

تمرین های **سری سوم** درس یادگیری ماشین

فرهاد دلیرانی
۹۶۱۳۱۱۲۵

از آنجایی که در شرح تمرین‌ها برای پایتون ورژن خاصی ذکر نشده است تمام کدها را با **پایتون سه‌وشش** نوشته ام و همین‌طور از `numpy`, `scipy.io` و `matplotlib` برای رسم نمودار استفاده کرده‌ام. برای سادگی در نصب آن پکیج‌ها از `anaconda3` استفاده کرده‌ام.

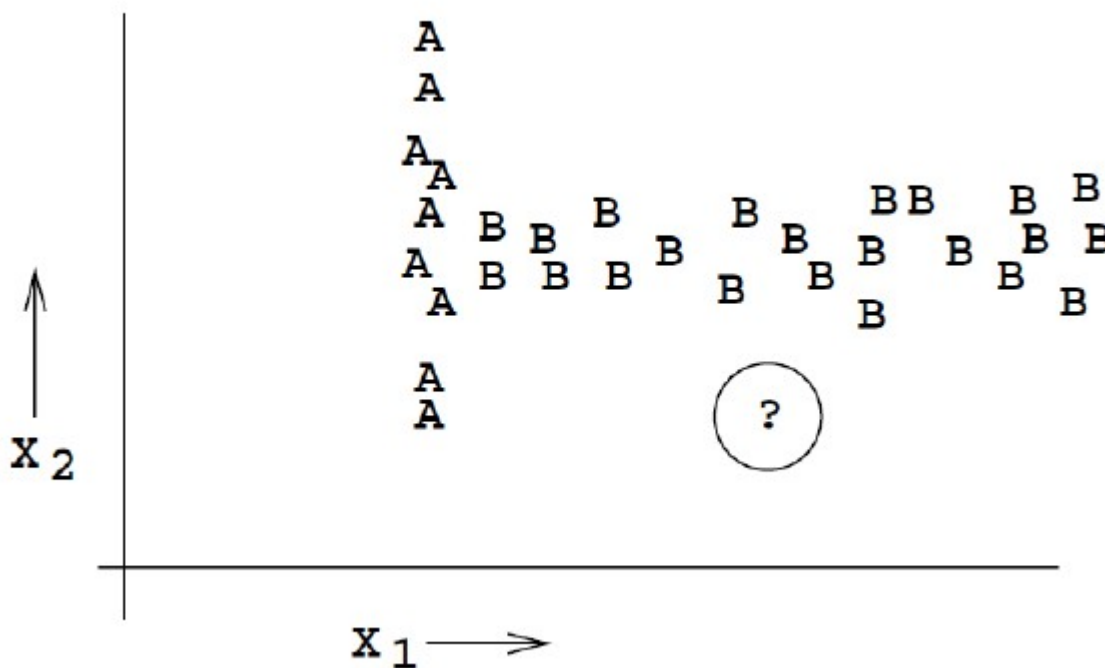
البته سوال آخر (شبکه‌ی بیز) را با پایتون 2 نوشته‌ام و از `numpy` `pgmpy` و `pandas` در آن استفاده کرده‌ام.

سوال ۱-

در Naive Bayes مرز تصمیم سه صورت می تواند باشد:

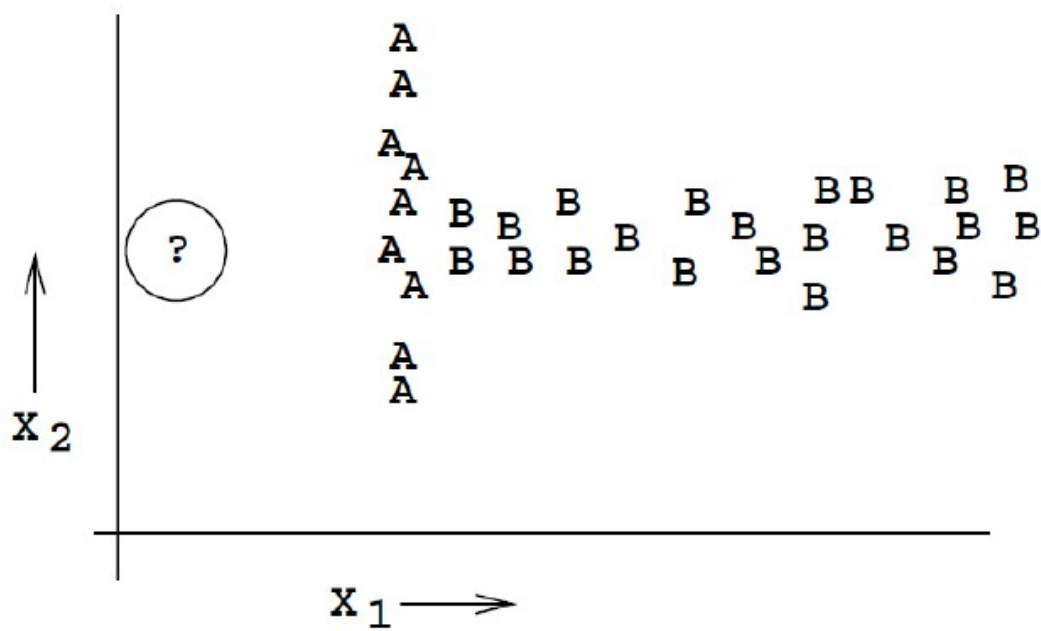
- اگر واریانس دو کلاس یکسان باشد، مرز جدا کننده یک خط (صفحه) است.
- اگر میانگین دو کلاس یکسان باشد، مرز جدا کننده یک دایره یا بیضی است.
- در حالت کلی مرز جدا کننده یک parabolic curve است.

(الف)



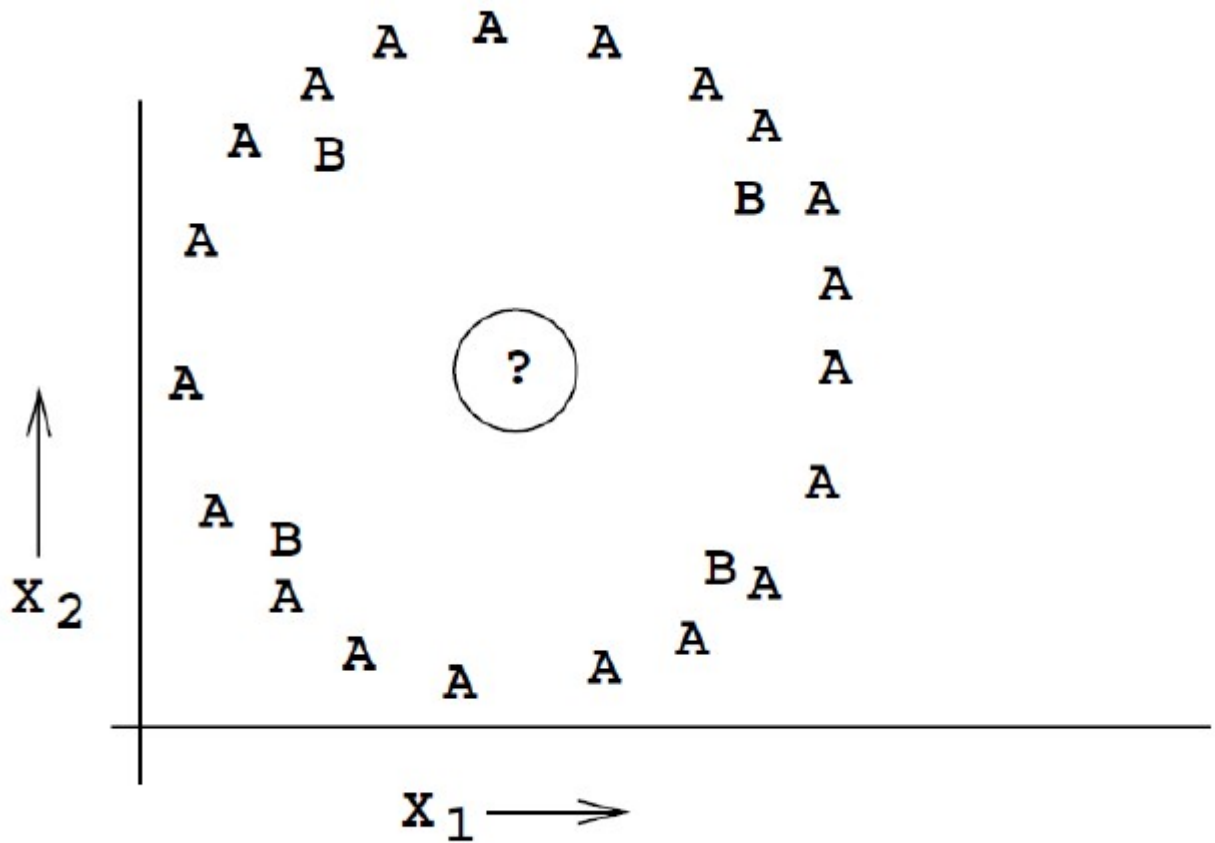
در شکل بالا اگر واریانس ها برابر باشد خطی نمونه های A و B را از هم جدا می کند. اگر هم واریانس ها برابر نباشند از آنجایی که میانگین ها هم یکسان نیست مرز تصمیم یک parabolic curve می شود. که در هر دو صورت نمونه ی عضو از کلاس B است.

(ب)



در شکل بالا اگر واریانس ها برابر باشد خطی نمونه های A و B را از هم جدا می کند که سمت چپ آن کلاس A می شود و سمت راست آن کلاس B می شود. اگر هم واریانس ها برابر نباشند از آنجایی که میانگین ها هم یکسان نیست مرز تصمیم یک parabolic curve می شود. که در هر دو صورت نمونه ای؟ عضوی از کلاس A است.

(ج)



در تصویر بالا به نظر می رسد میانگین داده ها یکسان است و همان طور که در ابتدای این سوال توضیح دادم هنگامی که میانگین ها یکسان باشند مرز تصمیم یک دایره یا بیضی است در نتیجه مرز تصمیم در شکل بالا یک دایره است که نمونه های B را از A جدا می کند و درون دایره کلاس B است و خارج آن کلاس A است. به همین سبب نمونه ی ؟ از کلاس B است.

سوال ۲)

الف) در صورتی که هیچ استقلال بین فیچرها نباشد برای هر فیچر 2^n پارامتر نیاز داریم و m کلاس داریم در نتیجه به $2^n * m$ پارامتر نیاز هست به علاوه ی prior های m کلاس که در کل $2^n * m + m$ پارامتر نیاز هست. اگر بخوایم آن را ساده تر نشان بدهیم می توانیم از $O(m * 2^n)$ استفاده کنیم.

ب) در صورتی که ویژگی ها با دادن کلاس مستقل از هم باشند می توان از نایبوز استفاده کرد برای این منظور برای هر ویژگی باید $p(\text{feature } i=1 | m=\text{true})$ و $p(\text{feature } i=1 | m=\text{false})$ و $p(\text{feature } i=0 | m=\text{true})$ و $p(\text{feature } i=0 | m=\text{false})$ را به دست آورد. در نتیجه برای هر فیچر به ازای تمامی کلاس ها باید $m * 4$ پارامتر به دست آورد و از آنجایی که n فیچر داریم باید $m * n * 4$ فیچر به دست بیاوریم و همین طور prior های m کلاس که در نهایت $m * n * 4 + m$ پارامتر باید تعیین شود. اگر بخوایم آن را ساده تر نشان بدهیم می توانیم از $O(m * n)$ استفاده کنیم.

ج) در صورتی که ویژگی ها با دادن کلاس مستقل از هم باشند و باینری هم نباشند می توان از Gaussian Naive Bayes استفاده کرد، که برای اینکار لازم است به ازای هر فیچر واریانس و میانگین رو به دست بیاوریم. اگر n فیچر داشته باشیم و m کلاس باید $m * n$ واریانس به دست بیاوریم و $m * n$ تا میانگین در نتیجه به $m * n * 2$ پارامتر نیاز داریم که باید تعیین شوند و همین طور prior های m کلاس که در نهایت $m * n * 2 + m$ پارامتر باید تعیین شود. اگر بخوایم آن را ساده تر نشان بدهیم می توانیم از $O(m * n)$ استفاده کنیم.

$$P(B|D=T) = \frac{P(B, D=T)}{P(D=T)}$$

$$P(D=T) = \sum_A \sum_B \sum_C P(A, B, C, D=T)$$

$$= \sum_A \sum_B \sum_C P(A) P(B|A) P(C|A) P(D=T|B, C)$$

$$= \sum_A \sum_B P(A) P(B|A) \sum_C P(D=T|B, C) P(C|A)$$

$$= \sum_A \sum_B P(A) P(B|A) (P(D=T|B, C=T) P(C=T|A) + P(D=T|B, C=F) P(C=F|A))$$

$$= \sum_A P(A) [P(B=T|A) (P(D=T|B=T, C=T) P(C=T|A) + P(D=T|B=T, C=F) P(C=F|A)) + P(B=F|A) (P(D=T|B=F, C=T) P(C=T|A) + P(D=T|B=F, C=F) P(C=F|A))]$$

$$= 0.8 [0.21(0.5 \times 0.25 + 0.15 \times 0.75) + 0.79(0.67 \times 0.25 + 0.95 \times 0.75)] + 0.2 [0.37(0.5 \times 0.3 + 0.15 \times 0.7) + 0.63(0.67 \times 0.3 + 0.95 \times 0.7)]$$

$$= 0.59606 + 0.127986 = 0.724046$$

$$P(B, D=T) = P(B=T, D=T) = \sum_A \sum_C P(A, B=T, C, D=T)$$

$$= \sum_A \sum_C P(A) P(B=T|A) P(C|A) P(D=T|B=T, C)$$

$$= \sum_A P(A) P(B=T|A) \sum_C P(C|A) P(D=T|B=T, C)$$

$$= \sum_A P(A) P(B=T|A) (P(C=T|A) P(D=T|B=T, C=T) + P(C=F|A) P(D=T|B=T, C=F))$$

$$= (0.2)(0.37)(0.3 \times 0.5 + 0.7 \times 0.15) + (0.8)(0.21)(0.25 \times 0.5 + 0.75 \times 0.15) = 0.05877$$

$$\Rightarrow P(B=T|D=T) = \frac{P(B=T, D=T)}{P(D=T)} = \frac{0.05877}{0.724046} = 0.081168$$

$$P(B=F|D=T) = 1 - 0.081168 = 0.91883124$$

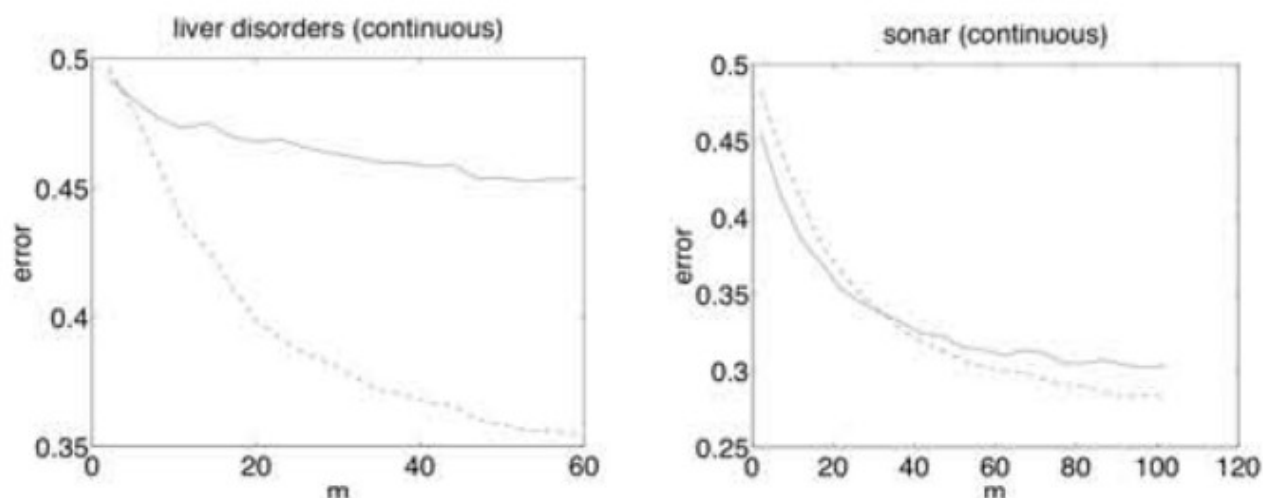
سوال (4)

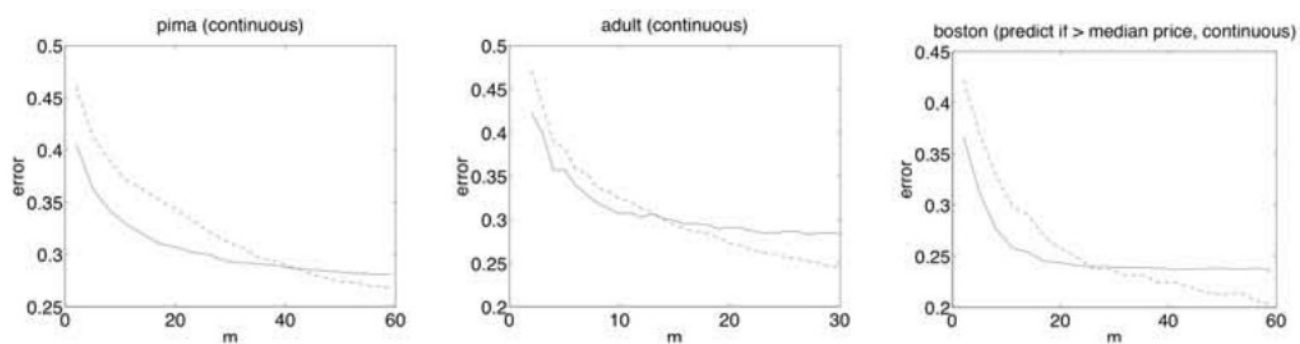
فرض کنید که می‌خواهیم داده‌ی X را دسته‌بندی کنیم و کلاس آن y است. مدل‌های generative توزیع توأم x و y را یاد می‌گیرد که آن را با $p(x,y)$ نشان می‌دهند ولی برای محاسبه‌ی کلاس یک ورودی آن را با استفاده از قانون بیز به صورت $p(y|x)$ در می‌آورند. ولی مدل‌های discriminative به‌طور مستقیم احتمال شرطی x به شرط اینکه y داده شده باشد را که به این صورت نمایش داده می‌شود $p(y|x)$ را یاد می‌گیرند.

سوال (5)

بیز ساده یک دسته‌بند generative است زیرا توزیع توأم x و y را فرا می‌گیرد و سپس با قانون بیز $p(y|x)$ را به دست می‌آورد. ولی Logistic Regression یک دسته‌بند discriminative است و مستقیماً $p(y|x)$ را به دست می‌آورد. بر اساس مقاله‌ی On Discriminative vs. Generative classifiers: A comparison of logistic regression and naive Bayes دسته‌بندهای generative خطای مجانبی بیشتری دارند ولی زودتر به این خطا می‌رسند در حالی که دسته‌بندهای discriminate خطای مجانبی کمتر و بهتری نسبت به generative ها دارند ولی دیرتر از آن‌ها به حد خطای خود می‌رسند.

در شکل‌های زیر چند نمونه از عملکرد Logistic Regression (خط چین) و Naive Bayes (خط صاف) را برای دیتاست‌های مختلف مشاهده می‌کنید که محور عمودی خطای کلی است و محور افقط تعداد المنت‌های دیتاست.





همین طور که در شکل‌ها پیداست naive bayes با تعداد کمتر sample به حد خطای خود رسیده است ولی logistic regression به داده‌های بیشتری نیاز داشته است. همین حد خطای logistic regression بهتر بوده است.

در نتیجه برای داده‌های کم بهتر از Naive Bayes استفاده کنیم و برای تعداد داده‌های زیاد از Logistic Regression استفاده شود.

پیاده سازی Logistic Regression:

برای راحتی بیشتر label ها را از ۱ و -۱ به ۱ و ۰ تبدیل کرده ام.

الف)

این قسمت از سوال خواسته است Logistic Regression را بدون regularization پیاده سازی کنیم. کدهای این قسمت از سوال در پوشه‌ی machineLearning3 و در فایل logistic_regression.py است. برای این منظور تابع‌های زیر را پیاده سازی کرده‌ام:

sigmoid(x)

این تابع برابر است با $1 / (e^x + 1)$ که در logistic regression برای محاسبه‌ی Cost function و گرادیان‌ها استفاده می‌شود.

cost_function(thetaVec, xMat, y)

این تابع هزینه‌ی logistic regression است که به صورت زیر محاسبه می‌شود:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m [-y^{(i)} \log(h_{\theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]$$

که به آن cross-entropy هم می‌گویند. در تصویر بالا $h(x^i)$ برابر است با `sigmoid(np.dot(x, thetaVec))`.

accuracy_of_test(thetaVec, xMat, y)

این تابع نتایج را بعد از آموزش Logistic Regression می‌گیرد و مجموعه‌ی تست و کلاس‌های متناظر با هر المنت آن را می‌گیرد بعد `sigmoid(np.dot(x, thetaVec))` را محاسبه می‌کند که یک عدد بین ۰ و ۱ است و با استفاده از cut-off تعیین می‌کند باید در کلاس یک دسته بندی شود یا کلاس صفر. سپس با توجه به کلاسی که به دست آمده و کلاس واقعی نمونه دقت را محاسبه می‌کند

gradients(thetaVec, xMat, y)

این تابع گرادیان cost function است که از طریق زیر محاسبه می‌شود:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \right)$$

در تصویر بالا $h(x^i)$ برابر است با `sigmoid(np.dot(x, thetaVec))`

gradient_descent(xMat, y, numberOfIter, learningRate)

این تابع با استفاده از تابع گرادیان بالا، و با توجه به ضریب یادگیری و حداکثر تعداد iteration ها گرادیان‌ها را محاسبه و آپدیت می‌کند.

logistic_gradient_descent(xTrain, yTrain, numberOfIter, learningRate, xTest, yTest)

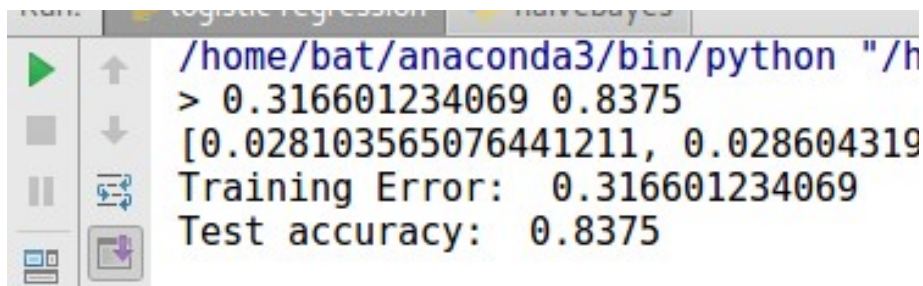
در این تابع ابتدا فیچر ها به صورت زیر scale می شوند:

$$(feature-mean)/(max-min)$$

سپس تابع **gradient_descent** که در بالا آن را معرفی کردم فراخوانده می شود و پارامترها به دست می آید و همین طور علاوه بر پارامترها میزان خطای مجموعه ی آموزش را هم محاسبه می کند. و بعد تابع **accuracy_of_test** فراخوانده می شود که دقت مدل آموزش داده شده را بر روی مجموعه تست مشخص می کند و در آخر پارامترها - scales factors - خطای مجموعه ی آموزش و دقت مجموعه ی تست بازگردانده می شود.

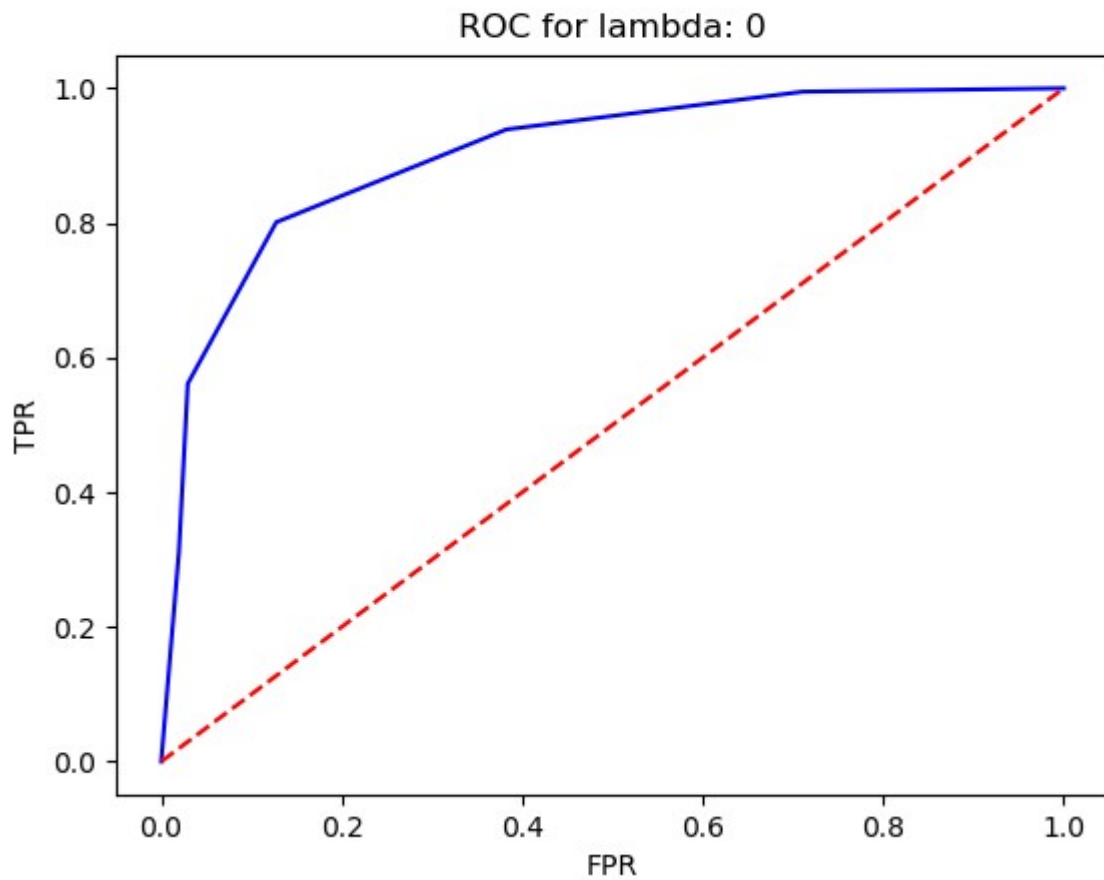
(runCode)

این تابع اصلی (Main) است که در این تابع دیتاست خوانده می شود و x_0 که برابر با یک است به داده ها افزوده می شود. کلاس ها از ۰-۱ و ۱ به ۰ تبدیل می شوند و تابع **logistic_gradient_descent** برای به دست آمدن پارامترها و میزان خطا و دقت مجموعه ی تست و آموزش فراخوانده می شود و در نهایت میزان دقت و خطا چاپ می شود که آن را در زیر مشاهده می کنید (ROC این بخش را در بخش ب این سوال رسم کرده ام):



```
/home/bat/anaconda3/bin/python "/h
> 0.316601234069 0.8375
[0.028103565076441211, 0.028604319
Training Error: 0.316601234069
Test accuracy: 0.8375
```

ROC این بخش در بخش ب محاسبه و رسم شده است:



بخش ب)

این بخش مانند بخش قبل است با این تفاوت که باید regularization انجام بدهیم و از ten fold cross validation استفاده کنیم. کدهای این بخش در درپوشه‌ی machineLearning3 و فایل **logistic regression regularization.py** قرار دارند.

برای این بخش تابع های زیر را نوشته:

sigmoid(x)

این تابع برابر است با $1 / (e^x + 1)$ که در logistic regression برای محاسبه‌ی Cost function و گرادیان‌ها استفاده می‌شود.

cost_function(thetaVec, xMat, y, _lambda)

این تابع هزینه‌ی logistic regression است که به صورت زیر محاسبه می‌شود:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m [-y^{(i)} \log(h_{\theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

که به آن cross-entropy هم می‌گویند. در تصویر بالا $h(x^{(i)})$ برابر است با `sigmoid(np.dot(x, thetaVec))`. تفاوت این تابع با تابع بخش قبل این است که regularization هم دارد.

accuracy_of_test(thetaVec, xMat, y, threshold)

این تابع نتایج را بعد از آموزش Logistic Regression می‌گیرد و مجموعه‌ی تست و کلاس‌های متناظر با هر المنت آن را می‌گیرد بعد `sigmoid(np.dot(x, thetaVec))` را محاسبه می‌کند که یک عدد بین 0 و 1 است و با استفاده از cut-off که برابر است با threshold تعیین می‌کند باید در کلاس یک دسته بندی شود یا کلاس صفر، سپس با توجه به کلاسی که به دست آمده و کلاس واقعی نمونه دقت را محاسبه می‌کند. Threshold به عنوان یک پارامتر به تابع داده شده است تا دقت برای cut-off های مختلف به دست بیاوریم و از آن برای رسم ROC استفاده کنیم.

tpr_fpr_of_test(thetaVec, xMat, y, threshold)

این تابع True positive Rate و False Positive Rate را با توجه به پارامترهای آموزش داده شده برای مجموعه داده‌های داده شده به تابع با توجه به threshold به دست می‌آورد.

gradients(thetaVec, xMat, y, _lambda)

این تابع گرادیان cost function است که از طریق زیر محاسبه می‌شود:

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad \text{for } j = 0$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \quad \text{for } j \geq 1$$

در تصویر بالا $h(x^{(i)})$ برابر است با `sigmoid(np.dot(x, thetaVec))`. تفاوت با تابع قسمت قبل این است که regularization لحاظ شده است.

gradient_descent(xMat, y, numberOfIter, learningRate, _lambda)

این تابع با استفاده از تابع گرادیان بالا، و با توجه به ضریب یادگیری و حداکثر تعداد iteration ها گرادیان‌ها را محاسبه و آپدیت می‌کند.

logistic_gradient_descent(xTrain, yTrain, numberOfIter, learningRate,

xTest, yTest, _lambda, threshold)

در این تابع ابتدا فیچر ها به صورت زیر scale می شوند:

$$(feature-mean)/(max-min)$$

سپس تابع **gradient_descent** که در بالا آن را معرفی کردم فراخوانده می شود و پارامترها به دست می آید و همین طور علاوه بر پارامترها میزان خطای مجموعه ی آموزش را هم محاسبه می کند. و بعد تابع **accuracy_of_test** فراخوانده می شود که دقت مدل آموزش داده شده را بر روی مجموعه تست مشخص می کند و بعد **tpr_fpr_of_test** فراخوانده می شود که میزان TPR و FPR برای مجموعه ی تست محاسبه می شود. و در آخر پارامترها – scales factors – خطای مجموعه ی آموزش و دقت مجموعه ی تست بازگردانده می شود.

ten_fold_cross_validation(learningFunction, _lambda, xtrain, ytrain, threshold, iteration, learningRate)

در این تابع **ten fold cross validation** را پیاده سازی شده است. داده ها ورودی به 10 قسمت تقسیم می شوند. و هر با یکی از آن قسمت ها تست می شود و باقی قسمت ها آموزش. سپس در به ازای ده بار مختلف **learningFunction** که همان **logistic_gradient_descent** فراخوانده می شود و میزان خطا و دقت برای قست آموزش و تست مشخص می شود. و در آخر میانگین ده قسمت برگردانده می شود.

(runCode)

این تابع اصلی (Main) است که در این تابع دیتاست خوانده می شود و x_0 که برابر با یک است به داده ها افزوده می شود. کلاس ها از ۱- و ۱ به ۰ و تبدیل می شوند سپس به ازای **lambda** های داده شده در سوال تابع **ten_fold_cross_validation** تا به ترین **lambda** انتخاب شود. بعد از اینکه به ترین **lambda** انتخاب شد تابع **logistic_gradient_descent** با **cut-off threshold** ها مختلف فراخوانده می شود تا ROC مدل را برای بهترین لامبدا رسم کنیم. همین طور ROC بخش قبل (بدون regularization) در این بخش محاسبه و رسم می شود. در ادامه خروجی ها را مشاهده می کنید. (در خروجی به جای **accuracy** نوشته ام **error accuracy** و از آنجایی که اجرای برنامه زمان بر است آن را اصلاح نکردم).

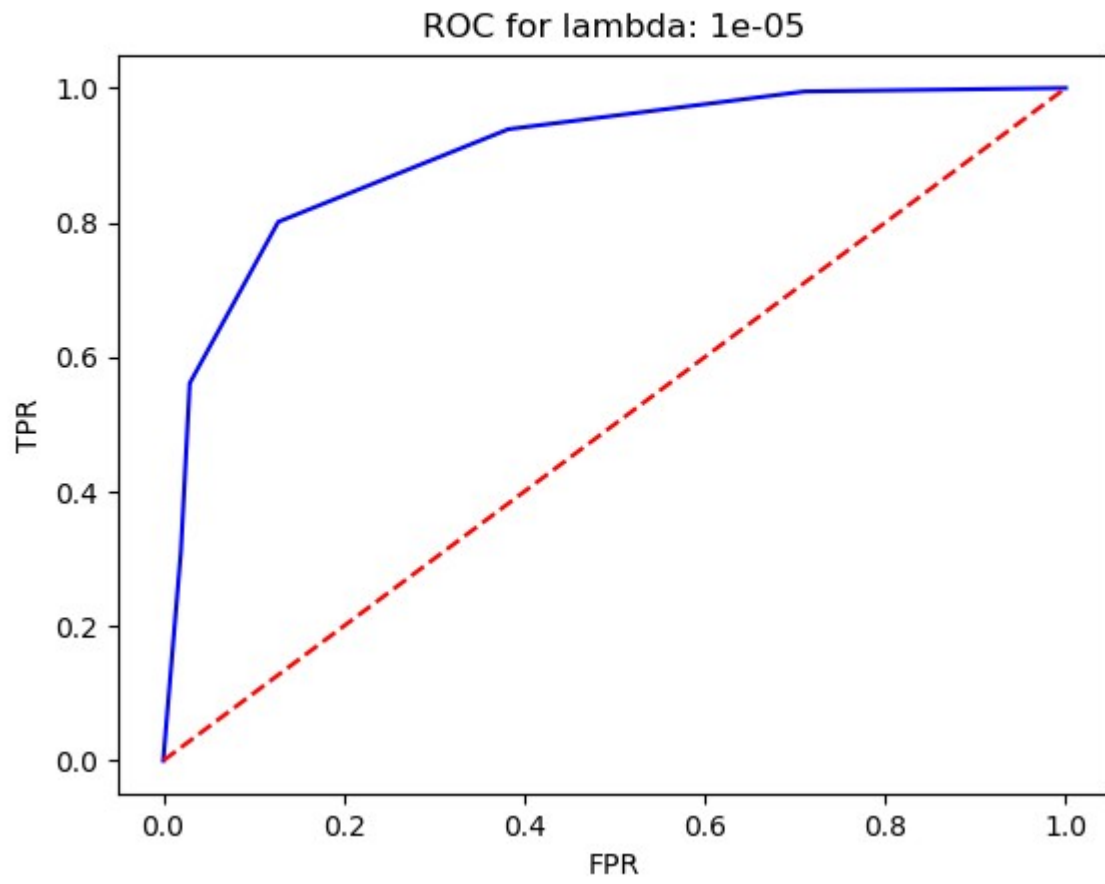
```

/home/bat/anaconda3/bin/python "/home/bat/Dropbox/codes/pythonCode/ma
=====Lambda: 1e-05 =====
10-fold-CV Training Error For Test:  0.18624999999999997
10-fold-CV Training Error Accuracy For Test:  0.81375
=====
=====Lambda: 0.0001 =====
10-fold-CV Training Error For Test:  0.18624999999999997
10-fold-CV Training Error Accuracy For Test:  0.81375
=====
=====Lambda: 0.001 =====
10-fold-CV Training Error For Test:  0.18687499999999999
10-fold-CV Training Error Accuracy For Test:  0.813125
=====
=====Lambda: 0.01 =====
10-fold-CV Training Error For Test:  0.19062500000000004
10-fold-CV Training Error Accuracy For Test:  0.809375
=====
=====Lambda: 0.1 =====
10-fold-CV Training Error For Test:  0.21875
10-fold-CV Training Error Accuracy For Test:  0.78125
=====
=====Lambda: 1 =====
10-fold-CV Training Error For Test:  0.29750000000000004
10-fold-CV Training Error Accuracy For Test:  0.7024999999999999
=====

=====Lambda: 10 =====
10-fold-CV Training Error For Test:  0.7162499999999999
10-fold-CV Training Error Accuracy For Test:  0.28375000000000006
=====
Best Lambda is: 1e-05, Its accuracy is:0.818125

```

در ابتدا داده‌ها را scale کرده‌ام به همین دلیل بهترین لاندا برای داده‌های اسکیل شده با زمانی که اسکیل انجام نشده است متفاوت است. در تصویر بالا خطا و دقت مجموعه‌ی تست آورده شده است و همین طور که در خروجی مشاهده می‌کنید بهترین ضریب 0.00001 است. در شکل زیر ROC معادل آن (بهترین لامبدا) را در مشاهده می‌کنید:



پیاده سازی Naive Bayes:

در این سوال باید برای دیتاست سوال قبل باید یک naive bayes classifier ایجاد کنیم. کدهای این سوال در درپوشه‌ی machineLearning3 و فایل naivebayes.py موجود است.
برای پیاده سازی این سوال تابع‌های زیر را نوشته‌ام:

find_priors(train)

این تابع داده‌های آموزش را می‌گیرد و priorهای کلاس ۱- و ۱+ را برمی‌گرداند. که این کار را با شمارش تعداد اعضای هر کلاس انجام می‌دهد.

find_distributions(train)

این تابع مجموعه‌ی آموزش را می‌گیرد و برای هر فیچر توزیع نرمال آن را محاسبه می‌کند که برای این منظور میانگین و واریانس هر فیچر را پیدا می‌کند.

univariate_normal(mean, variance, x)

این تابع میانگین و واریانس یک توزیع نرمال را می‌گیرد و مقدار نقطه‌ی x در توزیع نرمال را محاسبه می‌کند.

predictClassOfInstance(distributionsOfFeatures, instance, threshold, prior1, priorMines1)

این تابع توزیع نرمال فیچرها، یک نمونه که می‌خواهیم کلاس آن را تشخیص دهیم و prior کلاس‌ها را می‌گیرد. Threshold برای رسم roc است که مشخص می‌کند $p(w1|x)$ چه مقدار باید از $p(w2|x)$ بزرگ تر باشد تا کلاس نمونه عضو کلاس یک باشد. در حالت عادی اگر $p(w1|x)$ بزرگ تر از $p(w2|x)$ باشد نمونه عضو کلاس یک است.

این تابع مقادیر زیر را برای هر دو کلاس محاسبه می‌کند.

$$P(w1|x) = p(w1) * p(feature1_x|w1) * p(feature2_x|w1) * ... * p(featureN_x|w1)$$

$$p(w2|x) = p(w2) * p(feature1_x|w2) * p(feature2_x|w2) * ... * p(featureN_x|w2)$$

البته از آنجایی که تعداد ضرب‌ها زیاد است و عددها کوچک و اعشاری اند از هر دو Ln گرفته‌ام. $p(featurei_x|wj)$ برابر است با مقدار feature I نمونه‌ی x در توزیع نرمال فیچر I ام.

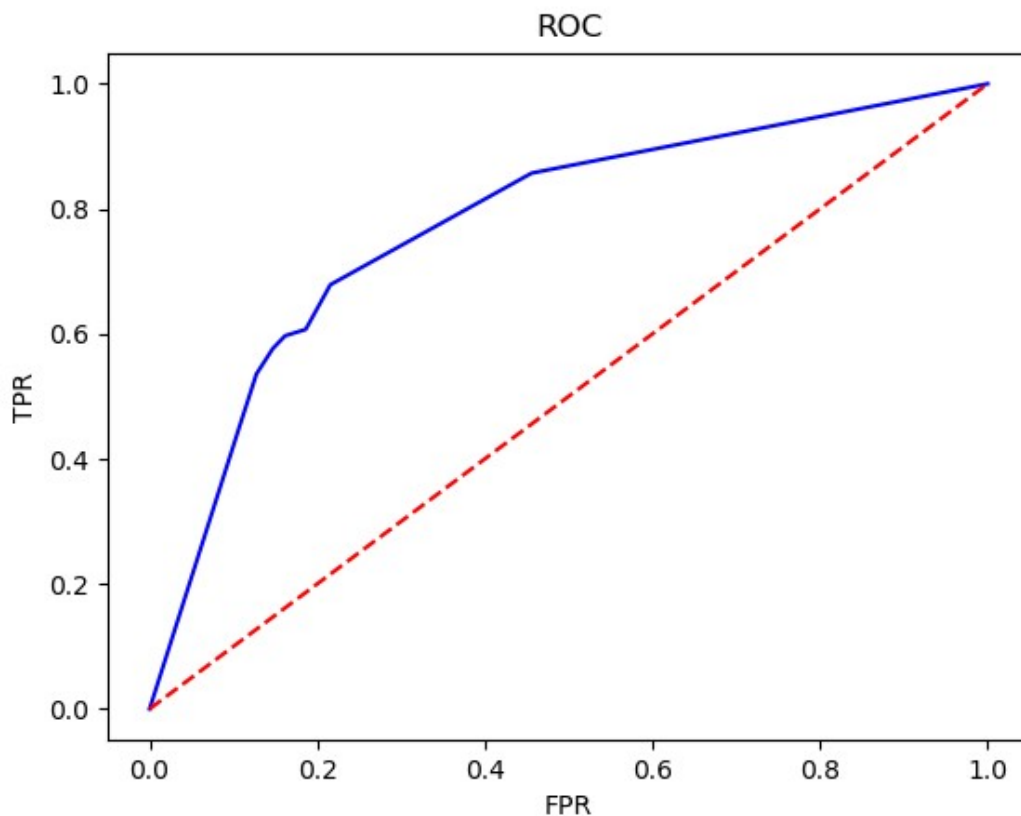
runCode

این تابع اصلی (main) است. ابتدا در آن دیتاست خوانده می‌شود. سپس با فراخوانی `find_priors` برای هر دو کلاس `prior` ها محاسبه می‌شوند. سپس با فراخوانی `find_distributions` برای هر فیچر میانگین و واریانس توزیع نرمال محاسبه می‌شود. سپس با فراخوانی `predictClassOfInstance` و با استفاده از توزیع های محاسبه شده کلاس هر عضو از مجموعه ی تست محاسبه می‌شود سپس با مقدار واقعی آن مقایسه می‌شود و دقت و خطای مجموعه ی تست محاسبه می‌شود. همین کار برای مجموعه ی آموزش تکرار می‌شود و خطا و دقت مجموعه ی آموزش برای آن محاسبه می‌شود. بعد از آن برای `threshold` های مختلف کلاس اجزای مجموعه ی تست محاسبه می‌شود و هر بار `True Positive Rate` و `False Positive Rate` برای آن ها محاسبه می‌شود تا برای رسم ROC مورد استفاده قرار بگیرد.

در دو شکل زیر خروجی حاصل از اجرای این کد را مشاهده می‌کنید:
خطا و دقت مجموعه ی آموزش و تست:

Accuracy Test:0.72, Error Test:0.28
Accuracy Train:0.80375, Error Train:0.19625000000000004

ROC به ازای `threshold` های مختلف:



پیاده سازی Bayes Network:

کدهای این بخش در پوشه machine3_cars موجود است. در این قسمت بر خلاف سایر قسمت‌ها از پایتون 2.7 استفاده کرده‌ام. از پکیج pgmpy برای ایجاد شبکه‌ی بیزین استفاده کرده‌ام.

قسمت الف)

در این قسمت باید یک نایبو بیز بسازیم که این کار را به **دو روش مختلف** انجام داده‌ام:

(۱) **روش اول**، به صورت عادی و با استفاده از رابطه‌ی $p(w|x) = p(w) * p(\text{feature1}|w) * \dots * p(\text{featureN}|w)$ که مانند سوال قبل است با این تفاوت که در سوال قبل از نایبو بیز گاوسی استفاده کردم ولی در اینجا از نایبو بیز گسسته. کد این راه حل اول در پوشه‌ی machineLearning3 و در فایل cars-naive-bayes.py قرار دارد و خروجی حاصل از اجرای آن برابر است با:

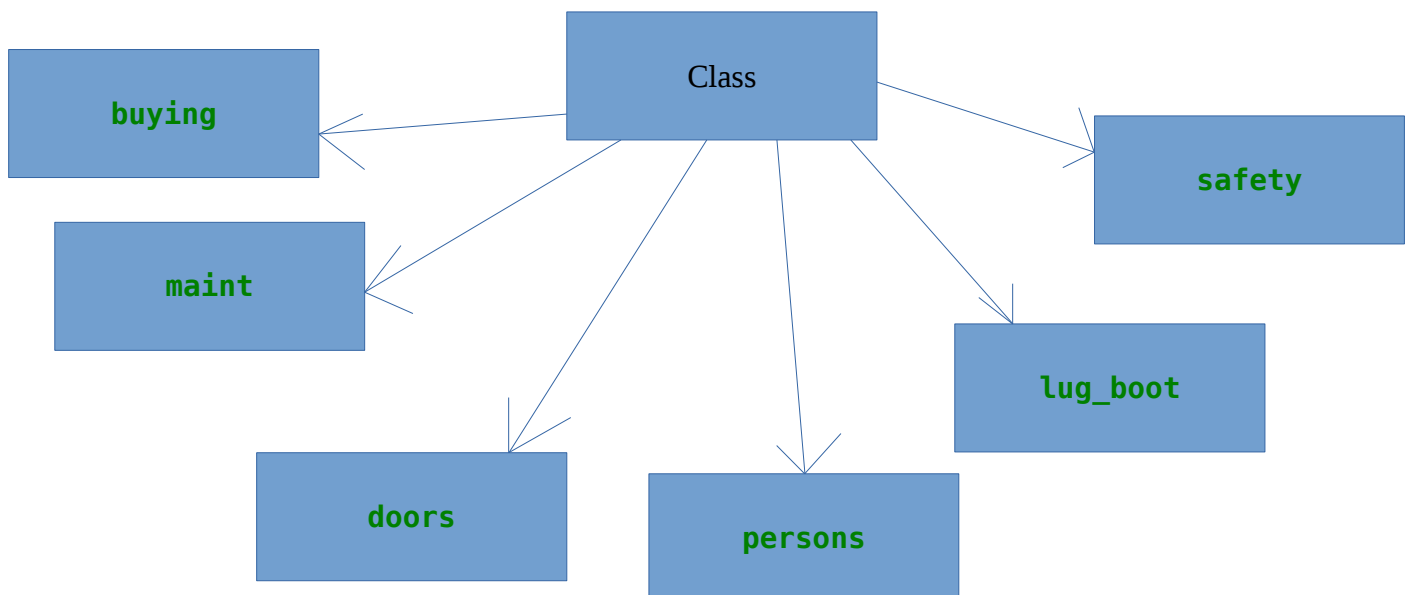
1-Fold-CV Accuracy: 0.77

1-Fold-CV Error: 0.22

این روش را با پایتون سه پیاده سازی کرده‌ام.

(۲) **در روش دوم**، که کدش در پوشه‌ی machine3_cars و در فایل pgmpyNaiveBayesCars.py موجود است، با استفاده از pgmpy یک شبکه‌ی بیزین به شکل زیر برای نایبو ساخته‌ام:

```
# Create Network
car_model = BayesianModel([('class', 'buying'), ('class', 'maint'),
                           ('class', 'doors'), ('class', 'persons'),
                           ('class', 'lug_boot'), ('class', 'safety')])
```



دو تابع برای این بخش نوشته‌ام:

runCode():

این تابع داده‌ها را از فایل می‌خواند و مقدار کلاس و ویژگی‌ها را بر اساس دیکشنری زیر جایگزین می‌کند:

```
attributes = [{ 'vhigh': 0, 'high': 1, 'med': 2, 'low': 3},          #buying
               #attribute
               { 'vhigh': 0, 'high': 1, 'med': 2, 'low': 3},      #maint
               { '2': 0, '3': 1, '4': 2, '5more': 3},           #doors
               { '2': 0, '4': 1, 'more': 2},                    #person
               { 'small': 0, 'med': 1, 'big': 2},                #lug_boot
               { 'low': 0, 'med': 1, 'high': 2},                 #safety
               { 'unacc': 0, 'acc': 1, 'good': 2, 'vgood': 3}    #classes
             ]
```

سپس تابع ten_fold_cross_validation را برای محاسبه‌ی خطا فرا می‌خواند.

تابع ten_fold_cross_validation

در این تابع داده‌ها به ده قسمت تبدیل شده‌اند. هر قسمت یک بار تست می‌شود و قسمت‌های دیگر مجموعه‌ی آموزش می‌شوند. و خطا و دقت بر روی آن اجرا می‌شود و در آخر میانگین خطا بازگردانده می‌شود. بعد از ساخت fold ها در هر کدام از ده بار اجرا شبکه‌ای مانند آنچه گفته شد ساخته می‌شود و شبکه با تابع car_model.fit(train_data) آموزش داده می‌شود و بعد کلاس داده‌های تست با کمترین تابع car_model.predict((test_data)) ایجاد و سنجیده می‌شود. و در آخر بعد از fold-10 میزان خطا بازگردانده می‌شود که در شکل زیر آن را مشاهده می‌کنید:

```
ten_fold_cross_
pgmpyNaiveBayesCars pgmpyNaiveBayesCars
> 0.866279069767
> 0.889534883721
> 0.848837209302
> 0.866666666667
10-Fold-CV Accuracy: 0.855271317829
10-Fold-CV Error: 0.144728682171
```

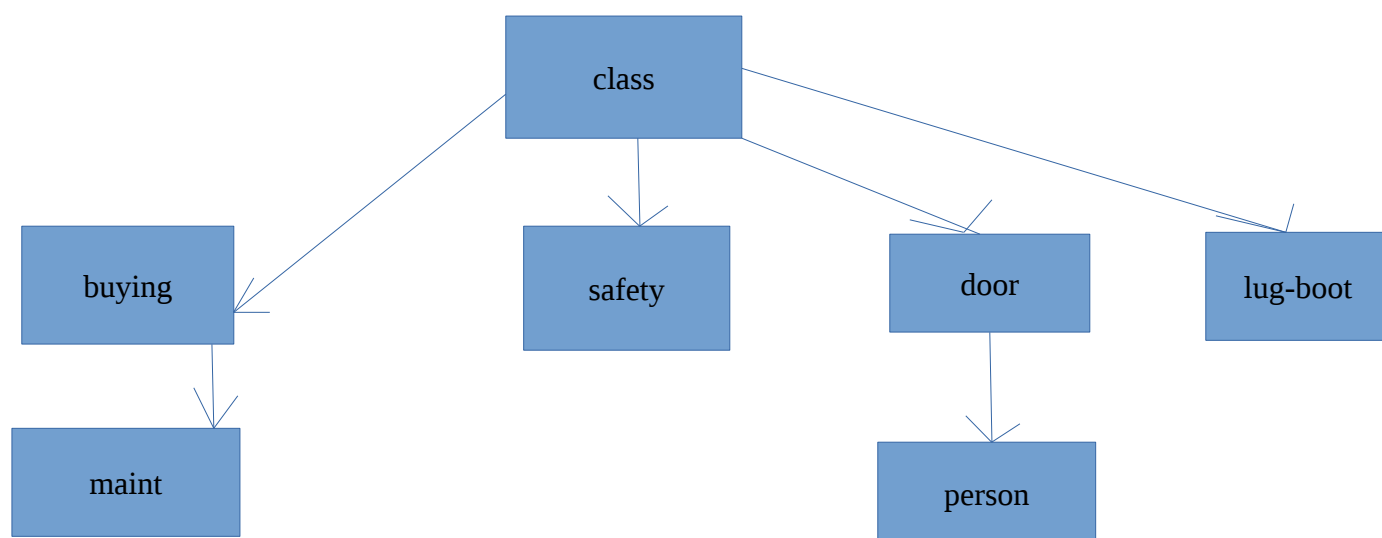
قسمت ب)

در این قسمت خواسته شده است که سه شبکه‌ی بیز بسازیم و آن را آموزش دهیم. کدهای این بخش در پوشه‌ی machine3_cars در فایل‌های زیر هستند.

- pgmpyBayesNetCars.py
- pgmpyBayesNetCars2.py
- pgmpyBayesNetCars3.py

از آنجایی که کدهایی را که برای این سه بخش نوشته‌ام فقط و فقط در بخش وصل کردن یال‌ها با روش دوم قسمت اول این سوال (نایبو بیز) هیچ گونه فرقی ندارند کد را توضیح نمی‌دهم فقط شبکه را رسم می‌کنم و خطای ten-fold را نمایش می‌دهم.

در pgmpyBayesNetCars.py شبکه‌ی بیز زیر را مورد استفاده قرار داده‌ام:

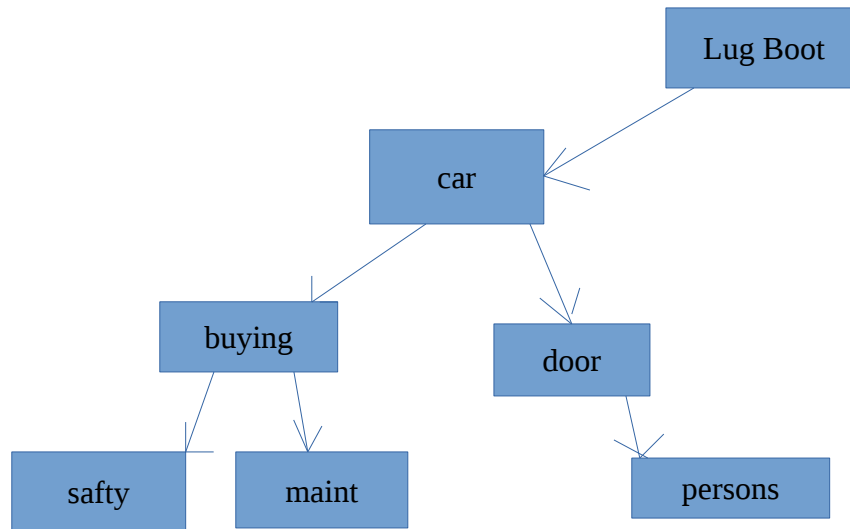


خطای این شبکه برابر است با:

```
pgmpyBayesNetCars  pgmpyNaiveBayesCars
> 0.639534883721
> 0.697674418605
> 0.720930232558
> 0.666666666667
10-Fold-CV Accuracy: 0.696317829457
10-Fold-CV Error: 0.303682170543

Process finished with exit code 0
```

در pgmpyBayesNetCars2.py شبکه‌ی بیز زیر را مورد استفاده قرار داده‌ام:

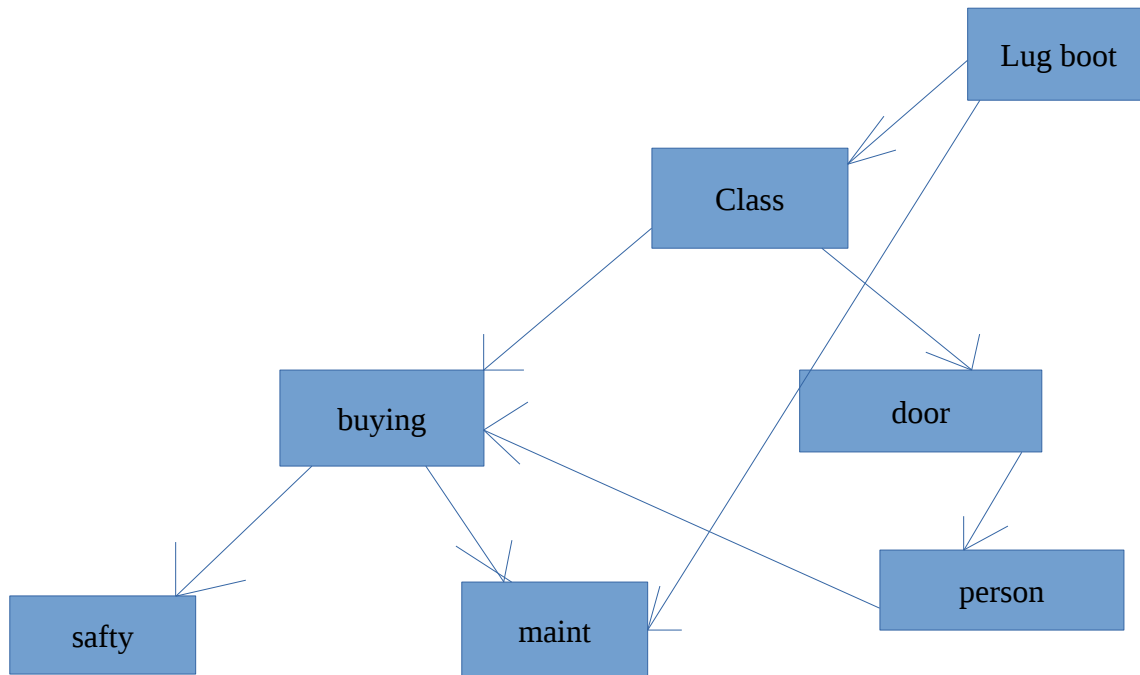


خطا و دقت fold-10

```
pgmpyBayesNetCars2  pgmpyNaiveBayesCars
> 0.674418604651
> 0.656976744186
> 0.773255813953
> 0.716666666667
10-Fold-CV Accuracy: 0.70015503876
10-Fold-CV Error: 0.29984496124

Process finished with exit code 0
```

در pgmpyBayesNetCars3.py شبکه‌ی بیز زیر را مورد استفاده قرار داده‌ام:



خطا در این مدل برابر است با

```
> 0.639534883721
> 0.732558139535
> 0.674418604651
> 0.727777777778
10-Fold-CV Accuracy: 0.700103359173
10-Fold-CV Error: 0.299896640827
```