Harshvardhan Thakur, 24231450, Computer Science Data Analytics

1. Understanding the Dataset

Given dataset is Perinatal Risk Information with some health parameters to classify women's maternal health using machine learning. Given data has Range Index of 1014 (0 to 1013) and total of 7 data columns with different datatypes as shown in Fig 1.1. Some columns such as Age, SystolicBP, DiastolicBP are independent variable and Type column is dependent variable and it is of categorical data, hence Classification model is preferable for this problem.

```
[10] df.info()
     ✓ 0.0s

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1014 entries, 0 to 1013
Data columns (total 7 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Age          1014 non-null   int64
 1   SystolicBP   1014 non-null   int64
 2   DiastolicBP  1014 non-null   int64
 3   BS           1014 non-null   float64
 4   BodyTemp     1014 non-null   float64
 5   HeartRate    1014 non-null   int64
 6   Type         1014 non-null   object
dtypes: float64(2), int64(4), object(1)
memory usage: 55.6+ KB
```

Fig 1.1 Dataframe information.

```
[125] df['Type'].value_counts()
      ✓ 0.0s

low risk     406
mid risk     336
high risk    272
Name: Type, dtype: int64
```

Fig. 2.1 Dependent Variable

2. Data Exploration and Pre-processing

- Dataset is balanced since the dependent variable 'Type' has roughly same number of instances. The categorical values ranging between 26.82% to 40.03% as shown in Fig. 2.1.
- There are no missing values in the data set as shown in Fig. 2.2, thus imputation is not required for this dataset.
- On further exploring, total of 562 duplicated values are present in the dataset as shown in Fig. 2.3.

```
[126] df.isnull().sum()
      ✓ 0.0s

Age            0
SystolicBP     0
DiastolicBP    0
BS             0
BodyTemp       0
HeartRate      0
Type           0
dtype: int64
```

Fig. 2.2 No Missing Values

```
[127] df.duplicated().sum()
      ✓ 0.0s

562
```

Fig. 2.3 Duplicated Values

- **Histplot:** sns.histplot() function to plot each attributes against 'Type' dependent variable to categorize the datapoints as shown in Fig. 2.4& 2.5 below, but from the graphs it is clearly visible datapoints are not easily separable into high, low, mid risk categories hence we use machine learning model that captures complexity such dataset.

Fig. 2.4 Age vs Type

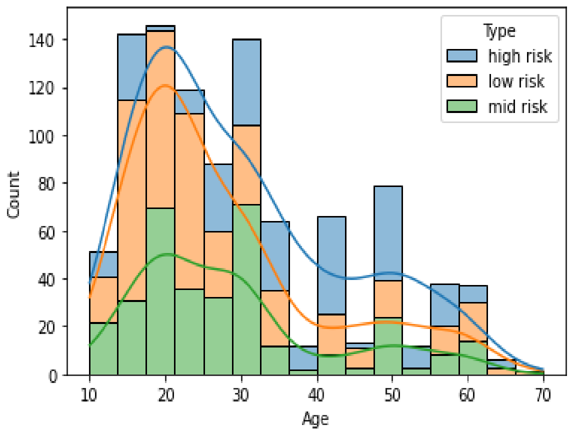
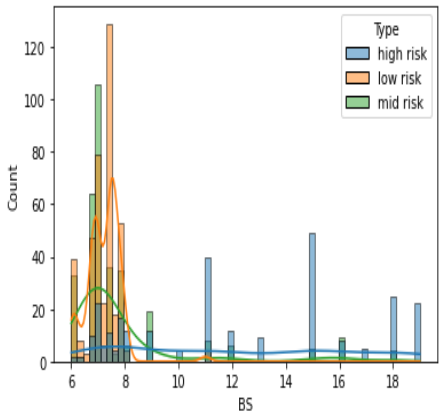


Fig. 2.5 BS vs Type



- Before Model Selection, we need to check for outliers as they can overfit the model performance. Many methods can be used to check for outliers such as IQR, line plots, z-score method etc. We made a custom function that takes each attribute and plots a line graph to check for outliers.

```
#Checking outliers using line graphs

def custom_graph(df):
    for column in df.columns:
        plt.figure(figsize=(10,6))
        plt.plot(df.index ,df[column], label=f'{column} values', linewidth=1)


        plt.title(f'{column} Line Plot')
        plt.xlabel('Values')
        plt.ylabel(column)
        plt.legend()

        plt.show()
custom_graph(df)
```
[128] ✓ 1.3s

Fig. 2.6 Custom Outlier Function

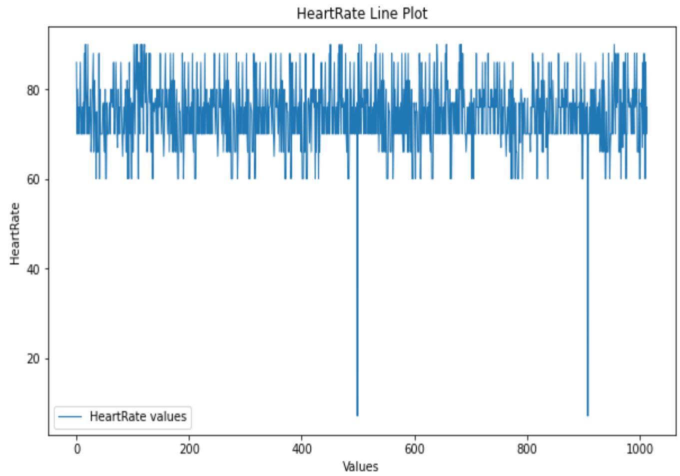


Fig. 2.7 Outlier Detection in HeartRate

- Although there's no significant change in model performance upon removing the outliers.
- **Correlation:** Correlation matrix of attributes to check which attributes are strongly correlated. SystolicBP and DiastolicBP have strong correlation coefficient. Although, matrix suggests that SystolicBP and DiastolicBP are strongly correlated and can be considered same in machine learning domain, however in medical domain both type of blood pressure is different and have some factors to them. Hence, we keep both the attributes.

```
cor_matrix=df.corr()
print(cor_matrix)
#SystolicBP and DiastolicBP have strong correlation coefficient
```
[47] ✓ 0.0s

	Age	SystolicBP	DiastolicBP	BS	BodyTemp	HeartRate
Age	1.000000	0.416045	0.398026	0.473284	-0.255323	0.079798
SystolicBP	0.416045	1.000000	0.787006	0.425172	-0.286616	-0.023108
DiastolicBP	0.398026	0.787006	1.000000	0.423824	-0.257538	-0.046151
BS	0.473284	0.425172	0.423824	1.000000	-0.103493	0.142867
BodyTemp	-0.255323	-0.286616	-0.257538	-0.103493	1.000000	0.098771
HeartRate	0.079798	-0.023108	-0.046151	0.142867	0.098771	1.000000

Fig. 2.8 Correlation Matrix.

3. Algorithm Selection and Application

- **Splitting the Data:** Before we feed dataset to the model, we split dataset into train, validation and test sets using train_test_split. First, data is split into test and temp and then further we split temp into train and validation data as shown in Fig. 3.1. Validation set prevents model from memorising the training set.

```
from sklearn.model_selection import train_test_split
```
[53] ✓ 0.0s

```
X_temp, X_test, y_temp, y_test = train_test_split(X, y, test_size=.20, random_state=1)
```
[54] ✓ 0.0s

```
X_train, X_val, y_train, y_val= train_test_split(X_temp, y_temp, test_size=.20, random_state=1)
```
[55] ✓ 0.0s

Fig. 3.1 Train Test Split.

- The problem is categorical, classification model such as Decision Tree and Random Forest perform better in such problems. Decision Tree handles binary classification as well as multi-class classification since the dependent variable is multi-class, and Decision Tree does not require data transformation and it handles imbalance data as well. However, Decision Tree can sometimes overfit to overcome that issue Random Forest model is used

```
Decision tree

[59]
dt= DecisionTreeClassifier(criterion='gini',random_state=1)
dt.fit(X_train,y_train)
✓ 0.0s
... DecisionTreeClassifier(random_state=1)

[88]
TS_DT= dt.score(X_train,y_train)
print("Training Score:", TS_DT)
✓ 0.0s
... Training Score: 0.9243827160493827

[90]
VS_DT= dt.score(X_val,y_val)
print("Validation Score:", VS_DT)
✓ 0.0s
... Validation Score: 0.8098159509202454
```

Fig. 3.2 Decision Tree Implementation

```
Random Forest

[101]
random_f = RandomForestClassifier(n_estimators = 50, random_state=1,max_features=5)
random_f = random_f.fit(X_train, y_train)
✓ 0.0s

[102]
TS=random_f.score(X_train, y_train)
print("Trainig Score:", TS)
✓ 0.0s
... Trainig Score: 0.9243827160493827

[103]
VS= random_f.score(X_val,y_val)
print("Validation Score:", VS)
✓ 0.0s
... Validation Score: 0.8159509202453987
```

Fig. 3.3 Random Forest Implementation

- **Hyperparameter Tuning:** RandomizedSearchCV is one of the hyperparameter which selects samples from the combinations of hyperparameter compared to which GridSearchCV tests all the possible combinations of hyperparameter. An n_estimator defines values of decision trees in random forest ensemble; for moderate dataset size values like 100,200,300 can help generalization of model.
  Upon fitting the random_search model, it fits data 3 times (as cv=3) and iterates over 100; making 300 fits.

```
#HYPERPARAMETER TUNING

from sklearn.model_selection import RandomizedSearchCV
✓ 0.0s

param_dist = {
    'n_estimators': [100, 200, 350, 475],
    'max_depth': [10, 20, 30, 40, None],      #None means no allowing tree
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False],
    'max_features': ['auto', 'sqrt', 'log2']
}
✓ 0.0s

random_search = RandomizedSearchCV(estimator=RandomForestClassifier(),
                                   param_distributions=param_dist,
                                   n_iter=100, cv=3, verbose=2,
                                   random_state=42, n_jobs=-1)
```

Fig. 3.4 Hyperparameter Space

```
random_search.fit(X_train, y_train)
✓ 28.6s                                                              Python
Fitting 3 folds for each of 100 candidates, totalling 300 fits

RandomizedSearchCV(cv=3, estimator=RandomForestClassifier(), n_iter=100,
                   n_jobs=-1,
                   param_distributions={'bootstrap': [True, False],
                                        'max_depth': [10, 20, 30, 40, None],
                                        'max_features': ['auto', 'sqrt',
                                                         'log2'],
                                        'min_samples_leaf': [1, 2, 4],
                                        'min_samples_split': [2, 5, 10],
                                        'n_estimators': [100, 200, 350, 475]},
                   random_state=42, verbose=2)

print("Parameters for RandomSearch: ", random_search.best_params_)
print("Score for RandomSearch: ", random_search.best_score_)
✓ 0.0s                                                               Python
Parameters for RandomSearch:  {'n_estimators': 100, 'min_samples_split': 5, 'min_samples_leaf': 1, 'max_features': 'log2', 'max_depth': None, 'bootstra
Score for RandomSearch:  0.7824074074074074
```

Fig. 3.5 Best Parameters and Score for RandomSearch

- Training, Validation and Testing Accuracy comes out to be 0.924 and 0.834 and 0.835 respectively indicating a good fit and model performance for Random Forest.

```
[290]
from sklearn.metrics import accuracy_score
✓ 0.0s

[291]
best_rf = random_search.best_estimator_
y_pred_h1 = best_rf.predict(X_train)

# Evaluate accuracy or any other metric
accuracy = accuracy_score(y_train, y_pred_h1)
print("Training Accuracy: ", accuracy)
✓ 0.0s
... Training Accuracy:  0.9243827160493827

[292]
best_rf = random_search.best_estimator_
y_pred_h2 = best_rf.predict(X_val)

# Evaluate accuracy or any other metric
accuracy = accuracy_score(y_val, y_pred_h2)
print("Validation Accuracy: ", accuracy)
✓ 0.0s
... Validation Accuracy:  0.8343558282208589
```

Fig. 3.6 Training and Validation Accuracy Of Random Forest after Tuning.

```
[316]
print(f"Classification Report for Testing Data:\n{classification_report(y_test, y_pred_test)}")
✓ 0.0s

... Classification Report for Testing Data:
              precision    recall  f1-score   support

   high risk       0.88      0.92      0.90        63
    low risk       0.83      0.82      0.82        77
    mid risk       0.75      0.73      0.74        63

    accuracy                           0.82       203
   macro avg       0.82      0.82      0.82       203
weighted avg       0.82      0.82      0.82       203
```

Fig. 4.1 Decision Tree Classification Report

```
best_rf = random_search.best_estimator_
y_pred_h3 = best_rf.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred_h3)
print("Testing Accuracy: ", accuracy)
```

```
Testing Accuracy:  0.8325123152709359
```

Fig. 3.7 Testing Accuracy

4. <u>**Model Evaluation and Comparative Analysis**</u>

- **Decision Tree and Random Forest (Tuned) comparison;** Decision Tree Classification report can be seen in Fig. 4.1, it's performance on training dataset and validation dataset comes out be 0.92 and 0.80 respectively which can be seen in the submitted code file. Fig. 4.2 shows classification report of Random Forest after Hyperparameter Tuning. Thus, on comparing the scores of both model their performance scores come out to be almost similar as shown in bar plot Fig. 4.3

```
print(f"Random Forest Tuning Report:\n{classification_report(y_test, y_pred_h3)}")
```

```
Random Forest Tuning Report:
              precision    recall  f1-score   support

   high risk       0.88      0.92      0.90        63
    low risk       0.84      0.83      0.84        77
    mid risk       0.77      0.75      0.76        63

    accuracy                           0.83       203
   macro avg       0.83      0.83      0.83       203
weighted avg       0.83      0.83      0.83       203
```

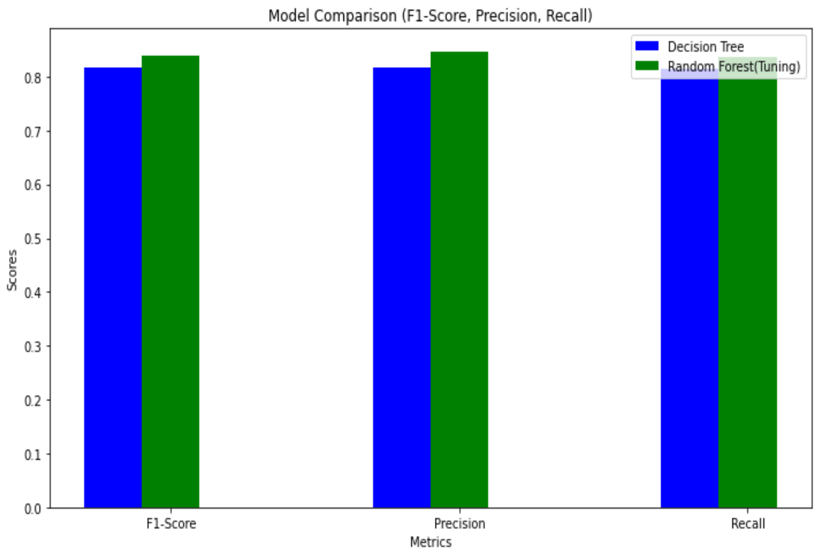


Fig. 4.2 Random Forest Hyperparameter Tuning Classification Report

Fig. 4.3 Model Comparison

5. <u>**Ethical Considerations**</u>

- For ML model to perform better, good dataset must be used with different attributes. We can't be sure whether this model will work for wider population as this model is trained on only 7 attributes. If such Machine Learning starts predicting wrong results; as consequences people might suffer major health issues.