۱- درخت تصمیم

برای این قسمت از فایل‌های زیر استفاده شده است.

- trainDT.m
- computeOptimalSplit.m
- batchClassifyWithDT.m
- classifyWithDT.m
- gatherTreeStats.m
- pruneSingleGreedyNode.m
- pruneAllNodes.m
- main1.m

درون تابع computeOptimalSplit.m را نوشتم و main1.m را برای اجرای موارد خواسته شده در این سوال، ایجاد کردم. توضیح همه‌ی برنامه‌ها در زیر آمده است.

فایل trainDT.m :

این فایل فقط شامل یک تابع به نام trainDT است.

- ورودی های آن عبارت اند از: x و y که پارامتر x ماتریس نمونه‌ها هست (هر سطر شامل یک نمونه است و ستون‌ها ویژگی های آن‌ها هستند) و y برچسب نمونه‌ها است.
- خروجی این تابع گره ریشه‌ی درخت یا زیردرخت تصمیمی است که از روی داده‌های ورودی ساخته شده است.
- درخت تصمیم به این صورت پیاده‌سازی شده است که برای دو نوع گره برگ و غیر برگ ساختاری به شکل زیر ایجاد می‌شود.
  - **گره برگ:**

۱. یک فیلد به نام isLeaf که برای این گره‌ها (گره‌های برگ) مقدارش برابر با true است.

۲. یک فیلد به نام label که برچسب آن گره برگ را نشان می‌دهد

۳. فیلدی به نام trainData که یک آرایه به طول دو است و خانه‌ی اول آن برچسب آن گره و خانه‌ی دوم آن تعداد نمونه‌های رسیده به آن گره را نشان می‌دهد.

- **گره غیر برگ:**

۱. مقدار فیلد isLeaf برای این گره‌ها false است.

۲. یک فیلد به نام attr دارند که شماره اندیس ویژگی مورد آزمایش در آن گره از درخت را نشان می‌دهد.

۳. فیلد thresh نشان‌دهنده‌ی مقدار آستانه‌ای است که برای آزمون روی ویژگی بالا استفاده می‌شود.

۴. فیلد child1 از جنس گره‌های درخت (برگ یا غیر برگ) است که شامل نمونه‌هایی است که در آن‌ها ویژگی بالا از مقدار آستانه‌ی گفته شده در بالا کمتر است.

x(attr) <= thresh

۵. فیلد child2 مانند قسمت قبل است با این تفاوت که شامل بقیه‌ی نمونه‌ها است.

x(attr) > thresh

۶. فیلد trainData نیز یک ماتریس دو سطری است که هر ستون آن شامل یک برچسب و تعداد نمونه‌های آن برچسب در آن گره از درخت است.

A	B	C	D	...
N_A	N_B	N_B	N_D	...

- مراحل انجام گرفته در این تابع به شرح زیر است:
  - در ابتدا سازگاری نمونه‌ها بررسی می‌شود. اگر دو نمونه وجود داشته باشند که در همه‌ی ویژگی‌ها مقدار یکسانی داشته باشند اما برچسب آن‌ها متفاوت باشد می‌گوییم نمونه‌ها ناسازگاراند. این مورد به این صورت بررسی می‌شود که به ازای هر نمونه‌ی s، آن نمونه را به اندازه‌ی همه‌ی نمونه‌ها تکرار می‌کنیم تا ماتریسی هم‌اندازه با x بدست آید.

```
repmat(x(s,:), [nSmps, 1])
```

سپس آن را با x مقایسه می‌کنیم و اندیس نمونه‌های کاملا مشابه با s را پیدا می‌کنیم.

```
matchRows = all(repmat(x(s,:), [nSmps, 1]) == x, 2);
```

حال بررسی می‌کنیم که تمام برچسب‌های این نمونه‌ها همه با هم برابر باشد (نمونه‌ی s نیز جزو همین نمونه‌ها ست).

```
matchedLabels = y(matchRows);
assert(all(matchedLabels == matchedLabels(1)), 'Training data is not consistent.');
```

- سپس بررسی می‌کنیم که آیا همه‌ی نمونه‌های رسیده به این مرحله (این گره) از یک برچسب هستند یا نه.

```
pureNode = all(y == y(1));
```

اگر همه دارای یک برچسب بودند این گره برگ است و به صورت زیر ساخته می‌شود.

```
if pureNode
   node.isLeaf = true;
   node.label = y(1);
   node.trainData = [y(1); length(y)];
```

- اگر همه‌ی نمونه‌ها از یک برچسب نبودند.

ابتدا با استفاده از computeOptimalSplit بهترین ویژگی برای این مرحله از ساخت درخت (این گره) و مقدار آستانه‌ی آن را محاسبه می‌کند.

```
[attr, thresh] = computeOptimalSplit(x, y);
```

سپس اندیس نمونه‌هایی که آن ویژگیشان کمتر از آستانه است را بدست می‌آورد.همین‌طور اندیس بقیه‌ی نمونه‌ها.

```
indsLessThanOrEqual = (x(:, attr) <= thresh);
indsAbove = (x(:, attr) > thresh);
```

می‌دانیم که این گره یک گره غیر برگ است.

```
node.isLeaf = false;
node.attr = attr;
node.thresh = thresh;
```

با صدا زدن دوباره‌ی همین تابع بر روی نمونه‌های جدا شده دو زیر درخت ایجاد می‌شود که ریشه‌های آن‌ها فرزندهای گره فعلی می‌شوند.

```
node.child1 = trainDT(x(indsLessThanOrEqual,:), y(indsLessThanOrEqual));
node.child2 = trainDT(x(indsAbove,:), y(indsAbove));
```

همین‌طور برای بدست آوردن فیلد trainData به صورت زیر عمل می‌کنیم. ابتدا با استفاده از تابع unique برچسب‌های موجود را بدست می‌آوریم و سپس با استفاده از تابع histc که هیستوگرام یک آرایه را برمی‌گرداند تعداد نمونه‌های هر برچسب را می‌شمریم.

```
uniqueLabels = unique(y);
node.trainData = [uniqueLabels'; histc(y, uniqueLabels)'];
```

فایل computeOptimalSplit.m :

این فایل شامل توابع زیر است:

- computeOptimalSplit
- computeGainAndThreshold
- Entropy

تابع اول تابعی است که در کد وجود داشت و می بایست کد آن را می نوشتیم. دو تابع دیگر را برای استفاده در تابع اصلی نوشتم.

- تابع computeOptimalSplit

ورودی این تابع x نمونه‌ها و y برچسب‌های آن‌هاست. خروجی آن بهترین ویژگی و مقدار آستانه‌ی آن است. (ویژگی و آستانه‌ای که بیشترین بهره‌ی اطلاعات را ایجاد می‌کند)
به ازای همه‌ی ویژگی ها ستون آن ویژگی (مقدار همه‌ی نمونه‌ها در آن ویژگی) و y را به تابع computeGainAndThreshold می دهیم. این تابع بهترین آستانه (ایجاد بیشترین بهره‌ی اطلاعات) برای آن ویژگی را بر می‌گرداند.
حال کافی است در یک حلقه برای همه‌ی ویژگی‌ها این تابع را صدا بزنیم و ویژگی ای که بیشترین بهره‌ی اطلاعاتی را دارد انتخاب کنیم.

- تابع computeGainAndThreshold

ورودی این تابع x یک بردار ستونی است که شامل مقدار همه‌ی نمونه‌ها برای یک ویژگی است و y برچسب نمونه‌ها. خروجی آن بهترین آستانه برای این ویژگی است. روند کار به این صورت است که ابتدا بررسی می‌کنیم که اگر همه‌ی مقادیر x یکسان است آن‌گاه از آستانه گذاری روی این ویژگی هیچ بهره‌ی اطلاعاتی نخواهیم داشت پس:

```
if all(x == x(1))
    gain = 0;
    th = x(1);
```

درغیر این‌صورت x و y را کنار هم در ستون‌های یک ماتریس دو ستونی قرار می‌دهیم. سپس آن را مرتب می‌کنیم.

```
z = [x,y];
z = sortrows(z);    %sort z's rows by first column (increamental)
```

حال کافی است نقاطی که در آن‌ها تغییر برچب مشاهده می‌شود را بررسی کنیم. برای این کار برچسب ها را از آن نقاط جدا کرده و به تابع Entropy می‌دهیم تا I را حساب کنیم.

```
I = (Entropy(y(1:i))*i/n) + (Entropy(y(i+1:n))*(n-i)/n);
```

کمترین I معادل با بیشترین بهره‌ی اطلاعات است. پس کمترین آن‌ها را پیدا می‌کنیم. و بهره‌ی اطلاعات را به این صورت محاسبه می‌کنیم. همچنین مقدار آستانه از میانگین مقدار ویژگی در دو طرف محل جداسازی بدست می‌آید.

```
gain = e - min;
th = (z(imin,1) + z(imin+1,1))/2;
```

- تابع Entropy

ورودی این تابع یک بردار از برچسب‌ها (در این سوال فقط 0 و 1) است و خروجی آن انتروپی است. روش کار آن به این صورت است که ابتدا بررسی می‌کنیم که اگر همه‌ی برچسب‌ها یکی بودند یعنی انتروپی صفر است.

```
if all(x == x(1))
    e = 0;
```

در غیر این صورت تعداد صفرهای بردار را میشماریم که در واقع تعداد اعضای یک برچسب است. حال با استفاده از فرمول انتروپی آن را محاسبه می‌کنیم.

$$EntropyS = -p\oplus \log 2p\oplus - p\ominus \log 2p\ominus$$

```
n = size(x,1);
nZeros = sum(x == 0);
p1 = nZeros/n;
p2 = (n - nZeros)/n;
t1 = p1*log2(p1);
t2 = p2*log2(p2);
e = -(t1 + t2);
```

فایل classifyWithDT.m :

این فایل شامل تابع classifyWithDT است. ورودی این تابع x یک نمونه و dT یک درخت تصمیم یا زیر درخت (ریشه درخت یا زیر درخت) است. که این تابع برچسب این نمونه را بدست می‌آورد و برمی‌گرداند. روش کار به این صورت است که ابتدا بررسی می‌شود که اگر این گره از درخت برگ است پس برچسب نمونه می‌شود برچسب همین گره برگ.

```
if dT.isLeaf
   y = repmat(dT.label, [nSmps, 1]);
```

در غیر این صورت با انجام آزمون روی نمونه با توجه به ویژگی و آستانه‌ی این گره، مشخص می‌شود که این نمونه باید به کدام زیر درخت برچسب گذاری شود. سپس همین تابع برای نمونه و زیر درخت مورد نظر صدا زده می‌شود.

```
if x(dT.attr) <= dT.thresh
      y = classifyWithDT(x, dT.child1);
   else
      y = classifyWithDT(x, dT.child2);
   end
```

فایل batchClassifyWithDT.m :

این فایل شامل تابع batchClassifyWithDT است. ورودی این تابع x کل نمونه‌ها و dT یک درخت تصمیم یا زیر درخت (ریشه درخت یا زیر درخت) است. که این تابع برچسب این نمونه‌ها را بدست می‌آورد و برمی‌گرداند. روند کار به این صورت است که به ازای هر نمونه تابع classifyWithDT را فراخوانی می‌کند.

فایل gatherTreeStats.m :

این فایل شامل تابع gatherTreeStats است. ورودی این تابع dT یک درخت تصمیم یا زیر درخت (ریشه درخت یا زیر درخت) است. خروجی آن treeStats یک ساختار است که تعداد گره‌های برگ و غیر برگ درخت را برمی‌گرداند. (کد آن واضح است)

فایل pruneAllNodes.m :

این فایل شامل تابع pruneAllNodes است. ورودی این تابع dT یک درخت تصمیم یا زیر درخت (ریشه درخت یا زیر درخت) است. خروجی آن یک آرایه از درخت‌هایی است که از هرس هر کدام از گره‌های غیر برگ این درخت بدست می‌آیند.

فایل pruneSingleGreedyNode.m :

این فایل شامل توابع pruneSingleGreedyNode و getAccuracy است.

- تابع pruneSingleGreedyNode

ورودی این تابع x نمونه‌ها، y برچسب نمونه‌ها و dT یک درخت تصمیم یا زیر درخت (ریشه درخت یا زیر درخت) است. خروجی آن یک درخت است که بهترین درخت از بین درخت‌هایی است که از هرس یک گره درخت dT بدست آمده‌اند. روند کار به این صورت است که با فراخوانی تابع pruneAllNodes

لیست تمام درخت‌هایی که از هرس یک گره از درخت اصلی ایجاد می‌شوند را بدست می‌آوریم. سپس با تابع getAccuracy مقدار دقت آن‌ها را بدست می‌آوریم و درختی که بیشترین دقت را دارد به عنوان خروجی مشخص می‌شود.

- تابع getAccuracy

ورودی این تابع x نمونه‌ها، y برچسب نمونه‌ها و dT یک درخت تصمیم یا زیر درخت (ریشه درخت یا زیر درخت) است. خروجی آن دقت درخت تصمیم dT برای داده‌های ورودی است. و روش کار به این‌صورت است که با استفاده از تابع batchClassifyWithDT برچسب پیشنهادی درخت برای نمونه‌ها را بدست می‌آوریم. سپس بردار برچسب‌های اصلی و برچسب‌های تخمین زده شده را از هم کم می‌کنیم. حال اگر تعداد صفرهای بردار حاصل را بشماریم تعداد تخمین‌های درست را شمرده ایم. پس دقت میشود تعداد صفرهای بردار حاصل تقسیم بر تعداد کل.

فایل main1.m :

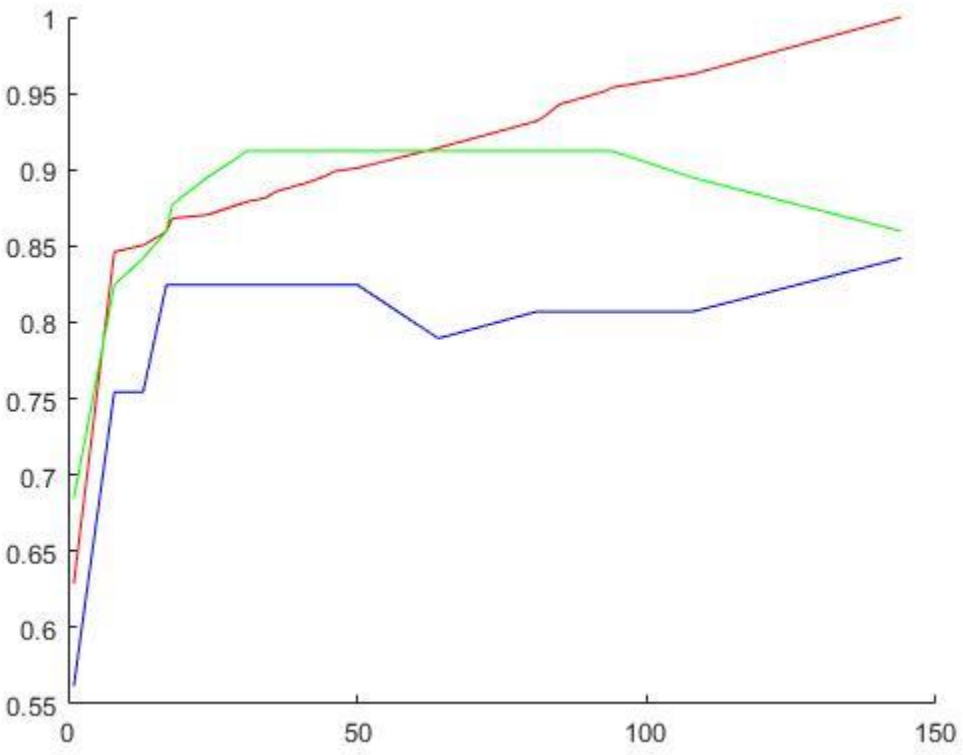
این فایل شامل توابع main1 و getAccuracy است.

- تابع main1

این تابع موارد خواسته شده در این سوال را به کمک توابع بالا اجرا می‌کند.
ابتدا داده‌های train را میخواند و با استفاده از تابع trainDT درخت تصمیم را می‌سازد. سپس با استفاده از gatherTreeStats تعداد گره‌های برگ و غیر برگ درخت را نمایش می‌دهد.
بعد از آن داده‌های test را می‌خواند و با استفاده از getAccuracy میزان دقت درخت را برای داده‌های train و test محاسبه می‌کند.
حال برای هرس درخت ابتدا داده‌های validation را می‌خواند. سپس در یک حلقه درخت را مدام هرس می‌کند تا به درختی برسد که یک گره دارد و از حلقه خارج شود.
برای رسم نمودار دقت برای هر درخت و به ازای داده‌های train، test و validation یک بردار سه سطری به نام accuracies تعریف کرده‌ایم. که در هر تکرار حلقه، با توجه به درخت هرس شده در آن مرحله دقت را برای هر سه مجموعه داده محاسبه می‌کنیم و در سطر مربوط به خودشان و در ستون مربوط به آن درخت قرار می‌دهیم.
همین‌طور بردار xAxis می‌کنیم که قرار است تعداد گره‌های برگ همه‌ی درخت‌های هرس شده در هر تکرار حلقه را ذخیره کند با استفاده از gatherTreeStats مقداردهی می‌شود.

تحلیل :

- درخت تصمیم ایجاد شده ۲۸۷ گره غیر برگ و ۱۴۴ گره برگ دارد.
- دقت این درخت بر روی داده‌های train برابر با ۱ است و برای داده‌های test برابر با ۰/۸۴۲۱۰۵ است.
- نمودار زیر دقت درخت تصمیم هرس شده را برای سه مجموعه داده‌ی train، نمودار قرمز رنگ، test، نمودار آبی رنگ و validation، نمودار سبز رنگ، نمایش می‌دهد.



- با توجه به شکل، مشخص است که وقتی اجازه می‌دهیم درخت بیش از اندازه بزرگ شود، درخت حاصل بیش از اندازه به داده‌های train وابسته می‌شود و به عبارتی پدیده‌ی بیش برازش رخ می‌دهد و درخت بسیار به داده‌های train منطبق می‌شود و این باعث می‌شود درخت حاصل در حالت کلی دقت کافی را نداشته باشد. در حالی که درختی با اندازه‌ی کوچکتر وجود دارد که در حالت عمومی، دقت بالاتری از درخت اصلی دارد.