Genetic Classification of Individuals Using Machine Learning

Date: -	9th November 2023
Team ID: -	Team-593068
Project Name: -	Genetic Classification of Individuals using Machine Learning
Maximum Marks: -	10 marks

Introduction: -

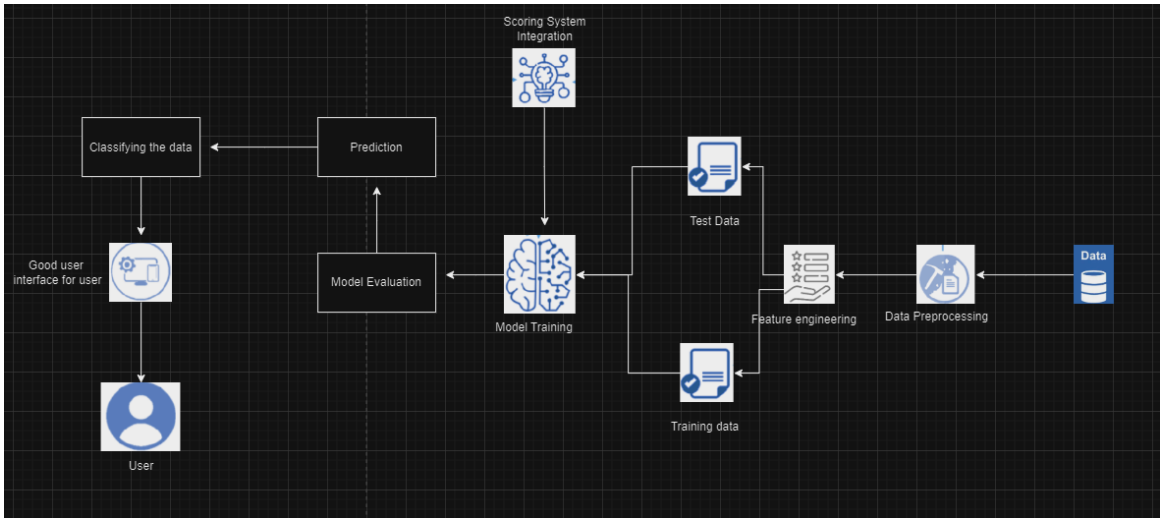
ClinVar is a public database of reports on the links between human variants and phenotypes, together with supporting evidence. Hence, ClinVar makes it easier to access and discuss the hypothesized connections between reported health status and human variation as well as the background to that interpretation. Submissions describing variations discovered in patient samples, claims made about their clinical importance, submitter information, and additional supporting data are processed by ClinVar. The HGVS standard is followed in the mapping and reporting of the alleles reported in submissions to reference sequences. After then, ClinVar displays the data for both interactive users and users who want to use ClinVar for other local apps and everyday workflows. ClinVar collaborates with relevant organizations to effectively address the needs of the medical genetics' community.

In this project, we will be predicted whether the various different submissions provided by different laboratories about the same genome variant are conflicting or in agreement. The criteria for conflicting reports are mentioned below. Each variant phenotype will be classified into 3 categories:

- Benign or likely benign
- VUS
- Pathogenic or likely pathogenic

If two laboratories provide different outcomes, then the result is said to be conflicting. This makes it a simple classification problem, but our issue begins with the sheer size of the dataset and the amount of actual useful data present in it.

Technical Architecture: -



Pre-requisites: -

For the proper executing of this project, the following software is necessary. It includes various libraries and applications to run the required files.

For this demonstration, we shall be using Google Colab and Visual Studia Code. Both of them are free to use and only VS Code needs to be downloaded as Colab can be used alongside Google Cloud.

1. For building the machine learning model, we will be utilizing Google Colab and the following python libraries:
   - Numpy - The primary Python framework for carrying out scientific computations is called NumPy. Large arrays of components and matrices are supported, and arithmetic operations can be performed on these arrays. In machine learning, data analytics, and other scientific domains, NumPy is frequently utilized.
   - Pandas - Pandas is a robust library for data analysis and manipulation. It offers data structures that are made to be readily manipulated in structured data, like DataFrame and Series. In the data science and machine learning fields, pandas is frequently employed for tasks including cleaning, filtering, and data analysis.
   - Matplotlib - Matplotlib is a Python charting toolkit that may be used to create static, interactive, and animated visualizations. You may make a range of plots and charts with it, such as bar charts, histograms, scatter plots, and line graphs. In scientific statistics and data analysis, Matplotlib is extensively utilized for data visualization.
   - Seaborn - Seaborn is a Matplotlib-based library for numerical data visualization. It offers a high degree of interaction for building engaging and educational mathematical models. Seaborn makes complex processes simpler and is frequently used to improve the visual appeal of Matplotlib plots.
   - Scikit Learn - Data mining and analysis may be done quickly and easily with the help of the machine learning framework Scikit-Learn. Numerous machine learning techniques for clustering, regression, classification, and other tasks are included. Easy to use and well-integrable with other scientific computing packages is the design of Scikit-Learn.
   - Pipeline - The pipeline allows Scikit-Learn to streamline a lot of processes.

```
pip install numpy
pip install pandas
pip install matplotlib
pip install seaborn
pip install scikit-learn==1.2.2
pip install pipeline
```

Following the code block above will install all the correct libraries in Colab.

2. For the web deployment, we shall be using both Colab and VS Code. Here, the following additional libraries are required:
   - Pickle - Pickle is used to serialize and de-serialize items. It enables the conversion of a Python object into a byte stream for sending or saving across a network or file. Machine learning models are frequently loaded and stored in pickles for use in other applications.
   - Flask - Python's Flask framework is a lightweight web application. Its user-friendly architecture eliminates the need for any specialized tools or libraries for tasks like database administration and form validation. Flask is a popular tool for Python web application and API development.

```
pip install pickle
```

```
pip install flask
```

These lines of code can be run in Colab as well as powershell terminal in VS Code to install the required libraries.

Project Objectives: -

In this project, we aim to:

- Understand the working of several different Classification algorithms and check application accuracies of each.
- Learn various different data cleaning techniques to be applicable on numerical and categorical data.
- Get applicative knowledge of web application and the usage of Flask for web deployment.

Project Flow: -

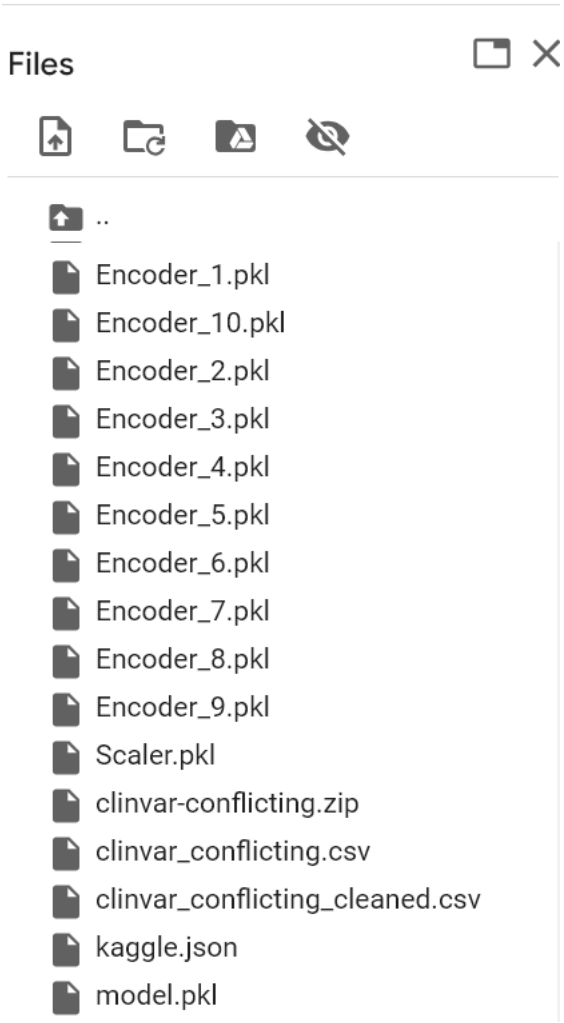
- The User submits the required input from a genetic report
- The data is cleaned and preprocessed and then is used to predict using the trained model
- The final predicting is showcased using the Flask UI

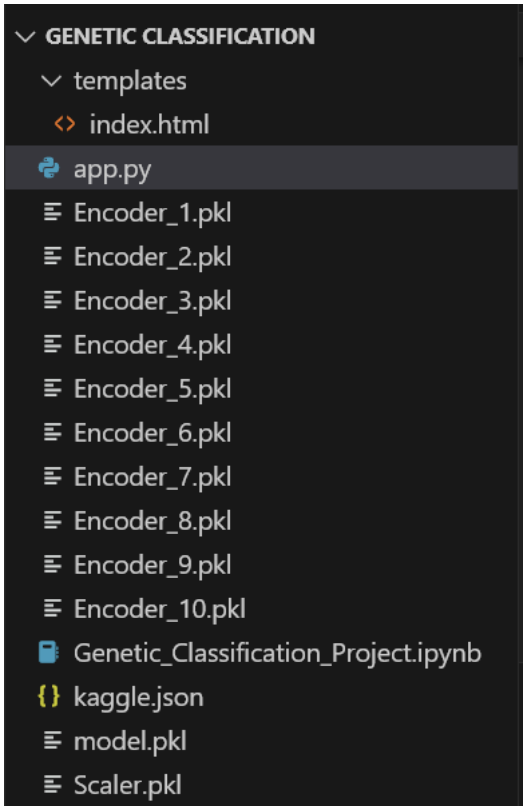
To be able to complete these tasks in this order, we must first set up each individual aspect of it. This includes:

- Data Collection
  - Using the ClinVar database as the source of the data
- Data Preprocessing
  - Data Cleaning to remove unnecessary features or features too complex
  - Encoding Categorical columns
  - Scaling Numeric columns
  - Splitting the dataset into training and testing categories
- Model Evaluation

    - Comparing various different Classification models
    - Using Data Visualization to select appropriate model and to identify key features
- Model Building
    - Importing the model libraries
    - Fitting the training data into the model to initialize the model
- Web Deployment
    - Converting encoders, scalers and model into .pkl files via dumping
    - Downloading those files and transferring them into the application folder
    - Building a HTML file
    - Creating a python application file

Project Structure: -





This is the list of all the final files required

- We need the HTML file saved in the templates folder for the GUI of the application.
- We need the encoders, the scalers and the model taken from the notebook file so as to maintain consistency between the model building file and the application file.
- We need the API file from Kaggle to download the dataset directly instead of having to load it manually

Milestone 1: Data Collection: -

```
[ ]  1 ! pip install -q kaggle
```

```
[ ]  1 !mkdir ~/.kaggle
```

```
[ ]  1 !cp kaggle.json ~/.kaggle
```

```
[ ]  1 !kaggle datasets download -d kevinarvai/clinvar-conflicting
```

```
Warning: Your Kaggle API key is readable by other users on this system! To fix this, you can run 'chmo
Downloading clinvar-conflicting.zip to /content
  0% 0.00/3.59M [00:00<?, ?B/s]
100% 3.59M/3.59M [00:00<00:00, 224MB/s]
```

```
[ ]  1 !unzip /content/clinvar-conflicting.zip
```

```
Archive:  /content/clinvar-conflicting.zip
  inflating: clinvar_conflicting.csv
```

```
[ ]  1 df = pd.read_csv("/content/clinvar_conflicting.csv")
     2 df
```

The above code is used to download and load the dataset into the notebook file. It contains 46 columns, out of which, the CLASS column is the target. The rest are the features. Most of the features are categorical but some of them are numerical.

Link for the dataset on Kaggle: https://www.kaggle.com/datasets/kevinarvai/clinvar-conflicting

Milestone 2: Data Preprocessing

Activity 1: Data Cleaning: -

```
1 df.isnull().sum()
```

```
CHROM                     0
POS                       0
REF                       0
ALT                       0
AF_ESP                    0
AF_EXAC                   0
AF_TGP                    0
CLNDISDB                  0
CLNDISDBINCL          65021
CLNDN                     0
CLNDNINCL             65021
CLNHGVS                   0
CLNSIGINCL            65021
CLNVC                     0
CLNVI                 37529
MC                      846
ORIGIN                    0
SSR                   65058
CLASS                     0
Allele                    0
Consequence               0
IMPACT                    0
SYMBOL                   16
Feature_type             14
Feature                  14
BIOTYPE                  16
EXON                   8893
INTRON                56385
cDNA_position          8884
CDS_position           9955
Protein_position       9955
Amino_acids           10004
Codons                10004
DISTANCE              65080
STRAND                   14
BAM_EDIT              33219
SIFT                  40352
PolyPhen              40392
MOTIF_NAME            65186
MOTIF_POS             65186
HIGH_INF_POS          65186
MOTIF_SCORE_CHANGE    65186
LoFtool                4213
CADD_PHRED             1092
CADD_RAW               1092
BLOSUM62              39595
dtype: int64
```

First, we drop the columns with majority null values.

```
[ ]  1 df = df.drop(['CLNDISDBINCL', 'CLNDNINCL', 'CLNSIGINCL', 'SSR', 'DISTANCE', 'MOTIF_NAME', 'MOTIF_POS', 'HIGH_INF_POS', 'MOTIF_SCORE_CHANGE'], axis=1)
     2 df.head()
```

	CHROM	POS	REF	ALT	AF_ESP	AF_EXAC	AF_TGP	CLNDISDB	CLNDN	
0	1	1168180	G	C	0.0771	0.10020	0.1066	MedGen:CN169374	not_specified	NC_000001.10:g.11
1	1	1470752	G	A	0.0000	0.00000	0.0000	MedGen:C1843891,OMIM:607454,Orphanet:ORPHA9877...	Spinocerebellar_ataxia_21|not_provided	NC_000001.10:g.14
2	1	1737942	A	G	0.0000	0.00001	0.0000	Human_Phenotype_Ontology:HP:0000486,MedGen:C00...	Strabismus|Nystagmus|Hypothyroidism|Intellectu...	NC_000001.10:g.17
3	1	2160305	G	A	0.0000	0.00000	0.0000	MedGen:C1321551,OMIM:182212,SNOMED_CT:83092002...	Shprintzen-Goldberg_syndrome|not_provided	NC_000001.10:g.21
4	1	2160305	G	T	0.0000	0.00000	0.0000	MedGen:C1321551,OMIM:182212,SNOMED_CT:83092002	Shprintzen-Goldberg_syndrome	NC_000001.10:g.21

5 rows × 37 columns

Then we will deal with each of the features individually via several techniques including dichotomization, threshold elimination, etc. The codes will be listed below:

CHROM

This feature contains repeated values for certain numbers in integer and string data type. So we convert it all to string.

```
[ ]  1 df['CHROM'] = df['CHROM'].astype(str)
```

POS

This feature has a large number of unique values. Hence we drop it.

```
[ ]  1 df = df.drop(['POS'], axis=1)
```

REF, ALT, Allele

These features have most of their values in [A, G, T, C]. So we combine the rest into an 'Other' category.

```
[ ]  1 df['REF'] = df['REF'].apply(lambda x: 'Other' if x not in ['A', 'C', 'G', 'T'] else x)
     2 df['ALT'] = df['ALT'].apply(lambda x: 'Other' if x not in ['A', 'C', 'G', 'T'] else x)
     3 df['Allele'] = df['Allele'].apply(lambda x: 'Other' if x not in ['A', 'C', 'G', 'T'] else x)
```

CLNDISDB

This feature contains ID's for diseases in other databases. Contains mostly unique values, so we drop.

```
[ ]  1 df = df.drop(['CLNDISDB'], axis=1)
```

CLNDN

Feature contains various disease names that go into thousands when encoded. We choose to drop.

```
[ ]  1 df = df.drop(['CLNDN'], axis=1)
```

CLNHGVS

This feature has 65188 unique values, hence is meaningless. We drop it.

```
[ ]  df = df.drop(['CLNHGVS'],axis=1)
```

CLNVS

We will combine low frequency classes into an 'Other' category.

```
[ ]  threshold = 100
     clnvs_counts = df['CLNVC'].value_counts()
     low_freq_classes = clnvs_counts[clnvs_counts < threshold].index
```

```
[ ]  df.loc[df['CLNVC'].isin(low_freq_classes), 'CLNVC'] = 'Other'
```

CLNVI, INTRON, BAM_EDIT, SIFT, PolyPhen, BLOSUM62

These features contain approximately 50% null values so instead of dropping them, we will be dichotomizing them.

Dichotomize - Assign all null values as 0 and the rest as 1, so as to distinguish between absent and present values.

```
[ ]  for variable in ['CLNVI', 'INTRON', 'BAM_EDIT', 'SIFT', 'PolyPhen', 'BLOSUM62']:
         df[variable] = df[variable].apply(lambda x: 1 if x == x else 0)
```

MC

Feature contains various variant names that go into thousands when encoded. We choose to drop.

```
[ ]  df = df.drop(['MC'], axis=1)
```

ORIGIN

We will be combining low frequency classes into one group. Since the majority instances are 1, we will categorize the rest as 0.

```
[ ]  threshold = 63940
     value_counts = df['ORIGIN'].value_counts()
     to_combine = value_counts[value_counts < threshold].index
     df['ORIGIN'] = df['ORIGIN'].replace(to_combine, 0)
```

Consequence

Feature contains various variant names that go into thousands when encoded. We choose to drop.

```
[ ]  df = df.drop(['Consequence'], axis=1)
```

SYMBOL

This feature has approximately 2300 unique values so we keep the top 100 as is, and combine the remaining into an 'Other' category.

```
threshold = 100
symbol_counts = df['SYMBOL'].value_counts()
low_freq_classes = symbol_counts[symbol_counts < threshold].index
```

```
df.loc[df['SYMBOL'].isin(low_freq_classes), 'SYMBOL'] = 'Other'
```

Feature_type

Almost all instances except 2 have the same value making this feature redundant. Hence, we drop.

```
df = df.drop(['Feature_type'], axis=1)
```

Feature

This feature contains ID's associated with gene name. We are dropping the numerical code and keeping gene location information intact to retain useful information instead of simply dropping.

```
import re
import numpy as np
df['Feature'] = df['Feature'].astype(str)
df['Feature'] = df['Feature'].apply(lambda x: re.sub(r'\d+', '', x) if isinstance(x, str) else x)
df['Feature'] = df['Feature'].apply(lambda x: x if ('XM' in x or 'NM' in x) else 'Other')
df['Feature'] = df['Feature'].replace('nan', np.nan)
```

```
df['Feature'] = df['Feature'].apply(lambda x: re.findall(r'(NM|XM)', str(x))[0] if re.findall(r'(NM|XM)', str(x)) else 'Other')
```

BIOTYPE

Almost all instances except 14 have the same value making this feature redundant. Hence, we drop.

```
df = df.drop(['BIOTYPE'], axis=1)
```

EXON

We will keep the first 100 values and combine the rest into an 'Other' category.

```
threshold = 100
exon_counts = df['EXON'].value_counts()
low_freq_classes = exon_counts[exon_counts < threshold].index
```

```
df.loc[df['EXON'].isin(low_freq_classes), 'EXON'] = 'Other'
```

cDNA_position, CDS_position, Protein_position

These features contain high amounts of unique values, upwards of 13000 values. So we decide to drop them.

```
[ ]  df = df.drop(['cDNA_position', 'CDS_position', 'Protein_position'], axis=1)
```

Amino_acids

This contains approximately 10000 null values. We will separate the top 100 individually, and group the remaining as 'Other' category including null values.

```
[ ]  top_100_list = df['Amino_acids'].value_counts()[0:100].index
```

```
[ ]  df['Amino_acids'] = df['Amino_acids'].apply(lambda x: x if x in top_100_list else 'Other')
```

Codons

This category contains approximately 10000 null values. If we keep the top 100 and group the rest, then we get extremely inbalanced data, with 'Other' containing 32000 values. So we drop it.

```
[ ]  df = df.drop(['Codons'], axis=1)
```

STRAND

It has 14 null values, so we drop the rows with the null values.

```
[ ]  df = df[df['STRAND'].notna()]
```

LoFtool, CADD_RAW, CADD_PHRED

We will replace the null values with the median for the respective feature.

```
[ ]  df['LoFtool'] = df['LoFtool'].replace(np.NaN, df['LoFtool'].median())
     df['CADD_RAW'] = df['CADD_RAW'].replace(np.NaN, df['CADD_RAW'].median())
     df['CADD_PHRED'] = df['CADD_PHRED'].replace(np.NaN, df['CADD_PHRED'].median())
```

This marks the end for data cleaning. We will save the cleaned dataset as a csv so that we can refer for any future requirement.

```
[ ]  df.to_csv('clinvar_conflicting_cleaned.csv')
```

Activity 2: Encoding the categorical columns: -

Here, we chose to use separate encoder objects for each categorical column so that when we receive the user input, we can use the same encoder to perform preprocessing upon it. It saves up time to prevent re-training the dataset in the application file, but consumes more space as the encoders will consume individual memory.

```
[ ]  1 from sklearn.preprocessing import LabelEncoder
     2 encoder_1 = LabelEncoder()
     3 encoder_2 = LabelEncoder()
     4 encoder_3 = LabelEncoder()
     5 encoder_4 = LabelEncoder()
     6 encoder_5 = LabelEncoder()
     7 encoder_6 = LabelEncoder()
     8 encoder_7 = LabelEncoder()
     9 encoder_8 = LabelEncoder()
    10 encoder_9 = LabelEncoder()
    11 encoder_10 = LabelEncoder()
```

```
[ ]  1 df['CHROM'] = encoder_1.fit_transform(df['CHROM'])
     2 df['REF'] = encoder_2.fit_transform(df['REF'])
     3 df['ALT'] = encoder_3.fit_transform(df['ALT'])
     4 df['CLNVC'] = encoder_4.fit_transform(df['CLNVC'])
     5 df['Allele'] = encoder_5.fit_transform(df['Allele'])
     6 df['IMPACT'] = encoder_6.fit_transform(df['IMPACT'])
     7 df['SYMBOL'] = encoder_7.fit_transform(df['SYMBOL'])
     8 df['Feature'] = encoder_8.fit_transform(df['Feature'])
     9 df['EXON'] = encoder_9.fit_transform(df['EXON'])
    10 df['Amino_acids'] = encoder_10.fit_transform(df['Amino_acids'])
```

Activity 3: Scaling the numerical columns: -

Once the categorical columns are encoded, we can begin scaling the numerical columns. Similar to encoding, we will be making use of the same scaler between our notebook file and the application folder. Before we begin scaling, we must split our target columns so that it doesn't get affect by the scaling.

X and Y Split

```
[ ]  1 X = df.drop(['CLASS'], axis=1)
```

```
[ ]  1 Y = df['CLASS']
```

Scaling the Dataset

```
[ ]  1 from sklearn.preprocessing import MinMaxScaler
     2 scaler = MinMaxScaler()
```

```
[ ]  1 X_scaled = pd.DataFrame(scaler.fit_transform(X), columns=X.columns)
```

Activity 4: Splitting the dataset into training and testing categories: -

Once the dataset is completely preprocessed, we can split it into train and test datasets. The training set is used to make the model while the test set is used to check the evaluation of the model.

Train Test Split

```
[ ]  from sklearn.model_selection import train_test_split
```

```
[ ]  X_train, X_test, y_train, y_test = train_test_split(X_scaled, Y, test_size=0.2, random_state=42)
```

Milestone 3: Model Evaluation: -

Activity 1: Comparing Classifiers: -

i. We will begin this step by calling all the classification models and creating a dictionary through which we can pipeline the evaluation process.

Defining the Classifiers

```
[ ]  from sklearn.pipeline import Pipeline
     from sklearn.linear_model import LogisticRegression
     from sklearn.tree import DecisionTreeClassifier
     from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, GradientBoostingClassifie
     from sklearn.svm import SVC
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.naive_bayes import GaussianNB
     from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

```
[ ]  classifiers = {
         'Logistic Regression': LogisticRegression(solver='saga', max_iter=1000),
         'Decision Tree': DecisionTreeClassifier(),
         'Random Forest': RandomForestClassifier(),
         'AdaBoost': AdaBoostClassifier(),
         'Gradient Boosting': GradientBoostingClassifier(),
         'SVM': SVC(),
         'KNN': KNeighborsClassifier(),
         'Naive Bayes': GaussianNB(),
     }
```

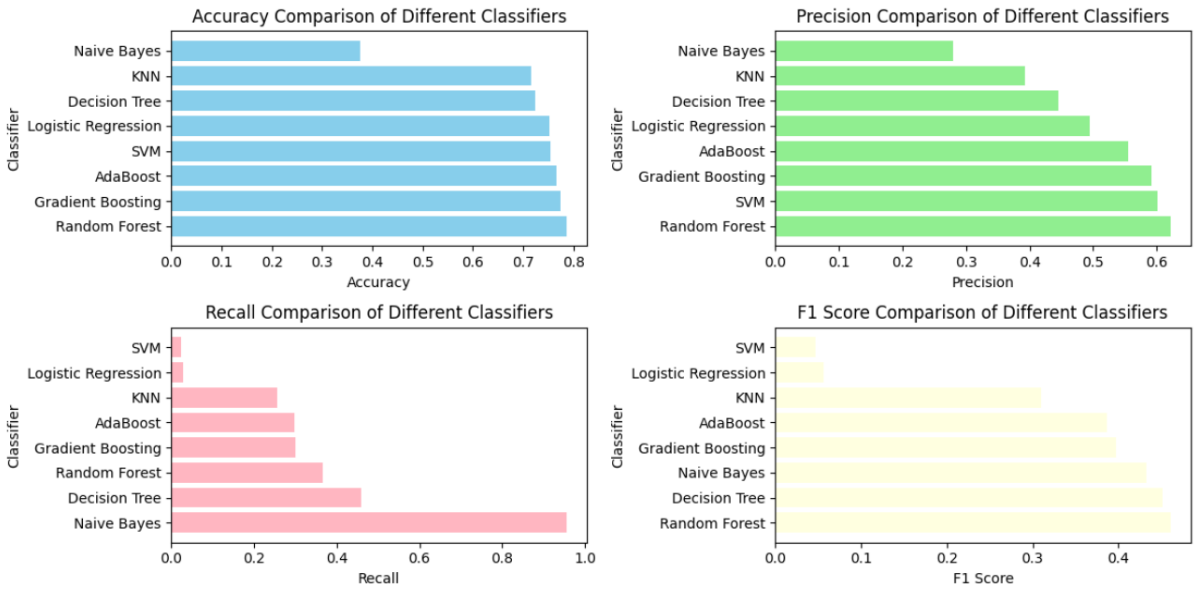
ii. Once they have been collected in the dictionary, we calculate the accuracy, precision, recall and f1 score of all the classifiers and figure out the most appropriate one for our purpose.

Calculating accuracy and precision for each classifier

```
[ ]  accuracy_results = []
     precision_results = []
     recall_results = []
     f1_score_results = []

     for key, classifier in classifiers.items():
         pipeline = Pipeline(steps=[('classifier', classifier)])
         pipeline.fit(X_train, y_train)
         y_pred = pipeline.predict(X_test)
         accuracy = accuracy_score(y_test, y_pred)
         precision = precision_score(y_test, y_pred)
         recall = recall_score(y_test, y_pred)
         f1 = f1_score(y_test, y_pred)
         accuracy_results.append((key, accuracy))
         precision_results.append((key, precision))
         recall_results.append((key, recall))
         f1_score_results.append((key, f1))
```

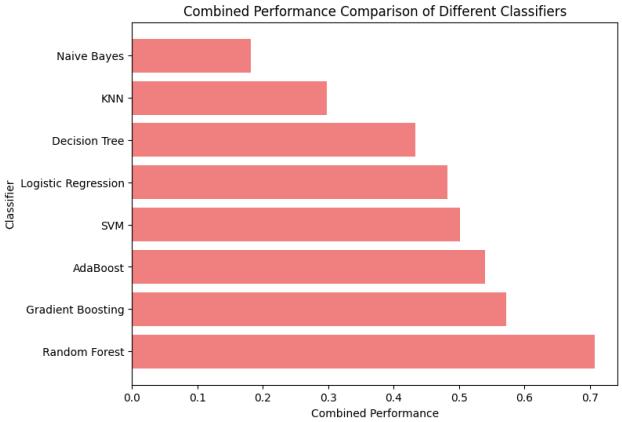
iii. After this, we graph the results of each of the metrics so see which classifiers perform the best among the ones we considered.

```
accuracy_results.sort(key=lambda x: x[1], reverse=True)
precision_results.sort(key=lambda x: x[1], reverse=True)
recall_results.sort(key=lambda x: x[1], reverse=True)
f1_score_results.sort(key=lambda x: x[1], reverse=True)


labels, values = zip(*accuracy_results)
plt.figure(figsize=(12, 6))
plt.subplot(2, 2, 1)
plt.barh(labels, values, color='skyblue')
plt.xlabel('Accuracy')
plt.ylabel('Classifier')
plt.title('Accuracy Comparison of Different Classifiers')

labels, values = zip(*precision_results)
plt.subplot(2, 2, 2)
plt.barh(labels, values, color='lightgreen')
plt.xlabel('Precision')
plt.ylabel('Classifier')
plt.title('Precision Comparison of Different Classifiers')

labels, values = zip(*recall_results)
plt.subplot(2, 2, 3)
plt.barh(labels, values, color='lightpink')
plt.xlabel('Recall')
plt.ylabel('Classifier')
plt.title('Recall Comparison of Different Classifiers')

labels, values = zip(*f1_score_results)
plt.subplot(2, 2, 4)
plt.barh(labels, values, color='lightyellow')
plt.xlabel('F1 Score')
plt.ylabel('Classifier')
plt.title('F1 Score Comparison of Different Classifiers')

plt.tight_layout()
plt.show()

```



iv. We also created a combined graph which took the average of all the values combined and gave us the approximate most accurate classifier.

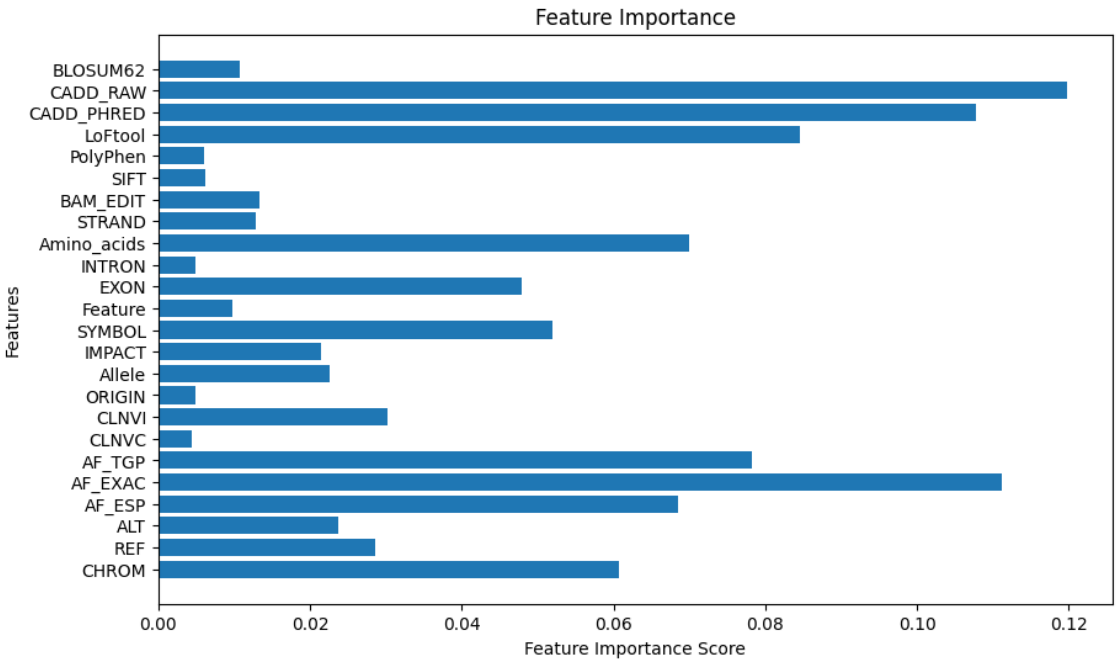
```
[ ]  combined_results = [(label, (accuracy + precision + recall + f1) / 4) for (label, accuracy), (_, precision), (_, recall), (_, f1) in zip(accuracy_results, precision_results,
     combined_results.sort(key=lambda x: x[1], reverse=True)

     labels, values = zip(*combined_results)
     plt.figure(figsize=(8, 6))
     plt.barh(labels, values, color='lightcoral')
     plt.xlabel('Combined Performance')
     plt.ylabel('Classifier')
     plt.title('Combined Performance Comparison of Different Classifiers')
     plt.show()
```



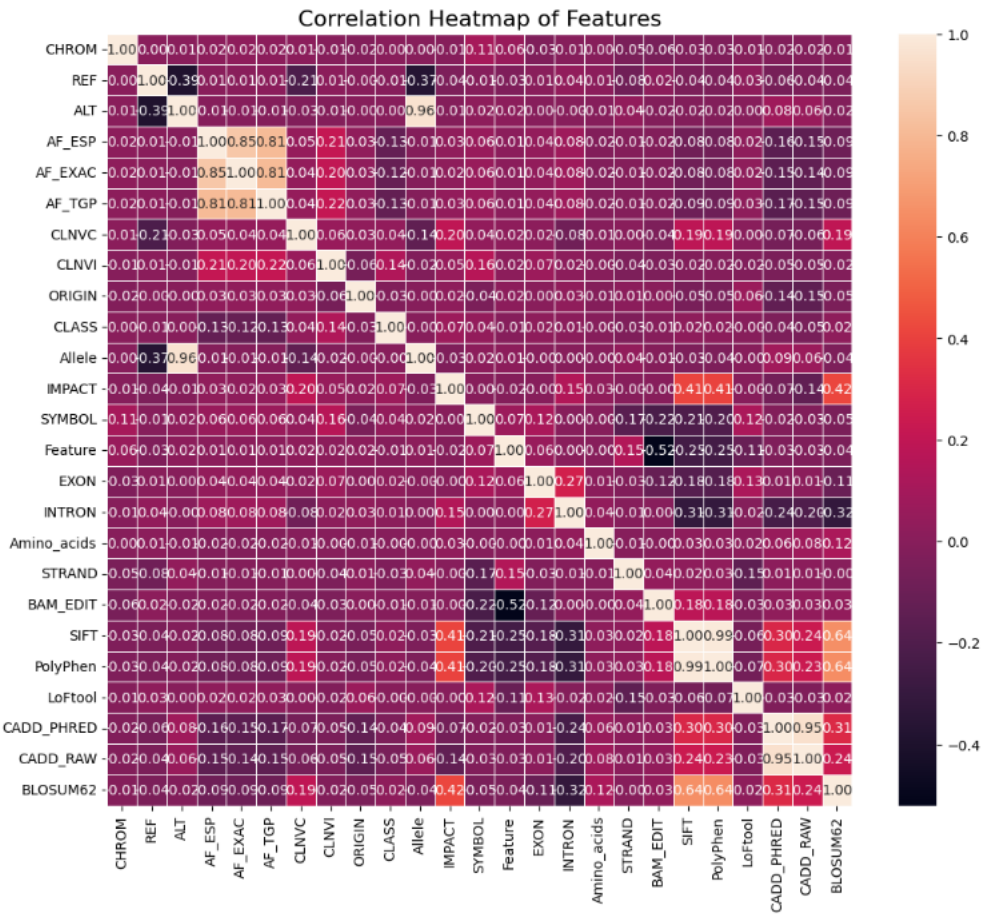
Activity 2: Data Visualization: -

Once we have figured out the best classification model, in this case, the Random Forest Classifier, we an see the feature importance for that model and decide on the important features using correlation graphs.

```
[ ]  plt.figure(figsize=(10, 6))
     model = RandomForestClassifier()
     model.fit(X, Y)
     feature_importance = model.feature_importances_
     feature_names = X.columns
     plt.barh(feature_names, feature_importance)
     plt.xlabel('Feature Importance Score')
     plt.ylabel('Features')
     plt.title('Feature Importance')
     plt.show()
```



```
[ ]  1 plt.figure(figsize=(12, 10))
     2 sns.heatmap(df[columns].corr(), annot=True, fmt='.2f', linewidths=0.5)
     3 plt.title('Correlation Heatmap of Features', fontsize=16)
     4 plt.show()
```



Milestone 4: Model Building: -

We call the required libraries and then fit the training data into the model.

Model Building

```
[ ]  1 rfc = RandomForestClassifier(n_estimators=100)
     2 rfc.fit(X_train, y_train)
```

```
▾ RandomForestClassifier
RandomForestClassifier()
```

```
[ ]  1 y_pred = rfc.predict(X_test)
     2 print(y_pred)
```

```
[1 0 0 ... 0 0 0]
```

Milestone 5: Web Deployment: -

Activity 1: Dumping into pickle files: -

Our first step is to convert the encoders, the scaler and the model into pickle files so that we can import them from our notebook to our application folder. This is necessary so that there are no inconsistencies between the training of the model and the prediction made by the UI we provide.

Dumping into a pickle file

```
[ ]  import pickle
```

```
[ ]  pickle.dump(rfc, open('model.pkl', 'wb'))
```

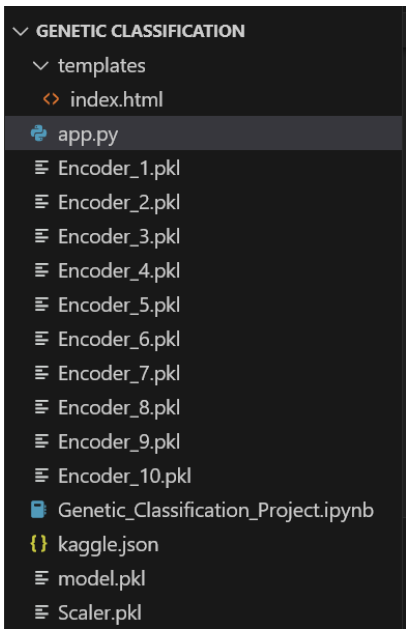
```
[ ]  pickle.dump(scaler, open('Scaler.pkl', 'wb'))
```

```
[ ]  pickle.dump(encoder_1, open('Encoder_1.pkl', 'wb'))
     pickle.dump(encoder_2, open('Encoder_2.pkl', 'wb'))
     pickle.dump(encoder_3, open('Encoder_3.pkl', 'wb'))
     pickle.dump(encoder_4, open('Encoder_4.pkl', 'wb'))
     pickle.dump(encoder_5, open('Encoder_5.pkl', 'wb'))
     pickle.dump(encoder_6, open('Encoder_6.pkl', 'wb'))
     pickle.dump(encoder_7, open('Encoder_7.pkl', 'wb'))
     pickle.dump(encoder_8, open('Encoder_8.pkl', 'wb'))
     pickle.dump(encoder_9, open('Encoder_9.pkl', 'wb'))
     pickle.dump(encoder_10, open('Encoder_10.pkl', 'wb'))
```

Activity 2: Downloading and transferring into application folder: -

Once compressed, we can download the files and then transfer them into the python application folder from where we can extract.

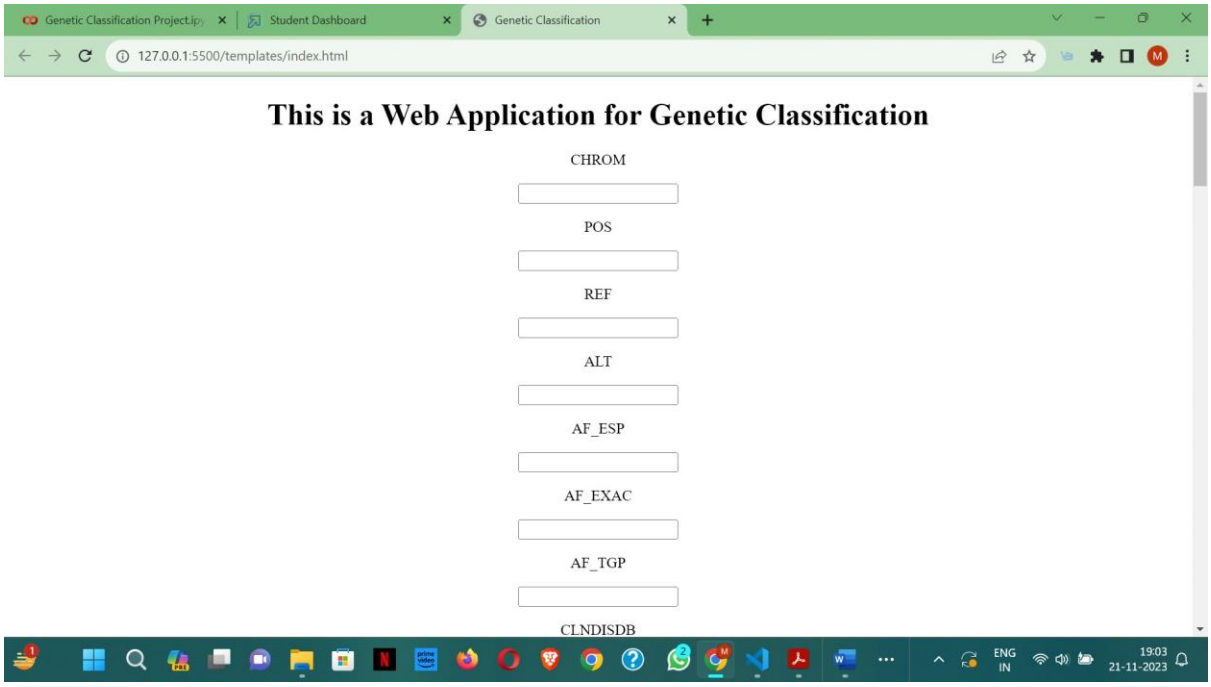
Once the transferring is done, the final application folder should look the following:



Activity 3: Creating HTML template: -

We create the front-end part of our web page using HTML

We use a single template called index.html, and it is simply used to receive input from user and to display the predicted output. This is how it looks like:



Activity 4: Creating python application file: -

i. Importing the necessary libraries

```
from flask import Flask, render_template, request
import pickle
import numpy as np
import pandas as pd
```

ii. Creating the flask application and loading our encoders, scaler and model

```
app = Flask(__name__)

model = pickle.load(open('model.pkl', 'rb'))
scaler = pickle.load(open('Scaler.pkl', 'rb'))
encoder_1 = pickle.load(open('Encoder_1.pkl', 'rb'))
encoder_2 = pickle.load(open('Encoder_2.pkl', 'rb'))
encoder_3 = pickle.load(open('Encoder_3.pkl', 'rb'))
encoder_4 = pickle.load(open('Encoder_4.pkl', 'rb'))
encoder_5 = pickle.load(open('Encoder_5.pkl', 'rb'))
encoder_6 = pickle.load(open('Encoder_6.pkl', 'rb'))
encoder_7 = pickle.load(open('Encoder_7.pkl', 'rb'))
encoder_8 = pickle.load(open('Encoder_8.pkl', 'rb'))
encoder_9 = pickle.load(open('Encoder_9.pkl', 'rb'))
encoder_10 = pickle.load(open('Encoder_10.pkl', 'rb'))
```

iii. Routing the HTML page along with accepting user input, data preprocessing

```python
@app.route('/')
def start():
    return render_template('index.html')


@app.route('/login', methods=['POST'])
def login():
    chrom = request.form["ch"] or np.NaN
    pos = request.form["ps"] or np.NaN
    ref = request.form["rf"] or np.NaN
    alt = request.form["at"] or np.NaN
    af_esp = request.form["ap"] or np.NaN
    af_exac = request.form["ac"] or np.NaN
    af_tgp = request.form["ag"] or np.NaN
    clndisdb = request.form["cb"] or np.NaN
    clndisdbincl = request.form["cl"] or np.NaN
    clndn = request.form["cn"] or np.NaN
    clndnincl = request.form["cc"] or np.NaN
    clnhgvs = request.form["cs"] or np.NaN
    clnsigincl = request.form["ci"] or np.NaN
    clnvc = request.form["cv"] or np.NaN
    clnvi = request.form["li"] or np.NaN
    mc = request.form["mc"] or np.NaN
    origin = request.form["on"] or np.NaN
    ssr = request.form["sr"] or np.NaN
    allele = request.form["ae"] or np.NaN
    consequence = request.form["cq"] or np.NaN
    impact = request.form["im"] or np.NaN
    symbol = request.form["sl"] or np.NaN
```

```python
    if ref not in ['A', 'C', 'G', 'T']:
        ref = 'Other'
    if alt not in ['A', 'C', 'G', 'T']:
        alt = 'Other'
    if allele not in ['A', 'C', 'G', 'T']:
        allele = 'Other'

    if clnvc in ['Insertion', 'Inversion', 'Microsatellite']:
        clnvc = 'Other'

    if pd.isnull(clnvi):
        clnvi = 0
    else:
        clnvi = 1

    if pd.isnull(intron):
        intron = 0
    else:
        intron = 1

    if pd.isnull(bam_edit):
        bam_edit = 0
    else:
        bam_edit = 1

    if pd.isnull(sift):
        sift = 0
```

```python
    CHROM_encoded = encoder_1.transform(np.array([chrom]))
    REF_encoded = encoder_2.transform(np.array([ref]))
    ALT_encoded = encoder_3.transform(np.array([alt]))
    CLNVC_encoded = encoder_4.transform(np.array([clnvc]))
    Allele_encoded = encoder_5.transform(np.array([allele]))
    IMPACT_encoded = encoder_6.transform(np.array([impact]))
    SYMBOL_encoded = encoder_7.transform(np.array([symbol]))
    Feature_encoded = encoder_8.transform(np.array([feature]))
    EXON_encoded = encoder_9.transform(np.array([exon]))
    Amino_acids_encoded = encoder_10.transform(np.array([amino_acids]))

    t = [
    [
        float(CHROM_encoded), float(REF_encoded), float(ALT_encoded), float(af_esp), float(af_exac), float
        float(clnvi), float(origin), float(Allele_encoded), float(IMPACT_encoded),
        float(SYMBOL_encoded), float(Feature_encoded), float(EXON_encoded), float(intron), float(Amino_aci
        float(bam_edit), float(sift), float(polyphen), float(loftool), float(cadd_phred), float(cadd_raw),
        float(blosum62)
    ]
    ]

    t_scaled = scaler.transform(t)
```

iv. Predicting the output and displaying it on the web page, along with running the application

```python
    output = model.predict(t_scaled)
    print(output)

    return render_template('index.html', y="The predicted answer is "+str(output[0]))


if __name__ == '__main__':
    app.run(debug=True)
```

v. Once the above steps are completed, we can run the python file and it will be hosted on the localhost 5000 with the following link http://127.0.0.1:5000. (Note: This link may change upon re-running so make sure to run on your own)

```
PS C:\Users\Public\Programming Codes\Genetic Classification> & C:/Users/moham/AppData/Local/Programs/Python/Pyt
hon311/python.exe "c:/Users/Public/Programming Codes/Genetic Classification/app.py"
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server i
nstead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 532-608-058
```

vi. The final output can be seen below