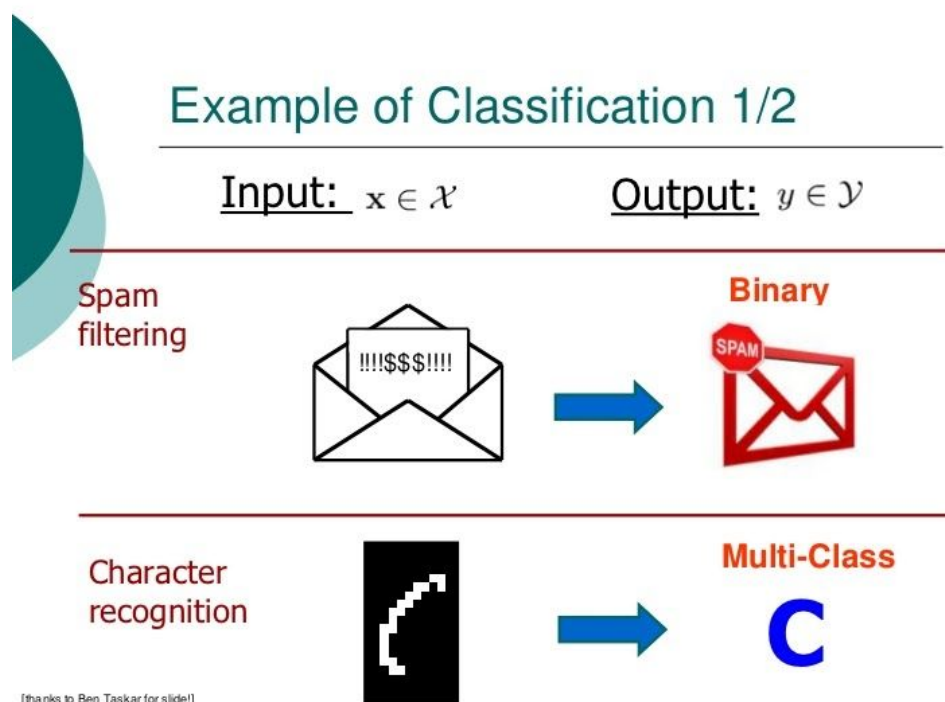


What is the classification?

Classification is a central topic in machine learning that has to do with teaching machines how to group together data by particular criteria. It is different from regression in the sense that target variables in classification are discrete in nature while in regression they are continuous. Remember that both regression and classification fall into the category of supervised learning approaches.

There is also an unsupervised version of the classification, called clustering where computers find shared characteristics by which to group data when categories are not specified (we will encounter it later, not in this concept).

Examples



Other common examples of classification come with classifying loan defaulters, predicting patients as having diabetes or not, etc.

All these types of classification problems can be effectively solved by Logistic Regression. Even though it contains the word "regression", do not take it as a regression algorithm. It is a linear model for classification and is widely used both in academia and industry mainly due to its simplicity and interpretability.

What is the difference between linear regression and logistic regression?

- *Outcome:* This is the fundamental and possibly the most intuitive difference between both algorithms. In linear regression, the outcome (dependent variable) is continuous. It can have any one of an infinite number of possible values, for instance, weight, height, number of hours, etc. Whereas in logistic regression, the outcome (dependent variable) has only a limited number of possible values. For instance, yes/no, true/false, red/green/blue, 1st/2nd/3rd/4th, etc.

- *Linear regression output as probabilities:* It's tempting to use the linear regression output as probabilities but it's a mistake because the output can be negative, and greater than 1 whereas probability can not. As regression might actually produce probabilities that could be less than 0, or even bigger than 1, logistic regression was introduced.

Equation: Linear regression has its own equation of the form: $y = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$. Logistic regression has a somewhat different equation as it interprets probability. You will learn more about it in the coming chapters where its equation is given by $y = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \dots + \theta_n x_n)}}$

- Logistic regression has a somewhat different equation as it interprets probability. You will learn more about it in the coming chapters where its equation is given by

NOTE: Although there is a regression at the end of its name Logistic Regression is used for classification purposes. At the same time, it is a form of the linear model only since the logistic function is a linear combination of weights.

In the last concept, you learnt about Linear Regression. Let's try to solve this problem with Linear Regression first.

Solving with linear regression

At first, let's take only the feature `entropy` to predict the `class` from it. We will look at how the decision boundary looks like and how will the line behave with an unseen data point. In the worst-case scenario, the unseen point can be an outlier. We want our model to generalize well and so, we will test for the worst-case scenario.

The code snippet is given below:

```
# import packages
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
%matplotlib inline

# instantiate model
linear_model = LinearRegression()

# fit the model
linear_model.fit(df[['entropy']], df[['class']])

# generate 1000 X-values
X_sample = np.linspace(-9, 3, 1000)

# calculate y-values for 1000 X-values
Y_sample = X_sample*linear_model.coef_[0] + linear_model.intercept_

# threshold for entropy
threshold = (0.5 - linear_model.coef_[0]) / linear_model.intercept_
print(threshold)

# scatter plot
```

```
plt.scatter(df[['entropy']], df[['class']], marker='x')

# axes specifications
plt.xlabel('Entropy')
plt.ylabel('Class {1:Authentic, 0:Fake}')
plt.ylim(-0.1, 1.1)

# threshold lines
plt.axvline(threshold, linestyle='--', color='green')
plt.axhline(0.5, linestyle='--', color='black')

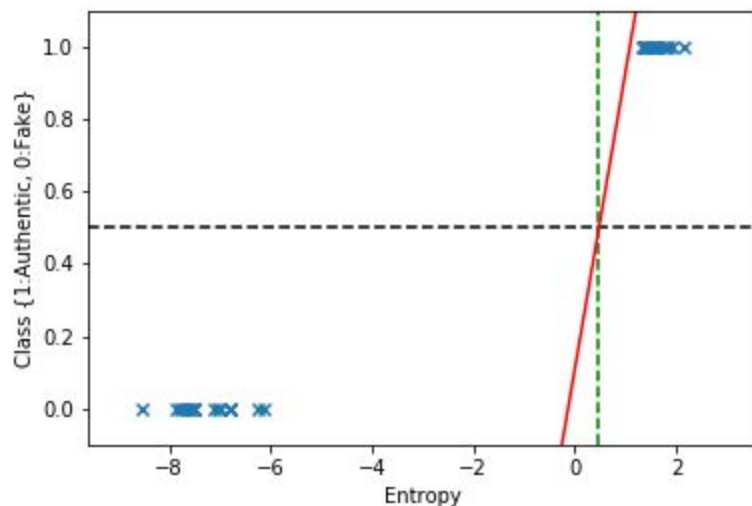
# line plot
plt.plot(X_sample, Y_sample, color='red')

# display plot
plt.show()
```

We get the output as:

```
Threshold value is: [0.47315246]
```

And the output image looks somewhat like this:



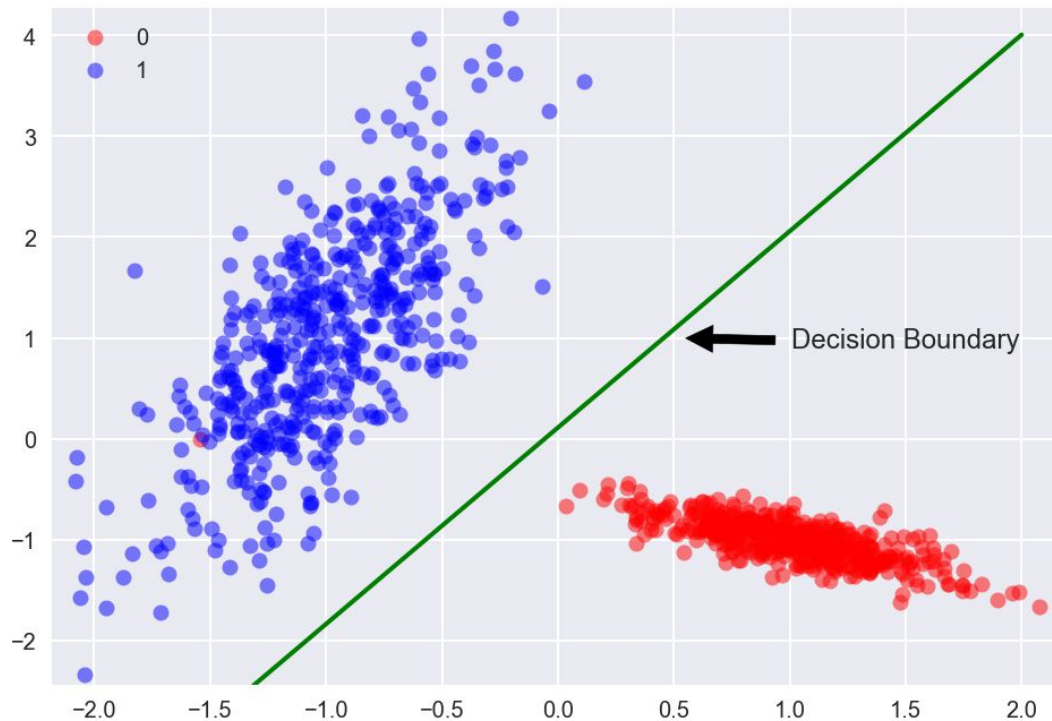
Context of the code snippet

- Trained a linear regression model `linear_model` on `entropy` and `class`
- Used `linear_model` to predict on 1000 samples (`X_sample`)
- The black horizontal line is the `class` threshold (set to 0.5 in our setting)
- The green vertical line is the corresponding threshold for `entropy`
- Redline is the decision boundary

Let's understand more about the decision boundary now.

Decision boundary: It is a sort of demarcation or criteria based on which we segregate our data into different groups. In the image below the straight line is a decision boundary separating the two sets of points colored

red and blue. We can also have non-linear decision boundaries (you will learn in the next chapter about it) as well as higher-dimensional ones.



In our setting `class` attribute has values either 0 or 1. Since we have fit a line, we assume that for y-value above 0.5, we consider it as 1 and 0 for values lower than 0.5. The threshold value is the value for `entropy` at which the `class` changes; which according to our code is approximately 0.47.

Linear Regression for Classification? Not a good idea

You saw that the threshold value for `entropy` was around 0.47. Upon addition of a new data point, we want the threshold values for `entropy` to change as little as possible so that it isn't affected by outliers. Let's add an outlier point and observe for ourselves whether or not the decision boundary and threshold changes.

The code snippet is given for you:

```
# instantiate the linear model
lm = LinearRegression()

# add outlier pointd to 'entropy' and 'class'
x = np.append(df.entropy.values, [35]).reshape(-1,1)
y = np.append(df['class'], [1]).reshape(-1,1)

# fit on new 'entropy' and 'class'
lm.fit(x, y)

# scatter plot
plt.scatter(x, y, marker='x')

# axes modification
plt.xlabel('Entropy')
plt.ylabel('Class {1:Authentic, 0:Fake}')
```

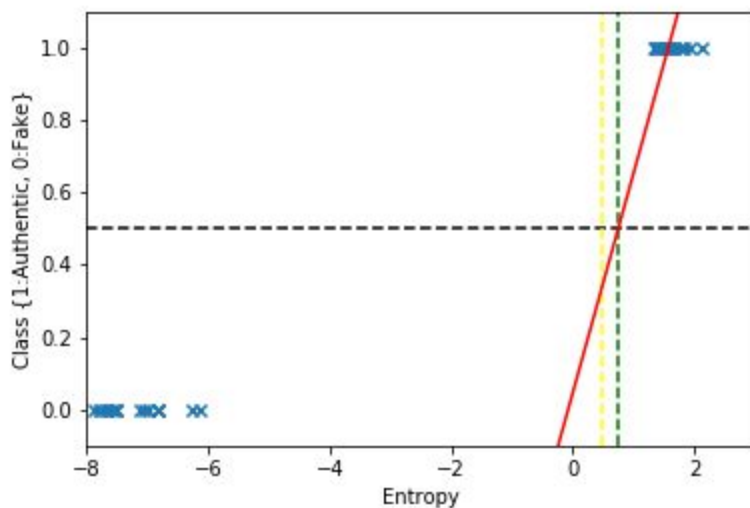
```
plt.ylim(-0.1, 1.1)

# new threshold
new_threshold = (0.5 - lm.coef_[0]) / lm.intercept_

# threshold lines
plt.axvline(new_threshold, linestyle='--', color='green')
plt.axhline(0.5, linestyle='--', color='black')
plt.axvline(threshold, linestyle='--', color='yellow')

# line plots
new_Y_sample = X_sample* lm.coef_[0]+lm.intercept_
plt.plot(X_sample, new_Y_sample, color='red')
plt.xlim(-8, 3)
plt.show()
```

The image looks somewhat like this:



From the image, it is pretty clear that the threshold value changes as evident from the line changing its color from yellow to green. As the threshold changes, so do the predictions and so linear regression is not a reliable model for this type of task.

Sigmoid

Linear Regression is not suitable for classification tasks mainly due to a couple of reasons:

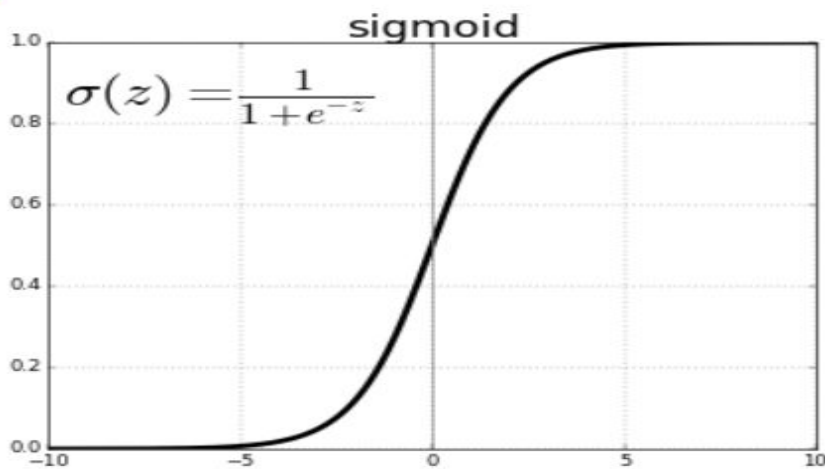
- Upon addition of outliers, the best-fit line changes which in turn changes the threshold for the decision boundary.

- Linear combination of features $\theta_0 + \sum_{i=1}^n \theta_i x_i$ spans from $-\infty$ to ∞ . But for a classification (binary) one you can have only two possible values (for example: 0 and 1)

Hence, linear regression is a very unstable process for classification tasks.

Cure for classification problems

We can overcome it with the help of the **sigmoid** function, also known as the S-curve. It looks somewhat like this:



In the figure, we consider negative labels as having the value 0 while the positive ones as being 1s.



Mathematical Form and Interpretation

$$\sigma(z) = \frac{1}{(1 + e^{(-z)})}$$

where

$$z = \theta_0 + \theta_1 x + \dots + \theta_n x$$

$$P(y = 1|x) = h_{\theta}(x) = \frac{1}{1 + \exp(-\theta^{\top} x)} \equiv \sigma(\theta^{\top} x)$$

$$P(y = 0|x) = 1 - P(y = 1|x) = 1 - h_{\theta}(x)$$

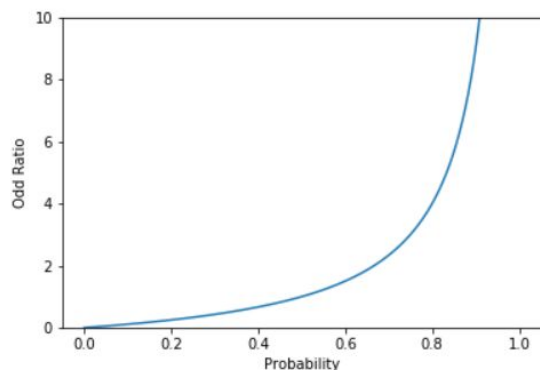
Odds ratio

So by now, you know that the sigmoid function gives us the probability of an instance belong to either of the classes. But ever wondered how we arrived at this step? Well, something called the odds-ratio helped us in arriving at the final form $g(z) = \frac{1}{(1+e^{(-z)})}$. So let us understand more about odds ratio now.

What is the odds ratio?

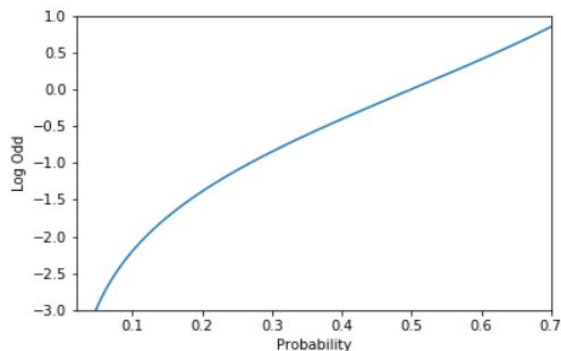
We will answer this question with the help of an example. Let the probability of success of an event is p ($0 \leq p \leq 1$). So, the probability of event failure is $1 - p$. The ratio of the probability of success to the probability of failure is called the odds ratio. Mathematically, it is equivalent to $\frac{p}{1-p}$. If some event has odds of 4, then it means that the chances of success are 4 times more likely than those of failure.

An interesting property of odds is that it is a monotonically increasing function. Odds increase as the probability increases or vice versa.



Log odds

The transformation from odds to log of odds is the log transformation. This is also a monotonic transformation i.e. greater the odds, greater the log of odds and vice versa.



Why take log odds?

It's difficult to model a variable with probability since it has a restricted range between 0 and 1. The log of the odds also called logits transformation is an attempt to get around the restricted range problem. It maps probability ranging between 0 and 1 to log-odds ranging from negative infinity to positive infinity.

A logistic regression model allows us to establish a relationship between a binary outcome variable and a group of predictor variables. It models the logit-transformed probability as a linear relationship with the predictor variables. Mathematically,

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

Observe how the right-hand side of the equation looks similar to the linear regression counterpart.

$$p = \frac{e^{\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n}}{1 + e^{\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n}}$$
$$p = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n)}}$$

which is the sigmoid function

Decision boundary for sigmoid function

The mathematical condition for decision boundary in case of sigmoid will occur when the probability is 0.5.

i.e.

$$\begin{aligned}y &= 0.5 \\ \Rightarrow \frac{1}{1 + e^{-z}} &= 0.5 \\ \Rightarrow 1 + e^{-z} &= 2 \\ \Rightarrow e^{-z} &= 1 \\ \Rightarrow z &= 0 \\ \Rightarrow \theta^T X &= 0\end{aligned}$$

Now, $\theta^T X = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$.

So, this condition i.e. $\theta_0 + \theta_1 x_1 + \dots + \theta_n x_n = 0$ is the equation for the decision boundary of sigmoid function.

By adding polynomial terms to the left hand side of the above equation we can also get a non-linear decision boundary. Lets say you have the function

$$h_{\theta}(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$$

Decision boundary for this equation is

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 = 0$$

Say θ was $[-1, 0, 0, 1, 1]$ then we have;

Predict that $y = 1$ if

$$-1 + x_1^2 + x_2^2 \geq 0$$

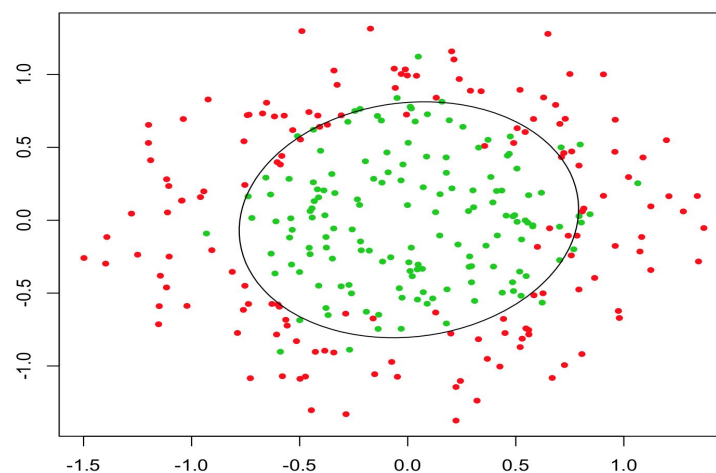
or

$$x_1^2 + x_2^2 \geq 1$$

Else predict

$$y = 0$$

The decision boundary for the above example is a circle of radius 1 and is shown in the image below.



Stability of sigmoid functions decision boundary

The main advantage of the sigmoid function is that always buckets values in the range of [0,1] and it does not get affected by outliers at all. Take a look at the code snippet below and the images that it produces:

```
# import packages
from sklearn.linear_model import LogisticRegression

logreg1 = LogisticRegression()
logreg2 = LogisticRegression()

# Outlier points added
x = np.append(df.entropy.values, [35]).reshape(-1,1)
y = np.append(df['class'], [1]).reshape(-1,1)

# fit model
logreg1.fit(df[['entropy']], df[['class']])
logreg2.fit(x, y)

# initialize figures
fig, (ax_1, ax_2) = plt.subplots(1, 2, figsize=(10,5))

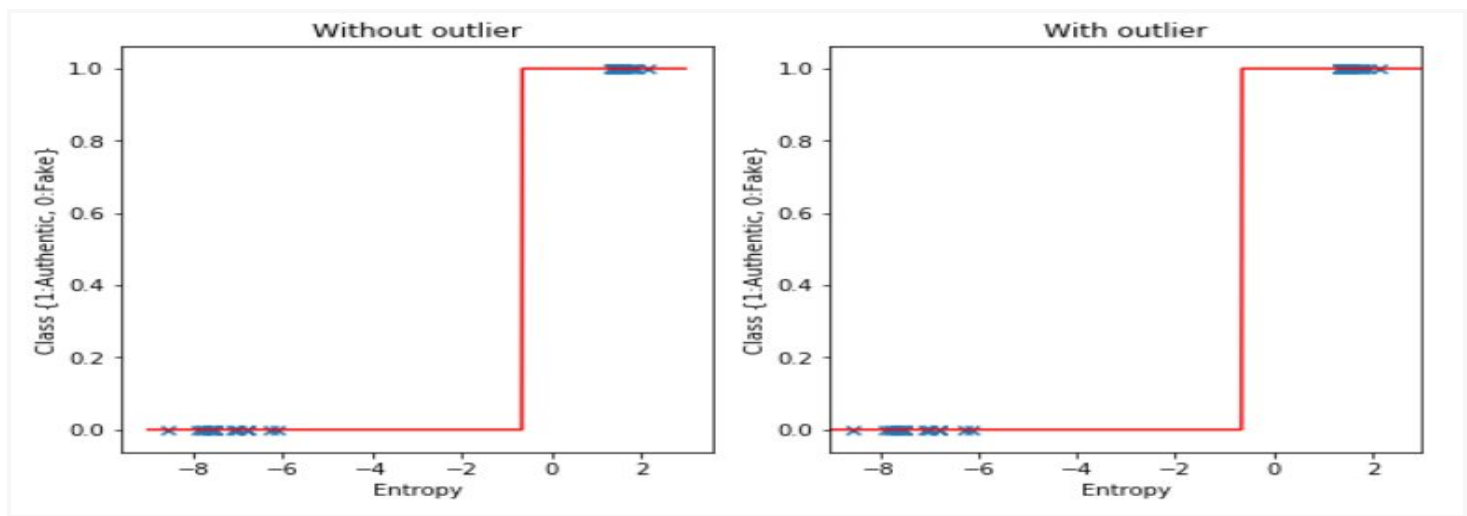
# scatter plot
ax_1.scatter(df[['entropy']], df[['class']], marker='x')
ax_2.scatter(x,y, marker='x')

# axes modifications
ax_1.set_title('Without outlier')
ax_2.set_title('With outlier')
ax_1.set_xlabel('Entropy')
ax_1.set_ylabel('Class {1:Authentic, 0:Fake}')
ax_2.set_xlabel('Entropy')
ax_2.set_ylabel('Class {1:Authentic, 0:Fake}')

# predictions
old_pred = logreg1.predict(X_sample.reshape(-1,1))
new_pred = logreg2.predict(X_sample.reshape(-1,1))

# line plots showing decision boundary for sigmoid
ax_1.plot(X_sample.reshape(-1,1), old_pred, color='red')
ax_2.plot(X_sample.reshape(-1,1), new_pred, color='red')
ax_2.set_xlim(-9, 3)

# display plot
plt.show()
```



The decision boundary is robust enough to deal with outliers.

Multiclass Classification

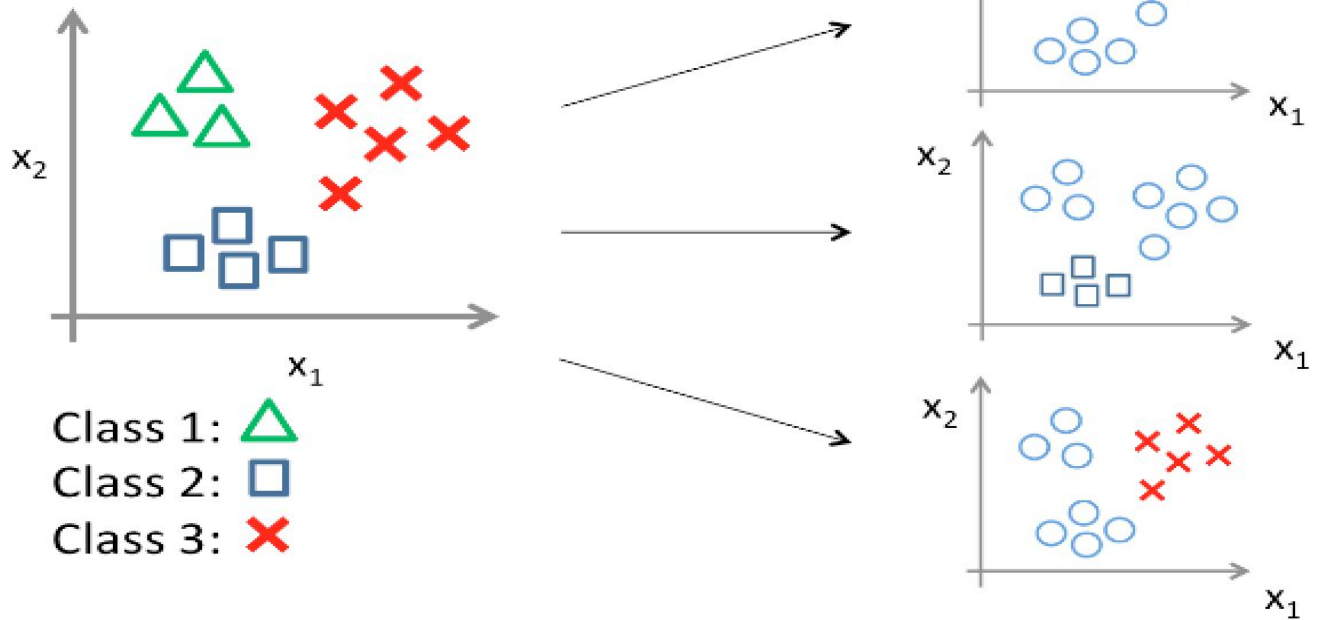
So far we have discussed binary classification only. But what if we have more than two target classes? How to approach such problems? For this kind of situations, we will be covering two methods; namely One-vs-All (One-vs-Rest) method and the Softmax method. Let's discuss them in details.

1. **One-vs-all Method**

Consider the situation where you have K classes ($K > 2$). You will create classifiers equal to the number of classes and use each of them to predict one class as 1 and other classes as 0s. Then you combine all the classifiers and obtain a single new classifier which assigns probabilities to instances belonging to every class. On encountering a new instance, it will output the probability of that instance belonging to every class and in general, we select it as the highest of the predicted probabilities. But like all other methods, this approach is also not bullet-proof. Particularly in problems with class imbalance, it doesn't have the desired effect.

For example, you have a situation at hand to classify shapes like triangles, squares and crosses. So for this classification problem, you will make 3 logistic classifiers. The first classifier will classify triangles as 1s and the other classes as 0s. Similarly, the second classifier will classify squares as 1s and other classes as 0s and the third one will classify crosses as 1s and other classes as 0s. We combine these three classifiers and obtain a single classifier which can now classify shapes into any of the three classes.

One-vs-all (one-vs-rest):



You can go through scikit-learn's official [documentation](#) of One-vs-rest classifier.

2. Softmax Function




Softmax is an extension of sigmoid function generalized for K classes. Consider you have m training examples $(x^1, y^1), (x^2, y^2), \dots, (x^n, y^n)$ and the class labels as $y^{(i)} \in \{1, 2, \dots, K\}$.

Now, given a new input x we want to predict the probability that it belongs each of the classes i.e. $P(y=k | x)$. The hypothesis is given by the equation:

$$h_{\theta}(x) = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \\ \vdots \\ P(y=K|x;\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^K \exp(\theta^{(j)\top} x)} \begin{bmatrix} e^{\theta^{(1)\top} x} \\ e^{\theta^{(2)\top} x} \\ \vdots \\ e^{\theta^{(K)\top} x} \end{bmatrix}$$

The hypothesis outputs a K dimensional vector whose elements sum to 1 giving us our K estimated probabilities. Here, $\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(K)} \in \mathbb{R}^n$ are the model parameters.

Every $\theta^{(i)}$ is again made up of n parameters and so it is convenient θ as a $n \times K$ matrix where $\theta = [\theta^{(1)} \quad \theta^{(2)} \quad \dots \quad \theta^{(K)}]$

QUESTIONS		YOUR ANSWER	CORRECT ANSWER
<p>1. A total predicted logit of 0 can be transformed to a probability of?</p> <p>Explanation: If $\text{logit} = 0$, then $\log(p/(1-p))=0$. So, $p/(1-p)=1$. Therefore, $p=0.5$.</p>		0	0.5
<p>2. In logistic regression, what do we estimate for each unit's change in X(feature)?</p> <p>Explanation: If you recollect the equation for logistic regression</p> $P(Y = 1/x_1, x_2, x_3, \dots) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots$ <p>For every unit change in one of the features, we estimate how much the natural logarithm for the odds of $Y=1$ changes.</p>		How much the natural logarithm of the odds for $Y = 1$	How much the natural logarithm of the odds for $Y = 1$
<p>3. Which of the following option is true?</p> <p>Explanation: Errors have to be normally distributed for Linear Regression; it is a core assumption. But that is not the case with Logistic Regression.</p>		Linear Regression errors values has to be normally distributed but in case of Logistic Regression it is not the case	Linear Regression errors values has to be normally distributed but in case of Logistic Regression it is not the case

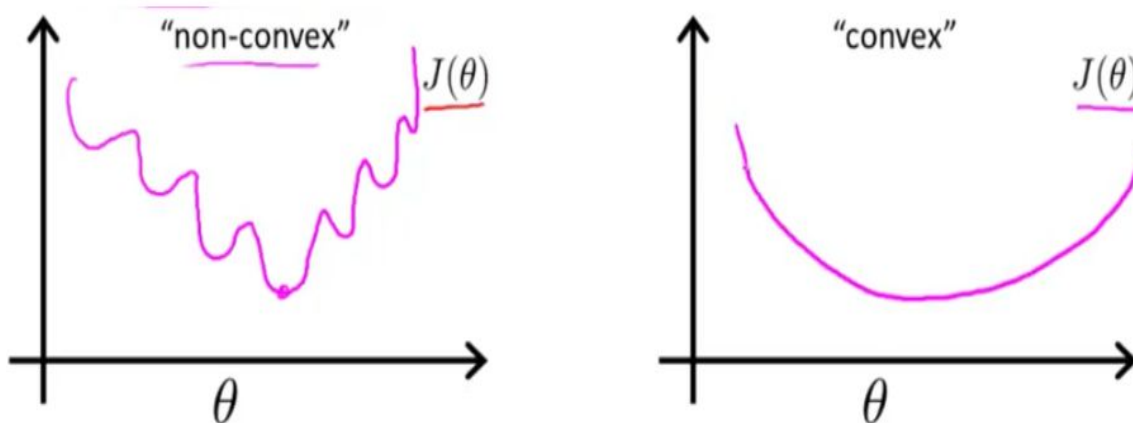
Cost function for Logistic Regression

From the previous tutorial on linear regression, we know that the cost function ($J(\theta)$) for m training examples with hypothesis $\theta(x_i)$ and actual target y_i is $J(\theta) = \frac{1}{m} \sum_{i=1}^m \frac{1}{2} (h_{\theta}(x_i) - y_i)^2$.

Now for linear regression, this is a convex cost function i.e. you are guaranteed to arrive at the global minimum cost. However, the same cannot be said for sigmoid function and applying the same for logistic regression where our hypothesis is $h_{\theta}(x) = \frac{1}{1+e^{-\theta^T X}}$

It is a non-convex function and chances are we might get stuck in some local minima while optimizing our solution.

It is depicted in the image below where we can see multiple local minima for a non-convex function. The image is taken from the massively popular lecture notes from Dr Andrew Ng.



Maximum Likelihood Estimation

In logistic regression instead of minimizing the least-squares error, we try to maximize the likelihood. Likelihood function determines how likely is the observation according to the model. It determines values for the model parameters such that they maximise the chance that the process described by the model produced the data that were actually observed.

Lets assume our hypothesis to be $h_{\theta}(x)$ with data points (x_i, y_i) where $0 \leq h_{\theta}(x) \leq 1$. The probability of observing positive class is given by $h_{\theta}(x)$ and that of observing negative class is given by $1 - h_{\theta}(x)$.

Assuming the underlying distribution as Bernoulli, the objective function becomes

$$\text{Likelihood (L)} = \prod_{i=1}^m h_{\theta}(x_i)^{y_i} (1 - h_{\theta}(x_i))^{1-y_i}$$

Notice that we take products and not the sum while calculating the likelihood as total likelihood is the product of the likelihood of every observation. For convenience, it is easy to deal with the log-form of likelihood. Taking the natural logarithm of the above equation gives us:

$$\ln(L) = \sum_{i=1}^m [y_i \ln h_{\theta}(x_i) + (1 - y_i) \ln(1 - h_{\theta}(x_i))]$$

Maximizing L is the same as minimizing -L and by taking the average over the entire set of m data points we obtain the new cost function as:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y_i \ln h_{\theta}(x_i) + (1 - y_i) \ln(1 - h_{\theta}(x_i))]$$

This term is our new cost function. Maximizing the log-likelihood will give us an optimal solution as it will result in the maximum likelihood (L) and give us the best parameters.

Gradient Descent to find best parameters

We have the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$

On calculating the derivative w.r.t. θ we obtain the best parameters for a single sample:

$$\frac{\partial}{\partial \theta_j} L(\theta) = -\left(y \frac{1}{\sigma(\theta^T X)} - (1 - y) \frac{1}{1 - \sigma(\theta^T X)}\right) \frac{\partial}{\partial \theta_j} \sigma(\theta^T X)$$

$$\frac{\partial}{\partial \theta_j} L(\theta) = -\left(y \frac{1}{\sigma(\theta^T X)} - (1 - y) \frac{1}{1 - \sigma(\theta^T X)}\right) \sigma(\theta^T X) (1 - \sigma(\theta^T X)) \frac{\partial}{\partial \theta_j} \theta^T X$$

$$\frac{\partial}{\partial \theta_j} L(\theta) = -(y(1 - \sigma(\theta^T X)) - (1 - y)\sigma(\theta^T X))x_j$$

$$\frac{\partial}{\partial \theta_j} L(\theta) = (\sigma(\theta^T X) - y)x_j$$

Adding up over all the samples we obtain:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x_i) - y_i)x_j$$

After that we can update the weights (θ s) as per: $\theta = \theta + \alpha \frac{\partial}{\partial \theta_j} J(\theta)$

Intuition behind the Cost Function

Now let's get into the intuition behind the cost function $J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y_i \ln h_{\theta}(x_i) + (1 - y_i) \ln(1 - h_{\theta}(x_i))]$

We will be considering two scenarios, one when the actual target is **1** and other when it is **0**. The intuition is explained with the help of the image below where the X-axis represents the predicted values and Y-axis represents the cost of $J(\theta)$.

Case I: $y = 1$

The cost function becomes $J(\theta) = -\frac{1}{m} \sum_{i=1}^m \ln h_{\theta}(x_i)$, since $(1 - y_i) = 0$.

Its graphical representation is given by the red line in the image below. Carefully observe that if we predict probabilities close to **0**, the cost is almost infinity; but as our prediction reaches **1**, it approaches **0**. Thus, it highly penalizes incorrect predictions.

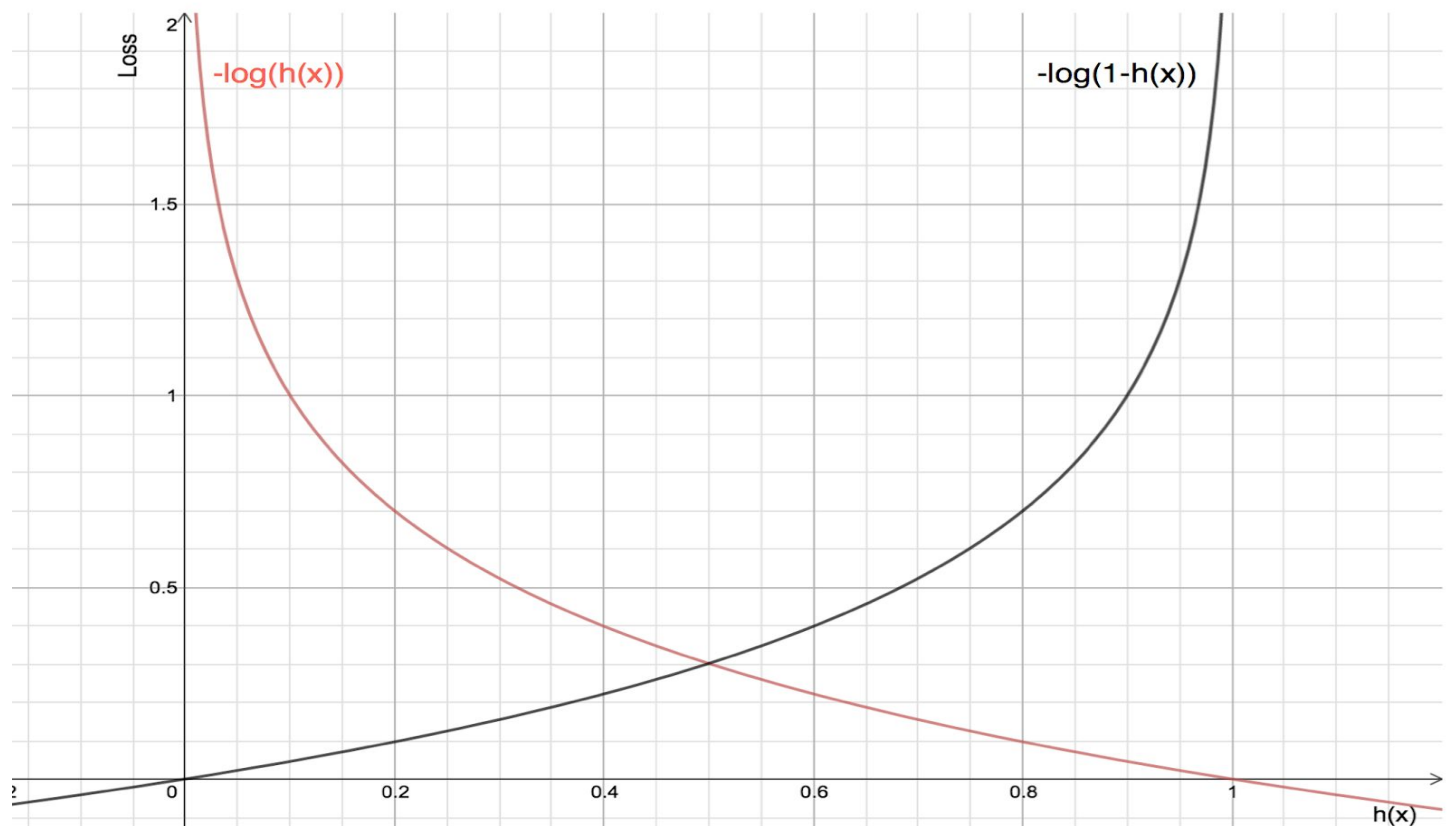
Case II: $y = 0$

Now, this cost function becomes $J(\theta) = -\frac{1}{m} \sum_{i=1}^m (1 - y_i) \ln(1 - h_{\theta}(x_i))$.

This is represented by the black line in the image below. For wrong predictions, it penalizes much more and for closer ones, it penalizes very less.

This is the intuition behind the cost function for logistic regression. After that the weights are being updated in the same manner as linear regression; the only difference being the cost function $J(\theta)$.

With regularization, there is just an added term in cost function and accordingly the weights will get updated so that it doesn't overfit.



Model building using scikit-learn

Till now, you have learnt about classification, sigmoid function and the modifications in the cost function for logistic regression. Now, you will perform necessary preprocessing steps and build a model on that data using scikit-learn.

Remember these steps as well as the sequence is not universal, there are various other things to check for.

Step 1: Split into train and test sets

Split the data into train and test sets with 20% data in the test set

Step 2: Standardize data

Although it is not a mandatory step, since we will be finding coefficients using gradient descent, it is recommended to use normalization so that it converges faster

Possible other measures:

- Check for multicollinearity Logistic regression requires there to be little or no multicollinearity among the independent variables. This means that the independent variables should not be too highly correlated with each other. If there exists some collinearity, avoid using them together.
- Check for missing values and treat them

In this topic, we will be only performing an only train-test split and standardize data with zero mean and unit variance.

After that, you can fit the model on training data and use that to predict unseen or test data.

Confusion matrix

Also called the error matrix, it is a table describing the performance of a supervised machine learning model on the testing data, where the true values are unknown. Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (and vice versa).

Let us understand it with the help of an example.

		Actual class	
		Cat	Non-cat
Predicted class	Cat	5 True Positives	2 False Positives
	Non-cat	3 False Negatives	17 True Negatives

Before going through the calculations, let's understand some terms:

- True Positives (TP): Actually `positive` and predicted `positive` (*CORRECT PREDICTIONS*)
- True Negatives (TN): Actually `negative` and predicted `negative` (*CORRECT PREDICTIONS*)
- False Positives (FP): Actually `negative` but predicted `positive` (*INCORRECT PREDICTIONS*)
- False Negatives (FN): Actually `positive` but predicted `negative` (*INCORRECT PREDICTIONS*)

In the above example with the binary outcome `Cat(1)` or `Non-cat(0)`:

- Actual number of cats = 8
- Actual number of non-cats = 19

Now, from the predictions based on model,

- Predicted cats = $5 + 2 = 7$
- Predicted non-cats = $3 + 17 = 20$

So,

- $TP = 5$ i.e. actual cats and also predicted cats.
- $FP = 2$ i.e. actual non-cats but predicted cats.
- $FN = 3$ i.e. actual cats but predicted non-cats.
- $TN = 17$ i.e. actual non-cats and predicted non-cats.

Confusion matrix represents confusion on unseen data.


Accuracy score, Precision, Recall and F score

Accuracy score

It is the fraction of correct predictions to the total number of predictions on unseen data. Mathematically,

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Remember in our earlier example, **TP** = 5, **FP** = 2, **FN** = 3, **TN** = 17

 So, accuracy score = $\frac{5+17}{5+2+3+17} = \frac{22}{27} = 0.8148$

When to use accuracy?

It is a relatively good metric where we have a symmetric distribution of the targets. but cannot handle cases where predictions for one of the instances is very less. For example: In the case of cancer detection, even if we sample data we will encounter very fewer cases of people having cancer (1). In such a case model will learn more about 0s i.e. non-cancerous and will output a good accuracy score even though it has missed out some actual cancer cases or all of them!

To resolve cases with unbalanced targets, precision, recall and f-1 score come in handy. You can use them according to the business requirement.

Precision

For every predicted class, it is the fraction of the correct predictions to the total number of predictions for that class. It answers the question Of all the values predicted as belonging to the class "X", what percentage is correct?

Mathematically,

$$Precision(P) = \frac{TP}{TP + FP}$$

The precision in our example would be $= \frac{5}{5+2} = \frac{5}{7} = 0.7143$

When to use precision?

Precision is a good measure to determine when the costs of **False Positive is high**.

For instance, email spam detection. In email spam detection, a false positive means that an email that is non-spam (actual negative) has been identified as spam (predicted spam). The email user might lose important emails if the precision is not high for the spam detection model.

Recall

For every actual class, it is the fraction of the number of correct predictions to the total number of actual instances of the class. It answers the question **Of all the instances of the class "X", what percentage did we predict correctly?**

Mathematically,

$$Recall(R) = \frac{TP}{TP + FN}$$

The recall in our previous example is $= \frac{5}{5+3} = \frac{5}{8} = 0.625$

When to use recall?

Recall shall be the model metric we use to select our best model when there is a **high cost associated with False Negative**. For instance, in fraud detection or sick patient detection; if a fraudulent transaction (Actual Positive) is predicted as non-fraudulent (Predicted Negative), the consequence can be very bad for the bank.

F score

It is the harmonic mean of the precision and recall for a classifier. Mathematically,

$$Fscore = \frac{2PR}{P + R}$$

Now, the F-score is $= \frac{2*0.7143*0.625}{0.7143+0.625} = \frac{0.892875}{1.3393} = 0.67$

When to use F score?

If you want to achieve a balance between precision and recall, use the F-1 score. But unfortunately, the F-score isn't the holy grail and has its tradeoffs. It favours classifiers that have similar precision and recall. This is a problem because you sometimes want a high precision and sometimes a high recall. The thing is that increasing precision results in a decreasing recall and vice versa. This is called the precision/recall tradeoff.

Go through official documentations for [accuracy](#), [confusion matrix](#), [F-1 score](#), [precision](#), [recall](#) to look at the syntax.

Find accuracy, precision, recall, f-score

In this task, you will calculate the accuracy score, precision, recall and f-score and also visualise the confusion matrix to see how your classifier is performing.

Instructions

- Use the `predictions` array and actual target labels `y_test` to calculate the confusion matrix accuracy, precision, recall, f-score and save them as `cf`, `acc`, `precision`, `recall` and `f_score` respectively.
- Go through official documentations for [accuracy](#), [confusion matrix](#), [F-1 score](#), [precision](#), [recall](#) to look at the syntax.

Skills Covered:

Machine Learning

Python

Hint

- Calculate `precision` as `precision_score(y_test, y_pred)`. Similarly

```
1 # import packages
2 from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
3 confusion_matrix
4 # confusion matrix
5 cf = confusion_matrix(y_test, y_pred)
6 print(cf.round(2))
7 # accuracy
8 acc = accuracy_score(y_test, y_pred)
9 print(acc.round(2))
10 # precision
11 precision = precision_score(y_test, y_pred)
12 print(precision.round(2))
13 # recall
14 recall = recall_score(y_test, y_pred)
15 print(recall.round(2))
16 # F-score
17 f_score = f1_score(y_test, y_pred)
18 print(f_score.round(2))
19 print(y_test.value_counts())
```

```
[[146  2]
 [ 2 125]]
0.99
0.98
0.98
0.98
0
148
1
127
Name: class, dtype: int64
```

ROC-AUC score

ROC stands for Receiver Operator Characteristic, is a curve that helps us visualize the performance of a binary classifier. The Area under the curve (AUC) of the ROC curve indicates the ability of the binary classifier to distinguish between both the classes. It is calculated using all possible threshold probabilities, unlike other metrics that use a fixed threshold.

Now, let us understand this using the actual ROC diagram which is a graphical plot of TPR on the Y-axis and FPR on the X-axis for various threshold settings. The TPR is nothing but the recall term that we had already discussed whereas FPR is the term which indicates how much the classifier incorrectly predicts a negative instance as positive.

The top left image shows the two distributions of negatives (left side gaussian) and positives (right side gaussian) and the corresponding ROC curve below. The threshold is the vertical line in the top image. By moving the line from left to right, we obtain different thresholds, calculate the respective TPRs and FPRs and plot it on the ROC curves. The AUC lies between 0 and 1.

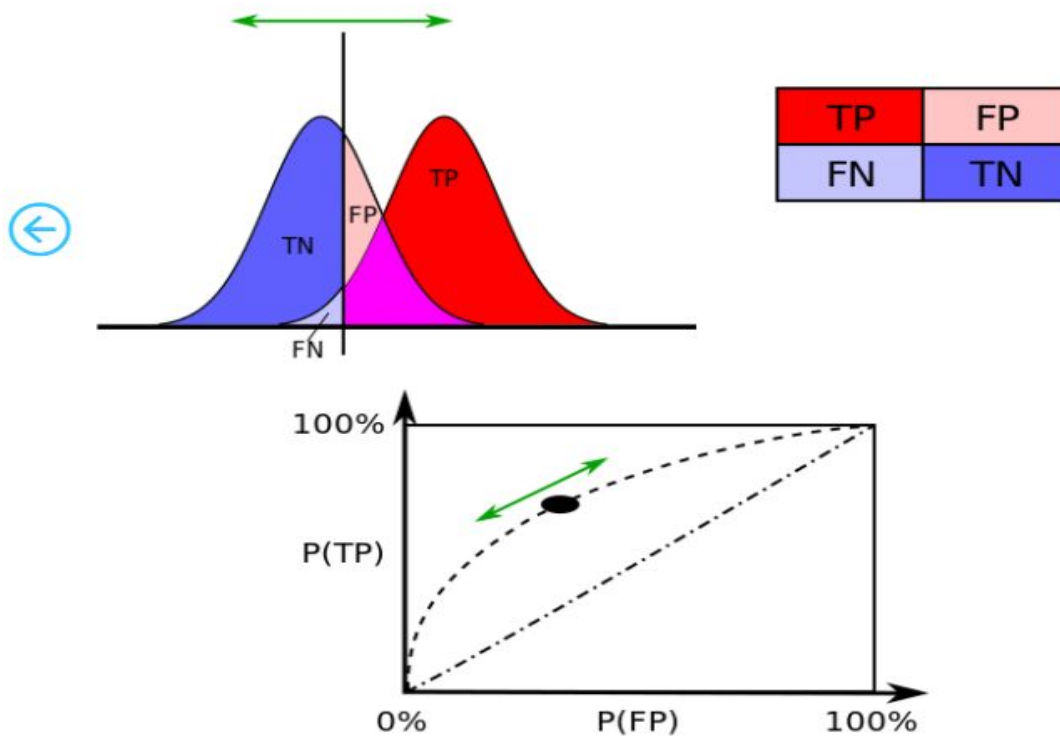
Evaluate your classifier based on the AUC score according to:

- .90-1 = excellent classifier

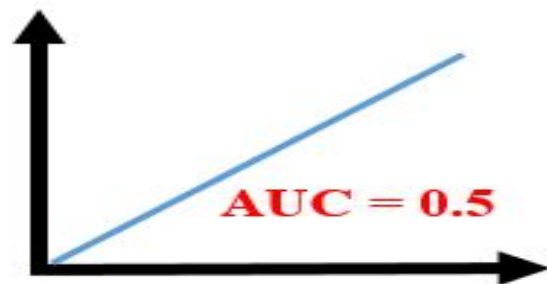
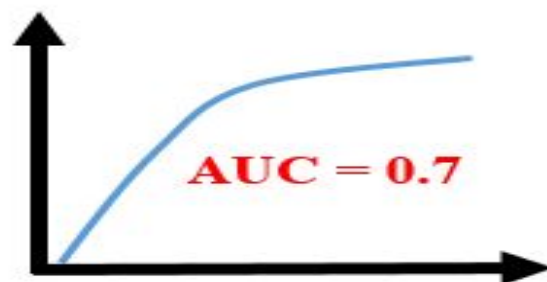
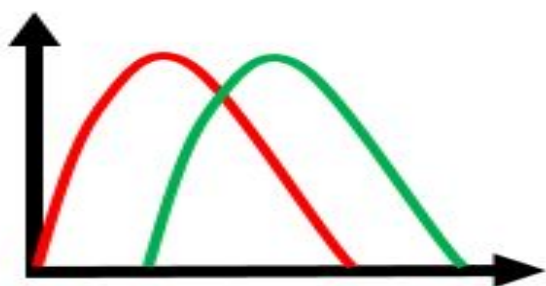
- .80-.90 = good classifier
- .70-.80 = fair classifier
- .60-.70 = poor classifier
- .50-.60 = fail classifier

$$TPR = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{FP + TN}$$



Now let us understand why a low AUC score represents a bad classifier and vice-versa.



In the first image, the classifier has an AUC score of 0.7 whereas the lower one has 0.5. Now, a higher AUC has classes which have a higher separation as evident from the image in the top left. More separation means the same FPR the TPR is higher and vice-versa. An ideal classifier separates both the classes perfectly and as a result, the TPR is 1 and FPR is 0. The worst classifier has an AUC score of 0.5 which happens when it is unable to make a distinction between the classes.

Advantages of using AUC score

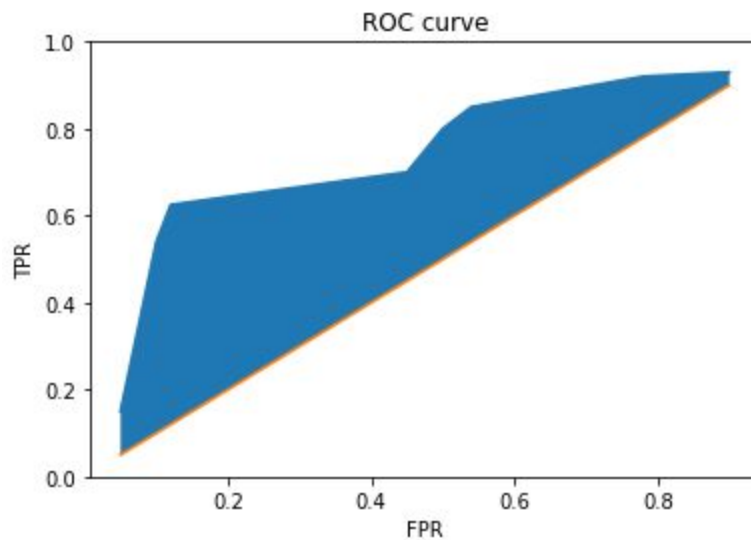
- It is useful even if predicted probabilities are not properly calibrated. For instance, our probabilities may lie in the range (0.8,0.95). This would have no effect on the AUC score.
- Useful metric even if the classes are imbalanced.

Example

In our previous (cat vs non-cat) example, we had TP = 5, FP = 2, FN = 3 and TN = 17. Here, TPR = 0.625 and FPR = 0.12 for a threshold of 0.5. Let's say that we have the following data for the different thresholds and their corresponding FPRs and TPRs

Threshold	FPR	TPR
0.06	0.05	0.15
0.1	0.1	0.54
0.3	0.45	0.7
0.45	0.5	0.8
0.6	0.54	0.85
0.85	0.78	0.92
0.95	0.9	0.93

Plotting them in an X-Y plot we get the following graph:



The blue shaded region represents the AUC with FPR and TPR on the X-axis and Y-axis respectively.

Calculate AUC

- Calculate the ROC AUC score using scikit learn and save it as `roc`. Go through its official [documentation](#)

```
# import packages

from sklearn.metrics import roc_auc_score

# Code starts here

roc = roc_auc_score(y_test, y_pred)

print(roc)

# Code ends here
```

Real Life Use Case

Business Objective

Health Care, today, is facing serious problems: quality of care does not meet patients' needs and costs are exploding. In the cardiology department of the XYZ hospital, discharged patients are advised to participate in a rehabilitation program. However, many of the discharged patients do not join the program or even quit before the program ends. In both cases, the hospital loses revenue; every visit is charged individually.

The hospital wants to increase the revenues by either attracting more patients to the rehabilitation program or reducing the fractions of drop-outs. Thus the hospital wanted to identify if the patient would join the rehabilitation program after discharge.

Solution Methodology

A data set with treated patients was available. The aim was to model the probability that a patient will join the program as a function of various numerical and categorical influence factors. This was achieved through a formal statistical analysis using logistic regression.

The logistic regression model revealed the important influence factors. The probability that a discharged patient will join the rehabilitation program was highly dependent on

- whether a person had a car at his/her disposal
- the distance of the patient's house from the rehabilitation centre
- the probability also decreased in a non-linear way with the age

The factors age & distance are something that cannot be controlled. So the hospital decided to devise a plan to influence mobility. After some analysis on the cost, the project lead came up with a carpooling procedure to a couple of patients with a car to patients without a car.

Business Impact

After a pilot phase of this project, a number of patients started using this carpooling service. Most of the patients also shared that they would not have opted for the rehabilitation program if this service was not there. The average number of patients attending the program in a month increased from 33 to 36 and the average number of sessions that a patient attends increased from 29 to 32.

TASK

Outlier Detection

Let's plot the `box plot` to check for the outlier.

Instructions:

- Plot the boxplot for `X_train['bmi']`.
- Set quantile equal to `0.95` for `X_train['bmi']` and store it in variable `q_value`.
- Check the value counts of the `y_train`.

Skills Covered:

Visualization Data Wrangling

Reference Solution

```
1 import matplotlib.pyplot as plt
2
3 # plot the boxplot
4
5 plt.boxplot(X_train['bmi'])
```

CODE **DATA** **RESET CODE** Code Saved

```
1 import matplotlib.pyplot as plt
2
3 # Code starts here
4 plt.boxplot(X_train['bmi'])
5 q_value = X_train['bmi'].quantile(0.95)
6 print(q_value)
7 print(y_train.value_counts(normalize = True))
8 # Code ends here
```

Congrats!

As we can see that there is an outlier. But if the BMI increases there is a very high chance that people will take the insurance. So, in this case, we are not going to remove the outlier we will keep it. So we can see that it is not necessary that outliers must be removed or processed in every dataset. It depends on the relation of the feature with the target.

CONTINUE

OUTPUT

RESULT

```
41.052249999999994
1 0.568224
0 0.431776
Name: insuranceclaim, dtype: float64
```


Predictor check!

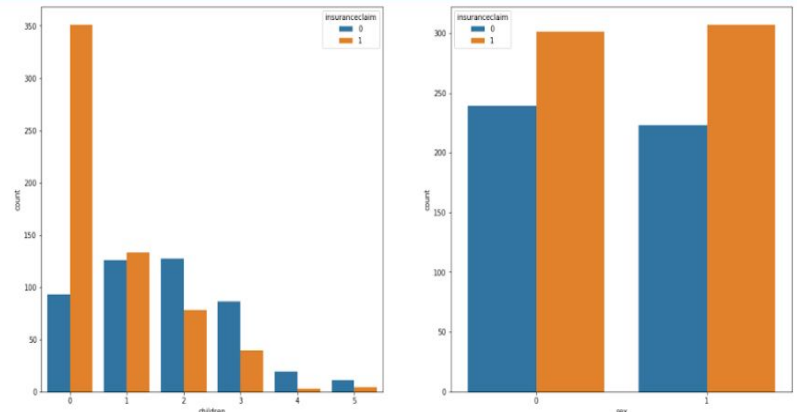
Let's check the `count_plot` for different features vs target variable `insuranceclaim`. This tells us which features are highly correlated with the target variable `insuranceclaim` and help us predict it better.

Instructions :

- Create a list `cols` store the columns `'children', 'sex', 'region', 'smoker'` in it.
- Create subplot with `(nrows = 2 , ncols = 2)` and store it in variable's `fig , axes`
- Create `for` loop to iterate through row.
- Create another `for` loop inside `for` to access column.
- create variable `col` and pass `cols[i * 2 + j]`.
- Using seaborn plot the `countplot` where `x=X_train[col], hue=y_train, ax=axes[i,j]`

Skills Covered:

```
2 import matplotlib.pyplot as plt
3 # code starts here
4 # store categorical variable
5 cols = ['children', 'sex', 'region', 'smoker']
6 # create subplot
7 fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(20,20))
8 # create loop for plotting countplot
9 for i in range(0,2):
10     for j in range(0,2):
11         col=cols[i*2 + j]
12         sns.countplot(x=X_train[col], hue=y_train, ax=axes[i,j])
```



Is my Insurance claim prediction right?

Now let's come to the actual task, using logistic regression to predict the `insuranceclaim`. We will select the best model by cross-validation using Grid Search.

Instructions:

- You are given a list of values for regularization parameters for the logistic regression model.
- Instantiate a logistic regression model with `LogisticRegression()` and pass the parameter as `random_state=9` and save it to a variable called `'lr'`.
- Inside `GridSearchCV()` pass `estimator` as the logistic model, `param_grid=parameters` to do grid search on the logistic regression model store the result in variable `grid`.
- Fit the model on the training data `X_train` and `y_train`.
- Make predictions on the `X_test` features and save the results in a variable called `'y_pred'`.
- Calculate accuracy for `grid` and store the result in the variable `accuracy`
- print `accuracy`.

```
1 from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
2 from sklearn.linear_model import LogisticRegression
3 from sklearn.metrics import accuracy_score
4
5
6 # parameters for grid search
7 parameters = {'C':[0.1,0.5,1,5]}
8
9 # Code starts here
10 lr = LogisticRegression(random_state=9)
11 grid = GridSearchCV(lr,param_grid = parameters)
12 grid.fit(X_train,y_train)
13 y_pred = grid.predict(X_test)
14 accuracy = accuracy_score(y_test,y_pred)
15 print(accuracy)
16 # Code ends here
```

Congrats! You have successfully applied the Logistic Regression Model and calculated the accuracy which comes to be 0.82 using GridSearchCV for prediction of `insuranceclaim`.

CONTINUE

RESULT

0.8246268656716418

Performance of a classifier !

Now let's visualize the performance of a binary classifier. Check the performance of the classifier using `roc auc curve`.

Instructions:

- Calculate the `roc_auc_score` and store the result in variable `score`.
- Predict the probability using `grid.predict_proba` on `X_test` and take the second column and store the result in `y_pred_proba`.
- Use `metrics.roc_curve` to calculate the `fpr` and `tpr` and store the result in variables `fpr, tpr, _`.
- Calculate the `roc_auc` score of `y_test` and `y_pred_proba` and store it in variable called `roc_auc`.
- Plot auc curve of `'roc_auc'` using the line `plt.plot(fpr,tpr,label="Logistic model, auc="+str(roc_auc))`.

Skills Covered:

Machine Learning

Python

```
1 from sklearn.metrics import roc_auc_score
2 from sklearn import metrics
3 score = roc_auc_score( y_test, y_pred)
4 print(score)
5 y_pred_proba = grid.predict_proba(X_test)[:,:1]
6 print(y_pred_proba[0])
7 fpr, tpr, _ = metrics.roc_curve(y_test, y_pred_proba)
8 roc_auc = roc_auc_score(y_test, y_pred_proba)
9 print(roc_auc)
10 plt.plot(fpr,tpr,label="Logistic model, auc="+str(roc_auc))
11 plt.legend(loc=4)
12 plt.show()
```

0.7901382488479263
0.38587218324522565
0.8970814132104454

