# Advanced EDA for Genomic Data Analysis: Identifying Genetic Variations Through Visualization

## Phase 4: Model Deployment and Interface Development

### 4.1 Overview of Model Deployment and Interface Development

In genomic data analysis, after training models to identify genetic variations, the next step is deploying these models for use in real-world applications. This involves making the model accessible and easy to use for researchers or clinicians.

**Model Deployment:**

Model deployment involves placing the trained model into a system where it can process new genomic data. This can be done through cloud services or local servers to ensure it can handle large datasets. It's important to monitor the model's performance, ensure data privacy, and allow updates as new data comes in.

**Interface Development:**

A user-friendly interface allows users to interact with the model and visualize the results. This is often done through a web application or dashboard, where users can see genetic variations through graphs, heatmaps, or interactive charts. This helps researchers understand complex data and make informed decisions based on genetic findings.

### 4.2 Deploying the Model

Deploying the model involves making it accessible for real-world use, so it can process new genomic data and provide predictions or insights.

Steps in Model Deployment:

1. **Setting Up the Environment**: Choose a platform (cloud or on-premise) where the model will run. It should be able to handle large genomic data and provide enough computing power.
2. **Containerization**: Use tools like Docker to package the model with all its dependencies. This makes sure it runs smoothly on different systems without compatibility issues.
3. **API Integration**: The model is made accessible through an API, allowing other systems to send data to the model and receive predictions. This enables seamless use in other applications or research tools.
4. **Monitoring and Maintenance**: Once the model is deployed, it needs to be monitored to ensure it works properly. This includes checking for errors and retraining the model with new data if necessary.
5. **Security and Privacy**: Since genomic data is sensitive, security measures like encryption and user authentication are essential to protect the data and comply with privacy laws.

**Source code :**

```
import os
import shutil
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from scipy.stats import ttest_ind
from utils.file import get_file_paths

class DataPreprocess:
    def __init__(self, data_dir: str, top_n_gene: list, gene_limit):
        # Data folder path which includes train-test dataset and class file
        self.data_dir = data_dir
        self.top_n_gene = top_n_gene
        self.gene_limit = gene_limit
        # Getting file names to dict and accessing them just typing name of file eg. 'train', 'test'
        self.file_paths = get_file_paths(self.data_dir)
        # Read train-test datasets and class file
        self.train, self.test = self.read_data()
        self.classes, self.encoder = self.read_classes()

        self.t_test_result = self.data_preprocess()
        self.top_n_values = self.get_top_n()
        self.save_top_n()

    def data_preprocess(self):
        self.remove_fold_data(5)
        self.threshold_data()
        self.remove_low_variance()
        return self.calculate_t(save_df=True)

    def read_data(self):
        # Reading train and test csv files and converting them into numpy
        # For now only selecting first 100 genes. It helps working faster for further steps
        if self.gene_limit == 0:
            train = pd.read_csv(self.file_paths['train']).to_numpy().T[:, :]
            test = pd.read_csv(self.file_paths['test']).to_numpy().T[:, :]
        else:
            train = pd.read_csv(self.file_paths['train']).to_numpy().T[:, :self.gene_limit]
            test = pd.read_csv(self.file_paths['test']).to_numpy().T[:, :self.gene_limit]
        return train, test

    def read_classes(self):
        # Open txt file as read mode
        file = open(self.file_paths['class'], "r")
        # Reading file contents as a list
        contents = file.read()
        # Separate new-lines
        classes = contents.split(sep="\n")[1:-1]
        # Creating encoder for classes
        le = LabelEncoder()
        # Fit all patient classes into label encoder and save them in new variable
        encoded = le.fit_transform(classes)
        return encoded, le

    def remove_fold_data(self, fold_n):
        # Loop over genes with all samples to find the index of genes that do not have enough fold
        genes_to_delete = [idx for idx, genes_row in enumerate(self.train.T)
                           if np.max(genes_row[1:]) < fold_n * np.min(genes_row[1:])]

        # Delete gene columns from training and test data
        self.train = np.delete(self.train, genes_to_delete, 1)
        self.test = np.delete(self.test, genes_to_delete, 1)

    def remove_low_variance(self):
        # Loop over genes with all samples to find the index of genes that do not have enough fold
        genes_to_delete = [idx for idx, genes_row in enumerate(self.train.T)
                           if np.std(genes_row[1:]) < 10]

        # Delete gene columns from training and test data
        self.train = np.delete(self.train, genes_to_delete, 1)
        self.test = np.delete(self.test, genes_to_delete, 1)

    def threshold_data(self):
        # Loop over genes with all samples to find the index of genes that do not have enough fold
        genes_to_delete = [idx for idx, genes_row in enumerate(self.train.T)
                           if np.max(genes_row[1:]) < 20 or np.min(genes_row[1:]) > 12000]

        # Delete gene columns from training and test data
        self.train = np.delete(self.train, genes_to_delete, 1)
        self.test = np.delete(self.test, genes_to_delete, 1)

    def calculate_t(self, save_df=False):
        # Placeholder for all class individual t test result
        total_t_result = []
        print(f'T-test Started on {len(set(self.classes))} class with {self.train.shape[1]} genes.\n')
        # Loop over all classes
        for cls in range(len(set(self.classes))):
            print(f'T-test on Class: {self.encoder.inverse_transform((cls,))[0]}')
            # Append class-based results in other list
            cls_t_result = []
            # Get indices of classes
            samp = np.where(self.classes == cls)[0] + 1
            # Take the first gene for t test
            for gene_0 in range(self.train.shape[1]):
                if np.any(self.train[1:, gene_0] < 20) or np.any(self.train[1:, gene_0] > 12000):
                    continue
                # Calculate t and p values when testing with all the remaining genes
                for gene_1 in range(gene_0 + 1, self.train.shape[1]):
                    if np.any(self.train[1:, gene_1] < 20) or np.any(self.train[1:, gene_1] > 12000):
                        continue
                    t_value, p_value = ttest_ind(self.train[samp, gene_0], self.train[samp, gene_1])
                    cls_t_result.append((self.encoder.inverse_transform((cls,))[0],
                                         self.train[0, gene_0], gene_0, self.train[0, gene_1], gene_1,
                                         t_value, p_value))
            total_t_result.append(cls_t_result)

        # If desired, save these results in an additional file
        if save_df:
            path = os.path.join(self.data_dir, "pp5i_t_result.gr.csv")
            cols = ['Class', 'Gene 1', 'Indices of Gene 1', 'Gene 2', 'Indices of Gene 2', 't-value', 'p-
value']
            data = np.squeeze(np.array(total_t_result).reshape((1, -1, 7)))
            df = pd.DataFrame(data, columns=cols)
            df.to_csv(path, index=False)
        print('\nT-test completed!')
        return np.array(total_t_result)

    def get_top_n(self):
        # Placeholder for top-n genes and their values
        top_n_values = {}
        for n in self.top_n_gene:
            # Placeholder for top genes in max n gene
            n_train_list = []
            # Loop over results and classes
            total_indices = []
            for encoded_class, cls_t_result in enumerate(self.t_test_result):
                indices_list = []
                cls_t_result = np.array(sorted(cls_t_result, key=lambda x: np.abs(float(x[5])),
reverse=True))
                for ind_0, ind_1 in cls_t_result[:, [2, 4]]:
                    if int(ind_0) not in indices_list and int(ind_0) not in total_indices:
                        indices_list.append(int(ind_0))
                        total_indices.append(int(ind_0))
                    if len(indices_list) == n:
                        break
                    if int(ind_1) not in indices_list and int(ind_1) not in total_indices:
                        indices_list.append(int(ind_1))
                        total_indices.append(int(ind_1))
                    if len(indices_list) == n:
                        break

                indices_list = list(self.train[0, indices_list])
                indices_list.append(self.encoder.inverse_transform((encoded_class,))[0])

            n_train_list.append(self.train[:, total_indices])
            n_train_list = np.concatenate(n_train_list, axis=1)
            cls_col = ['Classes'] +
list(np.squeeze(self.encoder.inverse_transform(np.sort(self.classes)).reshape((-1, 1))))
            n_train_list = np.concatenate((n_train_list, np.array(cls_col)[:, np.newaxis]), axis=1)

            top_n_values[n] = n_train_list

        return top_n_values

    def save_top_n(self):
        save_dir = os.path.join(self.data_dir, "top_n")
        if os.path.exists(save_dir):
            shutil.rmtree(save_dir)
            os.mkdir(save_dir)
        else:
            os.mkdir(save_dir)

        for n, n_values in enumerate(self.top_n_values.values()):
            cols = n_values[0, :]

            df_values = pd.DataFrame(n_values[1:, :], columns=cols)
            path = os.path.join(save_dir, f"pp5i_train.top{int(self.top_n_gene[n]):02}.gr.csv")
            df_values.to_csv(path, index=False)
        print(f'Files saved to: {save_dir}')

if __name__ == "__main__":
    dp = DataPreprocess('data', [2, 4, 6, 8, 10, 12, 15, 20, 25, 30], gene_limit=100)
```

**Deploy the API on Cloud**: To deploy an API for genomic data analysis on the cloud, follow these simplified steps:

1. **Choose a Cloud Provider**: Select a cloud platform like AWS, Google Cloud, or Azure for deployment.
2. **Prepare Your Model**: Package your model and required libraries. You can use Docker to ensure the model works across different environments.
3. **Create the API**: Develop the API using frameworks like Flask or FastAPI. Define endpoints (e.g., /predict) that will handle incoming genomic data and provide predictions.
4. **Set Up the Cloud Environment**:
    - Create a cloud instance (e.g., AWS EC2 or Google Cloud Compute Engine).
    - Install necessary libraries and set up a web server (e.g., Nginx or Apache).
5. **Deploy the API**: Upload the code to the cloud and start the API. Use cloud tools like **API Gateway** to manage and expose the API to users.

## Developing the Web Interface

This simple Streamlit app lets users upload a genomic data file, sends it to a prediction API, and displays the results. The app has a basic interface with a title, file uploader, and a "Predict" button. Once the file is uploaded and the button is clicked, the file is sent to the API endpoint, and the returned JSON result is shown on the screen. It’s a lightweight and user-friendly tool for quickly obtaining genomic data analysis results.

```
import streamlit as st
import requests

# URL of the deployed model API
API_URL = "http://your-api-url.com/predict"

st.title("Genomic Data Analysis")

st.write("Upload your genomic data file to get predictions.")
```

```
uploaded_file = st.file_uploader("Choose a genomic data file", type=["txt", "csv", "fasta", "vcf"])

if uploaded_file is not None:
    st.write("File uploaded successfully.")
    # Display file details
    st.write("Filename:", uploaded_file.name)

    # When the user clicks the predict button
    if st.button("Predict"):
        # Send the file to the model API
        files = {"file": uploaded_file.getvalue()}
        try:
            response = requests.post(API_URL, files={"file": uploaded_file})
            if response.status_code == 200:
                result = response.json()
                st.success("Prediction received:")
                st.json(result)
            else:
                st.error(f"Prediction failed with status code: {response.status_code}")
        except Exception as e:
            st.error(f"An error occurred: {e}")
```

## 4.3 Cloud Platform Considerations

When deploying a genomic data analysis model on a cloud platform, several factors should be considered to ensure performance, scalability, and security. Here are key considerations:

1. **Computational Resources**:
    - Genomic data analysis often involves processing large datasets, requiring significant computational power. Choose cloud platforms with high-performance compute options, such as AWS EC2, Google Cloud Compute Engine, or Azure Virtual Machines.
    - You may also need specialized hardware like GPUs for faster data processing, especially if using deep learning models.
2. **Storage**:
    - Genomic datasets can be large. Use scalable storage solutions like Amazon S3, Google Cloud Storage, or Azure Blob Storage to store and manage large files efficiently.
    - Ensure that the storage solution offers fast access and easy integration with your processing pipelines.
3. **Scalability**:
    - The cloud platform should allow you to scale resources up or down based on the workload. Platforms like AWS Auto Scaling, Google Cloud Functions, or Azure Scale Sets can automatically adjust computational resources as needed.
4. **Security and Privacy**:
    - Genomic data is sensitive, and it's crucial to comply with regulations such as HIPAA or GDPR. Ensure that the cloud platform provides robust security features, including data encryption, access controls, and secure communication channels.
    - Opt for private cloud or secure environments for handling sensitive data.

5. **Cost**:
    - Genomic analysis can be resource-intensive, and cloud resources can incur significant costs. Consider pricing models like pay-as-you-go or reserved instances based on your needs. Utilize cost management tools provided by cloud platforms to monitor and control spending.
6. **API Management**:
    - Use cloud-native API management services like AWS API Gateway, Google Cloud API Gateway, or Azure API Management to deploy, manage, and secure your APIs.
7. **Compliance**:
    - Ensure the cloud provider complies with relevant industry standards for data protection and privacy (e.g., HIPAA, ISO 27001, SOC 2) to meet regulatory requirements for genomic data handling.

## 4.4 Conclusion of Phase 4

Deploying a model for genomic data analysis on the cloud provides numerous benefits, including scalability, flexibility, and accessibility. By utilizing cloud platforms such as AWS, Google Cloud, or Azure, you can efficiently handle large genomic datasets and perform complex computations. The deployment process involves setting up APIs that allow seamless integration between the model and user interfaces, ensuring real-time predictions and insights. Proper consideration of computational resources, storage, security, and scalability is essential to ensure efficient, cost-effective, and secure deployment. By leveraging the cloud, genomic data analysis models can be accessed remotely, making them widely available for researchers, clinicians, and scientists to gain insights into genetic variations and improve healthcare outcomes.The overall project of advanced exploratory data analysis (EDA) in genomic data analysis offers significant value by providing deep insights into genetic data. The use of EDA techniques helps in understanding the complex relationships within genomic datasets, identifying patterns, and detecting genetic variations that are critical for scientific discovery and medical applications. By deploying predictive models on the cloud and building accessible user interfaces, we enable efficient analysis of large-scale genomic data. This project showcases the power of combining advanced data analysis methods, machine learning, and cloud technologies to drive progress in genomics, facilitating real-time decision-making and providing valuable tools for research and healthcare.