Data Science Analysis

Predicting Heart Strokes with Decision trees and Gradient boost using XGBoost

भारतीय प्रौद्योगिकी संस्थान हैदराबाद

Indian Institute of Technology Hyderabad

Arsalan Ahmad Sheikh

ES19BTECH11025

20-03-2022

ABSTRACT

A decision tree is a tree in which each internal node represents a test on a feature , each leaf node represents a class label (decision taken after computing all features) and branches represent conjunctions of features that lead to those class labels. The paths from root to leaf represent classification rules. Decision tree is one of the predictive modeling approaches used in *statistics*, *data mining* and *machine learning*.

CONTENTS

INTRODUCTION

Predicting heart strokes based on age and lifestyle factors has always been a major challenge for the professionals in the health department. In this paper, we aim to predict if a person has a heart stroke based on the given factors like *Age, Blood pressure, Cholesterol, Heart Rate, Angina (Reduced blood flow to the heart)* and likewise. We will use gradient trees and gradient boost along with regular data filtering and indexing.

DECISION TREES

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.

A decision tree uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.

GRADIENT BOOSTING and XGBOOST

With a regular machine learning model, like a decision tree, we'd simply train a single model on our dataset and use that for prediction. Even if we build an ensemble, all of the models are trained and applied to our data separately.

Boosting, on the other hand, takes a more *iterative* approach. It's still technically an ensemble technique in that many models are combined together to perform the final one, but takes a more clever approach.

Rather than training all of the models in isolation of one another, boosting trains models in succession, with each new model being trained to correct the errors made by the previous ones. Models are added sequentially until no further improvements can be made.

The advantage of this iterative approach is that the new models being added are focused on correcting the mistakes which were caused by other models.

Gradient Boosting specifically is an approach where new models are trained to predict the residuals (i.e errors) of prior models. I've outlined the approach in the diagram below.

```
# %%

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd




# %%

from sklearn.metrics import accuracy_score

from xgboost import XGBClassifier

from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import OneHotEncoder

from sklearn.compose import ColumnTransformer

from scipy.interpolate import make_interp_spline




df = pd.read_csv('./data/heart.csv')

df.columns


# x is independent variables

# y consists of dependent variable

x = df.iloc[:, :-1].values

y = df.iloc[:, -1].values
```

```
# This shows a list of features that consist of null values

df[df.columns].isnull().sum()

# Luckily we have no null values



# THIS BLOCK IS USED INCASE WE HAVE ANY NULL VALUES.

# Replacing the null value for that feature with the mean of all other
values

# imputer = SimpleImputer(missing_values=np.nan, strategy='mean')

# imputer.fit(x[:, [9]])

# x[:, [9]] = imputer.transform(x[:, [9]])



# Encoding categorical data

ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1, 2,
6, 8, 10])], remainder='passthrough')

x = np.array(ct.fit_transform(x))




# function for training and testing the model

def trainAndTest(testSize, x, y):

   x_train, x_test, y_train, y_test = train_test_split(

       x, y, test_size=testSize, random_state=0)
```

```
    sc = StandardScaler()
    x_train[:, 14:20] = sc.fit_transform(x_train[:, 14:20])
    x_test[:, 14:20] = sc.transform(x_test[:, 14:20])

    classifier = XGBClassifier(use_label_encoder = False, eval_metric =
"logloss")
    classifier.fit(x_train, y_train)
    y_pred = classifier.predict(x_test)

    accuracy_score_ = accuracy_score(y_test, y_pred)
    return("{:.0f}-{:.4f}".format((1-testSize)*100, accuracy_score_))


# Checking the accuracy of our results and plotting them wrt the
percentage of data trained on.
accuracy = []
trainingSize = []
for tt in range(10, 40):
    res = trainAndTest(tt*1.0/100, x, y)
    size, accuracy_score_ = res.split('-')
    trainingSize.append(float(size))
    accuracy.append(float(accuracy_score_))
    print("Accuracy for training on {}% of the data - {}".format(size,
accuracy_score_))
```

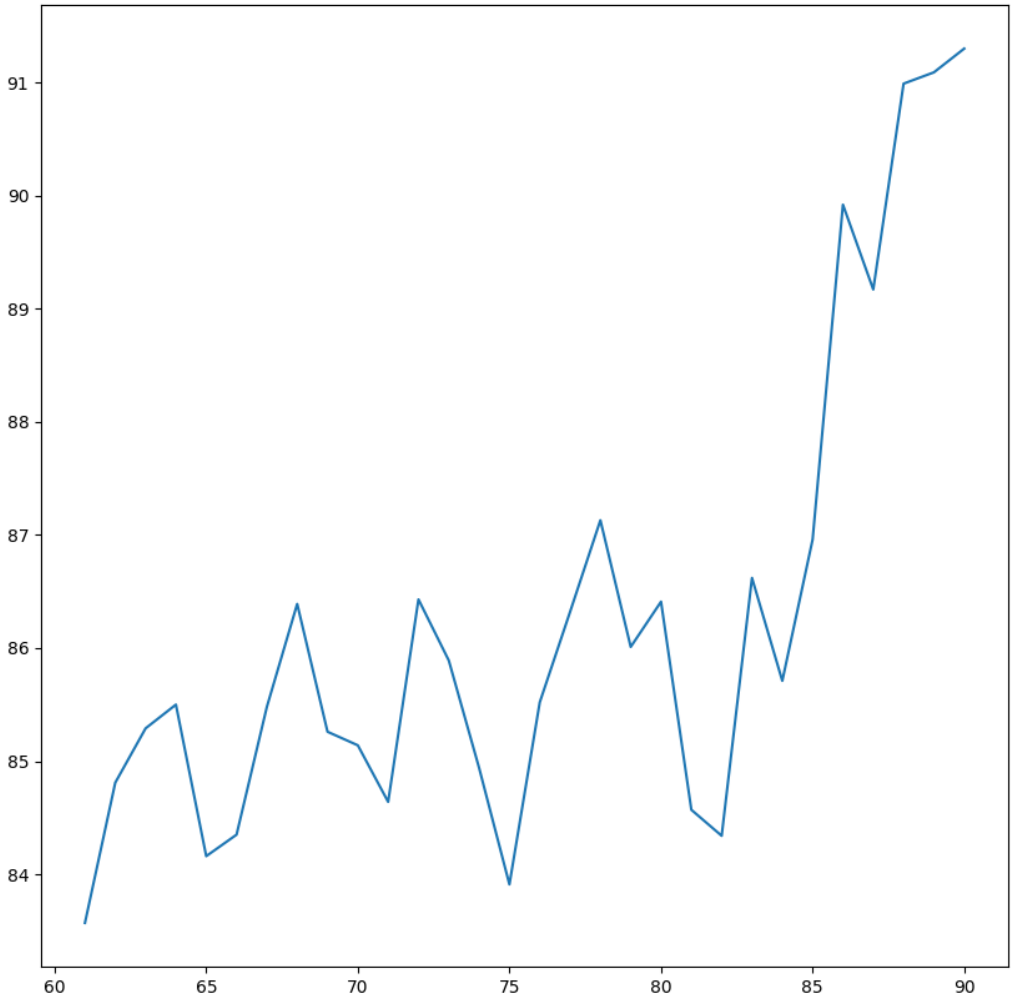
```
trainingSize = np.array(trainingSize)

accuracy = np.array(accuracy)

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

plt.plot(trainingSize, accuracy*100)

plt.show()
```



CONCLUSION

The accuracy of our model increases almost linearly with increasing size of training data but jumps to above 90% after using more than 85% as the training data.

This is an indication of either of the 2 things:

1. Our test data (of 15%) is too little to produce enough incorrect results as opposed to the correct results.
2. Our model gets better exponentially at predicting Heart Strokes in subjects.

GITHUB

All the data and files can be found on github at https://github.com/cybars69/dsa-project