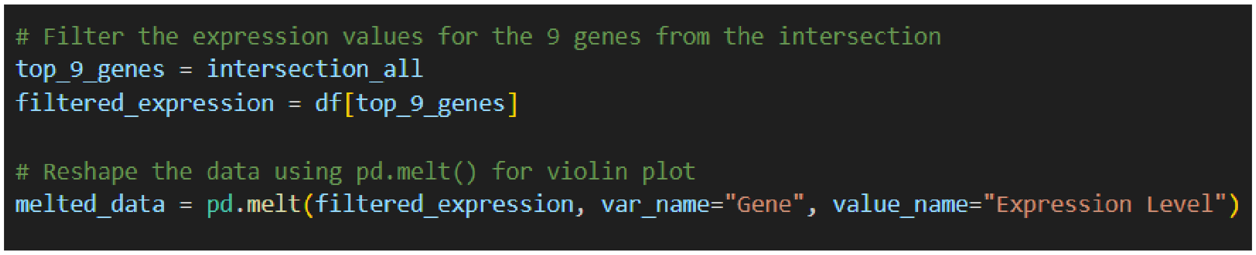
1. **Extract the intersection list:** Use the `intersection` method and convert it into a list (using `list(set(...))`) to find the common genes.
2. **Ensure the order is preserved:** Since sets do not maintain order, you can use the intersection results to index or filter the original DataFrame to get the expression data for those specific genes in the same order.
3. **Create the bar plot:** Now that you have the intersection genes, sort them based on their sum or mean values, ensuring the order is preserved for plotting.

  2. **When attempting violin plot**

For the bar plot, I retrieved the data of only 9 genes from the calculations for the top 10 genes by checking using .isin() and returning a new DataFrame containing only the top 9 genes from the original top_10_by_sum DataFrame. This is done similarly for the mean bar plot.

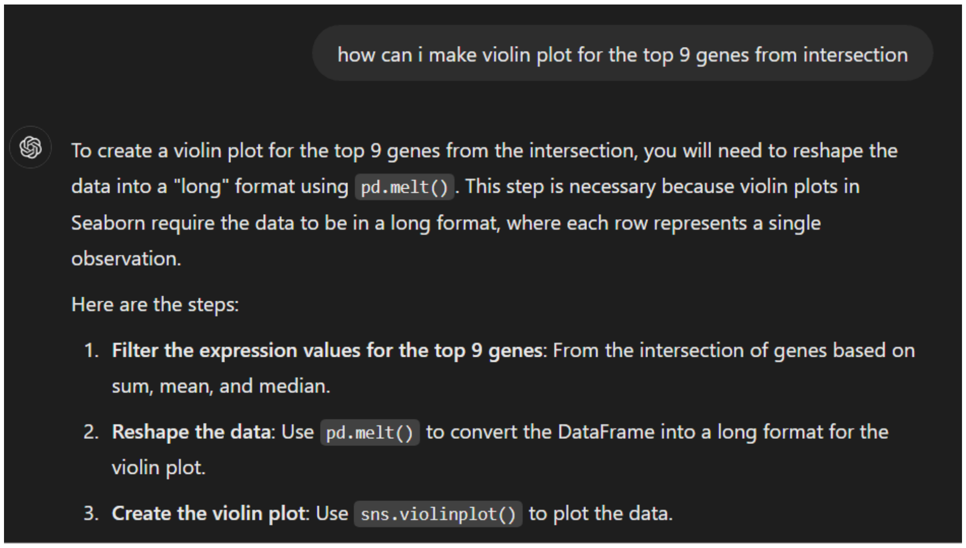

```python
# Bar Plot for the top 9 genes by sum of expression
plt.figure(figsize=(10, 6))
sns.barplot(x='Gene', y='Sum', data=top_10_by_sum[top_10_by_sum['Gene'].isin(top_9_genes)], hue='Gene', palette="Set2", legend=False)
```

When I attempted to create the violin plot, I couldn't directly use the data from the intersection but realized that the data needed to be transformed into a long format (one row per value), which is typically done using Pandas. As a result, I first converted the data into a DataFrame and then applied pd.melt to reshape it.



```
# Filter the expression values for the 9 genes from the intersection
top_9_genes = intersection_all
filtered_expression = df[top_9_genes]

# Reshape the data using pd.melt() for violin plot
melted_data = pd.melt(filtered_expression, var_name="Gene", value_name="Expression Level")
```

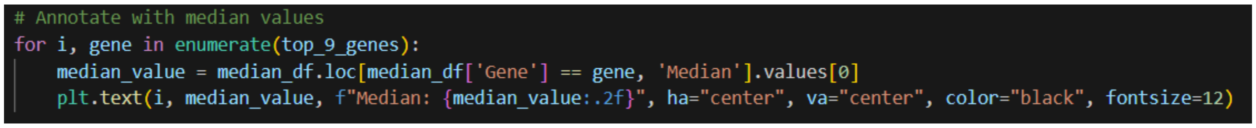


how can i make violin plot for the top 9 genes from intersection

To create a violin plot for the top 9 genes from the intersection, you will need to reshape the data into a "long" format using `pd.melt()`. This step is necessary because violin plots in Seaborn require the data to be in a long format, where each row represents a single observation.

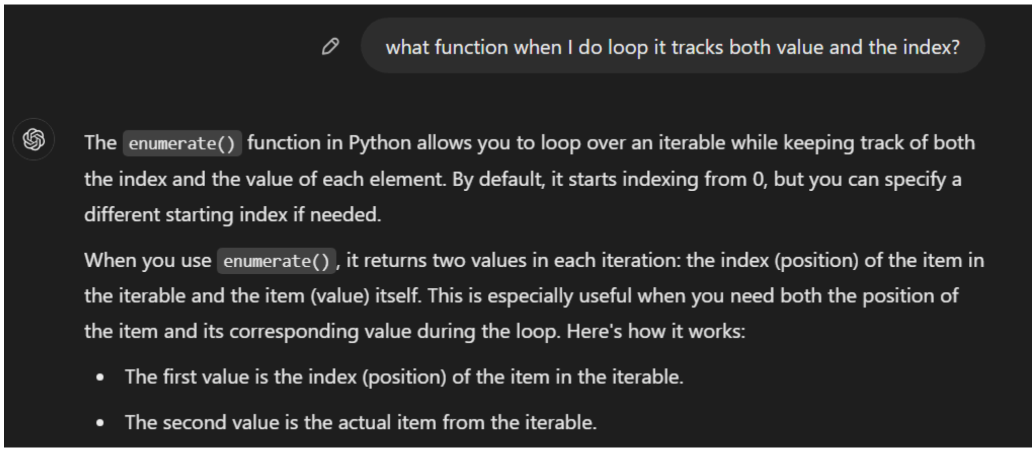
Here are the steps:

1. **Filter the expression values for the top 9 genes**: From the intersection of genes based on sum, mean, and median.
2. **Reshape the data**: Use `pd.melt()` to convert the DataFrame into a long format for the violin plot.
3. **Create the violin plot**: Use `sns.violinplot()` to plot the data.



```
# Annotate with median values
for i, gene in enumerate(top_9_genes):
    median_value = median_df.loc[median_df['Gene'] == gene, 'Median'].values[0]
    plt.text(i, median_value, f"Median: {median_value:.2f}", ha="center", va="center", color="black", fontsize=12)
```

It was quite challenging when I tried retrieving the median values and annotating on my violin plot. Then I found the built-in function enumerate() function that iterates over a list while keeping track of the index of the current item. I managed to apply it to my code that loops through top_9_genes, retrieving each gene's index and name.



what function when I do loop it tracks both value and the index?

The `enumerate()` function in Python allows you to loop over an iterable while keeping track of both the index and the value of each element. By default, it starts indexing from 0, but you can specify a different starting index if needed.

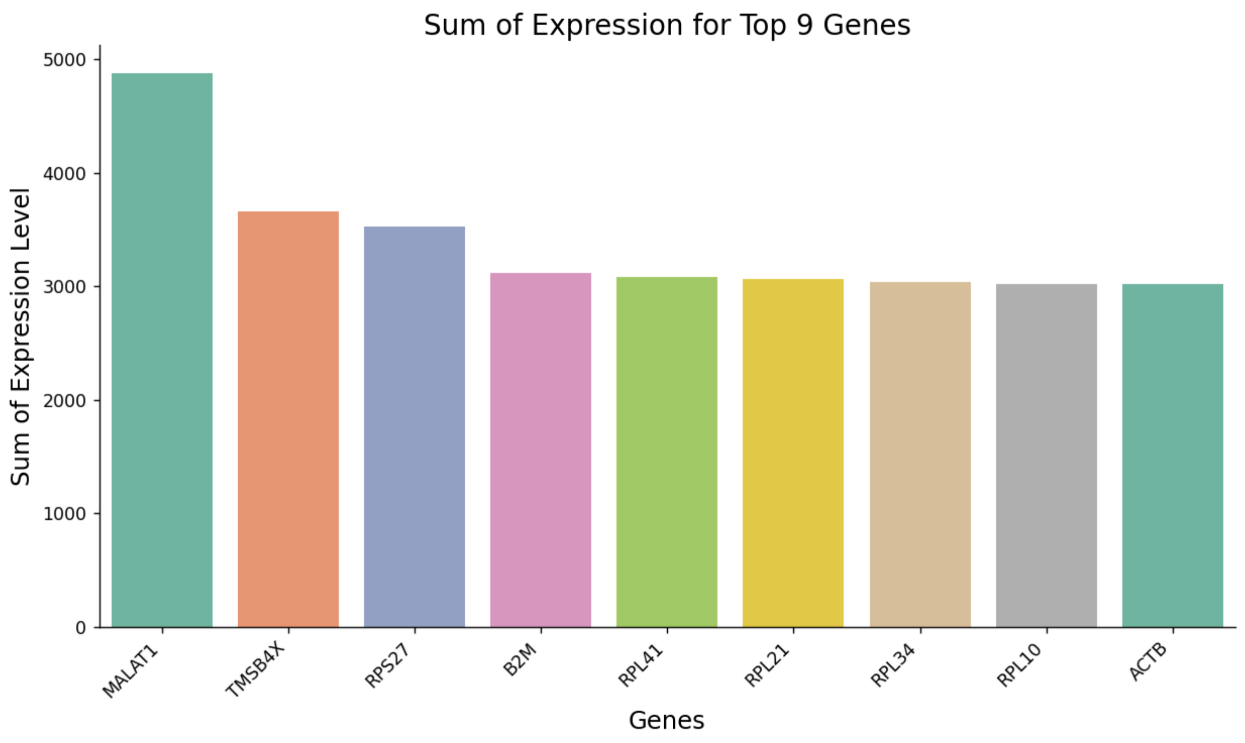
When you use `enumerate()`, it returns two values in each iteration: the index (position) of the item in the iterable and the item (value) itself. This is especially useful when you need both the position of the item and its corresponding value during the loop. Here's how it works:

- The first value is the index (position) of the item in the iterable.
- The second value is the actual item from the iterable.

5. Data Visualizations

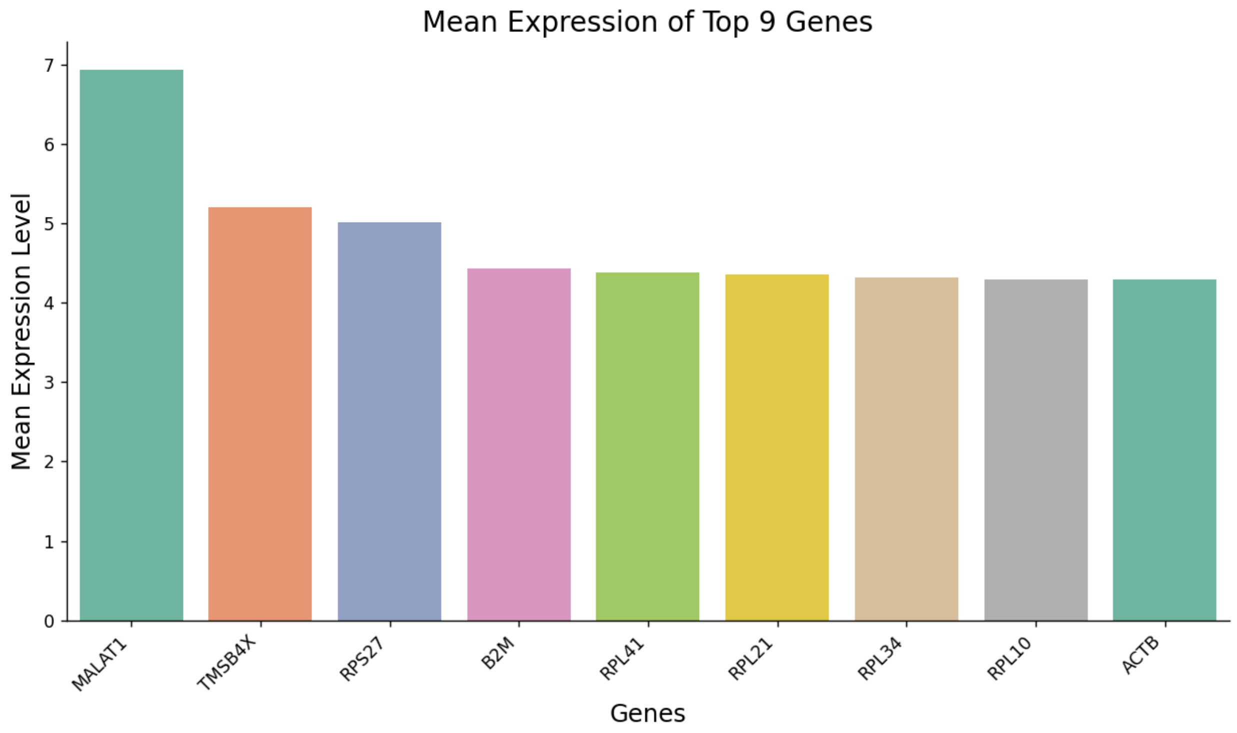
- **Visualization 1: Bar plot for sum of expression values**

A bar plot is used to display the total expression levels across the top 9 genes (sorted by their sum of expression) to make it easy to compare the genes and see which ones have the highest overall expression. The height of the bars represents the sum of expression values for each gene across all cells. Genes with higher bars have higher total expression, which suggest that they are more highly expressed across the dataset.



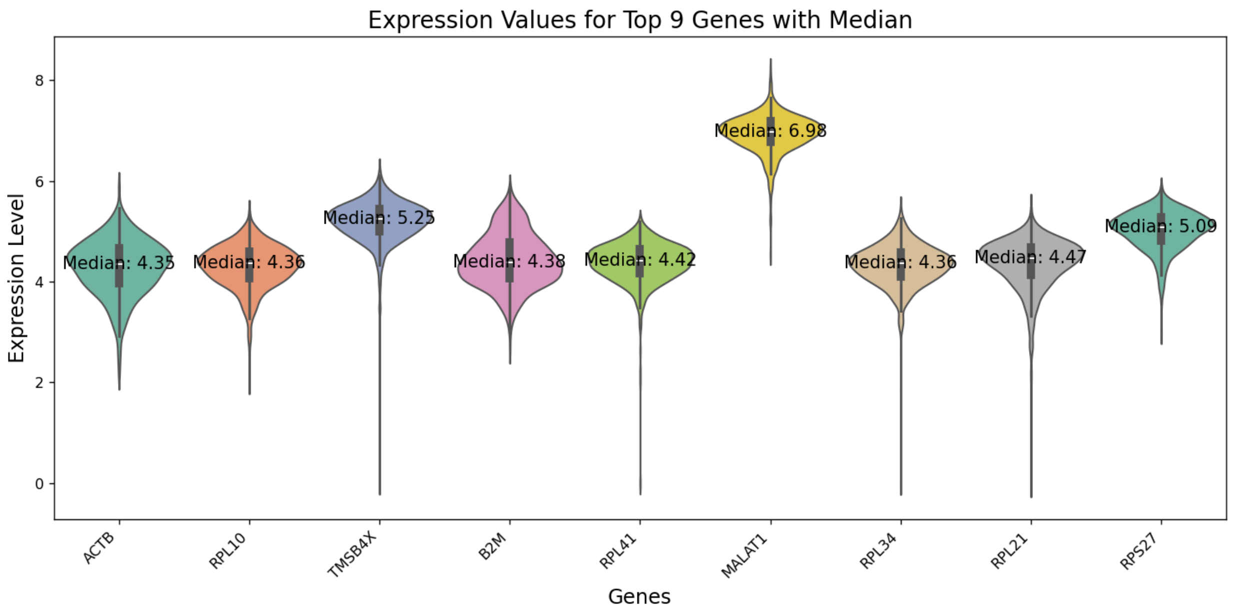
- **Visualization 2: Bar plot for mean expression**

Similarly, a bar plot is used to visualize mean expression level for each gene across all the cells. The plot helps to identify which genes have the highest average expression.



- **Visualization 3: Violin plot for expression distribution**

Violin plots give a fuller picture of the data by showing the distribution, density, and spread of the data. The violin plot illustrates the distribution of expression levels for each of the top 9 genes. The width of the "violin" represents how dense or clustered the observations are at that level. Wider sections indicate more cells with similar expression levels for that gene.



6. Results

My goal previously was to identify top 10 genes that are expressed in FL B cells. However, based on the sum and mean calculations, I could identify the same set of 10 genes that are expressed the most. However, when finding the intersection of the top 10 genes by sum and by mean with the top 10 genes by median, I discovered only 9 genes. This suggests that one gene is an outlier, being highly expressed in a small subset of cells with robust activity, rather than being uniformly expressed across all cells. Subsequently, I decided to move forward with data analysis of those 9 genes, which are RPL21, TMSB4X, MALAT1, B2M, RPS27, RPL41, ACTB, RPL34, RPL10.

Generally, the sum and average are highly sensitive to outliers. A few cells with extremely high expression levels can result in a skewed distribution of the gene's expression across the entire population, with many cells having low or no expression of a gene and a smaller subset of cells showing very high expression. Therefore, before drawing conclusions about these 10 genes, I calculated the median as a measure of central tendency. The median demonstrates the midpoint of the distribution, revealing whether a gene is expressed at moderate levels in the majority of cells. For example, although MALAT1 is ranked the first in both sum and mean analysis, by looking at the median in the violin plot, I was able to conclude it is consistently expressed at relatively high levels across most cells. Given its highest median value, MALAT1 indicates consistent activity in many cells.

Moreover, a long tail in the lower range for genes such as TMSB4X, RPL41, RPL34, and RPL21 suggests that there are few extreme values (outliers) that are much lower than the majority of the data points in the distribution. This implies that for these genes, while most of the data points are clustered around a certain expression level, there are a few cells with significantly lower expression levels compared to the rest. This demonstrates negative skewness with some extremely low values.

Interestingly, the violin plot for the gene B2M shows a greater spread on the upper side, suggesting a denser cluster of data points in the higher expression range. The "fatter" upper section of the B2M plot highlights that its expression is concentrated at higher levels in a significant proportion of samples.

In summary, the top 9 genes with high expression relatively uniformly in FL B cells are RPL21, TMSB4X, MALAT1, B2M, RPS27, RPL41, ACTB, RPL34, and RPL10. Based on the sum, average, and median analysis, MALAT1 is expressed the highest and consistently among all cells, despite a low negative skewness. Among 9 genes, TMSB4X, RPL41, RPL34, and RPL21 with a significantly longer tail at the bottom indicates left-skewed distribution. Lastly, the denser upper portion of the B2M violin plot highlights that its expression is elevated in a subset of cells.

7. Reflections

The process worked as expected in identifying the top genes with high expression and revealing patterns in their distribution across FL B cells. The combination of sum, mean, and median analyses was effective in capturing consistent high-expression genes like MALAT1.

However, some nuances, such as skewness in gene expression, suggest the need for further exploration to ensure that the conclusions accurately represent the data. We could incorporate statistical tests for skewness like normalization or validate findings using external datasets.