به نام خدا

1. استفاده از کرنل های مختلف در حل مسائل SVM در جهت پیاده سازی تکنیک جداساز SVM روی داده هایی هست که به صورت خطی جداپذیر نیستند. مورد استفاده این کرنل ها برای این است که داده ها را به فضایی بزرگتر انتقال می دهیم تا در آن فضا خطی جدایی پذیر باشند.  
   در مورد هر کدام از کرنل ها با توجه به ساختار تابعی منحصر به فردشان می توان زمان کاربردی نسبت داد.  
   برای مثال از کرنل RBF یا همان توابع Radial Basis Function بیشتر به خاطر شباهتش به الگوریتم های KNN استفاده می شود.
2. در این سوال یک مدل SVM روی داده های مورد نظر فیت می کنیم.  
   داده ی مورد نظر هم به این شکل تولید می شود که ابتدا از دیتاست train دو ستون three\_g و clock\_speed را به عنوان داده های محور X و ستون four\_g از همین دیتاست را به عنوان داده ی y در نظر می گیریم.  
   سپس داده های تست را هم به همین شکل برای محور های ایکس و ایگرگ، این بار از دیتاست تست موجود در فایل داده ی موبایل، برمیگزینیم. کد این بخش به شکل زیر است :

train\_data = pd.read\_csv('/content/gdrive/My Drive/train.csv')

test\_data = pd.read\_csv('/content/gdrive/My Drive/test.csv')

# provide with train and test data

X\_train\_data = train\_data[['three\_g', 'clock\_speed']]

y\_train\_data = train\_data['four\_g']

X\_test\_data = test\_data[['three\_g', 'clock\_speed']]

y\_test\_data = test\_data['four\_g']

سپس مدل کلاس بند SVM خود را به این شکل روی دیتای آموزش ( train ) خود فیت می کنیم و سپس با استفاده از تابع predict و در نظر گرفتن داده ی X\_test\_data به عنوان ورودی این تابع، از مدل خود می خواهیم که حدس و گمان خود را از داده ی X\_test\_data تولید کند و مجموعه ای به شکل y test data تولید کند. کد این بخش نیز به شکل زیر است :

svm\_model = SVC().fit(X\_train\_data, y\_train\_data)

svm\_pred = svm\_model.predict(X\_test\_data)

1. برای این سوال شش حالت مختلف SVM متشکل از چهار کرنل مختلف rbf , sigmoid , linear و polynomial با پارامتر های مختلف اجرا شد و مقدار خطای هر حالت با استفاده از تابع محاسبه ی خطا   
   ( Calculate\_error ) حساب شد.  
   تابع محاسبه خطا هم به این شکل است که مقادیر داده ی واقعی که همان با y\_test\_data است را با مقدار تخمین زده شده با استفاده از SVM را نظیر به نظیر مقایسه می کند و اگر برابر نبودند به متغیر error value که داخل خود تابع با مقدار صفر تعریف شده است، یک واحد اضافه می شود. در نهایت مقدار این متغیر تقسیم بر طول آرایه ی داده ی واقعی می شود تا مقدار ارور به صورت یک عدد بین صفر و یک به ما نمایش داده شود. داریم :

def calculate\_error(y\_true, y\_pred):

error\_value = 0

for i in range(0, len(y\_true)):

if y\_true[i] != y\_pred[i]:

error\_value = error\_value + 1

error\_percentage = (error\_value)/(len(y\_true))

error = ' error is ' + str(error\_percentage)

return error

کد حالت های مختلف SVM ، همراه با خروجی بدست آمده از تابع محاسبه خطا برای هر حالت نیز به شکل زیر است :

svm\_model2 = SVC(kernel='rbf', gamma= 0.02).fit(X\_train\_data, y\_train\_data)

svm\_pred2 = svm\_model2.predict(X\_test\_data)

calculate\_error(y\_test\_data, svm\_pred2)

error is 0.269

svm\_model3 = SVC(kernel='rbf', gamma= 0.1).fit(X\_train\_data, y\_train\_data)

svm\_pred3 = svm\_model3.predict(X\_test\_data)

calculate\_error(y\_test\_data, svm\_pred3)

error is 0.269

svm\_model4 = SVC(kernel='poly', degree= 1).fit(X\_train\_data, y\_train\_data)

svm\_pred4 = svm\_model4.predict(X\_test\_data)

calculate\_error(y\_test\_data, svm\_pred4)

error is 0.269

svm\_model5 = SVC(kernel='poly', degree= 2).fit(X\_train\_data, y\_train\_data)

svm\_pred5 = svm\_model5.predict(X\_test\_data)

calculate\_error(y\_test\_data, svm\_pred5)

error is 0.269

svm\_model6 = SVC(kernel='sigmoid').fit(X\_train\_data, y\_train\_data)

svm\_pred6 = svm\_model6.predict(X\_test\_data)

calculate\_error(y\_test\_data, svm\_pred6)

error is 0.341

svm\_model7 = SVC(kernel='linear').fit(X\_train\_data, y\_train\_data)

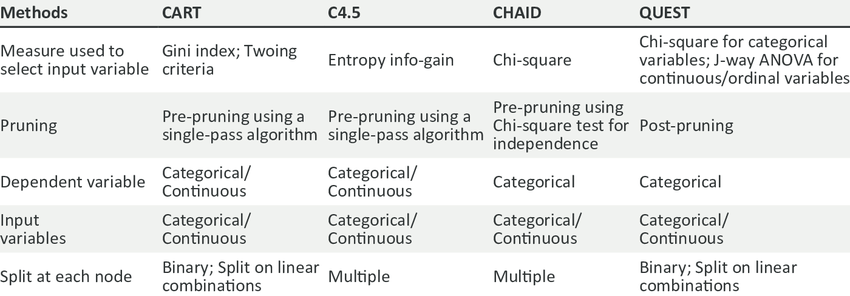
svm\_pred7 = svm\_model7.predict(X\_test\_data)

calculate\_error(y\_test\_data, svm\_pred7)

error is 0.269

1. در سوال قبلی ، مدل های 2 تا 6 ، همه از تکنیک حاشیه نرم یا همان soft margin استفاده کردند، چون کرنل های غیرخطی برای این مدل ها به کار گرفته شده است. اما مدل 7 به دلیل استفاده از کرنل خطی ، روش حاشیه سخت ( Hard Margin ) است.

7. تمام الگوریتم های مختلف ساخت درخت تصمیم برای حل جفت مسائل رگرسیون و کلاس بندی استفاده می شوند. همچنین تمام این الگوریتم ها برای هر دو نوع داده ی گسسته و پیوسته به کار گرفته شده اند. اما تفاوت هایی باهم دارند که این تفاوت ها در اصول ساخت یک درخت وجود دارد :  
برای مثال در بحث **تعیین معیار برای انتخاب ،** الگوریتم CART از روش GINI Index و Towing criteria استفاده می کند اما الگوریتم CHAID طبیعتا از روش chi-square استفاده می کند.  
چندی از تفاوت ها و شباهت های الگوریتم ها را می توانیم در عکس زیر ببینیم :



8. در این سوال با استفاده از کلاس Decision Tree Classifier، به   
ساخت درخت تصمیم می پردازیم. به دو روش این درخت ساخته شد :

الف) با استفاده از شاخص gini و ب) با استفاده از ساختار آنتروپی

ساختار کلی این سازش درخت، دارای چهار تابع است. تابع اول تابع آموزش بر اساس شاخص جینی نام دارد که کد آن بسیار ساده است و به شکل زیر قابل نمایش است :

# Function to perform training with giniIndex.

def train\_using\_gini(X\_train, X\_test, y\_train, max\_Depth, min\_samples\_Leaf):

# Creating the classifier object

clf\_gini = DecisionTreeClassifier(criterion = "gini",

random\_state = 100,max\_depth= max\_Depth,min\_samples\_leaf=min\_samples\_Leaf)

# Performing training

clf\_gini.fit(X\_train, y\_train)

return clf\_gini

همانطور که قابل ملاحظه است، این تابع داده های مورد نیاز برای ساخت درخت تصمیم را می گیرد و درخت تصمیم را بر اساس شاخص gini می سازد. سپس داده های آموزش X و Y را به مدل فیت می کند و در نهایت درخت را بر می گرداند.

تابع دوم ساختاری کاملا مشابه با تابع اول دارد و فقط به جای شاخص gini، از شاخص entropy برای ساخت مدل درخت تصمیم استفاده می شود. داریم:

def train\_using\_entropy(X\_train, X\_test, y\_train, max\_Depth, min\_samples\_Leaf):

# Decision tree with entropy

clf\_entropy = DecisionTreeClassifier(

criterion = "entropy", random\_state = 100,

max\_depth = max\_Depth, min\_samples\_leaf= min\_samples\_Leaf)

# Performing training

clf\_entropy.fit(X\_train, y\_train)

return clf\_entropy

تابع سوم برای حساب کردن تخمین ( prediction ) تعریف شده است. این تابع با استفاده از تابع predict مقادیری نظیر y test data را با مدل ساخته شده تخمین می زند و در نهایت به عنوان خروجی ، مقدار تخمین زده شده را برمی گرداند.

# Function to make predictions

def prediction(X\_test, clf\_object):

# Prediction on test with giniIndex

y\_pred = clf\_object.predict(X\_test)

print("Predicted values:")

print(y\_pred)

return y\_pred

تابع آخر برای حساب کردن دقت تخمین های بدست آمده نوشته شده است. امتیاز دقت تخمین ، ماتریس گیجی ( confusion matrix ) و گزارش کلاس بندی که شامل پارامتر های precision ، f1-score ، recall و support است، خروجی های این تابع است؛ که کد آن را می بینیم :

def cal\_accuracy(y\_test, y\_pred):

print("Confusion Matrix: ",

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

print ("Accuracy : ",

accuracy\_score(y\_test,y\_pred)\*100)

print("Report : ",

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

در آخر نیز تابعی صرفا برای جمع آوری خروجی توابع بالا می نویسیم که main نام دارد و به شرح زیر است :

def main():

clf\_gini = train\_using\_gini(X\_train\_data, X\_test\_data, y\_train\_data, 3, 5)

clf\_entropy = train\_using\_entropy(X\_train\_data, X\_test\_data, y\_train\_data, 3, 5)

# Operational Phase

print("Results Using Gini Index:")

# Prediction using gini

y\_pred\_gini = prediction(X\_test\_data, clf\_gini)

cal\_accuracy(y\_test\_data, y\_pred\_gini)

print("Results Using Entropy:")

# Prediction using entropy

y\_pred\_entropy = prediction(X\_test\_data, clf\_entropy)

cal\_accuracy(y\_test\_data, y\_pred\_entropy)

# Calling main function

if \_\_name\_\_=="\_\_main\_\_":

main()

در دو خط آخر کد بالا ، تابع main صدا زده شده است. پس خروجی تمامی توابع را به شکل زیر داریم :

Results Using Gini Index:

Predicted values:

[0 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 1 0 1

1 1 0 0 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 0 1 1 1

1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 0 1 1 0 0 1 1 1 0 1 0 1 0 1 0 0 1 0 1 0

0 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 0 1 0 1 1 0

1 1 1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 0 0 0 1

1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1 0 1 1 0 1 1 0 0 1

0 0 1 1 1 0 1 0 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 0 1 1 1 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 0 0 1 0 1 0

0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 1 1 1 0 1 1 1 1

1 1 0 0 1 1 1 0 1 1 0 1 1 0 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1

1 1 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1

1 0 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 1 1 1

0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1

0 1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 1 1 0 0 1 0 0 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 0 1 1 0 1 1 1

0 1 1 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

1 0 1 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 0

1 1 1 1 1 0 0 1 0 1 1 0 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1

1 1 1 1 0 1 1 0 0 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1

1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1 0 1 0 0 1 0 1 1 1 1 0 1

1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 1 0 1 0 0 1 1 1

0 1 1 1 0 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 1 0 1 1 0 1

1 0 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1

1 1 1 0 0 1 1 0 1 1 1 1 0 0 0 0 1 0 1 1 1 0 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1

1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1

1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 0

1]

Confusion Matrix: [[244 269]

[ 0 487]]

Accuracy : 73.1

Report : precision recall f1-score support

0 1.00 0.48 0.64 513

1 0.64 1.00 0.78 487

accuracy 0.73 1000

macro avg 0.82 0.74 0.71 1000

weighted avg 0.83 0.73 0.71 1000

Results Using Entropy:

Predicted values:

[0 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 1 0 1

1 1 0 0 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 0 1 1 1

1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 0 1 1 0 0 1 1 1 0 1 0 1 0 1 0 0 1 0 1 0

0 1 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 0 1 0 1 1 0

1 1 1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 0 0 0 1

1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1 0 1 1 0 1 1 0 0 1

0 0 1 1 1 0 1 0 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 0 1 1 1 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 0 0 1 0 1 0

0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 1 1 1 0 1 1 1 1

1 1 0 0 1 1 1 0 1 1 0 1 1 0 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1

1 1 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1

1 0 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 1 1 1

0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1

0 1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 1 1 0 0 1 0 0 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 0 1 1 0 1 1 1

0 1 1 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

1 0 1 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 0

1 1 1 1 1 0 0 1 0 1 1 0 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1

1 1 1 1 0 1 1 0 0 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1

1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1 0 1 0 0 1 0 1 1 1 1 0 1

1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 1 0 1 0 0 1 1 1

0 1 1 1 0 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 1 0 1 1 0 1

1 0 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1

1 1 1 0 0 1 1 0 1 1 1 1 0 0 0 0 1 0 1 1 1 0 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1

1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1

1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 0

1]

Confusion Matrix: [[244 269]

[ 0 487]]

Accuracy : 73.1

Report : precision recall f1-score support

0 1.00 0.48 0.64 513

1 0.64 1.00 0.78 487

accuracy 0.73 1000

macro avg 0.82 0.74 0.71 1000

weighted avg 0.83 0.73 0.71 1000

9. همانطور که در توابع اول و دوم سوال 8 مشاهده کردید ، تعداد سطح و تعداد برگ های درخت تصمیم قابل تنظیم توسط کاربر هستند. ( برای مثال در سوال قبلی درخت تصمیمی با سه سطح و پنج برگ ساختیم. ) در سوال 9 درخت تصمیم هایی با 4 سطح و 4 برگ و یکبار هم 4 سطح و 5 برگ می سازیم. جالب این است که خروجی های بدست آمده کاملا با خروجی حالت قبلی که در دو صفحه قبلی قابل مشاهده است؛ برابر است. اما این نتیجه به این معنی نیست که تغییر در تعداد برگ و سطح، تاثیری در نتیجه ندارد و تغییرات در پارامتر های این درخت، باعث تغییر نتیجه می شود.

10. بحث هرس کردن درخت زمان هایی نیاز است که تعداد کلاس ها یا در کل دسته بندی های بدست آمده توسط درخت زیادتر از حد معمول است. در واقع به این دلیل زیاد تر از حد معمول است که در کلاس های بدست آمده داده های کمی وجود دارد و برخی از این داده ها در کلاس های مختلف در حدی به هم شبیه هستند که بتوان آن ها را در یک کلاس قرار داد. در این حالت در قسمت برگ یا سطح های بالاتر ( معمولا تا دو سطح بالاتر از برگ ) بعضی از نود ها حذف می شوند و داده هایی که از آن مسیر به کلاس حذف شده رفته بودند به کلاسی که داده های مشابه داده های کلاس حذف شده دارد؛ انتقال می یابند.

13. روش درخت تصمیم هنوز به دلایل زیر قابل استفاده است:

1. پیاده سازی به مراتب راحت تر
2. رسیدن به دقت های بالا در همان مراحل اول آزمون و خطا
3. جداسازی و کلاس بندی مناسب و قابل تنظیم داده ها در این روش

در مدل های بالا نیز می بینید که دقت بدست آمده در مدل های دارای درخت تصمیم دقتی تقریبا مشابه مدل های بالا و بعضا بیشتر از بعضی از مدل ها دارد.( در مقایسه با روش SVR با کرنل sigmoid )

14. در این سوال هدف ، معرفی بعضی از الگوریتم های استخراج قانون که براساس روش رشد و هرس کردن کار می کنند است. ( روش های دیگری مانند روش های مبتنی بر درخت تصمیم هم وجود دارد مانند C4.5 )

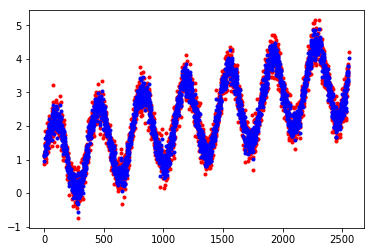
14.1 روش RIPPER :

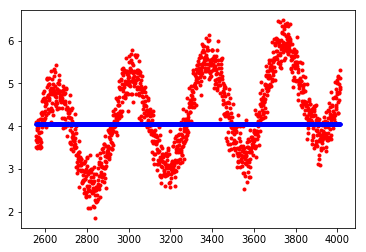
اساس این الگوریتم ، روش تقسیم و حل است. در واقع از تکنیک IREP یا همان کاهش مداوم خطای هرس برای تعیین قوانین اولی برای هر کلاس استفاده می کند. سپس هر قانون را جهت اصلاح بررسی می کند و در نتیجه هر قانون را به دو بخش قانون جایگزینی و قانون تجدیدنظر تقسیم می کند.  
در آخر این الگوریتم بررسی می کند تا قانون اصلی را نگه دارد یا یکی از قوانین جایگزینی و تجدیدنظر. به این شکل ابتدا به اندازه کافی قانون تولید می کند و در آخر قوانین اضافی را کنار می گذارد.

14.2 روش CAMUR :

این روش مخفف شده ی عبارت **کلاس بند براساس روش های جایگزین و چندتایی قانون مند** است و براساس روش RIPPER ساخته شده است. الگوریتم موردنظر با محاسبه ی یک مدل کلاس بند قانونمند قانون های ترکیبی و معادل یکدیگر زیادی را استخراج می کند و در ادامه از روش RIPPER برای هرس کردن قوانین استفاده می کند.

15. درختان تصمیم در مسائل رگرسیون برای داده های آموزش در مدل داده های سری زمانی مناسب است ؛ اما هر چه قدر که در آموزش مناسب است در مراحل ارزیابی ( validation ) و همینطور در ادامه در مرحله تست بسیار ضعیف است. مثلا در نمونه ای برای داده های غیر خطی از نوع متناوب با نوسان زیاد، یک خط روی این مدل به دست آورده است؛ که در رگرسیون ، این نتیجه بسیار افتضاح است :





عکس اول مربوط به آموزش و عکس دوم مربوط به مرحله دوم آموزش و قبل از تست ، یعنی مرحله ارزیابی است.

16. به آدرس اشاره شده مراجعه شد و داده های بیت کوین از سایت دانلود گردید. ستون Date به شکلی که در لینک منبع اشاره شده بود؛ مورد تغییر قرار گرفت که در نهایت به این شکل درآمد :

def mdy\_to\_ymd(d):

return datetime.strptime(d, '%b %d, %Y').strftime('%Y-%m-%d')

for i in range(0, 3941):

data['Date'][i] = mdy\_to\_ymd(data['Date'][i])

data['Date']

که خروجی دستور موجود در خط آخر نیز به شکل زیر است :

0 2021-05-01

1 2021-04-30

2 2021-04-29

3 2021-04-28

4 2021-04-27

...

3936 2010-07-22

3937 2010-07-21

3938 2010-07-20

3939 2010-07-19

3940 2010-07-18

Name: Date, Length: 3941, dtype: object

ستون های دیتاست همه از نوع رشته ای بودند. به همین دلیل قبل از انجام هر کاری از قبیل فیت کردن یک مدل یادگیری ماشین روی ستون های دیتاست، باید آن ها را به عدد ( ترجیحا float ) تبدیل می کردیم.

این تبدیل طی چند مرحله انجام شد:

* ابتدا از مقادیر ستون ها علامت کاما طبق فرایند زیر حذف شد:

data = data.apply(lambda x: x.str.replace(',', ''))

* سپس از مقادیر ستون Vol. ، حرف K که به معنای هزار است حذف شد که کد این بخش نیز به این ترتیب قابل مشاهده است :

data['Vol.'] = data['Vol.'].str.replace(r'\D','')

* بعد از انجام مقدمات تبدیل مقادیر رشته ای به عددی، حال نوبت به تبدیل این مقادیر به عدد می رسد :

data['High'] = data['High'] = pd.to\_numeric(data['High'], errors='coerce')

data['Open'] = data['Open'] = pd.to\_numeric(data['Open'], errors='coerce')

data['Low'] = data['Low'] = pd.to\_numeric(data['Low'], errors='coerce')

data['Vol.'] = data['Vol.'] = pd.to\_numeric(data['Vol.'], errors='coerce')

data['Price'] = data['Price'] = pd.to\_numeric(data['Price'], errors='coerce')

* بعد از انجام این تبدیل ، ممکن است برخی از مقادیری که در زمان رشته ای بودن دیتاست مقدار درستی نداشتند؛ مقدار nan بگیرند که این مشکل نیز به شکل زیر رفع شد:

**data = data.replace(np.nan, 0)**

حال، دیتاست ما آماده قرار گرفتن در معرض تکنیک های یادگیری ماشین است.

در ادامه باید داده ها به روش گفته شده جداسازی می شد که این کار نیز با دستورات زیر انجام شد :

X = data.drop(['Price', 'Date', 'Change %'], axis = 1)

y = data['Price']

X\_test, X\_train, y\_test, y\_train = train\_test\_split(X,y, test\_size = 0.9699 ,random\_state = 42, shuffle = False)

17. در این سوال از شش مدل استفاده شد که به ترتیب به آنها اشاره می کنیم

* رگرسیون خطی
* رگرسیون لاسو
* رگرسیون ریج
* روش SVR با کرنل rbf و پارامتر گامای یک صدم
* روش SVR با کرنل polynomial ( چند جمله ای )
* شبکه عصبی LSTM با لایه ورودی دارای یک نورون ، یک لایه ی مخفی ( Hidden ) با چهار نورون و لایه ی خروجی با یک نورون

مدل های یک تا پنج ، به یک شکل طراحی شده اند. ابتدا یک شی از کلاس تعریف شده برای تکنیک موردنظر در پایتون، می سازیم و سپس مدل خود را با استفاده از تابع fit، به داده فیت می کنیم. در ادامه با استفاده از تابع predict ، مقادیر تخمین زده شده برای مقدار y\_test را دریافت می کنیم.

در ادامه خطای مدل را با استفاده از روش RMSE بدست می آوریم. کد ها و خروجی های این پنج مدل به شکل زیر است :

lin\_regression = LinearRegression().fit(X\_train, y\_train)

lin\_pred = lin\_regression.predict(X\_test)

lin\_pred\_score = math.sqrt(mean\_squared\_error(y\_test, lin\_pred))

print('test score : %.2f RMSE' % (lin\_pred\_score))

test score : 961.10 RMSE

ridge = Ridge().fit(X\_train, y\_train)

ridge\_pred = ridge.predict(X\_test)

ridge\_pred\_score = math.sqrt(mean\_squared\_error(y\_test, ridge\_pred))

print('test score : %.2f RMSE' % (ridge\_pred\_score))

test score : 961.10 RMSE

lasso = Lasso().fit(X\_train, y\_train)

lasso\_pred = lasso.predict(X\_test)

lasso\_pred\_score = math.sqrt(mean\_squared\_error(y\_test, lasso\_pred))

print('test score : %.2f RMSE' % (lasso\_pred\_score))

test score : 994.15 RMSE

svr\_poly = SVR(kernel='poly').fit(X\_train,y\_train)

svr\_poly\_pred = svr\_poly.predict(X\_test)

svr\_poly\_pred\_score = math.sqrt(mean\_squared\_error(y\_test, svr\_poly\_pred))

print('test score : %.2f RMSE' % (svr\_poly\_pred\_score))

test score : 19534.72 RMSE

svr\_rbf = SVR(kernel='rbf', gamma=0.1).fit(X\_train, y\_train)

svr\_rbf\_pred = svr\_rbf.predict(X\_test)

svr\_rbf\_pred\_score = math.sqrt(mean\_squared\_error(y\_test, svr\_rbf\_pred))

print('test score : %.2f RMSE' % (svr\_rbf\_pred\_score))

test score : 49181.85 RMSE

حال به مدل LSTM می رسیم. قبل از این که مدل خود را روی داده فیت کنیم مراحلی را طی کردیم که همه آن مراحل را توضیح می دهیم :

* ابتدا تمام مقادیر دیتاست را در آرایه ذخیره می کنیم و ستون های اول   
  (Date) و آخر ( Change% ) را از این آرایه کنار می گذاریم و آرایه را به آرایه ای از جنس float تبدیل می کنیم :

dataset = data.values

dataset = np.delete(dataset,0,1)

dataset = np.delete(dataset,5,1)

dataset.astype('float32')

* آرایه بدست آمده را این بار به روش دستی ولی با همان مقادیر گفته شده در سوال 16 برای داده های آموزش و تست، جداسازی می کنیم:

train\_size = int(len(dataset) \* 0.9699)

test\_size = len(dataset) - train\_size

test, train = dataset[0:test\_size,0:1], dataset[test\_size:len(dataset),1:2]

print(len(train), len(test))

print(train.shape[1], test.shape[1])

که نتیجه ای به شکل زیر دارد :

3822 119

1 1

که این به این معنی است که آرایه ی جدا شده ی train دارای 3822 سطر و یک ستون و آرایه ی جدا شده ی test نیز دارای 119 سطر و یک ستون است.

سپس با استفاده از تابع create dataset این آرایه ها را به ماتریس تبدیل می کنیم. در ادامه به ماتریس های X train و X test بدست آمده دو ستون دیگر اضافه می کنیم که حالتی سه بعدی به خود بگیرد تا بتوانیم آنها را به تانسور تبدیل کنیم؛ چون ورودی اول تابع fit برای شبکه lstm باید تانسور باشد.

کد این بخش از تبدیلات به این صورت است :

def create\_dataset(dataset, look\_back=1):

dataX, dataY = [], []

for i in range(len(dataset)-look\_back):

a = dataset[i:(i+look\_back), 0]

dataX.append(a)

dataY.append(dataset[i + look\_back, 0])

return np.array(dataX), np.array(dataY)

look\_back = 1

trainX, trainY = create\_dataset(train, look\_back)

testX, testY = create\_dataset(test, look\_back)

trainY.shape

trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))

testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

tensor\_trainX = tf.convert\_to\_tensor(trainX, np.float32)

tensor\_testX = tf.convert\_to\_tensor(testX, np.float32)

و در نهایت مدل دنباله ای ( sequential ) شبکه عصبی LSTM را پیاده سازی می کنیم. تابع خطای مورد استفاده و همینطور بهینه ساز تابع خطای شبکه عصبی را هم به ترتیب mean square error و بهینه ساز adam در نظر می گیریم. در نهایت مدل خود را روی داده های جداسازی شده فیت می کنیم. کل داده های آموزش را برای مرحله آموزش استفاده می کنیم و این کار را در صد مرحله انجام می دهیم. پس داریم :

model = Sequential()

model.add(LSTM(4, input\_shape=(1, look\_back)))

model.add(Dense(1))

model.compile(loss='mean\_squared\_error', optimizer='adam')

model.fit(tensor\_trainX, trainY, epochs=100, batch\_size=1, verbose=2)

پس از اجرای این تکه کد، تخمین های خود را با استفاده از تابع predict بدست می آوریم و در نهایت مقدار خطای RMSE خروجی های خود را به شکل روش های قبل حساب می کنیم. کد این بخش هم به شکل زیر است :

# make predictions

trainPredict = model.predict(tensor\_trainX)

testPredict = model.predict(tensor\_testX)

#calculate root mean squared error

trainScore = math.sqrt(mean\_squared\_error(trainY, trainPredict[:,0]))

print('Train Score: %.2f RMSE' % (trainScore))

testScore = math.sqrt(mean\_squared\_error(testY, testPredict[:,0]))

print('Test Score: %.2f RMSE' % (testScore))

Train Score: 5345.90 RMSE

Test Score: 49464.87 RMSE

18. سوال 18 مربوط به روش های ensemble learning یا روش های ترکیبی یادگیری است که یک یادگیرنده ی ترکیبی در این حالت، به روش های مختلف از مخلوط یا ترکیب نتیجه ی چند یادگیرنده ی ضعیف ( تکنیک های تکین یادگیری ماشین ) یک نتیجه با واریانس و بایاس کم بدست می آورد.

از تکنیک های مرسوم، تکنیک های Bagging , Boosting و voting است که برای هر کدام از این سه روش مثال برای ارائه داریم :

**18.1: روش Voting :**

این روش بر اساس vote هایی که از مدل های تکین می گیرد خروجی نهایی را می سازد. برای پیاده سازی این روش از کتابخانه رگرسیون بر اساس رای یا Voting Regressor استفاده می کنیم.

(مدل های تکین انتخاب شده هم رگرسیون خطی، رگرسیون ریج و لاسو هستند که در واقع روش های یادگیری ترکیبی ما از ترکیب کردن نتیجه های این سه روش ، نتیجه های خود را اعلام می کنند.)

در طول اجرا به اروری برخوردیم که مشکل از لیبل ها و شمارنده ی آرایه ی y\_train بود. به همین دلیل از تابع label encoder استفاده شد تا لیبل های این آرایه را مناسب با تابع fit ، اصلاح کند.

داریم :

model\_1 = LinearRegression()

model\_2 = Lasso()

model\_3 = Ridge()

# Making the final model using voting regressor

final\_model = VotingRegressor(

estimators=[('lr', model\_1), ('ridge', model\_2), ('lasso', model\_3)])

lab\_enc = preprocessing.LabelEncoder()

encoded\_y\_train = lab\_enc.fit\_transform(y\_train)

# training all the model on the train dataset

final\_model.fit(X\_train, encoded\_y\_train)

# predicting the output on the test dataset

pred\_final = final\_model.predict(X\_test)

**18.2 روش Bagging :**

این روش کدی متفاوت با ساختار کد روش قبل دارد. اما قبل از کد اشاره ای به روش Bagging خواهیم داشت :

این روش بخش هایی از دیتاست را با طول های مشخص و مساوی انتخاب می کند و یک مدل یادگیری ماشین را روی این بخش های بدست آمده پیاده می کند. در نهایت خروجی هر درایه از آرایه ی مدل یادگیری ترکیبی ما، میانگینی خواهد بود از درایه های نظیر در نتایج بدست آمده از مدل اجرا شده روی بخش های دیتاست.  
به غیر از میانگین روش های دیگری نیز برای ترکیب نتیجه در این مرحله وجود دارد اما معمول ترین این روش ها، همان میانگین است.

کد این بخش به شکل زیر است :

final\_bag\_model = BaggingRegressor(base\_estimator=SVR(kernel='poly'),

n\_estimators=10, random\_state=0)

lab\_enc = preprocessing.LabelEncoder()

encoded\_y\_train = lab\_enc.fit\_transform(y\_train)

# training all the model on the train dataset

final\_bag\_model.fit(X\_train, encoded\_y\_train)

# predicting the output on the test dataset

pred\_final\_bag = final\_bag\_model.predict(X\_test)

**18.3 Boosting :**

ساختار کد این بخش بسیار شبیه به قبل است اما صرفا در این حالت به جای استفاده از یک مدل با ورودی های مختلف ، آن هم به طور موازی ، و میانگین گرفتن از خروجی ها ، این بار یک مدل چندین و چند بار به صورت متوالی ( سری ) مورد استفاده قرار می گیرد؛ به طوری که خروجی یک مرحله به عنوان ورودی مرحله بعدی به همان مدل داده می شود.  
روش های معمول Boosting در پایتون، AdaBoost و روش GradientBossting است که بنا بر خواسته تمرین در سوال بعدی از روش َAdaBoost استفاده شد. تمام این مراحل به صورت زیر قابل پیاده سازی هستند :

final\_Boost\_model = AdaBoostRegressor(base\_estimator=SVR(),

n\_estimators=10, random\_state=0)

lab\_enc = preprocessing.LabelEncoder()

encoded\_y\_train = lab\_enc.fit\_transform(y\_train)

# training all the model on the train dataset

final\_Boost\_model.fit(X\_train, encoded\_y\_train)

# predicting the output on the test dataset

pred\_final\_boost = final\_Boost\_model.predict(X\_test)

19. همانطور که اشاره شد، در این سوال از روش AdaBoost استفاده شد و خروجی های متعددی با پارامترهای مختلف از این روش گرفته شد؛ که در پایین می بینیم :

**حالت اول: استفاده از مدل SVR با کرنل rbf و گامای یک صدم :**

final\_Boost\_model1 = AdaBoostRegressor(base\_estimator=SVR(kernel='rbf', gamma= 0.01),

n\_estimators=10, random\_state=0)

lab\_enc = preprocessing.LabelEncoder()

encoded\_y\_train = lab\_enc.fit\_transform(y\_train)

# training all the model on the train dataset

final\_Boost\_model1.fit(X\_train, encoded\_y\_train)

# predicting the output on the test dataset

pred\_final\_boost1 = final\_Boost\_model1.predict(X\_test)

pred\_final\_boost1\_score = math.sqrt(mean\_squared\_error(y\_test, pred\_final\_boost1))

print('test score : %.2f RMSE' % (pred\_final\_boost1\_score))

test score : 48611.42 RMSE

**حالت دوم : مدل کرنل چند جمله ای درجه 2 ( سهمی ) :**

# Making the final model using Ada Boost regressor by polynomial kernel of svr and 10 weak learners

final\_Boost\_model2 = AdaBoostRegressor(base\_estimator=SVR(kernel='poly', degree = 2),

n\_estimators=10, random\_state=0)

lab\_enc = preprocessing.LabelEncoder()

encoded\_y\_train = lab\_enc.fit\_transform(y\_train)

# training all the model on the train dataset

final\_Boost\_model2.fit(X\_train, encoded\_y\_train)

# predicting the output on the test dataset

pred\_final\_boost2 = final\_Boost\_model2.predict(X\_test)

pred\_final\_boost2\_score = math.sqrt(mean\_squared\_error(y\_test, pred\_final\_boost2))

print('test score : %.2f RMSE' % (pred\_final\_boost2\_score))

test score : 39445.92 RMSE

**حالت سوم : مشابه حالت اول اما با دنباله ای به طول 20 یادگیرنده ضعیف:**

# Making the final model using Ada Boost regressor by rbf kernel of svr and 20 weak learners

final\_Boost\_model3 = AdaBoostRegressor(base\_estimator=SVR(kernel='rbf', gamma= 0.01),

n\_estimators=20, random\_state=0)

lab\_enc = preprocessing.LabelEncoder()

encoded\_y\_train = lab\_enc.fit\_transform(y\_train)

# training all the model on the train dataset

final\_Boost\_model3.fit(X\_train, encoded\_y\_train)

# predicting the output on the test dataset

pred\_final\_boost3 = final\_Boost\_model3.predict(X\_test)

pred\_final\_boost3\_score = math.sqrt(mean\_squared\_error(y\_test, pred\_final\_boost3))

print('test score : %.2f RMSE' % (pred\_final\_boost1\_score))

test score : 48611.42 RMSE

**حالت چهارم : مشابه حالت چهارم اما با دنباله ای به طول 20 یادگیرنده ضعیف :**

# Making the final model using Ada Boost regressor by polynomial kernel of svr and 20 weak learners

final\_Boost\_model4 = AdaBoostRegressor(base\_estimator=SVR(kernel='poly', degree = 2),

n\_estimators=20, random\_state=0)

lab\_enc = preprocessing.LabelEncoder()

encoded\_y\_train = lab\_enc.fit\_transform(y\_train)

# training all the model on the train dataset

final\_Boost\_model4.fit(X\_train, encoded\_y\_train)

# predicting the output on the test dataset

pred\_final\_boost4 = final\_Boost\_model2.predict(X\_test)

pred\_final\_boost4\_score = math.sqrt(mean\_squared\_error(y\_test, pred\_final\_boost4))

print('test score : %.2f RMSE' % (pred\_final\_boost4\_score))

test score : 39445.92 RMSE

20. الگوریتم Random Forest به عنوان یکی دیگر از روش های یادگیری ترکیبی است. این روش کاملا شبیه روش Bagging است؛ تنها تفاوت موجود در این روش ، استفاده از درخت های تصمیم به عنوان مدل های یادگیرنده ضعیف است. درخت های تصمیم در مسائل رگرسیون، پس از جداسازی داده ی آموزش، برای پیش بینی روی داده های تست به این شکل عمل می کنند:

نقاط داده ی تست هرکدام پس از دیگری در درخت تصمیم قرار می گیرند و پس از گذشتن از شروط نود های درخت به یکی از نود های برگ می رسند.

سپس مقدار y تخمینی با میانگین گرفتن از مقادیر y نقاط موجود در آن برگ به دست می آید.

حال کد این سوال را به شکل پیش رو میبینیم که دارای دو حالت از این الگوریتم با عمق درختان 2 سطحی و 4 سطحی است :

# Random Forest

random\_forest\_model = RandomForestRegressor(max\_depth=2,random\_state=0)

#encode our target

lab\_enc = preprocessing.LabelEncoder()

encoded\_y\_train = lab\_enc.fit\_transform(y\_train)

# train our random forest based model :

random\_forest\_model.fit(X\_train,encoded\_y\_train)

# predicting the output on the test dataset

random\_forest\_model\_pred = random\_forest\_model.predict(X\_test)

random\_forest\_model\_pred\_score = math.sqrt(mean\_squared\_error(y\_test, random\_forest\_model\_pred))

print('test score : %.2f RMSE' % (random\_forest\_model\_pred\_score))

test score : 47243.62 RMSE

# Random Forest (second edition with 4 layers)

random\_forest\_model4 = RandomForestRegressor(max\_depth=4,random\_state=0)

#encode our target

lab\_enc = preprocessing.LabelEncoder()

encoded\_y\_train = lab\_enc.fit\_transform(y\_train)

# train our random forest based model :

random\_forest\_model4.fit(X\_train,encoded\_y\_train)

# predicting the output on the test dataset

random\_forest\_model4\_pred = random\_forest\_model4.predict(X\_test)

random\_forest\_model4\_pred\_score = math.sqrt(mean\_squared\_error(y\_test, random\_forest\_model4\_pred))

print('test score : %.2f RMSE' % (random\_forest\_model4\_pred\_score))

test score : 46942.08 RMSE

21. خروجی های سوالات 17 تا 20 همه به شکل RMSE حساب شده که در تکه کد ها کاملا مشهود است. برای رسیدن به این خروجی هم از تابع آماده پایتون به اسم mean square error استفاده شد و در نهایت با استفاده از تابع math.sqrt ، خروجی RMSE تولید شد.

اما برای تابع مورد نظر سوال 21 نیز به این شکل عمل شد :

ابتدا فرمول خطای درصدی RMSE را با استفاده از توابع کتابخانه نامپای حساب می کنیم. سپس شرط گفته شده را بررسی می کنیم که اگر این مقدار حساب شده که در واقع امتیاز ( score ) مقدار تخمینی است، بیش از 0.95 بود، یعنی خطای ما کمتر از 5 صدم است و برچسب درست میگیریم. کد این بخش نیز به این شکل است :

def ignore\_5\_percent\_error (y\_true, y\_predict):

rmspe = np.sqrt(np.mean(np.square(((y\_true - y\_predict) / y\_true)), axis = 0))

if rmspe > 0.95 :

print('RMSPE value and Prediction status are ' + str(rmspe) + ' and True')

else :

print('RMSPE value and Prediction status are ' + str(rmspe) + ' and False')

return

ignore\_5\_percent\_error(y\_test, random\_forest\_model4\_pred)

RMSPE value and Prediction status are 0.9406361017366941 and False

به عنوان مثال هم خروجی رندوم فارست چهارسطحی هم به این تابع داده شد و به دلیل اینکه این خروجی از 0.95 کمتر بود خروجی نادرست گرفتیم.

22. یک ستون به دیتاست طوری اضافه می کنیم که اگر مقدار قیمت اپن از مقدار قبلیش بیشتر بود مقدار ستون جدید برابر با یک و اگر کمتر باشد مقدار منفی یک می گیرد. در صورت تساوی مقدار فعلی با مقدار بالایی ستون اپن هم مقدار صفر به ستون جدید تعلق می گیرد.

سپس شبکه عصبی LSTM را به همان روش قبلی فقط این بار با مقادیر تست و آموزش جدید - که به ترتیب ستون جدیدا ساخته شده و ستون اپن است - پیاده سازی می کنیم. داریم :

# splitting datas for new lstm network :

x22 = data22['Open'].values

y22 = data22['Open-diff'].values

xtest, xtrain, ytest, ytrain = train\_test\_split(x22, y22, test\_size = 0.9699, random\_state = 42, shuffle=False)

ytrain

array([-1.0, -1.0, -1.0, ..., 0.0, -1.0, 0], dtype=object)

# reshape input

xtrain = np.reshape(xtrain, (xtrain.shape[0], 1))

xtest = np.reshape(xtest, (xtest.shape[0], 1))

ytrain = np.reshape(ytrain, (y train.shape[0], 1))

ytrain.shape

(3823, 1)

# reshape input to be [samples, time steps, features]

xtrain = np.reshape(xtrain, (xtrain.shape[0], 1, xtrain.shape[1]))

xtest = np.reshape(xtest, (xtest.shape[0], 1, xtest.shape[1]))

ytrain = np.reshape(ytrain, (ytrain.shape[0], 1, ytrain.shape[1]))

Ytrain.shape

TensorShape([3823, 1, 1])

# create and fit the LSTM network

model = Sequential()

model.add(LSTM(4, input\_shape=(1, look\_back)))

model.add(Dense(1))

model.compile(loss='mean\_squared\_error', optimizer='adam')

model.fit(tensor\_xtrain, tensor\_ytrain, epochs=100, batch\_size=1, verbose=2)

Epoch 1/100

3823/3823 - 6s - loss: 0.8916

Epoch 2/100

3823/3823 - 4s - loss: 0.8904

Epoch 3/100

3823/3823 - 4s - loss: 0.8904

Epoch 4/100

3823/3823 - 4s - loss: 0.8904

Epoch 5/100

3823/3823 - 4s - loss: 0.8905

Epoch 6/100

3823/3823 - 4s - loss: 0.8905

Epoch 7/100

3823/3823 - 4s - loss: 0.8901

Epoch 8/100

3823/3823 - 4s - loss: 0.8904

Epoch 9/100

3823/3823 - 4s - loss: 0.8902

Epoch 10/100

3823/3823 - 4s - loss: 0.8904

Epoch 11/100

3823/3823 - 4s - loss: 0.8900

Epoch 12/100

3823/3823 - 4s - loss: 0.8896

Epoch 13/100

3823/3823 - 4s - loss: 0.8904

Epoch 14/100

3823/3823 - 4s - loss: 0.8900

Epoch 15/100

3823/3823 - 4s - loss: 0.8904

Epoch 16/100

3823/3823 - 4s - loss: 0.8903

Epoch 17/100

3823/3823 - 4s - loss: 0.8901

Epoch 18/100

3823/3823 - 4s - loss: 0.8898

Epoch 19/100

3823/3823 - 4s - loss: 0.8901

Epoch 20/100

3823/3823 - 4s - loss: 0.8903

Epoch 21/100

3823/3823 - 4s - loss: 0.8897

Epoch 22/100

3823/3823 - 4s - loss: 0.8897

Epoch 23/100

3823/3823 - 4s - loss: 0.8903

Epoch 24/100

3823/3823 - 4s - loss: 0.8902

Epoch 25/100

3823/3823 - 4s - loss: 0.8899

Epoch 26/100

3823/3823 - 4s - loss: 0.8903

Epoch 27/100

3823/3823 - 4s - loss: 0.8902

Epoch 28/100

3823/3823 - 4s - loss: 0.8901

Epoch 29/100

3823/3823 - 4s - loss: 0.8902

Epoch 30/100

3823/3823 - 4s - loss: 0.8903

Epoch 31/100

3823/3823 - 4s - loss: 0.8903

Epoch 32/100

3823/3823 - 4s - loss: 0.8903

Epoch 33/100

3823/3823 - 4s - loss: 0.8900

Epoch 34/100

3823/3823 - 4s - loss: 0.8900

Epoch 35/100

3823/3823 - 4s - loss: 0.8902

Epoch 36/100

3823/3823 - 4s - loss: 0.8901

Epoch 37/100

3823/3823 - 4s - loss: 0.8902

Epoch 38/100

3823/3823 - 4s - loss: 0.8903

Epoch 39/100

3823/3823 - 4s - loss: 0.8901

Epoch 40/100

3823/3823 - 4s - loss: 0.8902

Epoch 41/100

3823/3823 - 4s - loss: 0.8900

Epoch 42/100

3823/3823 - 4s - loss: 0.8904

Epoch 43/100

3823/3823 - 4s - loss: 0.8902

Epoch 44/100

3823/3823 - 4s - loss: 0.8902

Epoch 45/100

3823/3823 - 4s - loss: 0.8896

Epoch 46/100

3823/3823 - 4s - loss: 0.8903

Epoch 47/100

3823/3823 - 4s - loss: 0.8901

Epoch 48/100

3823/3823 - 4s - loss: 0.8899

Epoch 49/100

3823/3823 - 4s - loss: 0.8898

Epoch 50/100

3823/3823 - 4s - loss: 0.8903

Epoch 51/100

3823/3823 - 4s - loss: 0.8899

Epoch 52/100

3823/3823 - 4s - loss: 0.8904

Epoch 53/100

3823/3823 - 4s - loss: 0.8903

Epoch 54/100

3823/3823 - 4s - loss: 0.8900

Epoch 55/100

3823/3823 - 4s - loss: 0.8903

Epoch 56/100

3823/3823 - 4s - loss: 0.8902

Epoch 57/100

3823/3823 - 4s - loss: 0.8898

Epoch 58/100

3823/3823 - 4s - loss: 0.8900

Epoch 59/100

3823/3823 - 4s - loss: 0.8902

Epoch 60/100

3823/3823 - 4s - loss: 0.8902

Epoch 61/100

3823/3823 - 4s - loss: 0.8902

Epoch 62/100

3823/3823 - 4s - loss: 0.8901

Epoch 63/100

3823/3823 - 4s - loss: 0.8889

Epoch 64/100

3823/3823 - 4s - loss: 0.8906

Epoch 65/100

3823/3823 - 4s - loss: 0.8899

Epoch 66/100

3823/3823 - 4s - loss: 0.8899

Epoch 67/100

3823/3823 - 4s - loss: 0.8900

Epoch 68/100

3823/3823 - 4s - loss: 0.8902

Epoch 69/100

3823/3823 - 4s - loss: 0.8902

Epoch 70/100

3823/3823 - 4s - loss: 0.8902

Epoch 71/100

3823/3823 - 4s - loss: 0.8900

Epoch 72/100

3823/3823 - 4s - loss: 0.8900

Epoch 73/100

3823/3823 - 4s - loss: 0.8903

Epoch 74/100

3823/3823 - 4s - loss: 0.8901

Epoch 75/100

3823/3823 - 4s - loss: 0.8904

Epoch 76/100

3823/3823 - 4s - loss: 0.8898

Epoch 77/100

3823/3823 - 4s - loss: 0.8896

Epoch 78/100

3823/3823 - 4s - loss: 0.8902

Epoch 79/100

3823/3823 - 4s - loss: 0.8902

Epoch 80/100

3823/3823 - 4s - loss: 0.8897

Epoch 81/100

3823/3823 - 4s - loss: 0.8902

Epoch 82/100

3823/3823 - 4s - loss: 0.8902

Epoch 83/100

3823/3823 - 4s - loss: 0.8900

Epoch 84/100

3823/3823 - 4s - loss: 0.8902

Epoch 85/100

3823/3823 - 4s - loss: 0.8903

Epoch 86/100

3823/3823 - 4s - loss: 0.8902

Epoch 87/100

3823/3823 - 4s - loss: 0.8898

Epoch 88/100

3823/3823 - 4s - loss: 0.8902

Epoch 89/100

3823/3823 - 4s - loss: 0.8903

Epoch 90/100

3823/3823 - 4s - loss: 0.8901

Epoch 91/100

3823/3823 - 4s - loss: 0.8902

Epoch 92/100

3823/3823 - 4s - loss: 0.8902

Epoch 93/100

3823/3823 - 4s - loss: 0.8897

Epoch 94/100

3823/3823 - 4s - loss: 0.8903

Epoch 95/100

3823/3823 - 4s - loss: 0.8898

Epoch 96/100

3823/3823 - 4s - loss: 0.8903

Epoch 97/100

3823/3823 - 4s - loss: 0.8902

Epoch 98/100

3823/3823 - 4s - loss: 0.8901

Epoch 99/100

3823/3823 - 4s - loss: 0.8899

Epoch 100/100

3823/3823 - 4s - loss: 0.8899

<tensorflow.python.keras.callbacks.History at 0x7f8fa3baec10>

و در نهایت هم خروجی مدل lstm را به این شکل داریم :

# make predictions

trainPredict22 = model.predict(tensor\_xtrain)

testPredict22 = model.predict(tensor\_xtest)

ytrain = np.reshape(ytrain, (ytrain.shape[0], 1))

#calculate root mean squared error

trainScore22 = math.sqrt(mean\_squared\_error(ytrain, trainPredict22[:,0]))

print('Train Score: %.2f RMSE' % (trainScore22))

testScore22 = math.sqrt(mean\_squared\_error(ytest, testPredict22[:,0]))

print('Test Score: %.2f RMSE' % (testScore22))

Train Score: 0.94 RMSE

Test Score: 1.00 RMSE