**Basic Details:**

**Name:** Samala.Sudhakar

**Email:** [om.sudhakar@gmail.com](mailto:om.sudhakar@gmail.com)

**Unified Mentor ID:** UIMD10042529348

**Project :** Machine Learning Project

**Project Information:**

**Title:** Detect Thyroid Cancer Reoccurrence using patient data

**Objective:**

Building a machine learning model that can predict if a Thyroid Cancer survivor can relapse(his or her cancer reoccurs).

**Dataset**

This dataset contains data about thyroid checkups for people with a diagnosis and is a comprehensive collection of patient information, specifically focused on individuals diagnosed with cancer.

**Description of columns:**

● Age: The age at the time of diagnosis or treatment.

● Gender: The gender of the patient (male or female).

● Smoking: Whether the patient is a smoker or not.

● Hx Smoking: Smoking history of the patient (e.g., whether they have ever smoked).

● Hx Radiotherapy: History of radiotherapy treatment for any condition.

● Thyroid Function: The status of thyroid function, possibly indicating if there are any abnormalities.

● Physical Examination: Findings from a physical examination of the patient.

● Adenopathy: Presence or absence of enlarged lymph nodes (adenopathy) in the neck region.

● Pathology: Specific type of thyroid cancer determined by the pathological examination of biopsy samples.

● Focality: Whether the cancer is unifocal (limited to one location) or multifocal (present in multiple locations).

● Risk: The risk category of the cancer based on various factors, such as tumor size, extent of spread, and histological type.

● T: Tumor classification based on its size and extent of invasion into nearby structures.

● N: Nodal classification indicating the involvement of lymph nodes.

● M: Metastasis classification indicating the presence or absence of distant metastases.

● Stage: The overall stage of the cancer, typically determined by combining T, N, and M classifications.

● Response: Response to treatment, indicating whether the cancer responded positively, negatively, or remained stable after treatment.

● Recurred: Has the cancer recurred after initial treatment.

**Project Link:**

<https://github.com/AIforeverything/UnifiedMentorInternshipProjects/blob/c86c2928100b9b567ee2361675a7f402cc307a20/Detect%20Thyroid%20Cancer%20Reoccurrence%20using%20patient%20data/project1.ipynb>

<https://github.com/AIforeverything/UnifiedMentorInternshipProjects/blob/c86c2928100b9b567ee2361675a7f402cc307a20/categorical/categorical_model.py>

**Code**

**Steps Followed:**

**Step-1: Initially I have created a library for building a categorical machine learning model and used this library for building model.**

**categorical\_model.py**

# ## Step-1: Common virtual environment was created and activated: myenv

# ## pip install virtualenv

# ## virtualenv myenv

# ## .\myenv\Scripts\activate.ps1

def greet(name):

return f"good job {name}"

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import os

import sys

from pathlib import Path

import zipfile

import warnings

warnings.filterwarnings("ignore")

import sklearn

from sklearn.preprocessing import LabelEncoder, OneHotEncoder

from sklearn.preprocessing import MinMaxScaler

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

from sklearn.linear\_model import LogisticRegressionCV

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

from sklearn.naive\_bayes import GaussianNB

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC

from xgboost import XGBClassifier

from sklearn.model\_selection import GridSearchCV

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

import joblib

# import tensorflow as tf

# from tensorflow import keras

# from tensorflow.keras.models import Sequential

# from tensorflow.keras.layers import Dense

class categorical\_Model:

def \_\_init\_\_(self,model, target\_column, df):

"""

Initializes the categoricalTarget class.

Parameters:

model (str): The name of the model to be used.

target\_column (str): The name of the target column.

df (pd.DataFrame): The DataFrame containing the data.

"""

self.df = df

self.target\_column = target\_column

self.model = model

# Importing data into a dataframe from csv file in the directory

def readingData():

#checking the directory for .csv files

directory = Path('./')

# List all CSV files

for csv\_file in directory.glob('\*.csv'):

print(csv\_file.name)

df= pd.read\_csv(csv\_file)

return df

# # Data extraction from zipfile

def extractingZipFile(zipFilePath, extractTo):

"""

Extracts the contents of a zip file to a specified directory.

Parameters:

zipFilePath (str): The path to the zip file.

extractTo (str): The directory to extract the contents to.

"""

with zipfile.ZipFile(zipFilePath, 'r') as zip\_ref:

zip\_ref.extractall(extractTo)

# EDA (Exploratory Data Analysis)

# Checking missing values

def checkMissingValues(df):

"""

Checks for missing values in the DataFrame

Parameters:

df (pd.DataFrame): The DataFrame to check for missing values.

Returns:

missing values

"""

return df.isnull().sum()

# Removing duplicates

## function to check for duplicates and remove dupliates

def checkDuplicates(df):

"""

Checks for duplicate rows in the DataFrame and removes them.

Parameters:

df (pd.DataFrame): The DataFrame to check for duplicates.

Returns:

pd.DataFrame: The DataFrame with duplicates removed.

"""

duplicates = df.duplicated().sum()

if duplicates > 0:

df = df.drop\_duplicates()

print(f"Removed {duplicates} duplicate rows.")

else:

print("No duplicate rows found.")

return df

#Function for all columns

def allColumns(df):

return list(df.columns)

# Function for categorical columns

def catColumns(df):

catCol=df.select\_dtypes(include='object').columns

return catCol

# Function for Non-categorical columns

def nonCatColumns(df):

numeric\_col=df.select\_dtypes(include='number').columns

return numeric\_col

## function to check categorical columns and replacing them with numerical values

def checkCategoricalColumnsAndReplacingWithLE(df):

"""

Checks for categorical columns in the DataFrame and replaces them with numerical values.

Parameters:

df (pd.DataFrame): The DataFrame to check for categorical columns.

Returns:

pd.DataFrame: The DataFrame with categorical columns replaced with numerical values.

"""

categorical\_columns = df.select\_dtypes(include=['object']).columns

print(f"Categorical columns: {categorical\_columns}")

for col in categorical\_columns:

print(f"col.unique(): {df[col].unique()}")

print(f"col.value\_counts(): {df[col].value\_counts()}")

le = LabelEncoder()

df[col] = le.fit\_transform(df[col])

return df

# function to standardize Non Categorical columns

def standardizeNonCategoricalColumns(df):

minMax=MinMaxScaler()

numeric\_col=df.select\_dtypes(include='number').columns

df[numeric\_col]=minMax.fit\_transform(df[numeric\_col])

return df

## function to removing the missing values

def removeMissingValues(df):

"""

Removes rows with missing values from the DataFrame.

Parameters:

df (pd.DataFrame): The DataFrame to remove missing values from.

Returns:

pd.DataFrame: The DataFrame with missing values removed.

"""

df = df.dropna()

return df

#function to print the correlation matrix respect to the target column

def printCorrelationMatrix(df, target\_column):

"""

Prints the correlation matrix of the DataFrame with respect to the target column.

Parameters:

df (pd.DataFrame): The DataFrame to print the correlation matrix for.

target\_column (str): The name of the target column.

Returns:

pd.DataFrame: The correlation matrix.

"""

# print the correlation matrix with respect to the target column

print(f"Correlation matrix with respect to {target\_column}:")

print(df.corr()[target\_column].sort\_values(ascending=False))

corr\_text=df.corr()[target\_column].sort\_values(ascending=False)

# .to\_string() provides a nicely formatted text version of the DataFrame.

# This will produce a human-readable file.

# If we want a machine-readable format instead, consider .to\_csv("file.txt", sep='\t').

with open('correlation.txt', 'w') as f:

f.write(corr\_text.to\_string())

corr = df.corr()

plt.figure(figsize=(12, 8))

sns.heatmap(corr, annot=True, fmt=".2f", cmap='coolwarm')

plt.title(f"Correlation Matrix with respect to {target\_column}")

plt.show()

return corr

#checking missing values of each column

def missing\_columns(df):

return (df.isnull().sum())

#checking missing values of all columns

def missing\_columns\_total(df):

return (df.isnull().sum().sum())

## function to split the data into X,y

def splitDataIntoXy(df, target\_column):

"""

Splits the DataFrame into X and y.

retuns tuple

"""

X = df.drop(target\_column, axis=1)

y = df[target\_column]

return X,y

## function to split the data into train and test

def splitData(X,y):

"""

Splits the DataFrame into training and testing sets.

Parameters:

X,y

Returns:

tuple: The training and testing sets.

"""

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

return X\_train, X\_test, y\_train, y\_test

# function to train the model and compare the models and save the best model and the model report and the model performance

def trainModel(X\_train, X\_test, y\_train, y\_test):

"""

Trains the model and compares the models and saves the best model and the model report and the model performance.

Parameters:

X\_train (pd.DataFrame): The training data.

X\_test (pd.DataFrame): The testing data.

y\_train (pd.Series): The training labels.

y\_test (pd.Series): The testing labels.

Returns:

None

"""

models = {

"Logistic Regression": LogisticRegressionCV(max\_iter=10000),

"Decision Tree": DecisionTreeClassifier(),

"RandomForest": RandomForestClassifier(min\_samples\_split=5),

"Gradient Boosting": GradientBoostingClassifier(),

"Naive Bayes" :GaussianNB(),

"KNN" : KNeighborsClassifier(),

"Support Vector Machines" : SVC(),

"XGBoost": XGBClassifier()

}

best\_model = None

best\_accuracy = 0

for name, model in models.items():

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

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

print(f"{name} Accuracy: {accuracy:.4f}")

if accuracy > best\_accuracy:

best\_accuracy = accuracy

best\_model = model

best\_model\_name = name

print(f"Best Model: {best\_model.\_\_class\_\_.\_\_name\_\_} with accuracy: {best\_accuracy:.2f}")

# Save the best model

joblib.dump(best\_model\_name, f'{best\_model\_name}.pkl')

# Save the classification report

report = classification\_report(y\_test, y\_pred)

with open('classification\_report.txt', 'w') as f:

f.write(f"Model: {best\_model\_name} \n\n")

f.write(report)

# Save the confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

np.savetxt('confusion\_matrix.txt', cm, fmt='%d')

# function to load the model

def loadModel(model\_path):

"""

Loads the model from the specified path.

Parameters:

model\_path (str): The path to the model.

Returns:

model: The loaded model.

"""

model = joblib.load(model\_path)

return model

# making an object of the class to use the functions

def main():

# Unzip the file

file= categorical\_Model.extractingZipFile('./', "./")

# Reading the data

df = categorical\_Model.readingData()

# Checking for missing values

missing\_values = categorical\_Model.checkMissingValues(df)

print(f"Missing values: {missing\_values}")

# Checking for duplicates

df = categorical\_Model.checkDuplicates(df)

# Checking for categorical columns

df = categorical\_Model.checkCategoricalColumns(df)

# Removing missing values

df = categorical\_Model.removeMissingValues(df)

# Choosing the target column

target\_column = input("Enter the target column name: ")

if target\_column not in df.columns:

print(f"Target column '{target\_column}' not found in DataFrame.")

else:

print(f"Target column '{target\_column}' found in DataFrame.")

# Printing the correlation matrix

corr\_matrix = categorical\_Model.printCorrelationMatrix(df, target\_column)

# Splitting the data into train and test sets

X\_train, X\_test, y\_train, y\_test = categorical\_Model.splitData(df, target\_column)

# Training the model and saving the best model

categorical\_Model.trainModel(X\_train, X\_test, y\_train, y\_test)

**Step-2 : Code for model:**

#!/usr/bin/env python

# coding: utf-8

# # Objective

# ## Build a system that can predict if a Thyroid Cancer survivor can relapse(his or her cancer reoccurs)

# ### Dataset

# #### This dataset contains data about thyroid checkups for people with a diagnosis and is a comprehensive collection of patient information, specifically focused on individuals diagnosed with cancer

# ## Step-1: Common virtual environment was created and activated: myenv

# #### pip install virtualenv

# #### virtualenv myenv

# #### .\myenv\Scripts\activate.ps1

# ## Installing required libraries

# In[1]:

# %pip install -r requirements.txt

# ## Step-2: Importing required libraries

# In[25]:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import os

import sys

import zipfile

import warnings

warnings.filterwarnings("ignore")

import sklearn

from sklearn.preprocessing import LabelEncoder, OneHotEncoder

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

from sklearn.linear\_model import LogisticRegressionCV

from sklearn.ensemble import RandomForestClassifier

from xgboost import XGBClassifier

from sklearn.model\_selection import GridSearchCV

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

import joblib

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

# ## Step-3: Data extraction from zipfile

# In[4]:

def extractingZipFile(zipFilePath, extractTo):

    """

    Extracts the contents of a zip file to a specified directory.

    Parameters:

    zipFilePath (str): The path to the zip file.

    extractTo (str): The directory to extract the contents to.

    """

    with zipfile.ZipFile(zipFilePath, 'r') as zip\_ref:

        zip\_ref.extractall(extractTo)

extractingZipFile('thyroid\_cancer.zip', 'data')

# ## Step-4: Importing data into a dataframe

# In[13]:

def readingData(path):

    """

    Reads the data from a CSV file and returns it as a pandas DataFrame.

    Parameters:

    path (str): The path to the CSV file.

    Returns:

    pd.DataFrame: The data as a pandas DataFrame.

    """

    df = pd.read\_csv(path)

    return df

df=readingData("data/thyroid\_cancer/dataset.csv")

df.head()

# ## Step-4: EDA (Exploratory Data Analysis)

# In[14]:

df.info()

# In[15]:

df.describe()

# ## Step-4(a): Checking missing values

# In[16]:

def checkMissingValues(df):

    """

    Checks for missing values in the DataFrame

    Parameters:

    df (pd.DataFrame): The DataFrame to check for missing values.

    Returns:

    missing values

    """

    return df.isnull().sum()

missing\_values = checkMissingValues(df)

missing\_values

# #### No missing values were found

# ## Step-4(b): Removing duplicates

# In[17]:

## function to check for duplicates and remove dupliates

def checkDuplicates(df):

    """

    Checks for duplicate rows in the DataFrame and removes them.

    Parameters:

    df (pd.DataFrame): The DataFrame to check for duplicates.

    Returns:

    pd.DataFrame: The DataFrame with duplicates removed.

    """

    duplicates = df.duplicated().sum()

    if duplicates > 0:

        df = df.drop\_duplicates()

        print(f"Removed {duplicates} duplicate rows.")

    else:

        print("No duplicate rows found.")

    return df

df = checkDuplicates(df)

df.head()

# In[18]:

## function to check categorical columns and replacing them with numerical values

def checkCategoricalColumns(df):

    """

    Checks for categorical columns in the DataFrame and replaces them with numerical values.

    Parameters:

    df (pd.DataFrame): The DataFrame to check for categorical columns.

    Returns:

    pd.DataFrame: The DataFrame with categorical columns replaced with numerical values.

    """

    categorical\_columns = df.select\_dtypes(include=['object']).columns

    print(f"Categorical columns: {categorical\_columns}")

    for col in categorical\_columns:

        print(f"col.unique(): {df[col].unique()}")

        print(f"col.value\_counts(): {df[col].value\_counts()}")

        le = LabelEncoder()

        df[col] = le.fit\_transform(df[col])

    return df

df = checkCategoricalColumns(df)

df.head()

# In[19]:

df.info()

# In[20]:

df.corr()["Recurred"].sort\_values(ascending=False)

# ## Step-5: model building

# In[21]:

X=df.drop(columns=['Recurred'])

y=df['Recurred']

X.head()

# In[22]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape

# In[ ]:

## different models training using gridserachCV and evaluation

def train\_and\_evaluate\_model(model, param\_grid, X\_train, y\_train, X\_test, y\_test):

    """

    Trains and evaluates a machine learning model using GridSearchCV.

    Parameters:

    model (sklearn.base.BaseEstimator): The machine learning model to train.

    param\_grid (dict): The parameter grid for GridSearchCV.

    X\_train (pd.DataFrame): The training data.

    y\_train (pd.Series): The training labels.

    X\_test (pd.DataFrame): The testing data.

    y\_test (pd.Series): The testing labels.

    """

    grid\_search = GridSearchCV(model, param\_grid, cv=5, scoring='accuracy')

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

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

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

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

    print(f"Best parameters: {grid\_search.best\_params\_}")

    print(f"Accuracy: {accuracy}")

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

    print(confusion\_matrix(y\_test, y\_pred))

    return best\_model

# Logistic Calssifier

logistic\_model = LogisticRegressionCV(max\_iter=1000)

logistic\_param\_grid = {

    'Cs': [0.01, 0.1, 1, 10, 100],

    'penalty': ['l1', 'l2'],

    'solver': ['liblinear', 'saga']

}

logistic\_best\_model = train\_and\_evaluate\_model(logistic\_model, logistic\_param\_grid, X\_train, y\_train, X\_test, y\_test)

# Random Forest Classifier

rf\_model = RandomForestClassifier()

rf\_param\_grid = {

    'n\_estimators': [50, 100, 200],

    'max\_depth': [None, 10, 20, 30],

    'min\_samples\_split': [2, 5, 10],

    'min\_samples\_leaf': [1, 2, 4]

}

rf\_best\_model = train\_and\_evaluate\_model(rf\_model, rf\_param\_grid, X\_train, y\_train, X\_test, y\_test)

# XGBoost Classifier

xgb\_model = XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss')

xgb\_param\_grid = {

    'n\_estimators': [50, 100, 200],

    'max\_depth': [3, 5, 7],

    'learning\_rate': [0.01, 0.1, 0.2],

    'subsample': [0.8, 1.0]

}

xgb\_best\_model = train\_and\_evaluate\_model(xgb\_model, xgb\_param\_grid, X\_train, y\_train, X\_test, y\_test)

# Save the best model

def save\_model(model, model\_name):

    """

    Saves the trained model to a file.

    Parameters:

    model (sklearn.base.BaseEstimator): The trained model to save.

    model\_name (str): The name of the model file.

    """

    joblib.dump(model, model\_name)

save\_model(logistic\_best\_model, 'logistic\_model.pkl')

save\_model(rf\_best\_model, 'rf\_model.pkl')

save\_model(xgb\_best\_model, 'xgb\_model.pkl')

## function to print the model accuracy

def print\_model\_accuracy(model, X\_test, y\_test):

    """

    Prints the accuracy of the model on the test data.

    Parameters:

    model (sklearn.base.BaseEstimator): The trained model to evaluate.

    X\_test (pd.DataFrame): The testing data.

    y\_test (pd.Series): The testing labels.

    """

    y\_pred = model.predict(X\_test)

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

    print(f"Model accuracy: {accuracy}")

# In[27]:

print("LogisticRegressionCV\_best\_model: ")

print\_model\_accuracy(logistic\_best\_model, X\_test, y\_test)

print("RandomForestClassifier\_best\_model: ")

print\_model\_accuracy(rf\_best\_model, X\_test, y\_test)

print("XGBClassifier\_best\_model: ")

print\_model\_accuracy(xgb\_best\_model, X\_test, y\_test)

# In[ ]:

# loading the best model and checking precision,recall,f1-score, accuracy

def load\_model(model\_name):

    """

    Loads a trained model from a file.

    Parameters:

    model\_name (str): The name of the model file.

    Returns:

    sklearn.base.BaseEstimator: The loaded model.

    """

    return joblib.load(model\_name)

logistic\_model = load\_model('logistic\_model.pkl')

rf\_model = load\_model('rf\_model.pkl')

xgb\_model = load\_model('xgb\_model.pkl')

# # Step-6:  RandomForestClassifier has maximum

# Accuracy: 0.958904109589041

#               precision    recall  f1-score   support

#

#            0       0.96      0.98      0.97        51

#            1       0.95      0.91      0.93        22

#

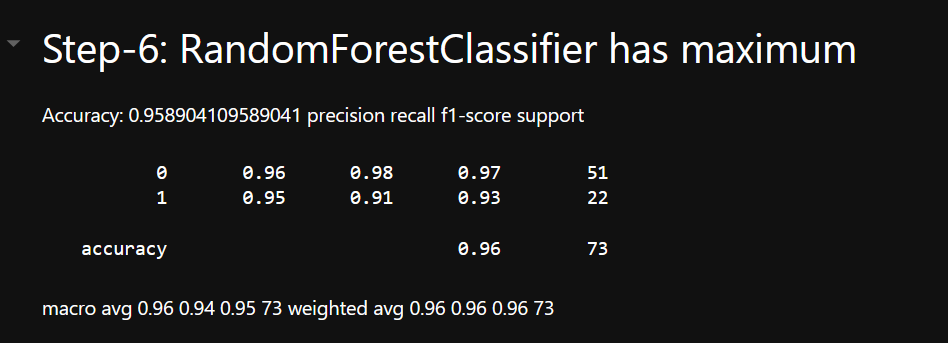
#     accuracy                           0.96        73

#    macro avg       0.96      0.94      0.95        73

# weighted avg       0.96      0.96      0.96        73

**Model Outcomes**

Different models are built using the dataset and found



Random forest model gave more accuracy.

**classification\_report :**

**Model: RandomForest**

RandomForestClassifier has maximum Accuracy: 0.958904109589041

              precision    recall  f1-score   support

           0       0.96      0.98      0.97        51

           1       0.95      0.91      0.93        22

    accuracy                           0.96        73

   macro avg       0.96      0.94      0.95        73

weighted avg       0.96      0.96      0.96        73