Project2_Healthcare

February 19, 2024

1 Healthcare(Course-end Project 2)

Description: NIDDK (National Institute of Diabetes and Digestive and Kidney Diseases) research creates knowledge about and treatments for the most chronic, costly, and consequential diseases.

The dataset used in this project is originally from NIDDK. The objective is to predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. Build a model to accurately predict whether the patients in the dataset have diabetes or not.

Dataset Description

The datasets consists of several medical predictor variables and one target variable (Outcome). Predictor variables includes the number of pregnancies the patient has had, their BMI, insulin level, age, and more. Variables Description Pregnancies Number of times pregnant Glucose Plasma glucose concentration in an oral glucose tolerance test BloodPressure Diastolic blood pressure (mm Hg) SkinThickness Triceps skinfold thickness (mm) Insulin Two hour serum insulin BMI Body Mass Index DiabetesPedigreeFunction Diabetes pedigree function Age Age in years Outcome Class variable (either 0 or 1). 268 of 768 values are 1, and the others are 0

[ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

[ ]:
```
df = pd.read_csv("health care diabetes.csv")
```

[ ]:
```
df.head()
```

[ ]:
```
df.info()
```

1.0.1 Project Task: Week 1

Data Exploration:

Perform descriptive analysis. Understand the variables and their corresponding values. On the columns below, a value of zero does not make sense and thus indicates missing value:

Glucose

BloodPressure

SkinThickness

Insulin

BMI

Visually explore these variables using histograms. Treat the missing values accordingly.

There are integer and float data type variables in this dataset. Create a count (frequency) plot describing the data types and the count of variables.

[ ]:
```
df.describe()
```

[ ]:
```
df.columns
```

[ ]:
```
df.shape
```

[ ]:
```
df.isnull().sum()
```

[ ]:
```
for i in df.columns:
    df[i].plot(kind = 'hist', label = 'labels')
    plt.show()
```

[ ]:
```
for i in df.columns:
    print(i)
    print(df[i].value_counts(normalize = True) * 100)
    print('\n\n')
```

[ ]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

data = {
    'Column': ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
               'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
    'Non-Null Count': [768, 768, 768, 768, 768, 768, 768, 768, 768],
    'Dtype': ['int64', 'int64', 'int64', 'int64', 'int64', 'float64',
              'float64', 'int64', 'int64']
}
df_info = pd.DataFrame(data)

# Creating the countplot
plt.figure(figsize=(10, 6))
sns.countplot(data=df_info, x='Dtype')
plt.title('Count of Variables by Data Type')
plt.xlabel('Data Type')
plt.ylabel('Count')
plt.show()
```

1.1 Data Exploration:

1.1.1 Check the balance of the data by plotting the count of outcomes by their value.

1.1.2 Describe your findings and plan future course of action.

[ ]:
```
import seaborn as sbn
import matplotlib.pyplot as plt
# check the balance of the data
plt.figure(figsize=(8,6))
sns.countplot(data= df, x = 'Outcome')
plt.title('Count of Outcomes')
plt.xlabel('Outcome')
plt.ylabel('Count')
plt.show()
```

1.1.3 Create scatter charts between the pair of variables to understand the relationships. Describe your findings.

[ ]:
```
sns.pairplot(df, diag_kind = 'kde', hue = 'Outcome')
plt.suptitle('Pairwise Scatter plots')
plt.show()
```

1.1.4 Perform correlation analysis. Visually explore it using a heat map.

[ ]:
```
correlation_matrix = df.corr()
correlation_matrix
```

[ ]:
```
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot = True, cmap='coolwarm', fmt = ".2f")
plt.title("Correlation Heatmap")
plt.show()
```

2 Strategy for Model Building:

2.0.1 Data Preprocessing:

We'll start by preprocessing the data, which includes handling missing values, encoding categorical variables, and scaling features if necessary.

[ ]:
```
col= ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']
df[col] = df[col].replace(0, None)
```

[ ]:
```
df.shape
```

[ ]:
```
df.isnull().sum()
```

[ ]:
```
(df.isna().sum()/ len(df))*100
```

[ ]:
```
df.dropna(subset = ['Glucose'], inplace = True)
```

[ ]:
```
df.shape
```

Code for missing value treatment

[ ]:
```
BP = int(round(df['BloodPressure'].mean()))
BP
```

[ ]:
```
df['BloodPressure'].fillna(BP, inplace = True)
```

[ ]:
```
df.shape
```

[ ]:
```
missing_SkinThickness = df[df['SkinThickness'].isnull()]
missing_SkinThickness
```

[ ]:
```
missing_skin_thickness_df = df[df['SkinThickness'].isnull()]

# Calculate the mode for each column in the filtered DataFrame
modes = missing_skin_thickness_df.mode(dropna=True).iloc[0]

print("Mode values for columns where SkinThickness is missing:")
print(modes)
```

[ ]:
```
# Select rows where 'SkinThickness' is missing
missing_skinthickness = df[df['SkinThickness'].isnull()]

# Calculate mean of other columns for rows where 'SkinThickness' is missing
mean_of_missing_skinthickness = missing_skinthickness.drop('SkinThickness',␣
 ↪axis=1).mean()

# Display mean of other columns
print(mean_of_missing_skinthickness)
```

[ ]:
```
a=df[(df["BMI"]==31.2)]
```

[ ]:
```
int(round(a["SkinThickness"].mean()))
```

[ ]:
```
for i in a.columns:
    print(i)
    print(a[i].value_counts())
    print('\n\n')
```

[ ]:
```
df['SkinThickness'].fillna(27, inplace = True)
```

[ ]:
```
df.isna().sum()
```

[ ]:
```
b=df[df['Insulin'].isna()]
```

[ ]:
```
b
```

[ ]:
```
for j in b.columns:
    print(j)
    print(b[j].value_counts(normalize = True)*100)
    print('\n\n')
```

[ ]:
```
(b['Glucose'].mean())
```

[ ]:
```
c = df[df['Glucose'].isin(range(110, 130))]
```

[ ]:
```
c
```

[ ]:
```
int(round(c['Insulin'].mean()))
```

[ ]:
```
df['Insulin'].fillna(154, inplace = True)
```

[ ]:
```
df.isna().sum()
```

[ ]:
```
df['BMI'].fillna(df['BMI'].mean(), inplace = True)
```

[ ]:
```
df.isna().sum()
```

[ ]:
```
df.dtypes
```

[ ]:
```
# Convert the 'Glucose' column to string type
df['Glucose'] = df['Glucose'].astype(str)

# Remove non-numeric characters and special characters
df['Glucose'] = df['Glucose'].str.replace('[^0-9]', '')

# Convert the data type of the column to numeric
df['Glucose'] = pd.to_numeric(df['Glucose'], errors='coerce')

# Convert the numeric data type to integer
df['Glucose'] = df['Glucose'].astype('Int64')
```

[ ]:
```
df.dtypes
```

[ ]:
```
corr =  df.corr()
corr
```

[ ]:
```
thresh = 0.4
corr[(corr> thresh) & (corr != 1)].fillna('')
```

From this correlation matrix it is found that the columns Pregnancies, Glucose, SkinThickness, Insulin and Age are most significant

[ ]:
```
sns.heatmap(corr, annot = True)
```

2.1 Balancing the dataset

[ ]:
```
df['Outcome'].value_counts(normalize = True)* 100
```

2.1.1 Conclusion:

From above observation it is observed that the data is little bit imbalanced(about +- 15%). Most of the models can handle this data, but to increase the accuracy lets balance the dataset. for this we will use the "oversampling the minority class or undersampling the majority class"

[ ]:
```
from sklearn.utils import resample
```

[ ]:
```
df_majority = df[df['Outcome'] == 0]
df_minority = df[df['Outcome'] ==1]

df_minority_upsampled = resample(df_minority, replace = True,
                                 n_samples = len(df_majority),
                                random_state = 42)

df_balanced = pd.concat([df_majority, df_minority_upsampled])
print(df_balanced['Outcome'].value_counts())
```

2.1.2 Feature Selection:

If the dataset has many features, we'll consider feature selection techniques like correlation analysis or feature importance to identify the most relevant features for our model.

2.1.3 Validation Framework:

We'll choose an appropriate validation framework to evaluate our models. This could involve techniques like train-test split, cross-validation, or stratified sampling.

2.1.4 Model Selection:

We'll select classification algorithms suitable for our problem. Given that we're comparing with KNN, we'll consider algorithms like Logistic Regression, Decision Trees, Random Forest, Support Vector Machines (SVM), and Gradient Boosting.

2.1.5 Hyperparameter Tuning:

We'll optimize the hyperparameters of the selected algorithms using techniques like grid search or random search to improve their performance.

2.1.6 Ensemble Methods:

Optionally, we may explore ensemble methods like bagging (e.g., Random Forest) or boosting (e.g., Gradient Boosting) to further enhance model performance. Applying an Appropriate Classification Algorithm:

Considering that we're comparing with KNN, we'll choose other classification algorithms mentioned earlier and train them on the preprocessed dataset. We'll use standard machine learning libraries like scikit-learn in Python to implement these algorithms. Comparing Various Models:

After training the models, we'll evaluate their performance using metrics such as accuracy, precision, recall, F1-score, and ROC-AUC score. We'll compare the performance of each model against the baseline (KNN) and identify the best-performing model.

Creating a Classification Report: The classification report will include metrics such as sensitivity (recall), specificity, precision, F1-score, and ROC-AUC score. Sensitivity measures the proportion of actual positives that are correctly identified, specificity measures the proportion of actual negatives that are correctly identified, and the ROC-AUC score quantifies the model's ability to discriminate between positive and negative classes.

Here's a brief outline of the parameters we'll use for evaluation: Sensitivity (Recall): TP / (TP + FN) Specificity: TN / (TN + FP) Precision: TP / (TP + FP) F1-score: Harmonic mean of precision and recall ROC-AUC Score: Area under the receiver operating characteristic curve, which represents the trade-off between sensitivity and specificity. We'll use these parameters to assess the performance of each classification algorithm and select the best one for our problem.

[ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import GradientBoostingClassifier

# Define the models you want to use
models = {
    'Random Forest': RandomForestClassifier(),
    'Logistic Regression': LogisticRegression(),
    'Support Vector Machine': SVC(),
    'Decision Tree': DecisionTreeClassifier(),
    'Gradient Boosting': GradientBoostingClassifier()
}

# List of most significant features
significant_features = ['Pregnancies', 'Glucose', 'SkinThickness', 'Insulin',␣
 ↪'Age']

# Selecting only the significant features from the dataset
X_selected = df[significant_features]
y_selected = df['Outcome']  # Use 'Outcome' as the target variable

# Splitting the selected features into training and testing sets
```

```
X_train_selected, X_test_selected, y_train_selected, y_test_selected =␣
 ↪train_test_split(X_selected, y_selected, test_size=0.2, random_state=42)

# Hyperparameter tuning using grid search or random search for each model
best_models = {}
for name, model in models.items():
    if name == 'Random Forest':
        param_grid = {
            'n_estimators': [10, 50, 100],
            'max_depth': [None, 5, 10]
        }
    elif name == 'Logistic Regression':
        param_grid = {
            'C': [0.1, 1, 10]
        }
    elif name == 'Support Vector Machine':
        param_grid = {
            'C': [0.1, 1, 10],
            'gamma': ['scale', 'auto'],
            'kernel': ['linear', 'rbf']
        }
    elif name == 'Decision Tree':
        param_grid = {
            'max_depth': [None, 5, 10]
        }
    elif name == 'Gradient Boosting':
        param_grid = {
            'n_estimators': [50, 100, 200],
            'max_depth': [3, 5, 10],
            'learning_rate': [0.01, 0.1, 1]
        }

    grid_search = GridSearchCV(model, param_grid, cv=5)
    grid_search.fit(X_train_selected, y_train_selected)
    best_models[name] = grid_search.best_estimator_

# Evaluate the performance of the best model with selected features
for name, model in best_models.items():
    y_pred_selected = model.predict(X_test_selected)
    print(f'Classification Report for {name}:')
    print(classification_report(y_test_selected, y_pred_selected))
```

Based on the above classification reports, here are the findings for each model:

2.1.7 Random Forest:

Precision for class 0 is 0.79, indicating that 79% of the instances predicted as class 0 were actually class 0. Recall for class 0 is 0.85, indicating that 85% of the actual class 0 instances were correctly

predicted as class 0. F1-score for class 0 is 0.82, which is the harmonic mean of precision and recall for class 0. Accuracy is 0.75, meaning that 75% of the instances were correctly classified. The model performs better for class 0 than for class 1.

2.1.8 Logistic Regression:

Similar to Random Forest, precision, recall, and F1-score for class 0 are higher compared to class 1. Accuracy is 0.79, indicating that 79% of the instances were correctly classified. The model performs slightly better than Random Forest for class 1.

2.1.9 Support Vector Machine (SVM):

Precision, recall, and F1-score for class 0 are higher compared to class 1, similar to the previous models. Accuracy is 0.79, similar to Logistic Regression. The model performs well for class 0 but relatively poorer for class 1.

2.1.10 Decision Tree:

Precision, recall, and F1-score for both classes are lower compared to the other models. Accuracy is 0.72, indicating lower overall performance compared to the other models. The model performs relatively poorly for both classes compared to the other models.

2.1.11 Gradient Boosting:

Precision, recall, and F1-score for class 0 are higher compared to class 1. Accuracy is 0.78, similar to Logistic Regression and SVM. The model performs relatively well for class 0 but relatively poorer for class 1. Overall, Logistic Regression, SVM, and Gradient Boosting show similar performance in terms of accuracy and class-wise metrics. Random Forest also performs well but slightly lower than the other three models. Decision Tree shows the lowest performance among all models.

[ ]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

# Train KNN model
knn_model = KNeighborsClassifier(n_neighbors=5)  # You can adjust the number of␣
 ↪neighbors as needed
knn_model.fit(X_train_selected, y_train_selected)

# Evaluate KNN model
y_pred_knn = knn_model.predict(X_test_selected)
classification_report_knn = classification_report(y_test_selected, y_pred_knn)

print("Classification Report for KNN:")
print(classification_report_knn)
```

Based on the comparison, Logistic Regression appears to perform the best among the models considered, with the highest accuracy, precision, and recall for both classes.

2.1.12 Project Task: Week 2

Data Modeling:

Devise strategies for model building. It is important to decide the right validation framework. Express your thought process.

Apply an appropriate classification algorithm to build a model.

Compare various models with the results from KNN algorithm.

Create a classification report by analyzing sensitivity, specificity, AUC (ROC curve), etc.

Please be descriptive to explain what values of these parameter you have used.

Data Reporting:

Create a dashboard in tableau by choosing appropriate chart types and metrics useful for the business. The dashboard must entail the following:

Pie chart to describe the diabetic or non-diabetic population

Scatter charts between relevant variables to analyze the relationships

Histogram or frequency charts to analyze the distribution of the data

Heatmap of correlation analysis among the relevant variables

Create bins of these age values: 20-25, 25-30, 30-35, etc. Analyze different variables for these age brackets using a bubble chart.

[ ]: