Basic Details:

Name: Samala.Sudhakar
Email: 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}")
```

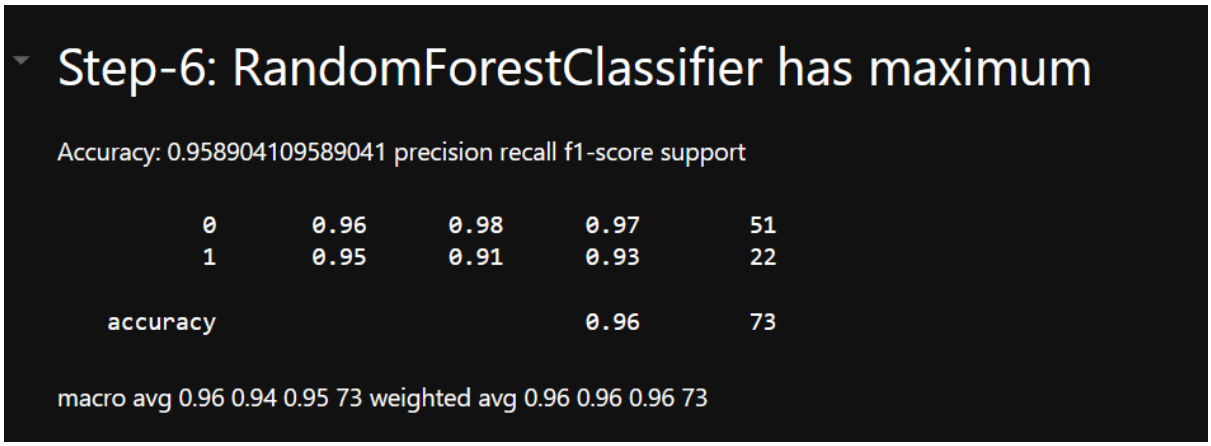
```
    print(classification_report(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
    return best_model
# Logistic Calssifier
logistic_model = LogisticRegressionCV(max_iter=1000)
logistic_param_grid = {
    'Cs': [0.01, 0.1, 1, 10, 100],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear', '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
```