به نام خدا

دانشکده مهندسی برق

تمرین پیاده سازی ۲

درس هوش مصنوعی

محمدامین حاجی خداوردیان

۹۷۱۰۱۵۱۸

استاد:دکتر عبدی

نیمسال اول ۱۴۰۰-۱۴۰۱

در این تمرین به ساخت و بررسی درخت تصمیم پرداختیم. در ابتدا کمی با بخش های مختلف کد پیاده سازی شده آشنا می‌شویم و در ادامه به بررسی و بیان مشکلات می‌پردازیم.

برای پیاده‌سازی درخت تصمیم ما نیاز به محاسبه آنتروپی(Entropy) و دست‌آورد اطلاعات(Information Gain) داریم. بنابراین باید در ابتدا آن ها را محاسبه کنیم. همانطور که در کلاس درس گفته شد برای محاسبه آنتروپی از رابطه زیر استفاده می‌کنیم:

$$\text{Entropy:} \quad H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k)$$

در قطعه کدی که در زیر آورده شده است همین بخش پیاده‌سازی شده است:

```
def Entropy(Example , Attribute):
    Column = Example[Attribute]
    NumData , remainder = pd.factorize(Column)
    Column = NumData
    Vk = np.bincount(Column) # Determine different values that our attribute can have
    Entropy = 0
    Probabilities = Vk / len(Column)  #Find probability of each Vk

    for Prob in Probabilities:
        if Prob > 0 :
            Entropy = Entropy + Prob*np.log2(Prob)
    return -Entropy
```

که در آن ابتدا مقادیر مختلفی که می‌تواند داشته باشد را به صورت اعداد طبیعی درآورده سپس تعداد هر کدام از اعداد مختلف را میابیم تا بتوانیم در ادامه احتمال هرکدام را محاسبه کنیم و سپس با استفاده از رابطه آورده شده به محاسبه آنتروپی می‌پردازیم.

برای محاسبه دست‌آورد اطلاعات نیز از رابطه گفته شده در درس استفاده کرده ایم که این رابطه به شکل زیر است:

$$Gain(A) = B(\frac{p}{p+n}) - Remainder(A)$$

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k+n_k}{p+n} B(\frac{p_k}{p_k+n_k})$$

این بخش نیز به صورت زیر پیاده سازی شده است:

```
def InformationGain(Examples , Attribute , ResultIndex):
    EntropyRoot = Entropy(Example=Examples , Attribute=ResultIndex)# Find entropy of root
    DiffValue = Examples[Attribute].unique()#find different Values that our attribute can have
    Vk = list()
    for Value in DiffValue:
        Vk.append(Examples[Examples[Attribute] == Value])# Differentiate Examples with respect to different Values of chosen Attribute
    Remainder = 0
    for i in range(len(Vk)):
        prob = Vk[i].shape[0]/Examples.shape[0] #Prob is pk+nk / p+n
        Remainder = Remainder + prob*Entropy(Example=Vk[i] , Attribute=ResultIndex)
    return EntropyRoot - Remainder
```

همانطور که دیده می‌شود ابتدا آنتروپی داده‌ها حساب شده و سپس با توجه به ویژگی(Attribute) داده شده به تابع که در واقع برحسب آن جداسازی قرار است رخ دهد مقادیری که این Attribute می‌تواند داشته باشد را محاسبه کرده و پس از آن با استفاده از تعداد داده‌های هر بخشی که این Attribute می‌سازد به تعداد کل احتمال انتخاب آن شاخه محاسبه شده و پس از آن با استفاده از رابطه داده شده در صفحه قبل به محاسبات می‌پردازیم.

حال به سراغ الگوریتم درخت تصمیم می‌رویم. با توجه به مطالب درس داریم:

function DECISION-TREE-LEARNING(*examples*, *attributes*, *parent_examples*) **returns** a tree

if *examples* is empty **then return** PLURALITY-VALUE(*parent_examples*)
else if all *examples* have the same classification **then return** the classification
else if *attributes* is empty **then return** PLURALITY-VALUE(*examples*)
else
  $A \leftarrow \text{argmax}_{a \in attributes}$ IMPORTANCE(a, *examples*)
  tree $\leftarrow$ a new decision tree with root test A
  for each value v_k of A **do**
    $exs \leftarrow \{e : e \in examples \text{ and } e.A = v_k\}$
    subtree $\leftarrow$ DECISION-TREE-LEARNING(exs, $attributes - A$, *examples*)
    add a branch to *tree* with label $(A = v_k)$ and subtree *subtree*
  return *tree*

Pseudo Code بالا نمایی از کل الگوریتمی است که ما برای پیاده سازی به آن نیاز داریم.

در ابتدا به سراغ بررسی تابع Plurality Value آورده شده در الگوریتم می‌رویم.

این بخش به صورت زیر پیاده‌سازی شده است:

```
def PluralityValue(ParentExamples , ResultIndex):
    length = ParentExamples.shape[0]
    Result = list(ParentExamples[ResultIndex])
    Value1 = Result[0]
    Sum1 = 0
    Sum2 = 0
    for i in range(length):
        if Result[i] == Value1:
            Sum1 = Sum1 + 1
        else:
            Value2 = Result[i]
            Sum2 = Sum2 + 1
    if Sum1 > Sum2:
        return node(Attribute=Value1)
    else:
        return node(Attribute=Value2)
```

که در آن با توجه به نمونه های موجود آن نمونه‌ای که تعداد بیشتری از آن را در اختیار داریم به عنوان خروجی به صورت یک گره از درخت اصلی در نظر می‌گیریم. درخت ما به صورت زیر است:

```
class node:
    def __init__(self , Attribute):
        self.Attribute = Attribute
        self.Examples = None
        self.Value = []
        self.Child = []
        self.InformationGain = None

    def add_child(self, obj):
        self.Child.append(obj)

    def add_Value(self, obj):
        self.Value.append(obj)
```

که هر کدام از ویژگی های آن به صورت زیر است:

- Attribute: ویژگی یا مقداری که درخت با توجه به آن دسته بندی شده است را نگه می‌دارد
- Examples: نمونه هایی که هر گره درخت دارد را نگه می‌دارد.
- Value: مقادیری که با دسته بندی با استفاده از Attribute به آن رسیده‌ایم را نگه می‌دارد.
- Child: فرزندان هر گره را نگه می‌دارد
- Information Gain: دست‌آورد اطلاعات هر گره را ذخیره می‌کند

الگوریتم گفته شده در صفحات قبل به صورت زیر پیاده‌سازی شده است:

```
def DecisionTreeLearning(Examples , Attribute , ParentExamples , ResultIndex , DiffValues , AttributeAll):
    Attributes = Attribute
    CheckVa = CheckAllSame(Examples , ResultIndex)
    if Examples.empty:
        return PluralityValue(ParentExamples , ResultIndex=ResultIndex)
    elif Attributes == []:
        return PluralityValue(ParentExamples , ResultIndex=ResultIndex)
    elif CheckVa != -1:
        return node(Attribute=CheckVa)
    else:
        ChoesnAttribute , InformationGainValue = Importance(Examples=Examples , Attributes=Attributes , ResultIndex=ResultIndex)
        Tree = node(Attribute=ChoesnAttribute)
        Tree.InformationGain = InformationGainValue
        Tree.Examples = Examples
        AttIndex = AttributeAll.index(ChoesnAttribute)
        DiffValue = DiffValues[AttIndex]#find different Values that our attribute can have
        Vks = list()
        for Value in DiffValue:
            Tree.add_Value(Value)
            Vks.append(Examples[Examples[ChoesnAttribute] == Value])# Differentiate Examples with respect to different Values of chosen Attribut
        AttributesT = copy.deepcopy(Attributes)
        AttributesT.remove(ChoesnAttribute)
        for Vk in Vks:
            SubTree = DecisionTreeLearning(Examples=Vk , Attribute=AttributesT , ParentExamples=Examples ,ResultIndex=ResultIndex , DiffValues=D
            SubTree.Examples = Vk
            Tree.add_child(SubTree)
        return Tree
```

همانطور که دیده می‌شود روند اصلی دقیقا مشابه الگوریتم گفته شده است حال به سراغ تابع Importance می‌رویم.

```
def Importance(Attributes , Examples , ResultIndex):
    InformationGainValues = list()
    for Attribute in Attributes:
        InformationGainValues.append(InformationGain(Examples=Examples , Attribute=Attribute , ResultIndex=ResultIndex))
    ImportantAttribute = argmax(InformationGainValues)
    InformationGainValue = max(InformationGainValues)
    return Attributes[ImportantAttribute] , InformationGainValue
```

در این تابع مجموعه تمام Attribute های موجود را گرفته و برای هر کدام Information Gain محاسبه می‌کنیم و Attribute ای که بیشترین دست‌آورد اطلاعات را دارد را به عنوان بهترین گزینه برای جداسازی برمی‌گردانیم.

یک تابع با عنوان CheckAllSame وجود دارد که چک می‌کند که اگر تمام خروجی‌ها یک مقدار را داشتند در این صورت گره ما برابر آن مقدار شود.

درخت تصمیم در صورتی که تعداد داده ها زیاد باشد نیاز به آن دارد که هرس شود. بنابراین ما بخشی طراحی کردیم که به هرس درخت می‌پردازد که در صفحه بعد آن را توضیح خواهیم داد.

کد هرس به شکل زیر است:

```
def Pruning(Tree , ResultIndex , VisitedChild):
    Vks = []
    for child in Tree.Child:
        Vks.append(child)
    flag = 0
    for child in Tree.Child:
        if child.Attribute == 1 or child.Attribute == 0:
            VisitedChild.append(child)
            flag = flag + 1

    if flag == len(Tree.Child):
        if Tree.InformationGain is None:
            return
        if Tree.InformationGain < 0.1:
            Attribute = PluralityValue(ParentExamples=Tree.Examples , ResultIndex=ResultIndex)
            Tree.Attribute = Attribute.Attribute
            Tree.InformationGain = None
            Tree.Child = []
            return
        else:
            return
```

```
    else:
        sum = 0
        for child in Tree.Child:
            if child in VisitedChild:
                sum = sum + 1
        if len(Tree.Child) == sum:
            VisitedChild.append(Tree)
        if Tree in VisitedChild:
            return
        else:
            for Vk in Vks:
                Pruning(Tree=Vk , ResultIndex=ResultIndex , VisitedChild=VisitedChild)
```

در آن ابتدا به سراغ گره‌هایی می‌رویم که فقط فرزند برگ دارند و با توجه به دست‌آورد اطلاعات آن‌ها اگر مقدارش کمتر از ۰.۱ بود گره حذف شده و با توجه به نمونه‌های موجود آن گره یک برگ را قرار می‌دهیم در غیر این صورت فرزندان را نشانه گذاری کرده و به عقب باز می‌گردیم برای اطمینان از هرس گره‌ها این عملیات را به دفعات زیاد انجام می‌دهیم.

در انتها برای بررسی میزان دقت درخت تصمیم نیاز به یک ارزیاب وجود دارد که نمونه ورودی را با توجه به درخت ما بررسی کند که این بخش را در ادامه به توضیح آن می‌پردازیم.

```
def EvaluateTree(Tree , Test):
    Attributes = list(Test.columns)
    ResultIndex = len(Attributes)-1
    Corr = 0
    InCorr = 0
    for i in range(len(Test)):
        Testrow = list(Test.iloc[i])
        flag = 1
        Root = copy.deepcopy(Tree)
        while flag:
            argVal = None
            TreeAttribute = Root.Attribute
            argAtt = Attributes.index(TreeAttribute)
            TestValue = Testrow[argAtt]
            TreeValues = Root.Value
            for j in range(len(TreeValues)):
                if TestValue == TreeValues[j]:
                    argVal = j
                    break
```

```
        if argVal == None:
            InCorr = InCorr + 1
            flag = 0
        else:
            if Root.Child[argVal].Attribute == 0 or Root.Child[argVal].Attribute == 1:
                if Testrow[ResultIndex] == Root.Child[argVal].Attribute:
                    Corr = Corr + 1
                    flag = 0
                else:
                    InCorr = InCorr + 1
                    flag = 0
            else:
                Root = Root.Child[argVal]
                continue
return Corr/(Corr + InCorr)
```

در این بخش در ابتدا با توجه به درخت به ریشه آن نگاه کرده و چک می‌کنیم کدام ویژگی مدنظر است سپس با توجه به مقداری که آن ویژگی در ورودی دارد به سراغ آن شاخه می‌رویم و این کار را تا جایی ادامه می‌دهیم که به برگ ها برسیم در صورتی که درخت تصمیم خروجی را درست حدس زد یک مقدار به حدس زده های درست اضافه می‌شود و اگر آن را غلط حدس زد در این صورت یک عدد به مقدار نادرست‌ها در انتها نسبت تعداد درست به کل را به عنوان دقت درخت برمی‌گردانیم.

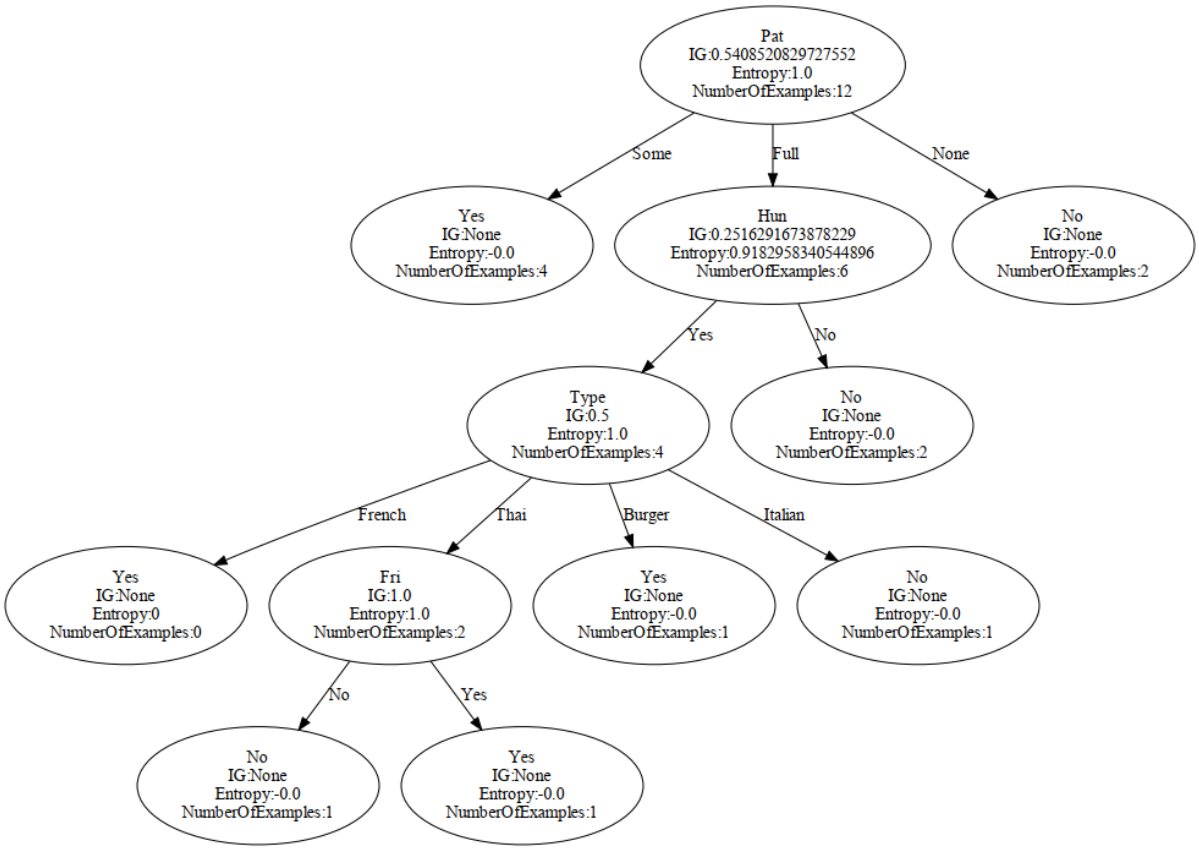
در ادامه به سراغ تست و نمایش درخت‌های گفته شده می‌پردازیم و در ابتدا به سراغ مثال رستوران که در کلاس درس به آن اشاره شد بحث می‌کنیم پس از آن به سراغ مسئله اصلی و مشکلاتی که در آن وجود دارد می‌رویم.

مسئله رستوران:

برای این بخش یک مجموعه داده با عنوان Test.csv قرار داده شده است و با توجه به قطعه کد زیر به سراغ ساخت آن درخت می‌رویم:

```
#Test For Restaurant
Examples = pd.read_csv('D:\Electrical_course\T7\AI\HW_pr\Imp2\Test.csv')
Attributes = list(Examples.columns)
ResultIndex = Attributes[len(Attributes)-1]
Attributes.remove(ResultIndex)
DiffValues = FindValues(Examples=Examples , Attributes=Attributes)
Tree = DecisionTreeLearning(Examples=Examples , Attribute=Attributes , ParentExamples=Examples , ResultIndex=ResultIndex , DiffValues=DiffValue
for i in range(2000):
    Pruning(Tree=Tree , ResultIndex=ResultIndex , VisitedChild=[])
TreeResturant = PrintTree(Tree , ResultIndex=ResultIndex)
TreeResturant.render('TreeRestaurant.gv', view=True)
```

درخت ساخته شده تحت عنوان یک pdf در محلی که کد در آن اجرا شده است ساخته شده است با باز کردن آن شکل درخت به صورت زیر است:



همانطور که در درخت صفحه قبل مشاهده می‌شود این درخت دقیقا مشابه آن چیزی است که در جزوه آورده شده است هم چنین در آن مقادیر Information Gain و آنتروپی گره اول آورده شده است که دقیقا برابر مطالب گفته شده در درس است:

$$Gain(Patrons) = 1 - \left[\frac{2}{12}B(\frac{0}{2}) + \frac{4}{12}B(\frac{4}{4}) + \frac{6}{12}B(\frac{2}{6})\right] \approx 0.541 \text{ bits},$$

درخت اطلاعاتی که در اختیار ما قرار می‌دهد:

- گره های درخت
  - Attribute
  - Information Gain
  - Entropy
  - NumberOfExamples یا تعداد نمونه هایی که در آن وجود دارد
- یال های درخت
  - دسته‌های مختلفی که با انتخاب یک Attribute خاص وجود دارد.

حال به سراغ مسئله دیگر خود می‌رویم

مسئله دیابت:

در این مسئله ابتدا نیاز است که گسسته سازی انجام شود و همانطور که در تمرین اشاره شد ما از روش بازه بندی های مساوی استفاده کرده ایم. این یک روش بسیار ساده است اما داده ها به خوبی دسته بندی نمی‌شوند به همین دلیل این روش دسته بندی قطعا در دقت درخت تاثیر می‌گذارد و آن را کاهش می‌دهد. برای گسسته سازی از تابع زیر استفاده شده است:

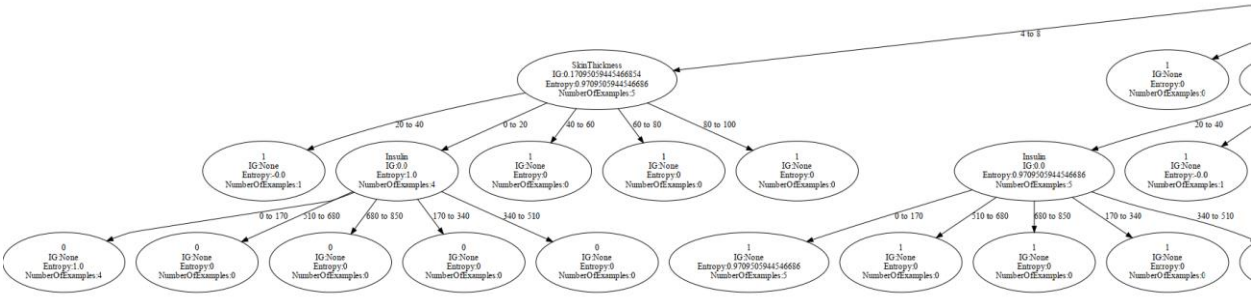
```
def Discretize(Data , n , attribute):
    Column = list(Data[attribute])
    MinValue = min(Column)
    MaxValue = max(Column)
    Step = (MaxValue - MinValue)/n
    for i in range(len(Column)):
        mini = MinValue
        if Column[i] < MinValue:
            Data.loc[i , attribute] = str(MinValue)+'<'
        if Column[i] > MaxValue:
            Data.loc[i , attribute] = '>'+str(MaxValue)
        for j in range(n):
            if mini <= Column[i] and Column[i] <= math.ceil(mini+Step):
                Data.loc[i , attribute] = str(mini)+' to '+str(math.ceil(mini+Step))
            mini = math.ceil(mini+Step)
    return
```

حال به سراغ تست آن با توجه به کد زیر می‌رویم:

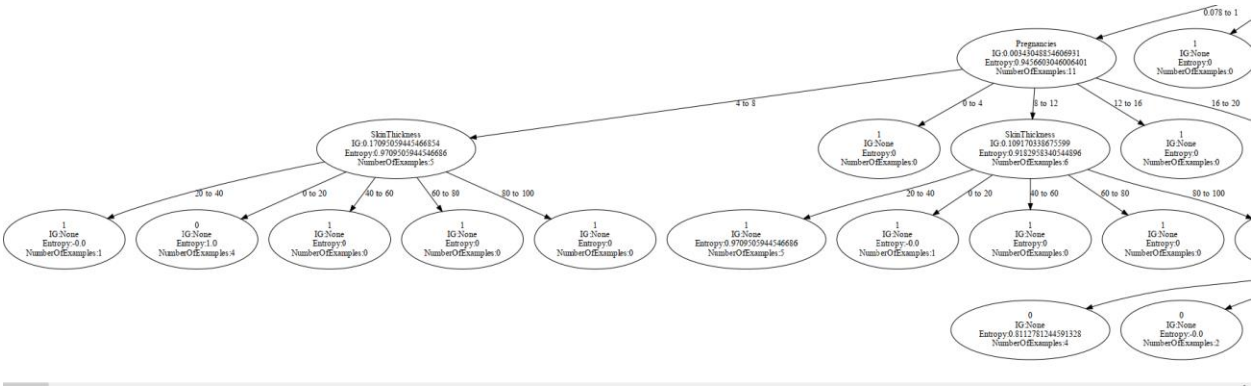
```
#Test For Diabetes Data
As = list()
Data = pd.read_csv('D:\Electrical_course\T7\AI\HW_pr\Imp2\diabetes.csv')
Attributes = list(Data.columns)
ResultIndex = Attributes[len(Attributes)-1]
Attributes.remove(ResultIndex)
for attribute in Attributes:
    Discretize(Data=Data , n=5 , attribute=attribute)
Data.to_csv('TestMake.csv' , index=False)
Examples = pd.read_csv('D:\Electrical_course\T7\AI\HW_pr\Imp2\TestMake.csv')
Attributes = list(Examples.columns)
ResultIndex = Attributes[len(Attributes)-1]
Attributes.remove(ResultIndex)
DiffValues = FindValues(Examples=Examples , Attributes=Attributes)
rng = np.random.RandomState()

train = Examples.sample(frac=0.8, random_state=rng)
test = Examples.loc[~Examples.index.isin(train.index)]
Tree = DecisionTreeLearning(Examples=train , Attribute=Attributes , ParentExamples=train , ResultIndex=ResultIndex , DiffValues=DiffValues ,
TreeDiabete1 = PrintTree(Tree , ResultIndex=ResultIndex)
TreeDiabete1.render('TreeDiabete1.gv', view=True)
for i in range(2000):
    Pruning(Tree=Tree , ResultIndex=ResultIndex , VisitedChild=[])
TreeDiabete2 = PrintTree(Tree , ResultIndex=ResultIndex)
TreeDiabete2.render('TreeDiabete2.gv', view=True)
As.append(EvaluateTree(Tree=Tree , Test=train))
As.append(EvaluateTree(Tree=Tree , Test=test))
print('Accuracy on Train Data:',As[0],'\n','Accuracy on Test Data:',As[1])
```

در تابع discretize با تغییر n می‌توان تعداد دسته بندی ها را مشخص کرد و در ادامه در بخش train می‌توان درصد داده های train را مشخص کرد که پیش فرض ۸۰ درصد است سپس یک بار درخت بدون هرس با نام TreeDiabete1 به صورت pdf ساخته می‌شود و یک بار پس از هرس با نام TreeDiabete2 و در انتها دقت درخت بر روی داده های Test و Train نمایش داده می‌شود

ابتدا یک خروجی با پیش فرض کد اجرا می‌کنیم. درخت دریافت شده به شدت بزرگ است و بدلیل تعداد بالای attribute ها و گستردگی داده ها این اتفاق رخ داده است درخت را به صورت کامل در این فایل نمی‌توان آورد ولی می‌توانید در فایل های موجود در پروژه با نام TreeDiabete1 آن را مشاهده نمایید. بخشی از آن به عنوان نمونه آورده شده است:



درخت پس از هرس نیز در فایل های موجود در پروژه با نام TreeDiabete2 وجود دارد و برای دیدن اینکه هرس به درستی اتفاق افتاده همان قسمتی که در صفحه قبل برای درخت قبل از هرس آورده شد را می‌آوریم:



توجه: نکته‌ای که وجود دارد این است که داده‌های train به صورت تصادفی انتخاب می‌شوند بنابراین نتایج یکسان با اجراهای متوالی نخواهیم داشت

ارزیابی درخت تصمیم ما به صورت زیر است:

```
Accuracy on Train Data: 0.9087947882736156
 Accuracy on Test Data: 0.7077922077922078
```

این دقت بر روی بازه های مساوی به تعداد ۵ و بر روی ۸۰ درصد داده ها به عنوان داده آموزشی Train بدست آمده است.

حال به سراغ تست بر روی چند بازه دیگر می‌رویم.

تعداد بازه ها ۳:

```
Accuracy on Train Data: 0.7833876221498371
 Accuracy on Test Data: 0.7727272727272727
```

تعداد بازه ها ۷:

```
Accuracy on Train Data: 0.9560260586319218
 Accuracy on Test Data: 0.6623376623376623
```

تعداد بازه ها ۹:

```
Accuracy on Train Data: 0.9853420195439739
 Accuracy on Test Data: 0.6818181818181818
```

تعداد بازه ها ۱۱:

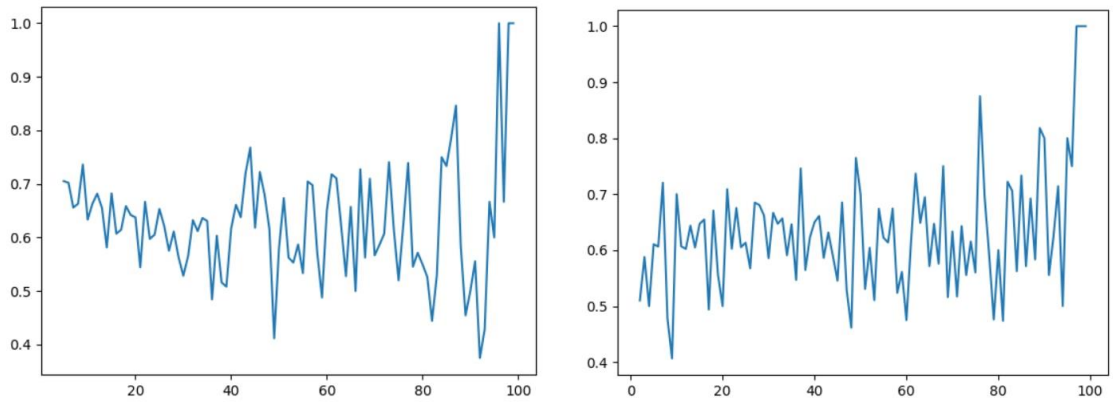
```
Accuracy on Train Data: 0.993485342019544
 Accuracy on Test Data: 0.6883116883116883
```

تعداد بازه ها ۱۳:

```
Accuracy on Train Data: 0.995114006514658
 Accuracy on Test Data: 0.6948051948051948
```

همانطور که دیده می‌شود با افزایش تعداد بازه‌ها دقت بر روی نمونه های آموزشی یا Train بیشتر شده و در واقع نوعی بیش برازش در حال رخ دادن است بنابراین بهتر است تعداد بازه ها را بیش از اندازه بزرگ نکنیم. علت دیگری که وجود دارد این است که داده‌ها به صورت یکنواخت پخش نشده است و بازه بندی‌های کوچک باعث می‌شود تعدادی از دسته‌ها داده های متناسبی در آن قرار نگیرد و دقت ما را کاهش دهد.

حال به سراغ تست تغییر درصد تعداد داده‌های train می‌رویم دو نمودار زیر بدست آمده است:



محور افقی نشان دهنده درصد داده‌های train است و محور عمودی نشان دهنده درصد دقت است. همانطور که دیده می‌شود اگر تعداد داده‌های train به بالای ۹۰ درصد کل داده ها برسد با تقریب بالایی همه test ها جواب درست را خواهند داد. از طرفی شکل به دلیل تصادفی بودن انتخاب داده‌های train و test به شکل های مختلف در آمده است.

در انتها برای بهبود درخت مناسب بود که شیوه دیگری را برای گسسته سازی داده‌ها انجام دهیم زیرا داده های داده شده تراکم یکسانی ندارند و در بازه‌های خاصی تعداد نمونه ها بیشتر است و بهتر است در آن بخش‌ها بازه‌های کوچکتر و در بخش‌هایی که تراکم نمونه کم است بازه‌ها بزرگتر باشند تا به این صورت عملیات یادگیری بهتر صورت گیرد.

یکی از چالش‌هایی که با آن مواجه شدیم افزایش دقت درخت بود در ابتدا کدی که برای این تمرین زده شده بود زیرشاخه‌های ویژگی‌ها به طور کامل آورده نشده بود و تنها زیرشاخه هایی که در داده train وجود داشت وارد درخت می‌شد این باعث می‌شود که اگر در داده test داده‌ای وجود داشته باشد که آن زیرشاخه را دارد نتواند جواب را چک کند. بنابراین به اصلاح آن پرداختیم.