HW_4_SP22

March 1, 2022

1 Name: Rohit Chandra

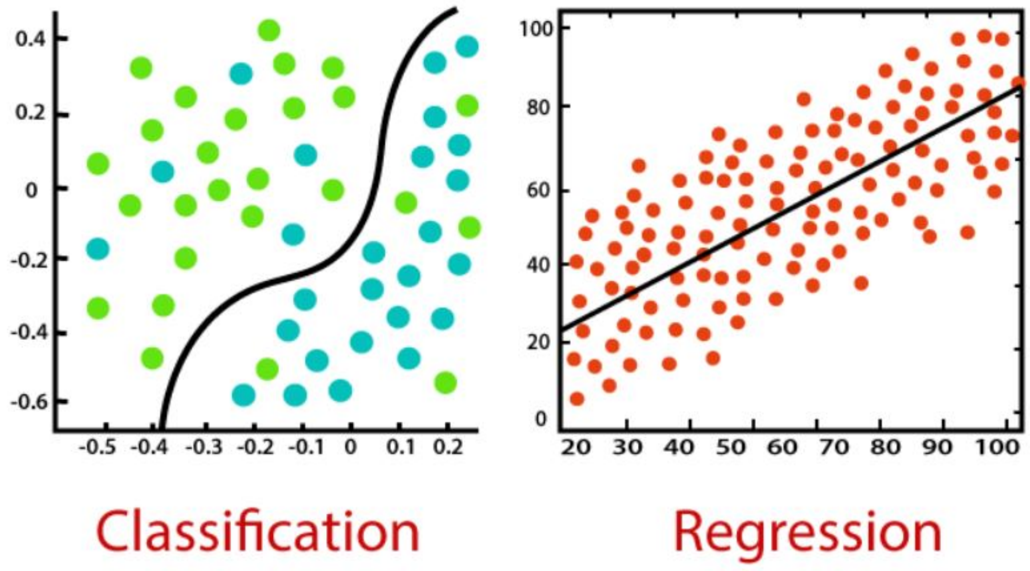
2 Classification Algorithms: Logistic Regression and Support Vector Machine

2.1 1 Explain what is classification and how it is different from regression

Classification is the process of finding a model that separates input data into multiple discrete classes or labels. In other words, a classification problem determines whether or not an input value can be part of a pre-identified group.

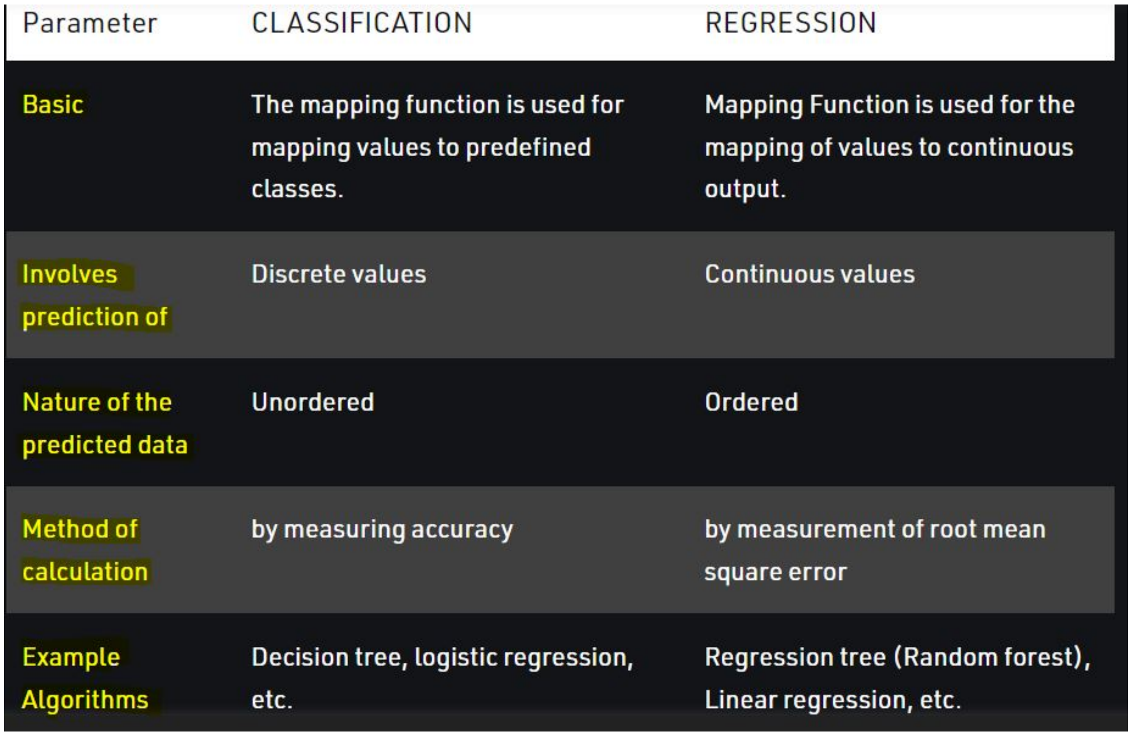
Consider the same dataset of all the students at a university. A classification task would be to use parameters, such as a student's weight, major, and diet, to determine whether they fall into the "Above Average" or "Below Average" category. Note that there are only two discrete labels in which the data is classified.

A classification algorithm is evaluated by computing the accuracy with which it correctly classified its input.



The following are the differences between regression and classification The **main difference between Regression and Classification algorithms** that **Regression algorithms**

are used to predict the continuous values such as price, salary, age, etc. and **Classification algorithms** are used to predict/Classify the discrete values such as Male or Female, True or False, Spam or Not Spam, etc.



Parameter	CLASSIFICATION	REGRESSION
Basic	The mapping function is used for mapping values to predefined classes.	Mapping Function is used for the mapping of values to continuous output.
Involves prediction of	Discrete values	Continuous values
Nature of the predicted data	Unordered	Ordered
Method of calculation	by measuring accuracy	by measurement of root mean square error
Example Algorithms	Decision tree, logistic regression, etc.	Regression tree (Random forest), Linear regression, etc.

2.2 2 Explain what is Logistic regression, its working and how it is different from linear regression

- This type of statistical analysis (also known as **logit model**) is often used for predictive analytics and modeling, and extends to applications in machine learning. In this analytics approach, the dependent variable is finite or categorical: either A or B (binary regression) or a range of finite options A, B, C or D (multinomial regression). It is used in statistical software to understand the relationship between the dependent variable and one or more independent variables by estimating probabilities using a logistic regression equation.
- This type of analysis can help you **predict the likelihood of an event happening or a choice being made.** For example, you may want to know the likelihood of a visitor choosing an offer made on your website — or not (dependent variable). Your analysis can look at known characteristics of visitors, such as sites they came from, repeat visits to your site, behavior on your site (independent variables).
- **Logistic regression** models help you determine a probability of what type of visitors are likely to accept the offer — or not. As a result, you can make better decisions about promoting your offer or make decisions about the offer itself.

How does Logistic Regression Work?

The logistic regression equation is quite similar to the linear regression model.

Consider we have a model with one predictor "x" and one Bernoulli response variable "$\hat{y}$" and p is the probability of $\hat{y}$=1. The linear equation can be written as:

$$p = b_0 + b_1 x \quad \text{--------> eq 1}$$

- The right-hand side of the equation (b0+b1x) is a linear equation and can hold values that exceed the range (0,1). But we know probability will always be in the range of (0,1).
- To overcome that, we predict odds instead of probability.
- **Odds: The ratio of the probability of an event occurring to the probability of an event not occurring.**
- **Odds = p/(1-p)**
- The equation 1 can be re-written as:

$$p/(1-p) = b_0 + b_1 x \quad \text{--------> eq 2}$$

- Odds can only be a positive value, to tackle the negative numbers, we predict the **logarithm of odds.**
- **Log of odds = ln(p/(1-p))**
- The equation 2 can be re-written as:

$$\ln(p/(1-p)) = b_0 + b_1 x \quad \text{--------> eq 3}$$

- To recover p from equation 3, we apply exponential on both sides.

$$\exp(\ln(p/(1-p))) = \exp(b_0 + b_1 x)$$
$$e^{\ln(p/(1-p))} = e^{(b0+b1x)}$$

- From the inverse rule of logarithms,

$$p/(1-p) = e^{(b0+b1x)}$$

- Simple algebraic manipulations

$$p = (1-p) * e^{(b0+b1x)}$$
$$p = e^{(b0+b1x)} - p * e^{(b0+b1x)}$$

- Taking p as common on the right-hand side

$$p = p * ((e^{(b0+b1x)})/p - e^{(b0+b1x)})$$
$$p = e^{(b0+b1x)} / (1 + e^{(b0+b1x)})$$

- Dividing numerator and denominator by e(b0+b1x) on the right-hand side

$$p = 1 / (1 + e^{-(b0+b1x)})$$

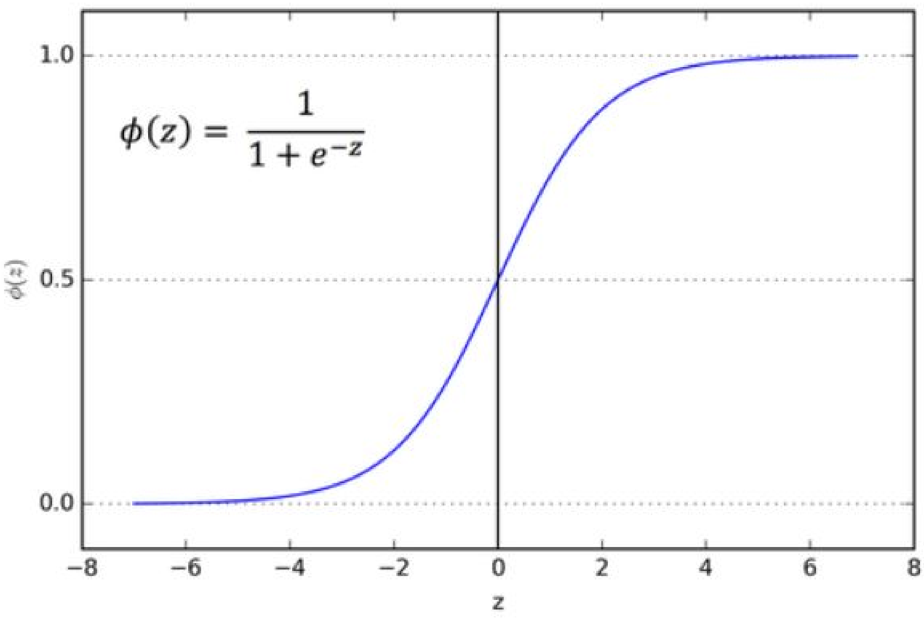
- Similarly, the equation for a logistic model with ‘n’ predictors is as below:

$$p = 1/\ (1 + e^{-(b0+b1x1+b2x2+b3x3+----+bnxn})$$

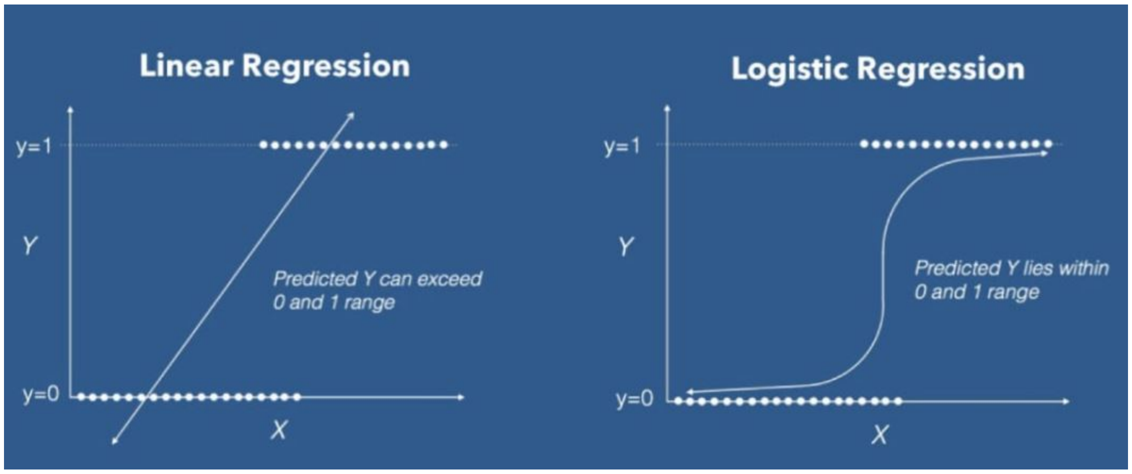
- The right side part is the sigmoid function. It helps to squeeze the output to be in the range between 0 and 1.

Sigmoid Function:

- The sigmoid function is useful to map any predicted values of probabilities into another value between 0 and 1.

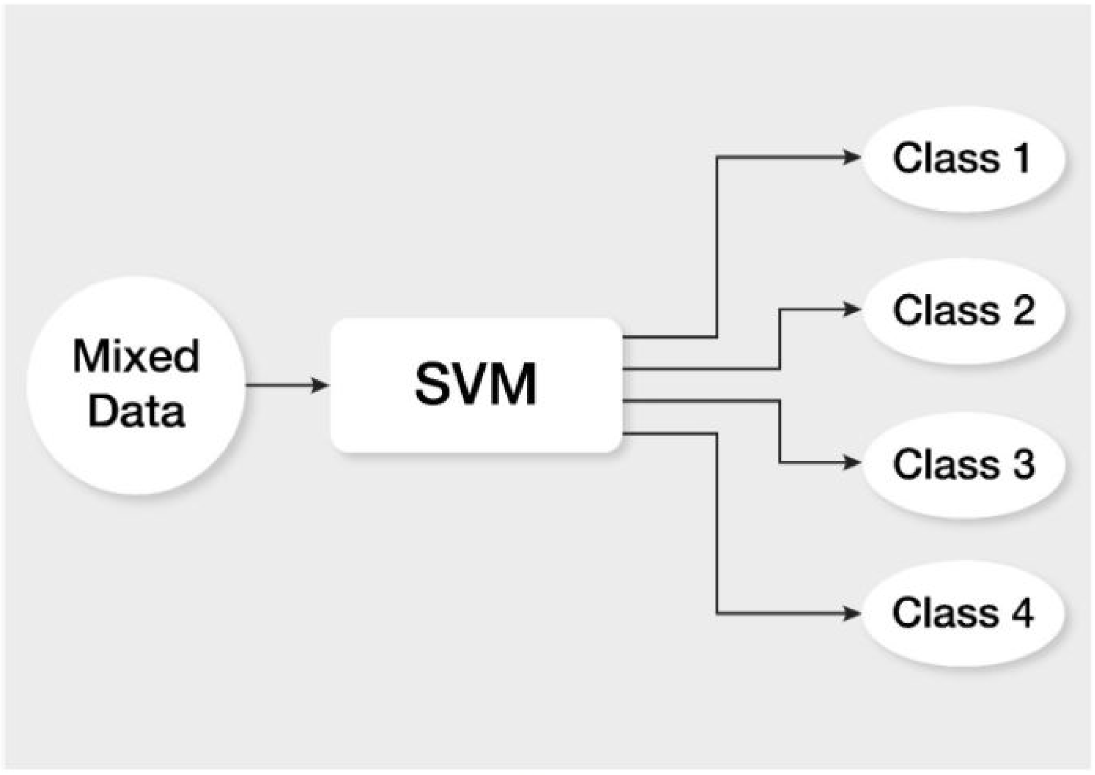


- We started with a linear equation and ended up with a logistic regression model with the help of a sigmoid function.
  - **Linear model: ŷ = b0+b1x**
  - **Sigmoid function: (z) = 1/(1+e−z)**
  - **Logistic regression model: ŷ = (b0+b1x) = 1/(1+e-(b0+b1x))**
- So, unlike linear regression, **we get an ‘S’ shaped curve** in logistic regression.



2.3 3 Explain what is Linear SVM and its working

- **SVM or Support Vector Machine:** is a linear model for classification and regression problems. It can solve linear and non-linear problems and work well for many practical problems.
- The idea of SVM is simple: The algorithm creates **a line or a hyperplane which separates the data into classes.**

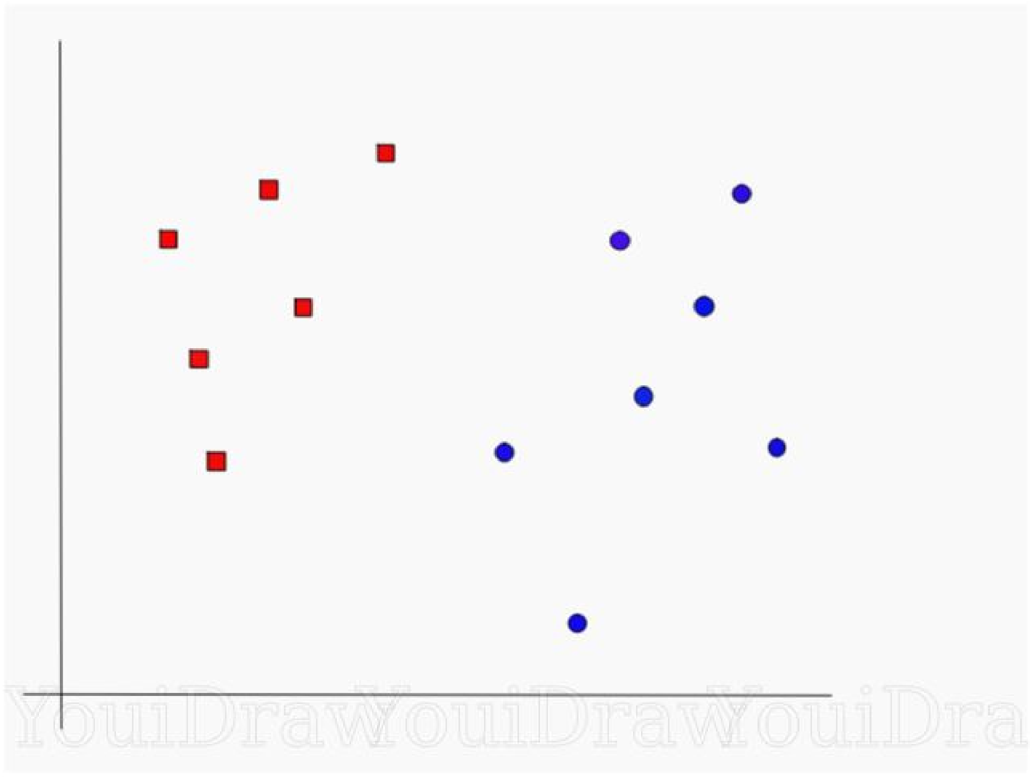


SVM classifier

THEORY:

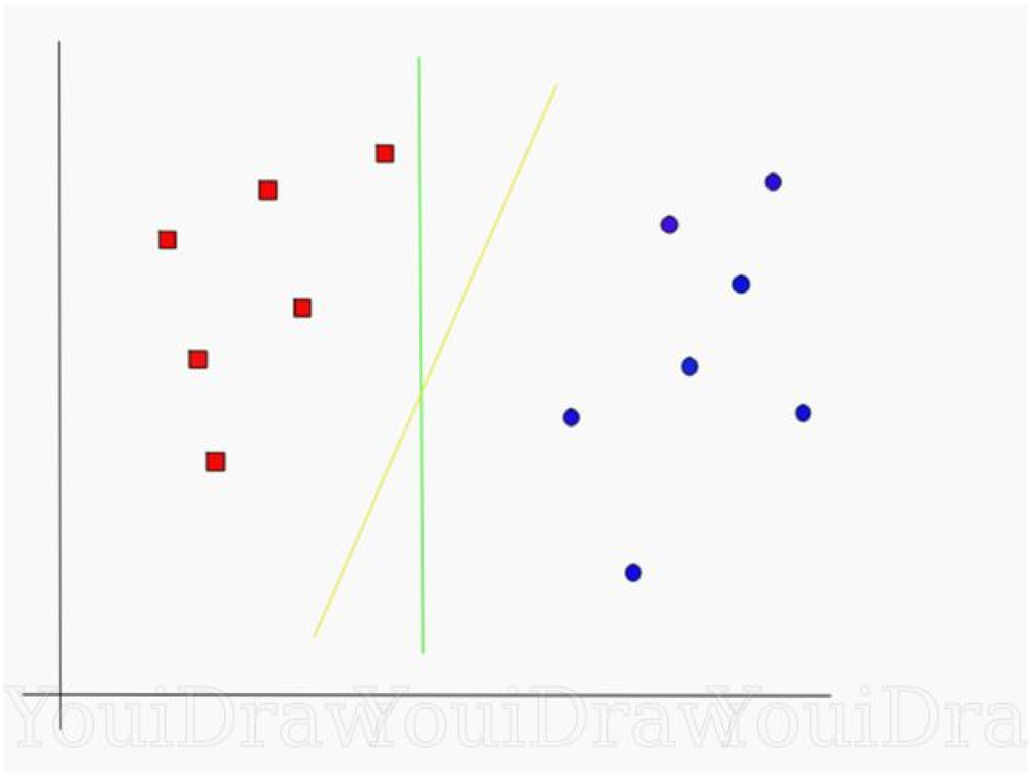
- At first approximation what SVMs do is to find a separating line(or hyperplane) between data of two classes. SVM is an algorithm that takes the data as an input and outputs a line that separates those classes if possible.

- Lets begin with a problem. Suppose you have a dataset as shown below and you need to classify the red rectangles from the blue ellipses(let's say positives from the negatives). So your task is to find an ideal line that separates this dataset in two classes (say red and blue).



Find an ideal line/ hyperplane that separates this dataset into red and blue categories

- Not a big task, right?
- But, as you notice there isn't a unique line that does the job. In fact, we have an infinite lines that can separate these two classes. So how does SVM find the ideal one??? Let's take some probable candidates and figure it out ourselves.

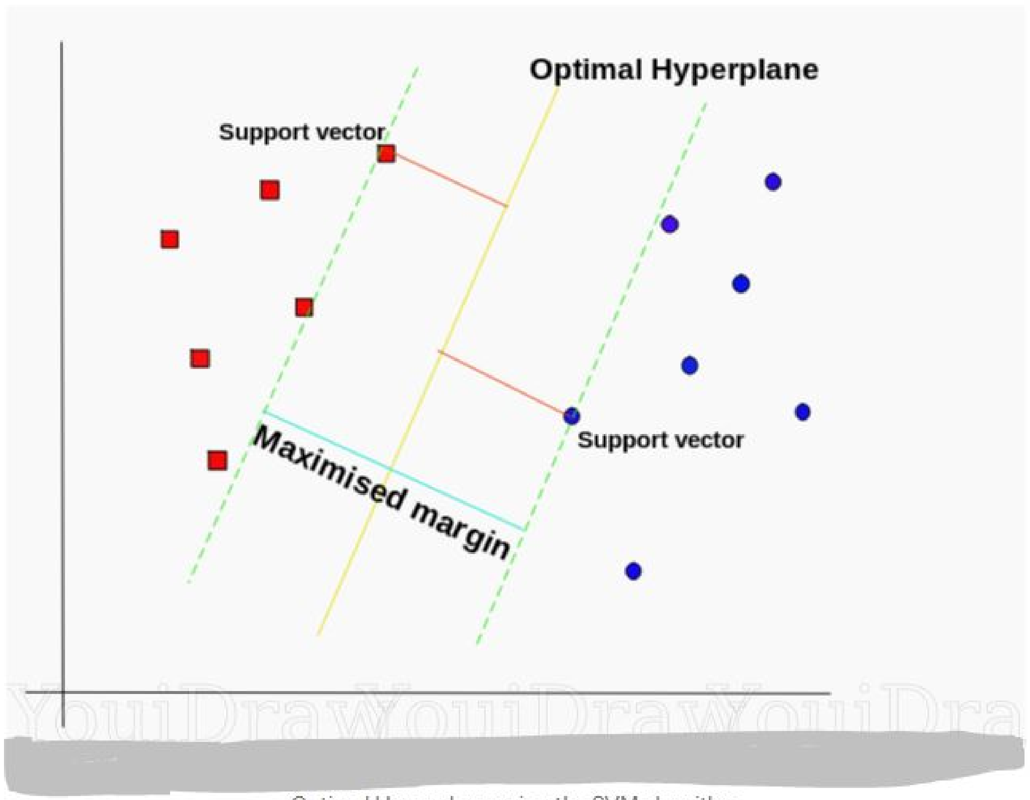


Which line according to you best separates the data???

- We have two candidates here, the green colored line and the yellow colored line. Which line according to you best separates the data?
- If you selected the yellow line then congrats, because thats the line we are looking for. It's visually quite intuitive in this case that the yellow line classifies better. But, we need something concrete to fix our line.
- The green line in the image above is quite close to the red class. Though it classifies the current datasets it is not a generalized line and in machine learning our goal is to get a more generalized separator.

SVM's way to find the best line

- According to the SVM algorithm we find the points closest to the line from both the classes.These points are called support vectors. Now, we compute the distance between the line and the support vectors. This distance is called the margin.
- Our goal is to maximize the margin. The hyperplane for which the margin is maximum is the optimal hyperplane.



Optimal Hyperplane using the SVM algorithm

Thus SVM tries to make a decision boundary in such a way that the separation between the two classes(that street) is as wide as possible.

2.4 4 What do you mean by kernel functions?

- Kernel is a way of computing the dot product of two vectors x and y in some (possibly very high dimensional) feature space, which is why kernel functions are sometimes called "generalized dot product".
- In machine learning, a "kernel" is usually used to refer to the kernel trick, a method of using a linear classifier to solve a non-linear problem. It entails transforming linearly inseparable data like (Fig. 3) to linearly separable ones (Fig. 2). The kernel function is what is applied on each data instance to map the original non-linear observations into a higher-dimensional space in which they become separable.

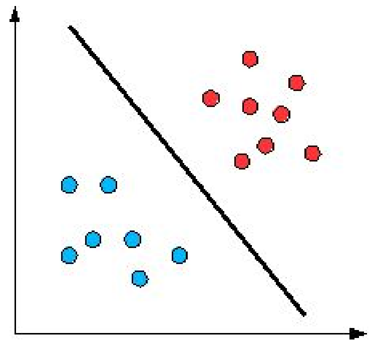


Fig. 2

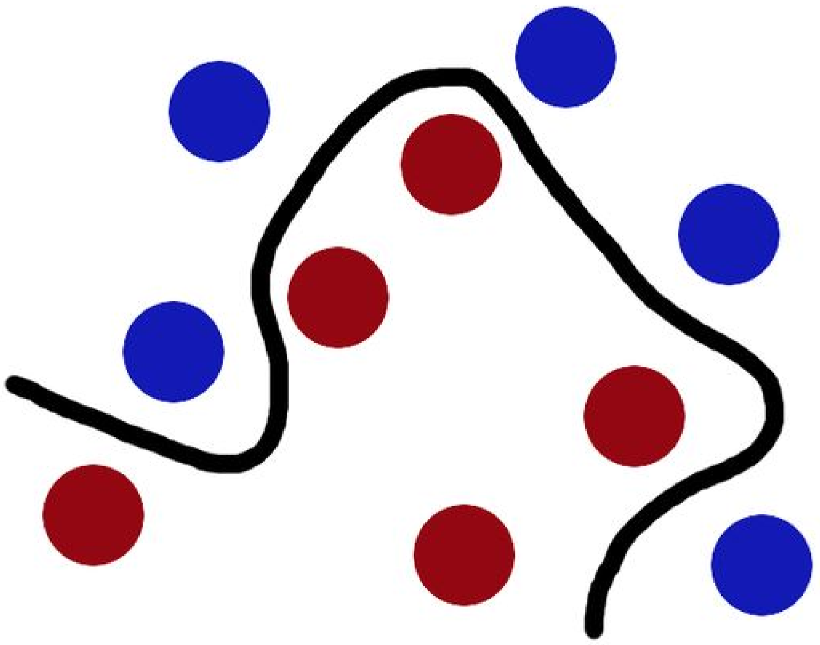
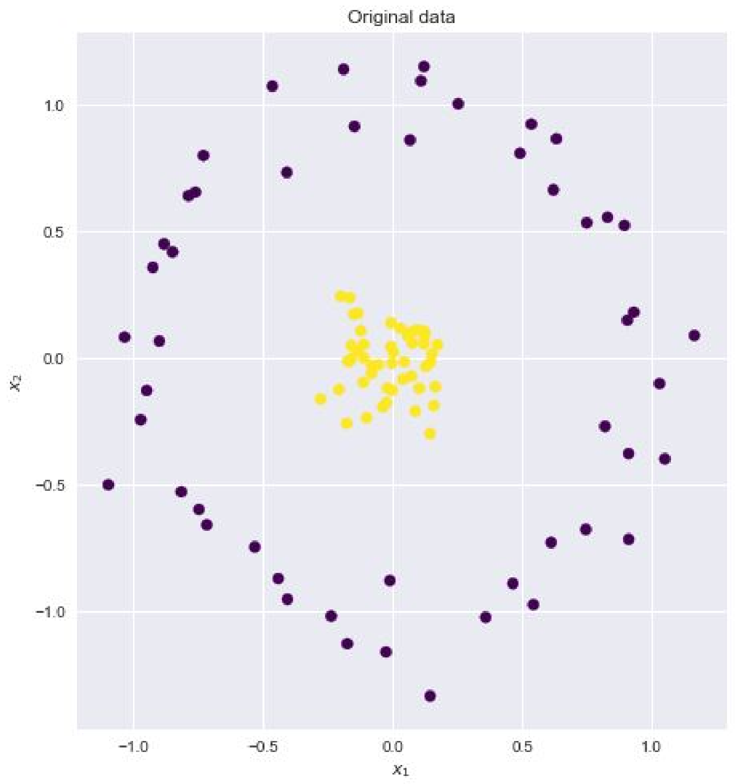


Fig. 3

Consider the following dataset where the yellow and blue points are clearly not linearly separable in two dimensions.



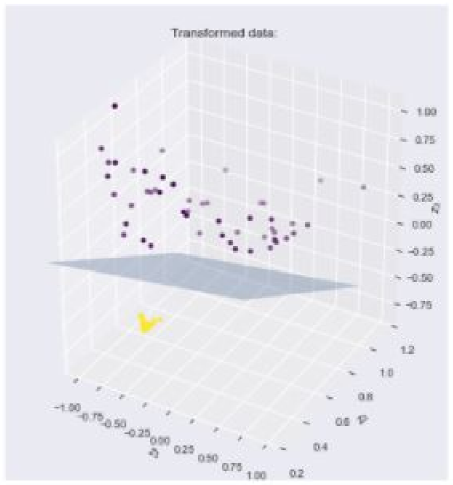
If we could find a higher dimensional space in which these points were linearly separable, then we could do the following:

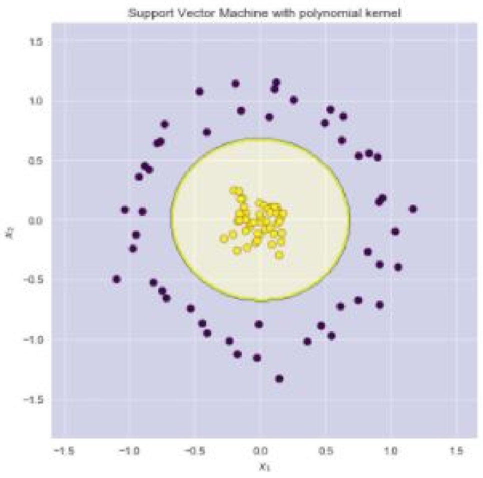
- Map the original features to the higher, transformer space (feature mapping)
- Perform linear SVM in this higher space
- Obtain a set of weights corresponding to the decision boundary hyperplane
- Map this hyperplane back into the original 2D space to obtain a non linear decision boundary

There are many higher dimensional spaces in which these points are linearly separable.

Visualizing the feature map and the resulting boundary line

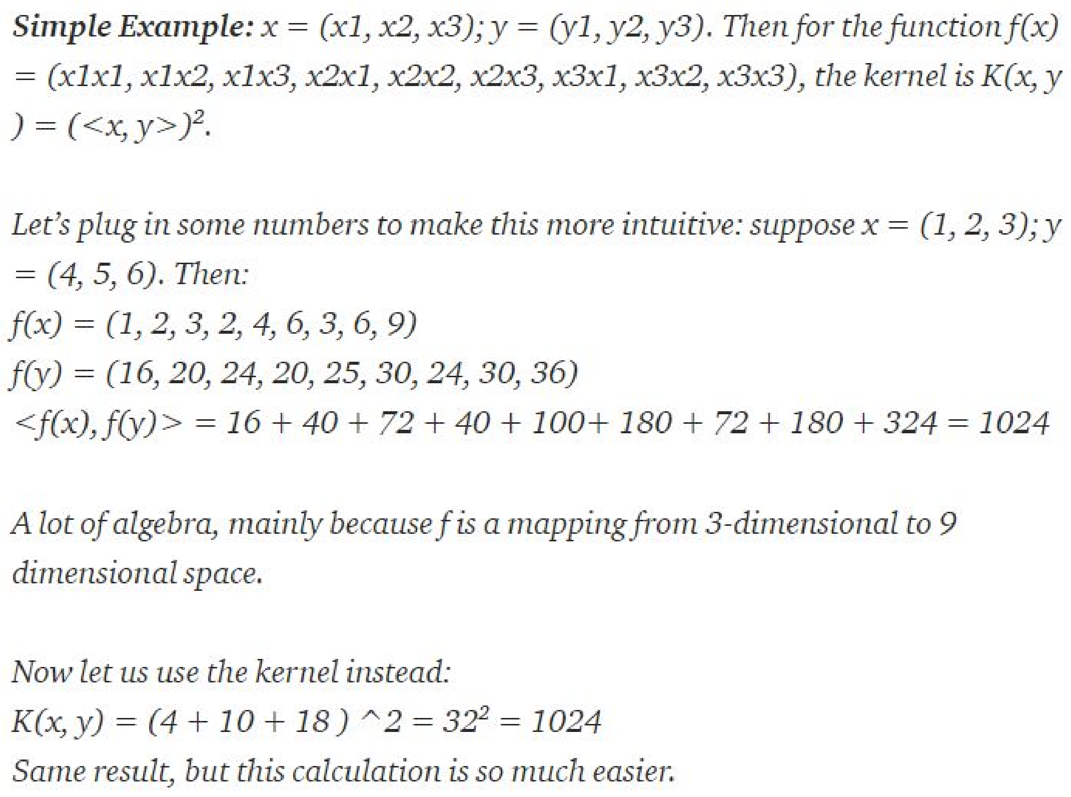
- Left-hand side plot shows the points plotted in the transformed space together with the SVM linear boundary hyperplane
- Right-hand side plot shows the result in the original 2-D space





2.5 5 Discuss how SVM makes use of kernel functions

- Briefly speaking, a **kernel is a shortcut that helps us do certain calculation faster which otherwise would involve computations in higher dimensional space.**
- **Mathematical definition:** K(x, y) = <f(x), f(y)>. Here K is the kernel function, x, y are n dimensional inputs. f is a map from n-dimension to m-dimension space. < x,y> denotes the dot product. usually m is much larger than n.
- **Intuition:** normally calculating <f(x), f(y)> requires us to calculate f(x), f(y) first, and then do the dot product. These two computation steps can be quite expensive as they involve manipulations in m dimensional space, where m can be a large number. But after all the trouble of going to the high dimensional space, the result of the dot product is really a scalar: we come back to one-dimensional space again! Now, the question we have is: do we really need to go through all the trouble to get this one number? do we really have to go to the m-dimensional space? The answer is no, if you find a clever kernel.



Simple Example: *x = (x1, x2, x3); y = (y1, y2, y3). Then for the function f(x) = (x1x1, x1x2, x1x3, x2x1, x2x2, x2x3, x3x1, x3x2, x3x3), the kernel is K(x, y) = (<x, y>)^2.*

Let's plug in some numbers to make this more intuitive: suppose x = (1, 2, 3); y = (4, 5, 6). Then:
f(x) = (1, 2, 3, 2, 4, 6, 3, 6, 9)
f(y) = (16, 20, 24, 20, 25, 30, 24, 30, 36)
<f(x), f(y)> = 16 + 40 + 72 + 40 + 100+ 180 + 72 + 180 + 324 = 1024

A lot of algebra, mainly because f is a mapping from 3-dimensional to 9 dimensional space.

Now let us use the kernel instead:
K(x, y) = (4 + 10 + 18) ^2 = 32^2 = 1024
Same result, but this calculation is so much easier.

- **Additional beauty of Kernel:** kernels allow us to do stuff in infinite dimensions! Sometimes going to higher dimension is not just computationally expensive, but also impossible. f(x) can be a mapping from n dimension to infinite dimension which we may have little idea of how to deal with. Then kernel gives us a wonderful shortcut.
- **Relation to SVM: now how is related to SVM?** The idea of SVM is that y = w phi(x) +b, where w is the weight, phi is the feature vector, and b is the bias. if y> 0, then we classify datum to class 1, else to class 0. We want to find a set of weight and bias such that the margin is maximized. Previous answers mention that kernel makes data linearly separable for SVM. I think a more precise way to put this is, kernels do not make the data linearly separable. The feature vector phi(x) makes the data linearly separable. Kernel is to make the calculation process faster and easier, especially when the feature vector phi is of very high dimension (for example, x1, x2, x3, ..., x_D^n, x1^2, x2^2, ..., x_D^2).
- **Why it can also be understood as a measure of similarity:** if we put the definition of kernel above, <f(x), f(y)>, in the context of SVM and feature vectors, it becomes <phi(x), phi(y)>. The inner product means the projection of phi(x) onto phi(y). or colloquially, how much overlap do x and y have in their feature space. In other words, how similar they are.

2.6 6 Discuss the following terms: Accuracy, Precision, Recall, F1 score, Specificity, Sensitivity, AUROC, PRAUC

Accuracy

- Classification Accuracy is what we usually mean, when we use the term accuracy. It is the ratio of number of correct predictions to the total number of input samples.

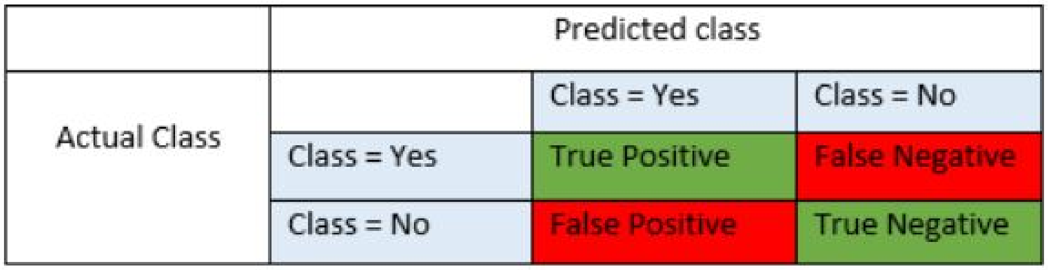
$$Accuracy = \frac{Number\ of\ Correct\ predictions}{Total\ number\ of\ predictions\ made}$$

Note:

- It works well only if there are equal number of samples belonging to each class.
- **For example,** consider that there are 98% samples of class A and 2% samples of class B in our training set. Then our model can easily get 98% training accuracy by simply predicting every training sample belonging to class A. When the same model is tested on a test set with 60% samples of class A and 40% samples of class B, then the test accuracy would drop down to 60%. Classification Accuracy is great, but gives us the false sense of achieving high accuracy.
- The real problem arises, when the cost of misclassification of the minor class samples are very high. If we deal with a rare but fatal disease, the cost of failing to diagnose the disease of a sick person is much higher than the cost of sending a healthy person to more tests.
- Hence, accuracy is not the be-all and end-all model metric to use when selecting the best model

Confusion Matrix:

- A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known.



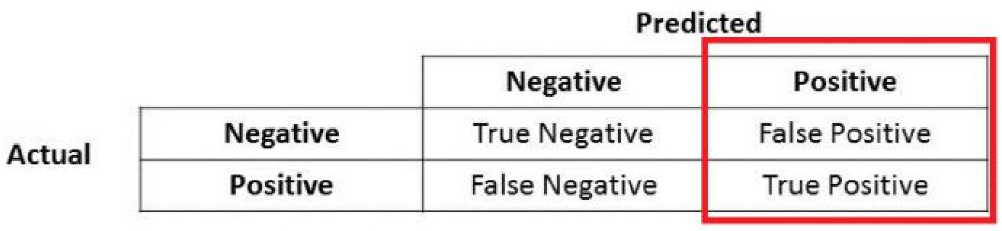
	Predicted class		
Actual Class		Class = Yes	Class = No
	Class = Yes	True Positive	False Negative
	Class = No	False Positive	True Negative

There are 4 important terms :

- **True Positives** : The cases in which we predicted YES and the actual output was also YES.
- **True Negatives** : The cases in which we predicted NO and the actual output was NO.
- **False Positives** : The cases in which we predicted YES and the actual output was NO.
- **False Negatives** : The cases in which we predicted NO and the actual output was YES.

Precision:

- Precision talks about how precise/accurate your model is out of those predicted positive, how many of them are actual positive.
- 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.



		Predicted	
		Negative	Positive
Actual	Negative	True Negative	False Positive
	Positive	False Negative	True Positive

True Positive + False Positive = Total Predicted Positive

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

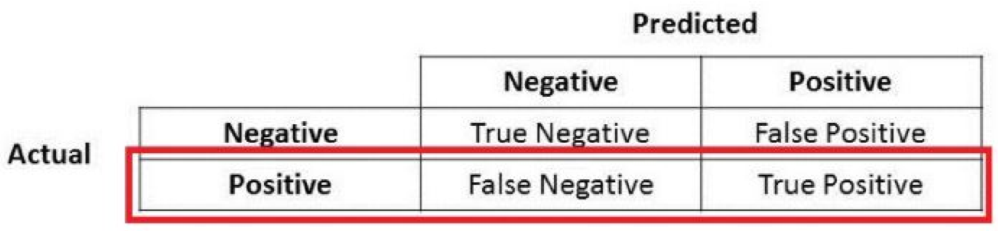
$$= \frac{True\ Positive}{Total\ Predicted\ Positive}$$

Recall:

- Recall actually calculates how many of the Actual Positives our model capture through labeling it as Positive (True Positive). we know that 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.
- Similarly, in sick patient detection. If a sick patient (Actual Positive) goes through the test and predicted as not sick (Predicted Negative). The cost associated with False Negative will be extremely high if the sickness is contagious.

$$\text{Recall} = \frac{\textit{True Positive}}{\textit{True Positive}+\textit{False Negative}}$$

$$= \frac{\textit{True Positive}}{\textit{Total Actual Positive}}$$



		Predicted	
		Negative	**Positive**
Actual	**Negative**	True Negative	False Positive
	Positive	False Negative	True Positive

True Positive + False Negative = Actual Positive

F1 score:

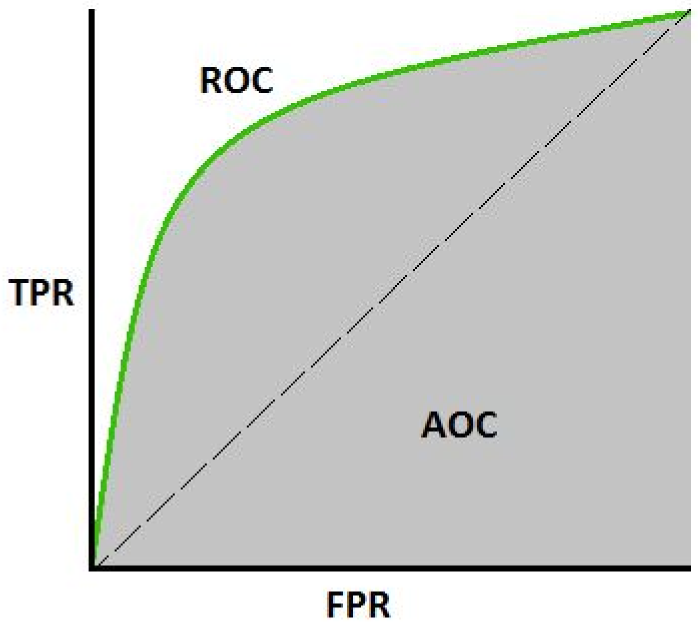
- F1 Score is needed when you want to seek a balance between Precision and Recall.
- Right…so what is the difference between F1 Score and Accuracy then? We have previously seen that accuracy can be largely contributed by a large number of True Negatives which in most business circumstances, we do not focus on much whereas False Negative and False Positive usually has business costs (tangible & intangible) thus **F1 Score might be a better measure to use if we need to seek a balance between Precision and Recall AND there is an uneven class distribution (large number of Actual Negatives).**
- F1 Score is used to **measure a test's accuracy**
- F1 Score is the **Harmonic Mean between precision and recall. The range for F1 Score is [0, 1].** It tells you how precise your classifier is (how many instances it classifies correctly), as well as how robust it is (it does not miss a significant number of instances).
- High precision but lower recall, gives you an extremely accurate, but it then misses a large number of instances that are difficult to classify. **The greater the F1 Score, the better is the performance of our model.** Mathematically, it can be expressed as :

$$\text{F1} = 2 \times \frac{\textit{Precision} * \textit{Recall}}{\textit{Precision}+\textit{Recall}}$$

AUROC:

- AUC - ROC curve is a performance measurement for the classification problems at various threshold settings.

- **ROC is a probability curve and AUC represents the degree or measure of separability.**
- It tells how much the model is capable of distinguishing between classes.
- Higher the AUC, the better the model is at predicting 0 classes as 0 and 1 classes as 1. By analogy, the Higher the AUC, the better the model is at distinguishing between patients with the disease and no disease.
- The ROC curve is plotted with TPR against the FPR where TPR is on the y-axis and FPR is on the x-axis.



True Positive Rate TPR(Sensitivity) :

- True Positive Rate is defined as TP/ (FN+TP).
- True Positive Rate corresponds to the proportion of positive data points that are correctly considered as positive, with respect to all positive data points.

$$TruePositiveRate = \frac{TruePositive}{FalseNegative + TruePositive}$$

True Negative Rate (Specificity) :

- True Negative Rate is defined as TN / (FP+TN).
- False Positive Rate corresponds to the proportion of negative data points that are correctly considered as negative, with respect to all negative data points.

$$TrueNegativeRate = \frac{TrueNegative}{TrueNegative + FalsePositive}$$

False Positive Rate :

- False Positive Rate is defined as FP / (FP+TN).
- False Positive Rate corresponds to the proportion of negative data points that are mistakenly considered as positive, with respect to all negative data points.

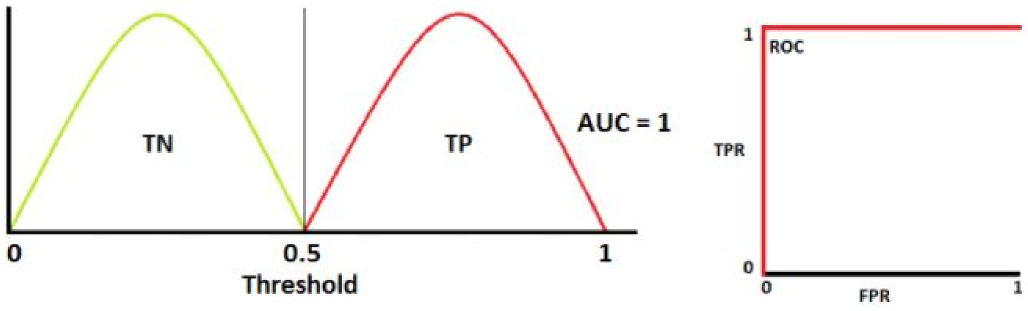
$$FalsePositiveRate = \frac{FalsePositive}{TrueNegative + FalsePositive}$$

Note:

- False Positive Rate and True Positive Rate both have values in the range [0, 1].
- FPR and TPR both are computed at varying threshold values such as (0.00, 0.02, 0.04, ..., 1.00) and a graph is drawn.
- AUC is the area under the curve of plot False Positive Rate vs True Positive Rate at different points in [0, 1].

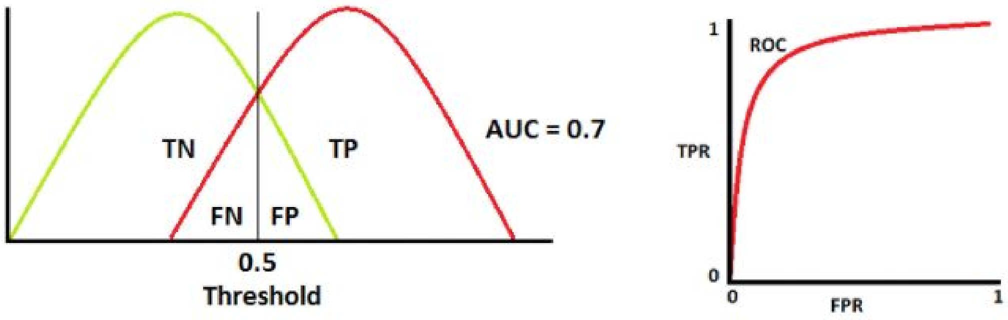
How to speculate about the performance of the model?

- An excellent model has AUC near to the 1 which means it has a good measure of separability. A poor model has an AUC near 0 which means it has the worst measure of separability. In fact, it means it is reciprocating the result. It is predicting 0s as 1s and 1s as 0s. And when AUC is 0.5, it means the model has no class separation capacity whatsoever.
- Let's interpret the above statements.As we know, ROC is a curve of probability. So let's plot the distributions of those probabilities:
- Note: Red distribution curve is of the positive class (patients with disease) and the green distribution curve is of the negative class(patients with no disease).



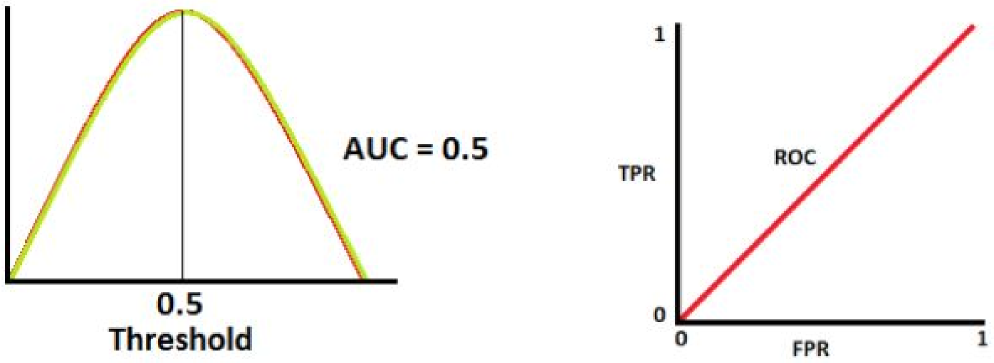
Observation:

- This is an ideal situation. When two curves don't overlap at all means model has an ideal measure of separability. It is perfectly able to distinguish between positive class and negative class.



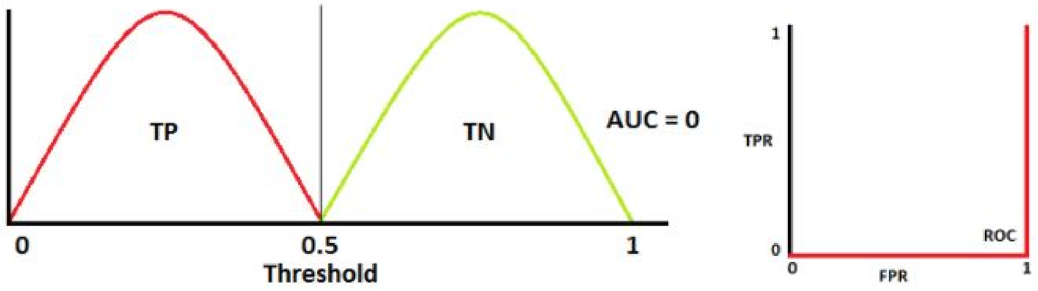
Observation:

- When two distributions overlap, we introduce type 1 and type 2 errors. Depending upon the threshold, we can minimize or maximize them. When AUC is 0.7, it means there is a 70% chance that the model will be able to distinguish between positive class and negative class.



Observation:

- This is the worst situation. When AUC is approximately 0.5, the model has no discrimination capacity to distinguish between positive class and negative class.

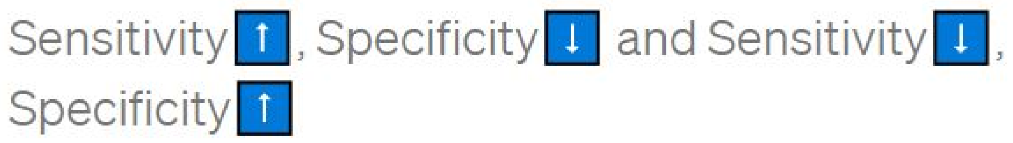


Observation:

- When AUC is approximately 0, the model is actually reciprocating the classes. It means the model is predicting a negative class as a positive class and vice versa.

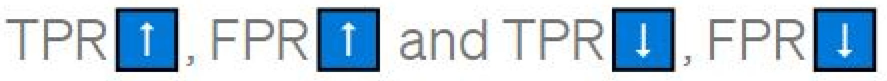
The relation between Sensitivity, Specificity, FPR, and Threshold.

- Sensitivity and Specificity are inversely proportional to each other. So when we increase Sensitivity, Specificity decreases, and vice versa.



Sensitivity↑, Specificity↓ and Sensitivity↓, Specificity↑

- When we decrease the threshold, we get more positive values thus it increases the sensitivity and decreasing the specificity.
- Similarly, when we increase the threshold, we get more negative values thus we get higher specificity and lower sensitivity.
- As we know FPR is 1 - specificity. So when we increase TPR, FPR also increases and vice versa.



TPR↑, FPR↑ and TPR↓, FPR↓

How to use the AUC ROC curve for the multi-class model?

- In a multi-class model, we can plot the N number of AUC ROC Curves for N number classes using the One vs ALL methodology.
- So for example, If you have three classes named X, Y, and Z, you will have one ROC for X classified against Y and Z, another ROC for Y classified against X and Z, and the third one of Z classified against Y and X.

PR AUC | Average Precision:

- Similarly to ROC AUC in order to define PR AUC we need to define what Precision-Recall curve.
- It is a curve that combines precision (PPV) and Recall (TPR) in a single visualization. For every threshold, you calculate PPV and TPR and plot it. The higher on y-axis your curve is the better your model performance.
- You can use this plot to make an educated decision when it comes to the classic precision/recall dilemma. Obviously, the higher the recall the lower the precision. Knowing at which recall your precision starts to fall fast can help you choose the threshold and deliver a better model.

- You can also think of PR AUC as the average of precision scores calculated for each recall threshold.

ROC AUC vs PR AUC:

- What is common between ROC AUC and PR AUC is that they both look at prediction scores of classification models and not thresholded class assignments. What is different however is that ROC AUC looks at a true positive rate TPR and false positive rate FPR while PR AUC looks at positive predictive value PPV and true positive rate TPR.
- Because of that if you care more about the positive class, then using PR AUC, which is more sensitive to the improvements for the positive class, is a better choice. One common scenario is a highly imbalanced dataset where the fraction of positive class, which we want to find (like in fraud detection), is small. I highly recommend taking a look at this kaggle kernel for a longer discussion on the subject of ROC AUC vs PR AUC for imbalanced datasets.
- If you care equally about the positive and negative class or your dataset is quite balanced, then going with ROC AUC is a good idea.

2.7 7 perform classification on zoo dataset from kaggle using logistic regression after performing appropriate data pre processing and hyper parameter tuning and evaluate using the technique you feel is fit for this task and give your comments.

[60]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, KFold, cross_validate
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪accuracy_score
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
```

[5]:
```
#save the csv in a dataframe
zoo_df = pd.read_csv("D:/Masters/SJSU/Academics/sem_2/CMPE_257_ML/
 ↪CMPE_257_ML_git/Assignments/HW4/Data/zoo.csv")
```

[6]:
```
zoo_df.head()
```

[6]:
```
  animal_name  hair  feathers  eggs  milk  airborne  aquatic  predator  \
0    aardvark     1         0     0     1         0        0         1
1    antelope     1         0     0     1         0        0         0
2        bass     0         0     1     0         0        1         1
3        bear     1         0     0     1         0        0         1
4        boar     1         0     0     1         0        0         1

   toothed  backbone  breathes  venomous  fins  legs  tail  domestic  catsize  \
0        1         1         1         0      0      4      0         0        1
1        1         1         1         0      0      4      1         0        1
2        1         1         0         0      1      0      1         0        0
3        1         1         1         0      0      4      0         0        1
4        1         1         1         0      0      4      1         0        1

   class_type
0           1
1           1
2           4
3           1
4           1
```

[7]:
```
# shape of the entire dataset
zoo_df.shape
```

[7]:
```
(101, 18)
```

[8]:
```
zoo_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 101 entries, 0 to 100
Data columns (total 18 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   animal_name  101 non-null    object
 1   hair         101 non-null    int64
 2   feathers     101 non-null    int64
 3   eggs         101 non-null    int64
 4   milk         101 non-null    int64
 5   airborne     101 non-null    int64
 6   aquatic      101 non-null    int64
 7   predator     101 non-null    int64
 8   toothed      101 non-null    int64
 9   backbone     101 non-null    int64
 10  breathes     101 non-null    int64
 11  venomous     101 non-null    int64
 12  fins         101 non-null    int64
 13  legs         101 non-null    int64
 14  tail         101 non-null    int64
 15  domestic     101 non-null    int64
 16  catsize      101 non-null    int64
 17  class_type   101 non-null    int64
dtypes: int64(17), object(1)
memory usage: 14.3+ KB
```

Observation:

- There are no Null values

```
[9]: # summary of the dataset

     zoo_df.describe()
```

```
[9]:              hair    feathers        eggs        milk    airborne     aquatic  \
     count  101.000000  101.000000  101.000000  101.000000  101.000000  101.000000
     mean     0.425743    0.198020    0.584158    0.405941    0.237624    0.356436
     std      0.496921    0.400495    0.495325    0.493522    0.427750    0.481335
     min      0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
     25%      0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
     50%      0.000000    0.000000    1.000000    0.000000    0.000000    0.000000
     75%      1.000000    0.000000    1.000000    1.000000    0.000000    1.000000
     max      1.000000    1.000000    1.000000    1.000000    1.000000    1.000000

              predator     toothed    backbone    breathes    venomous        fins  \
     count  101.000000  101.000000  101.000000  101.000000  101.000000  101.000000
     mean     0.554455    0.603960    0.821782    0.792079    0.079208    0.168317
     std      0.499505    0.491512    0.384605    0.407844    0.271410    0.376013
     min      0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
     25%      0.000000    0.000000    1.000000    1.000000    0.000000    0.000000
     50%      1.000000    1.000000    1.000000    1.000000    0.000000    0.000000
     75%      1.000000    1.000000    1.000000    1.000000    0.000000    0.000000
     max      1.000000    1.000000    1.000000    1.000000    1.000000    1.000000

                  legs        tail    domestic     catsize  class_type
     count  101.000000  101.000000  101.000000  101.000000  101.000000
     mean     2.841584    0.742574    0.128713    0.435644    2.831683
     std      2.033385    0.439397    0.336552    0.498314    2.102709
     min      0.000000    0.000000    0.000000    0.000000    1.000000
     25%      2.000000    0.000000    0.000000    0.000000    1.000000
     50%      4.000000    1.000000    0.000000    0.000000    2.000000
     75%      4.000000    1.000000    0.000000    1.000000    4.000000
     max      8.000000    1.000000    1.000000    1.000000    7.000000
```

```
[10]: # check the target feature

      zoo_df['class_type'].value_counts()
```

```
[10]: 1    41
      2    20
      4    13
      7    10
      6     8
      3     5
      5     4
      Name: class_type, dtype: int64
```

[11]:
```
# plot the target variable distribution

x = zoo_df['class_type'].value_counts().index.tolist()
y = zoo_df['class_type'].value_counts().tolist()

fig = px.bar(x=x, y=y, color=x, title="Animal Class Type Distribution",
             labels={
                 'x': 'Animal Class',
                 'y': 'count'
                 },)
fig.show()
```

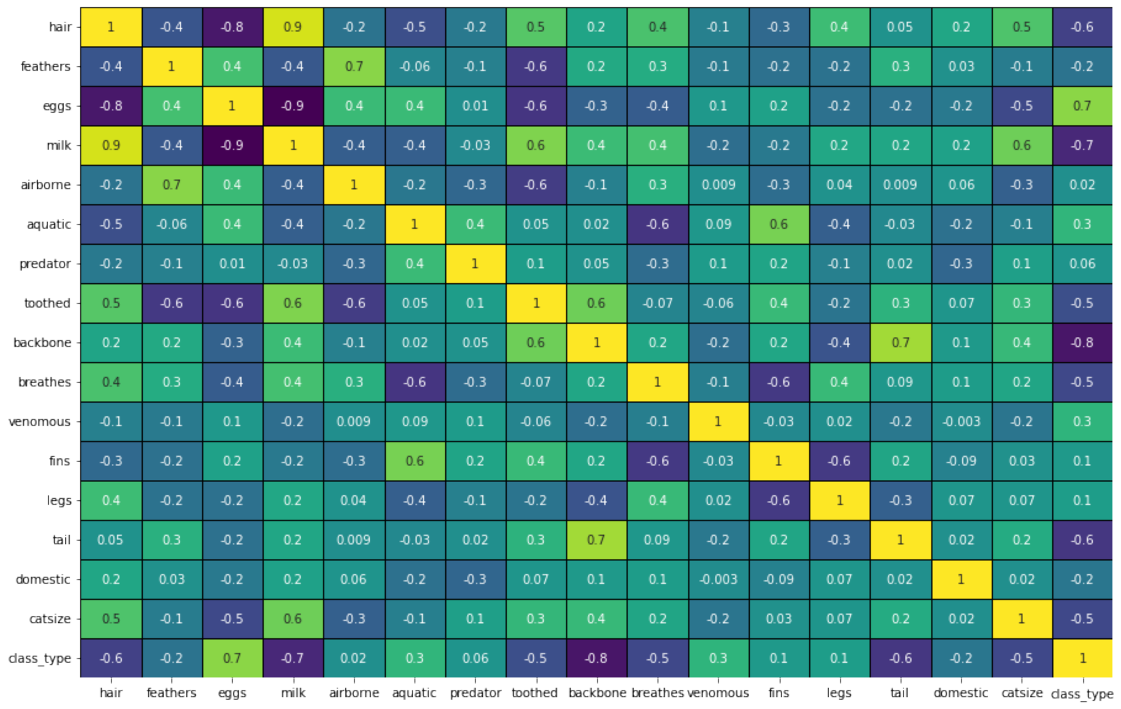
```
<IPython.core.display.Javascript object>
```

[12]:
```
zoo_df['animal_name'].value_counts()
```

[12]:
```
frog        2
pony        1
sealion     1
seal        1
seahorse    1
           ..
gorilla     1
goat        1
gnat        1
girl        1
wren        1
Name: animal_name, Length: 100, dtype: int64
```

[13]:
```
# correlation heatmap

fig, ax = plt.subplots(figsize=(15,10))
sns.heatmap(zoo_df.corr(), annot=True, fmt='.1g', cmap="viridis", cbar=False,␣
 ↪linewidths=0.5, linecolor='black');
```



Observation:

- Dropping **hair** feature since it's high correlated with milk feature
- Dropping **milk** feature since it's highly correlated with eggs
- Dropping **animal_name** as it is insignificant

[14]:
```
# drop the above columns

zoo_df.drop(['animal_name','hair', 'milk'], axis=1, inplace=True)
```

[15]:
```
# drop the target feature from the input data

X = zoo_df.drop("class_type", axis=1)
X.head()
```

[15]:
```
   feathers  eggs  airborne  aquatic  predator  toothed  backbone  breathes  \
0         0     0         0        0         1        1         1         1
1         0     0         0        0         0        1         1         1
2         0     1         0        1         1        1         1         0
3         0     0         0        0         1        1         1         1
4         0     0         0        0         1        1         1         1

   venomous  fins  legs  tail  domestic  catsize
0         0     0     4     0         0        1
```

```
1          0      0      4      1          0         1
2          0      1      0      1          0         0
3          0      0      4      0          0         1
4          0      0      4      1          0         1
```

[16]:
```
# saving the target feature in y variable

y = zoo_df["class_type"]
y.head()
```

[16]:
```
0    1
1    1
2    4
3    1
4    1
Name: class_type, dtype: int64
```

Splitting the data into training and test datasets

Here, we are trying to predict the class type of the animal using the given data. Hence, the class_type will be the y label and rest of the data will be the X or the input data

[18]:
```
# split the dataset into train test

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4,␣
 ↪random_state = 0)

print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(60, 14)
(60,)
(41, 14)
(41,)
```

[19]:
```
# Train the logsitic regression model

LRmodel = LogisticRegression(max_iter = 1500)
LRmodel.fit(X_train, y_train)
```

[19]:
```
LogisticRegression(max_iter=1500)
```

[20]:
```
# Predicting for the training set
y_train_pred = LRmodel.predict(X_train)
```

[21]:
```
# Checking the evaluation metrics of the train set
```

```
print(classification_report(y_train, y_train_pred))
```

```
              precision    recall  f1-score   support

           1       0.96      1.00      0.98        25
           2       1.00      1.00      1.00        13
           3       0.00      0.00      0.00         1
           4       1.00      1.00      1.00         5
           5       1.00      1.00      1.00         3
           6       1.00      1.00      1.00         5
           7       1.00      1.00      1.00         8

    accuracy                           0.98        60
   macro avg       0.85      0.86      0.85        60
weighted avg       0.97      0.98      0.98        60

D:\IDEs\Anaconda\lib\site-packages\sklearn\metrics\_classification.py:1318:
UndefinedMetricWarning:

Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.

D:\IDEs\Anaconda\lib\site-packages\sklearn\metrics\_classification.py:1318:
UndefinedMetricWarning:

Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.

D:\IDEs\Anaconda\lib\site-packages\sklearn\metrics\_classification.py:1318:
UndefinedMetricWarning:

Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
```

```
[22]: #calculating the confusion matrix for train set

      confusion_matrix(y_train, y_train_pred)
```

```
[22]: array([[25,  0,  0,  0,  0,  0,  0],
             [ 0, 13,  0,  0,  0,  0,  0],
             [ 1,  0,  0,  0,  0,  0,  0],
             [ 0,  0,  0,  5,  0,  0,  0],
             [ 0,  0,  0,  0,  3,  0,  0],
             [ 0,  0,  0,  0,  0,  5,  0],
             [ 0,  0,  0,  0,  0,  0,  8]], dtype=int64)
```

[23]:
```
# Predicting for the test set

y_test_pred = LRmodel.predict(X_test)
```

[24]:
```
# Checking the evaluation metrics of the test set

print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.88      0.94      0.91        16
           2       0.88      1.00      0.93         7
           3       0.00      0.00      0.00         4
           4       0.80      1.00      0.89         8
           5       1.00      1.00      1.00         1
           6       0.75      1.00      0.86         3
           7       1.00      0.50      0.67         2

    accuracy                           0.85        41
   macro avg       0.76      0.78      0.75        41
weighted avg       0.78      0.85      0.81        41

D:\IDEs\Anaconda\lib\site-packages\sklearn\metrics\_classification.py:1318:
UndefinedMetricWarning:

Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.

D:\IDEs\Anaconda\lib\site-packages\sklearn\metrics\_classification.py:1318:
UndefinedMetricWarning:

Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.

D:\IDEs\Anaconda\lib\site-packages\sklearn\metrics\_classification.py:1318:
UndefinedMetricWarning:

Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
```

[25]:
```
#calculating the confusion matrix for test set

confusion_matrix(y_test, y_test_pred)
```

[25]:
```
array([[15,  1,  0,  0,  0,  0,  0],
       [ 0,  7,  0,  0,  0,  0,  0],
       [ 2,  0,  0,  2,  0,  0,  0],
       [ 0,  0,  0,  8,  0,  0,  0],
       [ 0,  0,  0,  0,  1,  0,  0],
       [ 0,  0,  0,  0,  0,  3,  0],
       [ 0,  0,  0,  0,  0,  1,  1]], dtype=int64)
```

[26]:
```
# Now instead of train and test set, using cross validation

# since sv is 4 it'll give 4 different models

# CV is used to check homogeneity, overfitting or high variance or to tune␣
↪hyperparameters

model = LogisticRegression(max_iter = 1000)
scores = cross_validate(model, X, y, cv = 4, scoring = 'accuracy',␣
↪return_train_score = True)
scores
```

[26]:
```
{'fit_time': array([0.0654614 , 0.01991487, 0.01564956, 0.03124118]),
 'score_time': array([0.0010767, 0.       , 0.       , 0.       ]),
 'test_score': array([0.96153846, 1.        , 0.84      , 0.88      ]),
 'train_score': array([0.97333333, 0.97368421, 0.98684211, 0.98684211])}
```

Observation:

- The accuracy of the **2nd model has the highest accuracy** on both train and test data
- **3rd model is performing worst.** We can infer that during that split the train and test data are not equally distributed

[27]:
```
#Method 2 - performing cross validation using Kfold with code

# Kfold gives the indices of training and testing dataset.

# This is similar to cross_validate()

acc_score = []

cv = KFold(n_splits = 4, random_state = 100, shuffle = True)

#loop runds for 4 times since n_splits = 4
for train_index , test_index in cv.split(X):

    X_train , X_test = X.iloc[train_index,:], X.iloc[test_index,:]
    y_train , y_test = y[train_index] , y[test_index]

    model = LogisticRegression(max_iter = 1000)
    model.fit(X_train,y_train)
    pred_values = model.predict(X_test)
    acc = accuracy_score(y_test, pred_values)
    acc_score.append(acc)

avg_acc_score = sum(acc_score)/4

print('accuracy of each fold - {}'.format(acc_score))
print('Avg accuracy : {}'.format(avg_acc_score))
```

```
accuracy of each fold - [0.9615384615384616, 0.92, 0.88, 0.92]
Avg accuracy : 0.9203846153846154
```

Observation:

- The accuracy for the 3rd fold is **88%** so we can infer that the train and test split were not equally distributed
- The avg accuracy is **92%** which is quite good

[ ]:

```

```

2.8 8 perform classification on MNIST dataset using Linear SVM and then using Kernels and compare the methods

[29]:

```
train_data = pd.read_csv("D:/Masters/SJSU/Academics/sem_2/CMPE_257_ML/
 ↪CMPE_257_ML_git/Assignments/HW4/Data/train.csv") #reading the csv files␣
 ↪using pandas
test_data = pd.read_csv("D:/Masters/SJSU/Academics/sem_2/CMPE_257_ML/
 ↪CMPE_257_ML_git/Assignments/HW4/Data/test.csv")
```

[30]:

```
# print the dimension or shape of train data

train_data.shape
```

[30]:

```
(42000, 785)
```

[31]:

```
# print the dimension or shape of test data

test_data.shape
```

[31]:

```
(28000, 784)
```

[32]:

```
# printing first five columms of train_data

train_data.head()
```

[32]:

```
   label  pixel0  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  \
0      1       0       0       0       0       0       0       0       0
1      0       0       0       0       0       0       0       0       0
```

```
2      1       0       0       0       0       0       0       0       0
3      4       0       0       0       0       0       0       0       0
4      0       0       0       0       0       0       0       0       0

   pixel8  …  pixel774  pixel775  pixel776  pixel777  pixel778  pixel779  \
0       0  …         0         0         0         0         0         0
1       0  …         0         0         0         0         0         0
2       0  …         0         0         0         0         0         0
3       0  …         0         0         0         0         0         0
4       0  …         0         0         0         0         0         0

   pixel780  pixel781  pixel782  pixel783
0         0         0         0         0
1         0         0         0         0
2         0         0         0         0
3         0         0         0         0
4         0         0         0         0

[5 rows x 785 columns]
```

[33]:
```
# printing first five columns of test_data

test_data.head()
```

[33]:
```
   pixel0  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  \
0       0       0       0       0       0       0       0       0       0
1       0       0       0       0       0       0       0       0       0
2       0       0       0       0       0       0       0       0       0
3       0       0       0       0       0       0       0       0       0
4       0       0       0       0       0       0       0       0       0

   pixel9  …  pixel774  pixel775  pixel776  pixel777  pixel778  pixel779  \
0       0  …         0         0         0         0         0         0
1       0  …         0         0         0         0         0         0
2       0  …         0         0         0         0         0         0
3       0  …         0         0         0         0         0         0
4       0  …         0         0         0         0         0         0

   pixel780  pixel781  pixel782  pixel783
0         0         0         0         0
1         0         0         0         0
2         0         0         0         0
3         0         0         0         0
4         0         0         0         0

[5 rows x 784 columns]
```

[34]:
```
train_data.isnull().sum()
```

[34]:
```
label       0
pixel0      0
pixel1      0
pixel2      0
pixel3      0
           ..
pixel779    0
pixel780    0
pixel781    0
pixel782    0
pixel783    0
Length: 785, dtype: int64
```

Observation:

- There are no missing values in train dataset

[35]:
```
test_data.isnull().sum()
```

[35]:
```
pixel0      0
pixel1      0
pixel2      0
pixel3      0
pixel4      0
           ..
pixel779    0
pixel780    0
pixel781    0
pixel782    0
pixel783    0
Length: 784, dtype: int64
```

Observation:

- There are no missing values in test dataset

[36]:
```
train_data.describe()
```

[36]:
```
              label   pixel0   pixel1   pixel2   pixel3   pixel4   pixel5  \
count  42000.000000  42000.0  42000.0  42000.0  42000.0  42000.0  42000.0
mean       4.456643      0.0      0.0      0.0      0.0      0.0      0.0
std        2.887730      0.0      0.0      0.0      0.0      0.0      0.0
min        0.000000      0.0      0.0      0.0      0.0      0.0      0.0
25%        2.000000      0.0      0.0      0.0      0.0      0.0      0.0
50%        4.000000      0.0      0.0      0.0      0.0      0.0      0.0
75%        7.000000      0.0      0.0      0.0      0.0      0.0      0.0
max        9.000000      0.0      0.0      0.0      0.0      0.0      0.0
```

```
       pixel6   pixel7   pixel8  …      pixel774      pixel775  \
count  42000.0  42000.0  42000.0  …  42000.000000  42000.000000
mean       0.0      0.0      0.0  …      0.219286      0.117095
std        0.0      0.0      0.0  …      6.312890      4.633819
min        0.0      0.0      0.0  …      0.000000      0.000000
25%        0.0      0.0      0.0  …      0.000000      0.000000
50%        0.0      0.0      0.0  …      0.000000      0.000000
75%        0.0      0.0      0.0  …      0.000000      0.000000
max        0.0      0.0      0.0  …    254.000000    254.000000

           pixel776     pixel777      pixel778      pixel779  pixel780  \
count  42000.000000  42000.00000  42000.000000  42000.000000   42000.0
mean       0.059024      0.02019      0.017238      0.002857       0.0
std        3.274488      1.75987      1.894498      0.414264       0.0
min        0.000000      0.00000      0.000000      0.000000       0.0
25%        0.000000      0.00000      0.000000      0.000000       0.0
50%        0.000000      0.00000      0.000000      0.000000       0.0
75%        0.000000      0.00000      0.000000      0.000000       0.0
max      253.000000    253.00000    254.000000     62.000000       0.0

       pixel781  pixel782  pixel783
count   42000.0   42000.0   42000.0
mean        0.0       0.0       0.0
std         0.0       0.0       0.0
min         0.0       0.0       0.0
25%         0.0       0.0       0.0
50%         0.0       0.0       0.0
75%         0.0       0.0       0.0
max         0.0       0.0       0.0

[8 rows x 785 columns]
```

[38]:
```
test_data.describe()
```

[38]:
```
       pixel0   pixel1   pixel2   pixel3   pixel4   pixel5   pixel6   pixel7  \
count  28000.0  28000.0  28000.0  28000.0  28000.0  28000.0  28000.0  28000.0
mean       0.0      0.0      0.0      0.0      0.0      0.0      0.0      0.0
std        0.0      0.0      0.0      0.0      0.0      0.0      0.0      0.0
min        0.0      0.0      0.0      0.0      0.0      0.0      0.0      0.0
25%        0.0      0.0      0.0      0.0      0.0      0.0      0.0      0.0
50%        0.0      0.0      0.0      0.0      0.0      0.0      0.0      0.0
75%        0.0      0.0      0.0      0.0      0.0      0.0      0.0      0.0
max        0.0      0.0      0.0      0.0      0.0      0.0      0.0      0.0

       pixel8   pixel9  …      pixel774      pixel775      pixel776  \
count  28000.0  28000.0  …  28000.000000  28000.000000  28000.000000
```

```
mean       0.0      0.0  ...     0.164607     0.073214     0.028036
std        0.0      0.0  ...     5.473293     3.616811     1.813602
min        0.0      0.0  ...     0.000000     0.000000     0.000000
25%        0.0      0.0  ...     0.000000     0.000000     0.000000
50%        0.0      0.0  ...     0.000000     0.000000     0.000000
75%        0.0      0.0  ...     0.000000     0.000000     0.000000
max        0.0      0.0  ...   253.000000   254.000000   193.000000

           pixel777      pixel778  pixel779  pixel780  pixel781  pixel782  \
count  28000.000000  28000.000000   28000.0   28000.0   28000.0   28000.0
mean       0.011250      0.006536       0.0       0.0       0.0       0.0
std        1.205211      0.807475       0.0       0.0       0.0       0.0
min        0.000000      0.000000       0.0       0.0       0.0       0.0
25%        0.000000      0.000000       0.0       0.0       0.0       0.0
50%        0.000000      0.000000       0.0       0.0       0.0       0.0
75%        0.000000      0.000000       0.0       0.0       0.0       0.0
max      187.000000    119.000000       0.0       0.0       0.0       0.0

       pixel783
count   28000.0
mean        0.0
std         0.0
min         0.0
25%         0.0
50%         0.0
75%         0.0
max         0.0

[8 rows x 784 columns]
```

[37]:
```
order = list(np.sort(train_data['label'].unique()))
print(order)
```

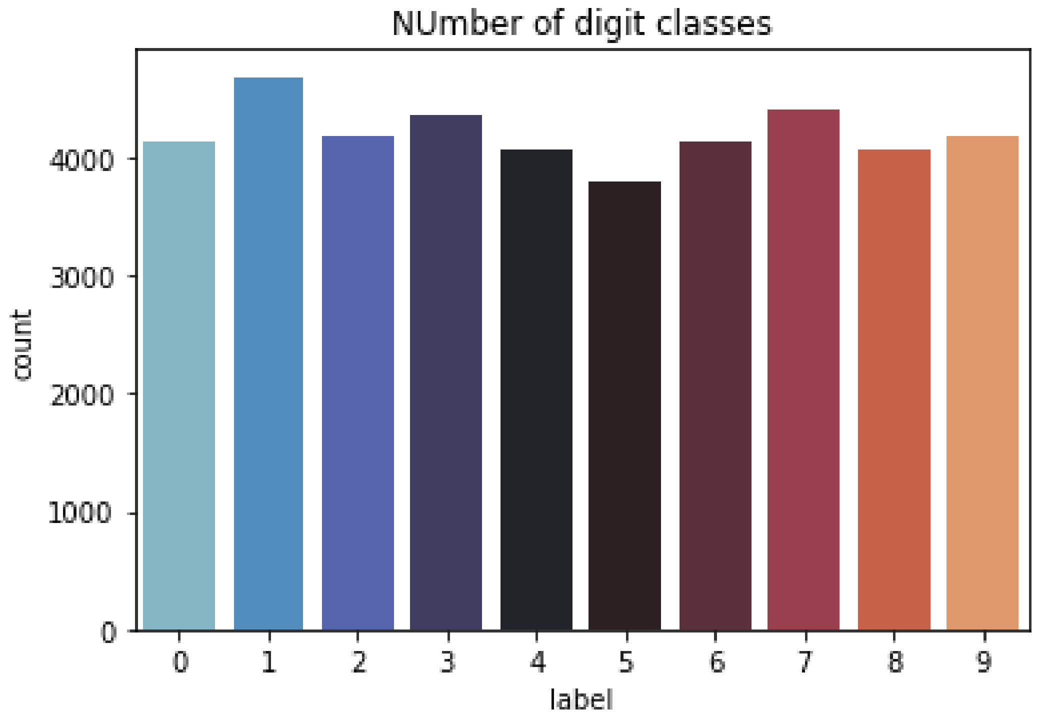
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[40]:
```
## Visualizing the number of class and counts in the datasets
plt.plot(figure = (16,10))
g = sns.countplot( train_data["label"], palette = 'icefire')
plt.title('Number of digit classes')
train_data.label.astype('category').value_counts()
```

```
D:\IDEs\Anaconda\lib\site-packages\seaborn\_decorators.py:43: FutureWarning:

Pass the following variable as a keyword arg: x. From version 0.12, the only
valid positional argument will be `data`, and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
```

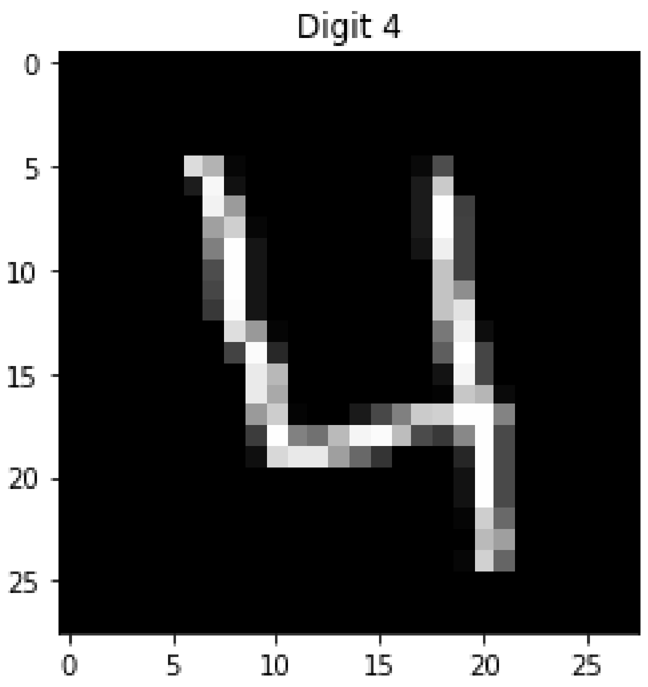
[40]:
```
1    4684
7    4401
3    4351
9    4188
2    4177
6    4137
0    4132
4    4072
8    4063
5    3795
Name: label, dtype: int64
```



[41]:
```
# Plotting some samples as well as converting into matrix

four = train_data.iloc[3, 1:]
four.shape
four = four.values.reshape(28,28)
plt.imshow(four, cmap='gray')
plt.title("Digit 4")
```

[41]:
```
Text(0.5, 1.0, 'Digit 4')
```



```
[42]: # average feature values
      round(train_data.drop('label', axis=1).mean(), 2)
```

```
[42]: pixel0      0.0
      pixel1      0.0
      pixel2      0.0
      pixel3      0.0
      pixel4      0.0
                 …
      pixel779    0.0
      pixel780    0.0
      pixel781    0.0
      pixel782    0.0
      pixel783    0.0
      Length: 784, dtype: float64
```

Observation:

- In this case, the average values do not vary a lot (e.g. having a diff of an order of magnitude). Nevertheless, it is better to rescale them.

```
[43]: ## Separating the X and Y variable

      y = train_data['label']
```

```
## Dropping the variable 'label' from X variable
X = train_data.drop(columns = 'label')

## Printing the size of data
print(train_data.shape)
```

```
(42000, 785)
```

[47]:
```
# Splitting into train and test for model 1 and reseting the index to avoid␣
↪jumbled index

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
↪random_state = 100)

X_train = X_train.reset_index(drop = True)
X_test = X_test.reset_index(drop = True)
y_train = y_train.reset_index(drop = True)
y_test = y_test.reset_index(drop = True)

print("shape of X_train is: ", X_train.shape)
print("shape of X_test is: ", X_test.shape)
print("shape of y_train is: ", y_train.shape)
print("shape of y_test is: ", y_test.shape)
```

```
shape of X_train is:  (29400, 784)
shape of X_test is:  (12600, 784)
shape of y_train is:  (29400,)
shape of y_test is:  (12600,)
```

[48]:
```
# define standard scaler
std_scaler = StandardScaler()
# transform data
X_train_scaled = std_scaler.fit_transform(X_train)
X_test_scaled = std_scaler.transform(X_test)
```

Model Building

Let’s fist build two basic models - linear and non-linear with default hyperparameters, and compare the accuracies

[49]:
```
# linear model

svc_model_linear = SVC(kernel='linear')
svc_model_linear.fit(X_train_scaled, y_train)

# predict train data
y_train_pred = svc_model_linear.predict(X_train_scaled)
```

```
# predict test data
y_test_pred = svc_model_linear.predict(X_test_scaled)
```

[50]:
```
# Checking the evaluation metrics of the train set

print(classification_report(y_train, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2890
           1       1.00      1.00      1.00      3260
           2       1.00      1.00      1.00      2978
           3       0.99      0.99      0.99      3055
           4       1.00      1.00      1.00      2868
           5       0.99      0.99      0.99      2617
           6       1.00      1.00      1.00      2887
           7       0.99      0.99      0.99      3117
           8       0.99      0.99      0.99      2827
           9       0.99      0.99      0.99      2901

    accuracy                           0.99     29400
   macro avg       0.99      0.99      0.99     29400
weighted avg       0.99      0.99      0.99     29400
```

[51]:
```
# Checking the evaluation metrics of the test set

print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       0.95      0.97      0.96      1242
           1       0.95      0.98      0.97      1424
           2       0.87      0.89      0.88      1199
           3       0.88      0.89      0.89      1296
           4       0.89      0.93      0.91      1204
           5       0.89      0.88      0.88      1178
           6       0.96      0.95      0.95      1250
           7       0.93      0.93      0.93      1284
           8       0.90      0.83      0.87      1236
           9       0.92      0.88      0.90      1287

    accuracy                           0.91     12600
   macro avg       0.91      0.91      0.91     12600
weighted avg       0.91      0.91      0.91     12600
```

Observation:

- The accuracy is nearly 99 % on train data and 91% on test data
- the model correctly classified the classes 0 , 1, 2, 4, 6 with 100% accuracy

[52]:
```
clf1 = svm.SVC(kernel = 'linear', C = 5, gamma= 0.05)
clf1.fit(X_train_scaled, y_train)

# predict train data
y_train_pred = clf1.predict(X_train_scaled)


# predict test data
y_test_pred = clf1.predict(X_test_scaled)
```

```
<IPython.core.display.Javascript object>
```

[53]:
```
# Checking the evaluation metrics of the train set for kernel = linear, C = 5,␣
↪gamma = .05

print(classification_report(y_train, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2890
           1       1.00      1.00      1.00      3260
           2       1.00      1.00      1.00      2978
           3       0.99      1.00      1.00      3055
           4       1.00      1.00      1.00      2868
           5       1.00      1.00      1.00      2617
           6       1.00      1.00      1.00      2887
           7       0.99      1.00      1.00      3117
           8       1.00      1.00      1.00      2827
           9       1.00      0.99      0.99      2901

    accuracy                           1.00     29400
   macro avg       1.00      1.00      1.00     29400
weighted avg       1.00      1.00      1.00     29400
```

[54]:
```
# Checking the evaluation metrics of the test set for kernel = linear, C = 5,␣
↪gamma = .05

print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       0.95      0.97      0.96      1242
           1       0.95      0.98      0.97      1424
           2       0.86      0.89      0.88      1199
           3       0.88      0.88      0.88      1296
           4       0.89      0.93      0.91      1204
           5       0.88      0.87      0.88      1178
           6       0.96      0.95      0.95      1250
           7       0.91      0.93      0.92      1284
           8       0.90      0.83      0.86      1236
           9       0.92      0.86      0.89      1287

    accuracy                           0.91     12600
   macro avg       0.91      0.91      0.91     12600
weighted avg       0.91      0.91      0.91     12600
```

Observation:

- The accuracy increased when compared to the previous model for classes 0 - 8 with hyper-parameters C = 5, gamma = .05

[57]:
```
clf2 = SVC( kernel = 'poly', gamma = 0.1, random_state = 0)
clf2.fit(X_train_scaled, y_train)

# predict train data
y_train_pred = clf2.predict(X_train_scaled)


# predict test data
y_test_pred = clf2.predict(X_test_scaled)
```

[58]:
```
# Checking the evaluation metrics of the train set for kernel = poly and gamma=␣
↪0.1

print(classification_report(y_train, y_train_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2890
           1       1.00      1.00      1.00      3260
           2       1.00      1.00      1.00      2978
           3       1.00      1.00      1.00      3055
           4       1.00      1.00      1.00      2868
           5       1.00      1.00      1.00      2617
           6       1.00      1.00      1.00      2887
           7       1.00      1.00      1.00      3117
           8       1.00      1.00      1.00      2827
           9       1.00      1.00      1.00      2901

    accuracy                           1.00     29400
   macro avg       1.00      1.00      1.00     29400
weighted avg       1.00      1.00      1.00     29400
```

[59]:
```
# Checking the evaluation metrics of the test set for kernel = linear, C = 5,␣
↪gamma = 0.1

print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1242
           1       0.99      0.99      0.99      1424
           2       0.97      0.96      0.96      1199
           3       0.97      0.97      0.97      1296
           4       0.96      0.97      0.96      1204
           5       0.98      0.97      0.97      1178
           6       0.99      0.99      0.99      1250
           7       0.97      0.97      0.97      1284
           8       0.96      0.97      0.96      1236
           9       0.95      0.96      0.96      1287

    accuracy                           0.97     12600
   macro avg       0.97      0.97      0.97     12600
weighted avg       0.97      0.97      0.97     12600
```

Observation:

- The best accuracy is for kernel polynomial, C = 5, gamma = 0.1 since it classified all the classes correctly when compared to previous models for both train and test dataset

[ ]:

[ ]:

Sources to learn more about the topics:

https://www.ibm.com/topics/logistic-regression

https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

https://towardsdatascience.com/kernel-function-6f1d2be6091

https://www.quora.com/What-are-kernels-in-machine-learning-and-SVM-and-why-do-we-need-them/answer/Lili-Jiang?srid=oOgT

https://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/

https://neptune.ai/blog/f1-score-accuracy-roc-auc-pr-auc

[ ]: