Prediction of Parkinson’s Disease using Audio Frequency Analysis and Machine Learning

MODEL CREATION CODE:

**# *Import libraries***

import numpy as np

import pandas as pd

import plotly.express as px

import matplotlib.pyplot as plt

from sklearn import svm

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.naive\_bayes import GaussianNB

from sklearn.linear\_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.ensemble import GradientBoostingClassifier, AdaBoostClassifier, RandomForestClassifier, ExtraTreesClassifier

from xgboost import XGBClassifier

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy\_score, classification\_report

from imblearn.over\_sampling import SMOTE

import joblib

***# Ignore all warnings***

import warnings

warnings.filterwarnings('ignore')

df = pd.read\_csv("Parkinsson disease.csv")

df.drop(['name'], axis=1, inplace=True)

df

***# Exploratory Data Analysis (EDA)***

px.histogram(df, x="status", color="status", width=500, height=500).show()

***# Correlation matrix before balancing the data.***

df\_corr = df.corr()

fig = px.imshow(df\_corr, x=df\_corr.columns, y=df\_corr.columns, title='Correlation Matrix',

                labels=dict(x="Features", y="Features", color="Correlation"), height=1500, width=1500, text\_auto=True)

fig.update\_yaxes(automargin=True)

fig.show()

***# Handling imbalanced data using SMOTE***

smote = SMOTE()

X = df[df.drop('status', axis=1).columns]

y = df['status']

X\_smote, y\_smote = smote.fit\_resample(X, y)

***# New DataFrame after SMOTE***

df\_new = X\_smote.assign(status=y\_smote)

px.histogram(df\_new, x="status", color="status", width=500, height=500).show()

X = df\_new.drop(['status'], axis=1)

y = df\_new['status']

***# Feature Selection using Correlation***

corr\_matrix = X.corr().abs()

upper = corr\_matrix.where(np.triu(np.ones(corr\_matrix.shape), k=1).astype(bool))

selected\_columns = [column for column in upper.columns if any(upper[column] > 0.8)]

print("Selected columns:", selected\_columns)

X = X[selected\_columns]

***# Normalizing the data using StandardScaler***

scaler = StandardScaler()

X\_scaled = pd.DataFrame(scaler.fit\_transform(X), columns=X.columns)

X\_scaled

***# Split the data into training and testing sets***

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.20, shuffle=True, stratify=y)

***# Models and hyperparameters***

models = [

    ('Random Forest', RandomForestClassifier(), {

        'n\_estimators': [100, 200, 300],

        'max\_depth': [None, 10, 20],

        'min\_samples\_leaf': [1, 2],

        'min\_samples\_split': [2, 4],

        'criterion': ['gini', 'entropy'],

        'n\_jobs': [-1],

    }),

    ('Extra Trees', ExtraTreesClassifier(), {

        'n\_estimators': [100, 200, 300],

        'max\_depth': [None, 10, 20],

        'min\_samples\_leaf': [1, 2],

        'min\_samples\_split': [2, 4],

        'criterion': ['gini', 'entropy'],

        'n\_jobs': [-1],

    }),

    ('SVM', svm.SVC(), {

        'C': [0.1, 1, 10],

        'kernel': ['linear', 'rbf'],

        'gamma': ['scale', 'auto'],

}),

    ('Gradient Boosting', GradientBoostingClassifier(), {

        'n\_estimators': [100, 200, 300],

        'max\_depth': [3, 5],

        'learning\_rate': [0.01, 0.1],

    }),

    ('Decision Tree', DecisionTreeClassifier(), {

        'max\_depth': np.arange(3, 16, 2),

        'min\_samples\_leaf': [3, 5, 10],

        'min\_samples\_split': [8, 10, 12],

        'criterion': ['gini', 'entropy'],

    }),

    ('KNN', KNeighborsClassifier(), {

        'n\_neighbors': np.arange(2, 21, 2),

        'weights': ['uniform', 'distance'],

        'algorithm': ['auto', 'ball\_tree', 'kd\_tree', 'brute'],

        'leaf\_size': [5, 10, 15],

    }),

    ('Logistic Regression', LogisticRegression(), {

        'penalty': ['l1', 'l2'],

        'C': [0.01, 0.1, 1, 10],

        'solver': ['newton-cg', 'lbfgs', 'liblinear'],

        'max\_iter': [100, 500],

        'random\_state': [42],

    }),

('AdaBoost', AdaBoostClassifier(), {

        'n\_estimators': [50, 100, 150],

        'learning\_rate': [0.1, 0.5, 1.0],

    }),

    ('XGBoost', XGBClassifier(), {

        'n\_estimators': [100, 200, 300],

        'max\_depth': [3, 5],

        'learning\_rate': [0.01, 0.1],

        'subsample': [0.7, 0.8, 0.9],

        'colsample\_bytree': [0.7, 0.8, 0.9],

    }),

    ('Gaussian NB', GaussianNB(), {})

]

best\_accuracy = 0.0

accuracies = []

model\_names = []

for model\_name, model, param\_grid in models:

    grid\_search = GridSearchCV(model, param\_grid=param\_grid, scoring='accuracy', cv=5, n\_jobs=-1, verbose=1)

    grid\_search.fit(X\_train, y\_train)

    print(f"Best hyperparameters for {model\_name}:", grid\_search.best\_params\_)

    best\_model = grid\_search.best\_estimator\_

    y\_pred = best\_model.predict(X\_test)

    print(classification\_report(y\_test, y\_pred))

    test\_accuracy = accuracy\_score(y\_test, y\_pred)

    print(f"Test accuracy of the best {model\_name} model:", test\_accuracy)

    accuracies.append(test\_accuracy)

    model\_names.append(model\_name)

    if test\_accuracy > best\_accuracy:

        best\_accuracy = test\_accuracy

        best\_model\_name = model\_name

        final\_model = best\_model

***# Plot the accuracy of different models***

plt.figure(figsize=(10, 6))

plt.bar(model\_names, accuracies, color='skyblue')

plt.xlabel('Model')

plt.ylabel('Accuracy')

plt.title('Model Accuracy Comparison')

plt.xticks(rotation=45)

plt.show()

print(f"The {best\_model\_name} model is the best model...")

*# Saving the best model and normalizer*

joblib.dump(final\_model, 'best\_model.pkl')

joblib.dump(scaler, 'normalizer.pkl')

print("Model saved successfully!!!")

TESTING MODEL CODE:

import joblib

import pandas as pd

import warnings

warnings.filterwarnings('ignore')

***# Load the saved model and normalizer***

model = joblib.load('best\_model.pkl')

scaler = joblib.load('normalizer.pkl')

***# Selected features***

selected\_features = ['MDVP:Jitter(Abs)', 'MDVP:RAP', 'MDVP:PPQ', 'Jitter:DDP', 'MDVP:Shimmer', 'MDVP:Shimmer(dB)',

                     'Shimmer:APQ3', 'Shimmer:APQ5', 'MDVP:APQ', 'Shimmer:DDA', 'NHR', 'HNR', 'PPE']

***# Get input from the user for the selected features***

user\_input = {}

for feature in selected\_features:

    value = float(input(f"Enter the value for '{feature}': "))

    user\_input[feature] = value

***# Create a DataFrame with the user input***

user\_df = pd.DataFrame([user\_input])

***# Normalize the user input using the saved normalizer***

user\_scaled = scaler.transform(user\_df)

***# Make a prediction using the loaded model***

prediction = model.predict(user\_scaled)

***# Print the result***

if prediction[0] == 0:

    print("Status: Healthy")

else:

    print("Status: Parkinson's Disease")