MODULE II:

Classification

The Classification algorithm is a Supervised Learning technique that is used to identify the category of new observations on the basis of training data. In Classification, a program learns from the given dataset or observations and then classifies new observation into a number of classes or groups. Such as, Yes or No, 0 or 1, Spam or Not Spam, cat or dog, etc. Classes can be called as targets/labels or categories.

Unlike regression, the output variable of Classification is a category, not a value, such as "Green or Blue", "fruit or animal", etc. Since the Classification algorithm is a Supervised learning technique, hence it takes labeled input data, which means it contains input with the corresponding output.

MNIST dataset

The MNIST database (Modified National Institute of Standards and Technology database) is a large collection of handwritten digits. It has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger NIST Special Database 3 (digits written by employees of the United States Census Bureau) and Special Database 1 (digits written by high school students) which contain monochrome images of handwritten digits. The digits have been size-normalized and centered in a fixed-size image. The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

Programming snippet to interact with MNIST dataset

```
from sklearn.datasets import fetch_openml
mnist=fetch_openml("mnist_784",as_frame=False )
#as_frame=False as we need to process image as numpy arrays

#extracting features and target
x,y=mnist.data,mnist.target

#split dataset into train and test
xtrain,xtest,ytrain,ytest=x[:60000],x[60000:],y[:60000],y[60000:]

#Code to display a digit
import matplotlib.pyplot as plt
def show_digit(img):
 imgdata=img.reshape(28,28)
 plt.imshow(imgdata,cmap="binary")

show_digit(x[1])
```

Training a Binary Classifier

In a binary classification task, the goal is to classify the input data into two mutually exclusive categories. The training data in such a situation is labeled in a binary format: true and false; positive and negative; 0 and 1; spam and not spam, etc. depending on the problem being tackled. For instance, we might want to detect whether a given image is a truck or a boat. Logistic Regression and Support Vector Machines algorithms are natively designed for binary classifications. However, other algorithms such as K-Nearest Neighbors and Decision Trees can also be used for binary classification.

Python code for a binary classifier for digits (5 or not-5)

```
ytrain5=(ytrain=='5')
ytest5=(ytest=='5')
from sklearn.linear_model import SGDClassifier
sg=SGDClassifier()
sg.fit(xtrain,ytrain5)
sg.predict([x[1]])
```

Performance Measures of Binary classifiers

Crossvalidation

Cross validation is a technique used in machine learning to evaluate the performance of a model on unseen data. It involves dividing the available data into multiple folds or subsets, using one of these folds as a validation set, and training the model on the remaining folds. This process is repeated multiple times, each time using a different fold as the validation set. Finally, the results from each validation step are averaged to produce a more robust estimate of the model's performance. The main purpose of cross validation is to prevent overfitting, which occurs when a model is trained too well on the training data and performs poorly on new, unseen data. By evaluating the model on multiple validation sets, cross validation provides a more realistic estimate of the model's generalization performance, i.e., its ability to perform well on new, unseen data.

#Python code for cross-validation on digit binary classifier that computes accuracy of classification

```
from sklearn.model_selection import cross_val_score
cross_val_score(sg,xtrain,ytrain5,cv=3,scoring='accuracy')
```

#Python code for prediction with cross validation

```
from sklearn.model_selection import cross_val_predict
ypred=cross_val_predict(sg,xtrain,ytrain5,cv=3)
```

Confusion Matrix

A confusion matrix is a matrix that summarizes the performance of a machine learning model on a set of test data. It is a means of displaying the number of accurate and inaccurate instances based on the model's predictions. It is often used to measure the performance of classification models, which aim to predict a categorical label for each input instance.

The matrix displays the number of instances produced by the model on the test data.

- True positives (TP): occur when the model accurately predicts a positive data point.
- True negatives (TN): occur when the model accurately predicts a negative data point.
- False positives (FP): occur when the model predicts a positive data point incorrectly.
- False negatives (FN): occur when the model mispredicts a negative data point.

Table for confusion matrix

Actual	Predicted	
	Non-5	5
Non-5	TN	FP
5	FN	TP

#Python code to generate Confusion matrix

```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(ytrain5,ypred)
print(cm)
```

Precision

Precision is defined as the ratio of correctly classified positive samples (True Positive) to a total number of classified positive samples (either correctly or incorrectly).

Precision = True Positive/True Positive + False Positive

Precision = TP/TP+FP

precision helps us to visualize the reliability of the machine learning model in classifying the model as positive

Recall

The recall is calculated as the ratio between the numbers of Positive samples correctly classified as Positive to the total number of Positive samples. The recall measures the model's ability to detect positive samples. The higher the recall, the more positive samples detected.

Recall = True Positive/True Positive + False Negative

Recall = TP/TP+FN

Unlike Precision, Recall is independent of the number of negative sample classifications. Further, if the model classifies all positive samples as positive, then Recall will be 1.

#Python code for precision and recall

```
from sklearn.metrics import precision_score,recall_score,f1_score
precision_score(ytrain5,ypred)
recall_score(ytrain5,ypred)
```

Difference between Precision and Recall in Machine Learning

Precision	Recall
It helps us to measure the ability to classify positive samples in the model.	It helps us to measure how many positive samples were correctly classified by the ML model.
While calculating the Precision of a model, we should consider both Positive as well as Negative samples that are classified.	While calculating the Recall of a model, we only need all positive samples while all negative samples will be neglected.
When a model classifies most of the positive samples correctly as well as many false-positive samples, then the model is said to be a high recall and low precision model.	When a model classifies a sample as Positive, but it can only classify a few positive samples, then the model is said to be high accuracy, high precision, and low recall model.
The precision of a machine learning model is dependent on both the negative and positive samples.	Recall of a machine learning model is dependent on positive samples and independent of negative samples.
In Precision, we should consider all positive samples that are classified as positive either correctly or incorrectly.	The recall cares about correctly classifying all positive samples. It does not consider if any negative sample is classified as positive.

F1-score

Precision and recall offer a trade-off, i.e., one metric comes at the cost of another. More precision involves a harsher critic (classifier) that doubts even the actual positive samples from the dataset, thus reducing the recall score. On the other hand, more recall entails a lax critic that allows any sample that resembles a positive class to pass, which makes border-case negative samples classified as "positive," thus reducing the precision. Ideally, we want to maximize both precision and recall metrics to obtain the perfect classifier.

The F1 score combines precision and recall using their harmonic mean, and maximizing the F1 score implies simultaneously maximizing both precision and recall. Thus, the F1 score has become the choice of researchers for evaluating their models in conjunction with accuracy.
The F1 score is calculated as the harmonic mean of the precision and recall scores, as shown below. It ranges from 0-100%, and a higher F1 score denotes a better quality classifier.

$$\text{F1 Score} = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

#API for f1-score

```
f1_score(ytrain5,ypred)
```

Precision-recall trade-off

The classification threshold is an important parameter when building and evaluating classification models. It can significantly impact the model's performance and the decisions made based on its predictions.

A typical default choice is to use a threshold of 0.5.

In the spam example, that would mean that any email with a predicted probability greater than 0.5 is classified as spam and put in a spam folder. Any email with a predicted probability of less than or equal to 0.5 is classified as legitimate.

Each metric has its limitations. Precision prioritizes “correctness” but may not account for the cost of missing positive cases. Recall emphasizes “completeness” but may result in falsely flagging some instances. Both types of errors can be expensive, depending on the specific use case. Since precision and recall measure different aspects of the model quality, this leads to the precision-recall trade-off. You must balance their importance and account for it when training and evaluating ML models.

To balance precision and recall, you should consider the costs of false positives and false negatives errors. This is highly custom and depends on the business context. You might make different choices when solving the same problem in different companies.

Optimize for recall

Say your task is to score the customers likely to buy a particular product. You then pass this list of high-potential customers to a call center team to contact them. You might have thousands of customers registering on your website every week, and the call center cannot reach all of them. But they can easily reach a couple of hundred.

Every customer that buys the product will make an effort well worth it. In this scenario, the cost of false positives is low (just a quick call that does not result in a purchase), but the value of true positives is high (immediate revenue).

In this case, you'd likely optimize for recall. You want to make sure you reach all potential buyers. Your only limit is the number of people your call center can contact weekly. In this case, you can set a lower decision threshold. Your model might have low precision, but this is not a big deal as long as you reach your business goals and make a certain number of sales.

Optimize for precision

Let's say you are working for a food delivery company. Your team is developing a machine learning model to predict which orders might be delivered in under 20 minutes based on factors such as order size, restaurant location, time of day, and delivery distance.

You will use this prediction to display a "fast delivery" label next to a potential order.

In this case, optimizing for precision makes sense. False positives (orders predicted to be completed fast but actually delayed) can result in a loss of customer trust and ultimately lead to decreased sales. On the other hand, false negatives (orders predicted to take longer but completed in under 20 minutes) will likely have no consequences at all, as the customer would simply be pleasantly surprised by the fast delivery. Optimizing for precision typically means setting a higher classification threshold.

Balance precision and recall

consider a scenario where you are developing a model to detect fraudulent transactions in a banking system. In this case, the cost of false positives is high, as it can lead to blocking legitimate transactions and causing inconvenience to the customers. On the

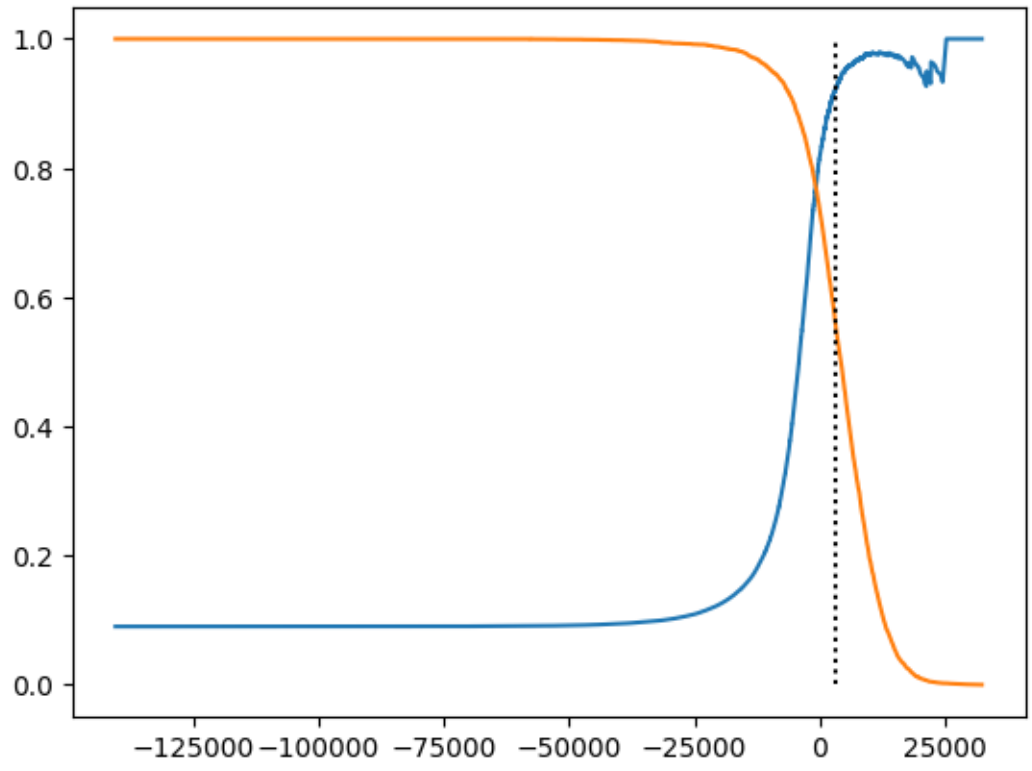
other hand, the cost of false negatives is also significant, as it can result in financial loss and lost trust due to fraudulent transactions.
In this case, you need to strike a balance between precision and recall. While you need to detect as many fraudulent transactions as possible (high recall), you must also ensure that you don't flag legitimate transactions as fraudulent too often (high precision). The threshold for detecting a fraudulent transaction must be set carefully, considering the costs associated with both types of errors.
Since the fraud cost can differ, you can also set different thresholds based on the transaction amounts. For example, you can set a lower decision threshold for high-volume transactions since they come with a higher potential financial loss. For smaller amounts (which are also more frequent), you can set the threshold higher to ensure you do not inconvenience customers too much.

Precision-recall curve

One approach is the precision-recall curve. It shows the value pairs between precision and recall at different thresholds.
#Code for precision-recall curve

```
yscores=cross_val_predict(sg,xtrain,ytrain5,cv=3,
                          method='decision_function')
from sklearn.metrics import precision_recall_curve
p,r,t=precision_recall_curve(ytrain5,yscores)
plt.plot(t,p[:-1],label="T vs P")
plt.plot(t,r[:-1],label="T vs R")
plt.vlines(3000,0,1.0,"k","dotted",label="threshold line")
plt.show()
```

Output:



An ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

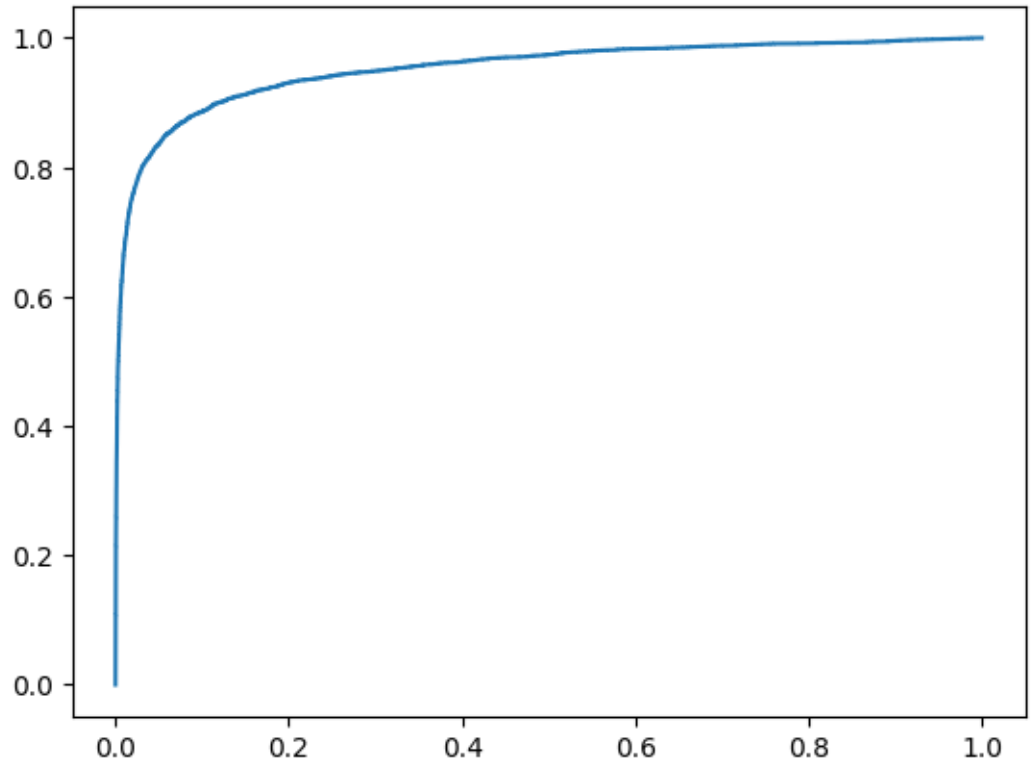
- True Positive Rate
- False Positive Rate

An ROC curve plots TPR vs. FPR at different classification thresholds. Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives.

#Python code for ROC curve

```
from sklearn.metrics import roc_curve
fpr,tpr,t=roc_curve(ytrain5,yscores)
plt.plot(fpr,tpr,label="FPR vs TPR")
plt.show()
```

Output:



AUC: Area Under the ROC Curve

AUC stands for "Area under the ROC Curve." That is, AUC measures the entire two-dimensional area underneath the entire ROC curve (think integral calculus) from (0,0) to (1,1).

AUC provides an aggregate measure of performance across all possible classification thresholds. One way of interpreting AUC is as the probability that the model ranks a random positive example more highly than a random negative example. AUC is desirable for the following two reasons:

- AUC is scale-invariant. It measures how well predictions are ranked, rather than their absolute values.
- AUC is classification-threshold-invariant. It measures the quality of the model's predictions irrespective of what classification threshold is chosen.

#Python code for AUC

```
from sklearn.metrics import roc_auc_score
a=roc_auc_score(ytrain5,yscores)
print(a)
```

Multiclass classification

Binary classification are those tasks where examples are assigned exactly one of two classes. Multi-class classification is those tasks where examples are assigned exactly one of more than two classes.

- Binary Classification: Classification tasks with two classes.
- Multi-class Classification: Classification tasks with more than two classes.

Some algorithms are designed for binary classification problems. Examples include:

- Logistic Regression
- Perceptron
- Support Vector Machines

As such, they cannot be used for multi-class classification tasks, at least not directly.
Instead, heuristic methods can be used to split a multi-class classification problem into multiple binary classification datasets and train a binary classification model each.
Two examples of these heuristic methods include:

- One-vs-Rest (OvR)
- One-vs-One (OvO)

One-Vs-Rest for Multi-Class Classification

One-vs-rest (OvR for short, also referred to as One-vs-All or OvA) is a heuristic method for using binary classification algorithms for multi-class classification.
It involves splitting the multi-class dataset into multiple binary classification problems. A binary classifier is then trained on each binary classification problem and predictions are made using the model that is the most confident.
For example, given a multi-class classification problem with examples for each class 'red,' 'blue,' and 'green'. This could be divided into three binary classification datasets as follows:

- Binary Classification Problem 1: red vs [blue, green]
- Binary Classification Problem 2: blue vs [red, green]
- Binary Classification Problem 3: green vs [red, blue]

A possible downside of this approach is that it requires one model to be created for each class. For example, three classes requires three models. This could be an issue for large datasets (e.g. millions of rows), slow models (e.g. neural networks), or very large numbers of classes (e.g. hundreds of classes).

#Python code for One-Vs-Rest

```
from sklearn.multiclass import OneVsRestClassifier
oc=OneVsRestClassifier(SVC())
oc.fit(xtrain[:2000],ytrain[:2000])
oc.predict([x[0]])
oc.decision_function([x[0]]).round(2)
```

One-Vs-One for Multi-Class Classification

One-vs-One (OvO for short) is another heuristic method for using binary classification algorithms for multi-class classification.

Like one-vs-rest, one-vs-one splits a multi-class classification dataset into binary classification problems. Unlike one-vs-rest that splits it into one binary dataset for each class, the one-vs-one approach splits the dataset into one dataset for each class versus every other class.

For example, consider a multi-class classification problem with four classes: 'red,' 'blue,' and 'green,' 'yellow.' This could be divided into six binary classification datasets as follows:

- Binary Classification Problem 1: red vs. blue
- Binary Classification Problem 2: red vs. green
- Binary Classification Problem 3: red vs. yellow
- Binary Classification Problem 4: blue vs. green
- Binary Classification Problem 5: blue vs. yellow
- Binary Classification Problem 6: green vs. yellow

The formula for calculating the number of binary datasets, and in turn, models, is as follows:

- (NumClasses * (NumClasses – 1)) / 2

Classically, this approach is suggested for support vector machines (SVM) and related kernel-based algorithms. This is believed because the performance of kernel methods does not scale in proportion to the size of the training dataset and using subsets of the training data may counter this effect.

#Python code for One-Vs-One approach

```
from sklearn.svm import SVC
sc=SVC()
sc.fit(xtrain[:2000],ytrain[:2000])
sc.predict([x[0]])
sds=sc.decision_function([x[0]])
sds.round(2)
classid=sds.argmax()
classid
```

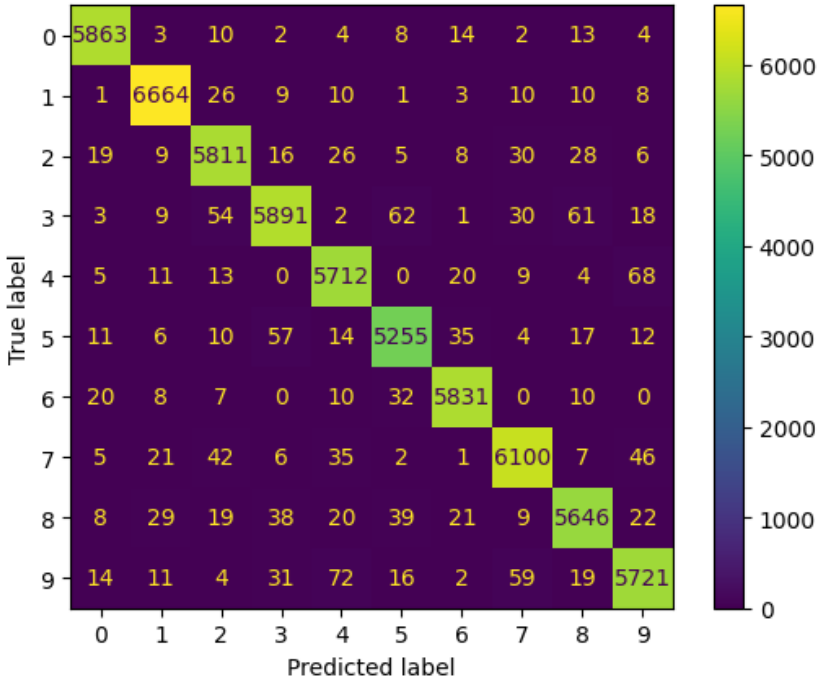
Error Analysis

Error analysis is the process to isolate, observe and diagnose erroneous ML predictions thereby helping understand pockets of high and low performance of the model. When it is said that "the model accuracy is 90%" it might not be uniform across subgroups of data and there might be some input conditions which the model fails more.

#Python code to display number of correct and wrong digit classifications

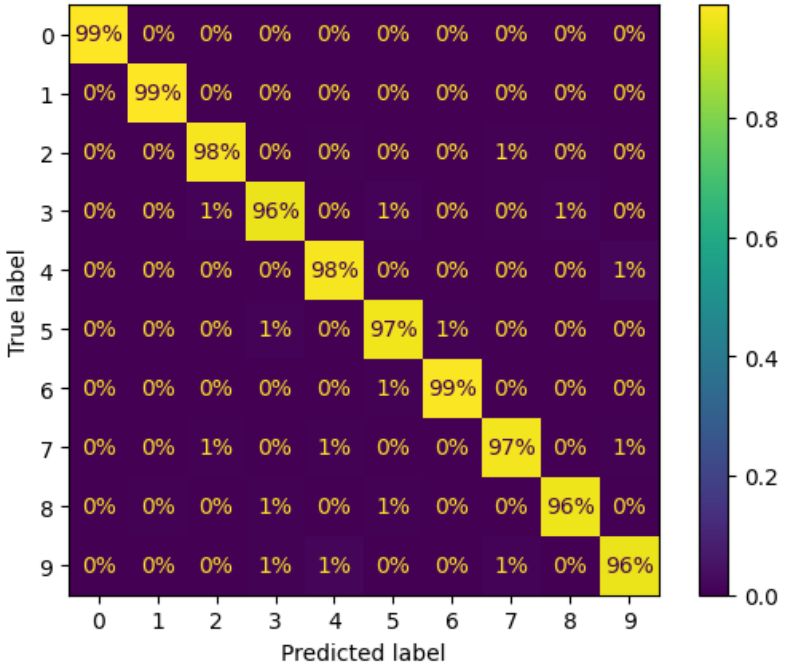
```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import ConfusionMatrixDisplay
ypred=cross_val_predict(sc,xtrain,ytrain,cv=3)
ConfusionMatrixDisplay.from_predictions(ytrain,ypred)
```

Output:



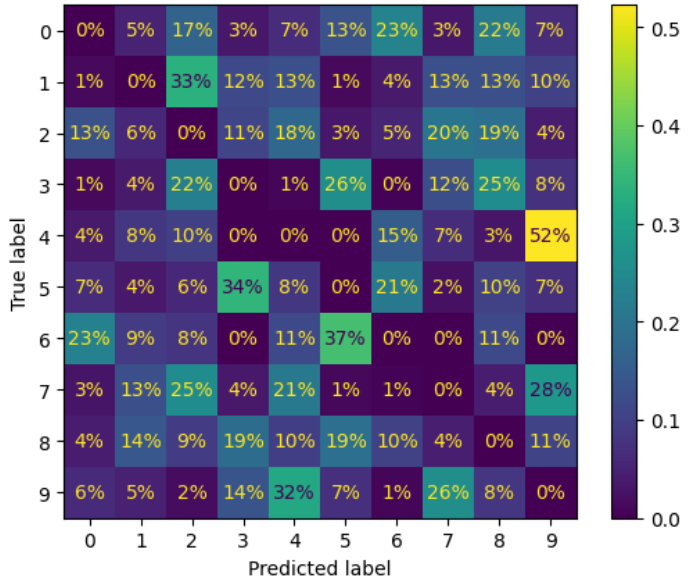
#Python code to display normalized errors

```
ConfusionMatrixDisplay.from_predictions(ytrain,ypred,normalize="true",values_format=".0%")
plt.show()
```



#Python code to zoom-up the major error classifications

```
sample_weight=(ypred!=ytrain)
ConfusionMatrixDisplay.from_predictions(ytrain,ypred,normalize="true",sample_weight=sample_weight,values_format=".0%")
plt.show()
```



Multilabel classification:

It is used when there are two or more classes and the data we want to classify may belong to none of the classes or all of them at the same time, e.g. to classify which traffic signs are contained on an image. In multi-label classification, the training set is composed of instances each associated with a set of labels, and the task is to predict the label sets of unseen instances through analyzing training instances with known label sets.

Difference between multi-class classification & multi-label classification is that in multi-class problems the classes are mutually exclusive, whereas for multi-label problems each label represents a different classification task, but the tasks are somehow related.

For example, multi-class classification makes the assumption that each sample is assigned to one and only one label: a fruit can be either an apple or a pear but not both at the same time. Whereas, an instance of multi-label classification can be that a text might be about any of religion, politics, finance or education at the same time or none of these.

Multioutput Algorithms

Multioutput algorithms are a type of machine learning approach designed for problems where the output consists of multiple variables, and each variable can belong to a different class or have a different range of values. In other words, multioutput problems involve predicting multiple dependent variables simultaneously.

Two main types of Multioutput Problems:

- Multioutput Classification: In multioutput classification, each instance is associated with a set of labels and the goal is to predict these labels simultaneously.
- Multioutput Regression: In multioutput regression, the task is to predict multiple continuous variables simultaneously.

MODULE-III

Linear Regression

Linear regression is a type of supervised machine learning algorithm that computes the linear relationship between the dependent variable and one or more independent features by fitting a linear equation to observed data. Linear regression is not merely a predictive tool; it forms the basis for various advanced models. Techniques like regularization and support vector machines draw inspiration from linear regression, expanding its utility. Additionally, linear regression is a cornerstone in assumption testing, enabling researchers to validate key assumptions about the data.

Linear regression model prediction:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- $\hat{y}$ is the predicted value.

n is the number of features.

x_i is the i^{th} feature value.

θ_j is the j^{th} model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$).

Linear Regression model prediction (vectorized form)

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

MSE cost function for a Linear Regression model

$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

Normal Equation

$$\hat{\boldsymbol{\theta}} = \left(\mathbf{X}^T \mathbf{X}\right)^{-1} \mathbf{X}^T \mathbf{y}$$

#Python code to predict Linear regression using Normal Equation

```
import numpy as np
n=100
x=2*np.random.randn(n,1)
y=4+3*x+np.random.randn(n,1)
from sklearn.preprocessing import add_dummy_feature
x1=add_dummy_feature(x)
theta=np.linalg.inv(x1.T@x1)@x1.T@y
xnew=np.array([[0],[2]])
xnew1=add_dummy_feature(xnew)
ypred=xnew1@theta
ypred
```

#Python code to perform Linear Regression based on the library function

```
from sklearn.linear_model import LinearRegression
lr=LinearRegression()
lr.fit(x,y)
print(lr.intercept_, lr.coef_)
```

Gradient Descent (GD)

It is a widely used optimization algorithm in machine learning and deep learning that minimises the cost function of a neural network model during training. It works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached. The learning happens during the backpropagation while training the neural network-based model. There is a term known as Gradient Descent, which is used to optimize the weight and biases based on the cost function. The cost function evaluates the difference between the actual and predicted outputs.

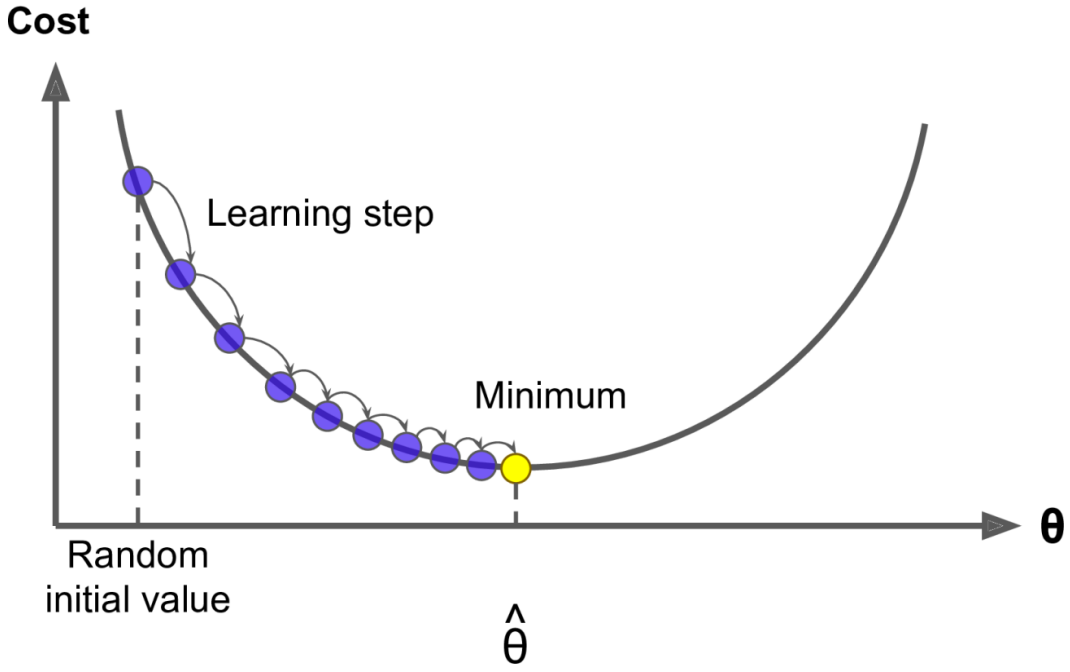
Gradient Descent is a fundamental optimization algorithm in machine learning used to minimize the cost or loss function during model training.

- It iteratively adjusts model parameters by moving in the direction of the steepest decrease in the cost function.
- The algorithm calculates gradients, representing the partial derivatives of the cost function concerning each parameter.

These gradients guide the updates, ensuring convergence towards the optimal parameter values that yield the lowest possible cost.

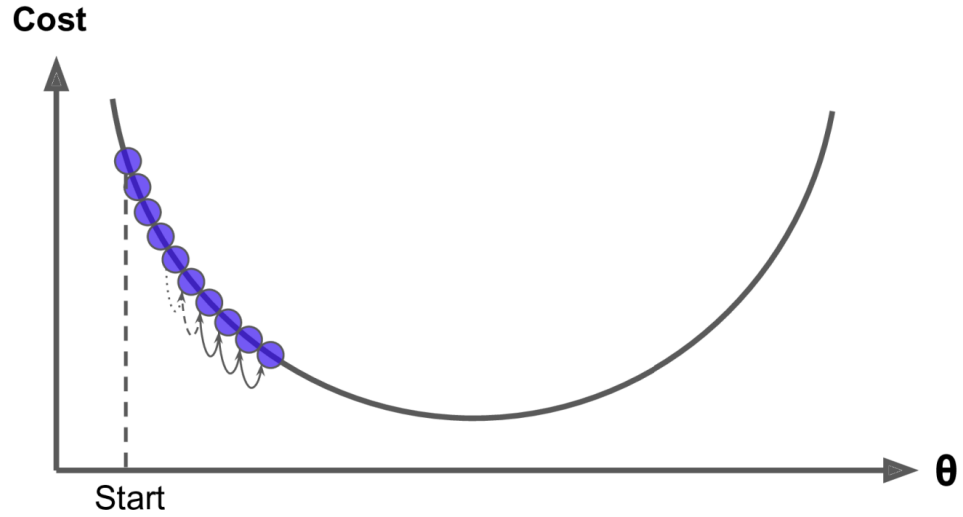
Gradient Descent is versatile and applicable to various machine learning models, including linear regression and neural networks.

The path of Gradient descent is depicted in the following figure:

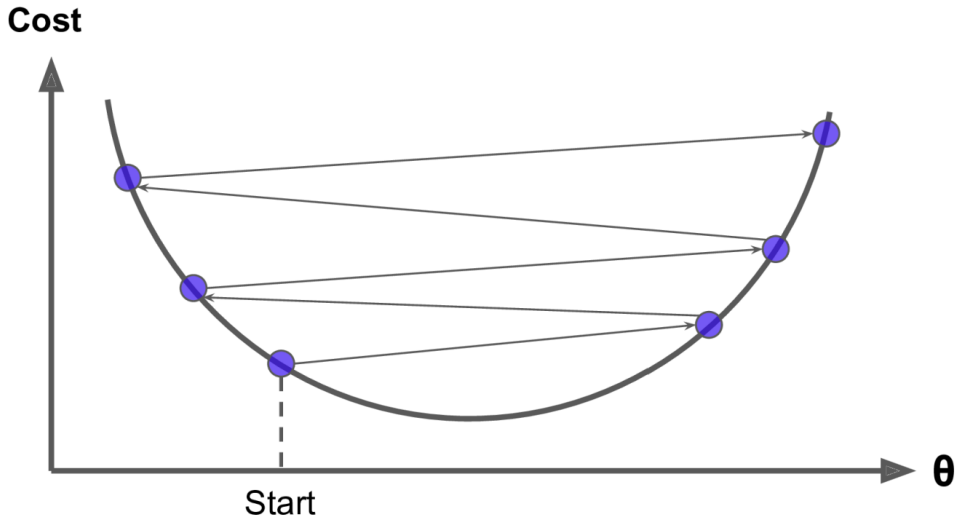


Impact of Learning rate:

If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time:

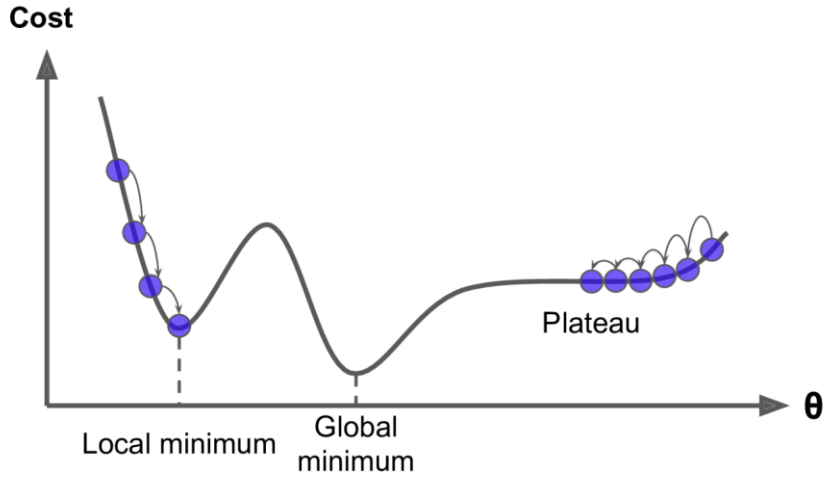


On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution

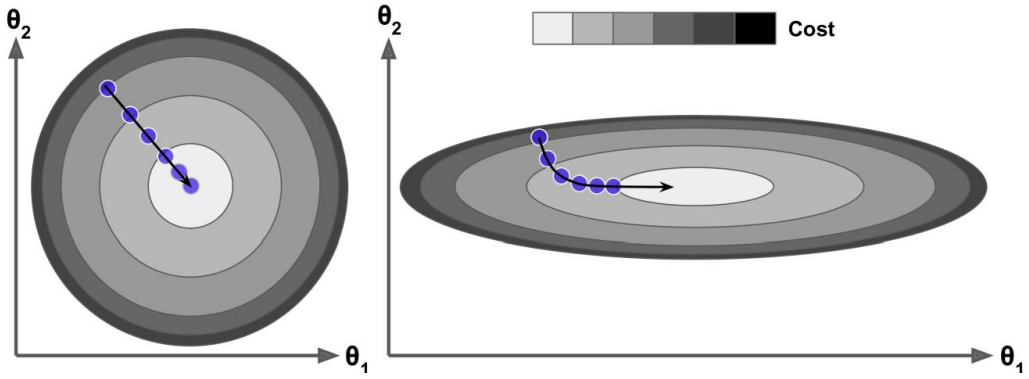


Challenges in Gradient Descent:

if the random initialization starts the algorithm on the left, then it will converge to a local mini- mum, which is not as good as the global minimum. If it starts on the right, then it will take a very long time to cross the plateau, and if you stop too early you will never reach the global minimum.



Impact of scaling on Gradient Descent:
Gradient Descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right)



Batch Gradient Descent

In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch.
Cost function in Batch Gradient descent:

$$\frac{\partial}{\partial \theta_j}\mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^T\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

Gradient vector of the cost function

$$\nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

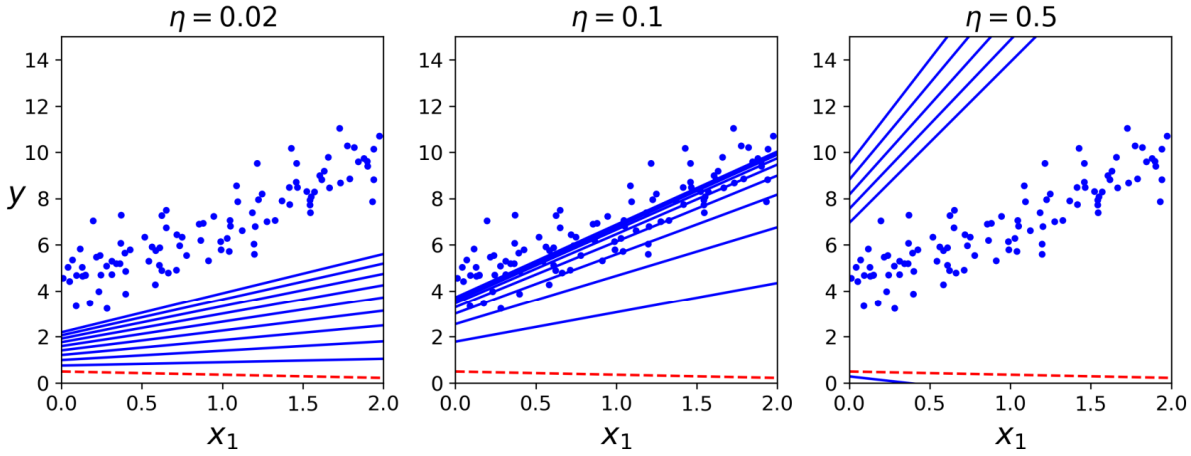
Gradient Descent step

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$$

#Python code to implement Full/Batch Gradient Descent

```
from sklearn.preprocessing import add_dummy_feature
eta=0.1
n=100
x=2*np.random.randn(n,1)
y=4+3*x+np.random.randn(n,1)
x1=add_dummy_feature(x)
m=len(x1)
theta=np.random.randn(2,1)
epochs=1000
for epoch in range(epochs):
 grad=2/m*x1.T@(x1@theta-y)
 theta=theta-eta*grad
theta
```

Full/Batch Gradient Descent with various learning rates:



On the left, the learning rate is too low: the algorithm will eventually reach the solu- tion, but it will take a long time. In the middle, the learning rate looks pretty good: in just a few iterations, it has already converged to the solution. On the right, the learn- ing rate is too high: the algorithm diverges.

Stochastic Gradient Descent

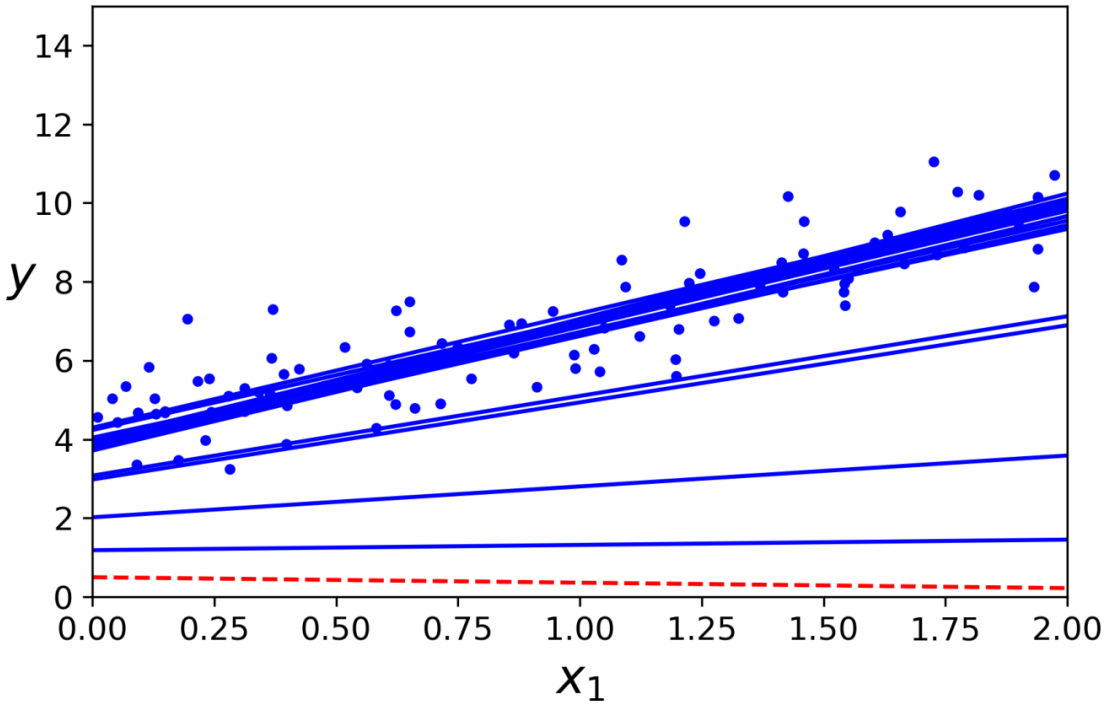
In Batch Gradient Descent we were considering all the examples for every step of Gradient Descent. But what if our dataset is very huge. Deep learning models crave for data. The more the data the more chances of a model to be good. Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples. This does not seem an efficient way. To tackle this problem we have Stochastic Gradient Descent. In Stochastic Gradient Descent (SGD), we consider just one random sample at a time to take a single step. Also because the cost is so fluctuating, it will never reach the minima but it will keep dancing around it.

#Python code to implement Stochastic Gradient Descent

```
from sklearn.preprocessing import add_dummy_feature
eta=0.1
n=100
x=2*np.random.randn(n,1)
y=4+3*x+np.random.randn(n,1)
x1=add_dummy_feature(x)
m=len(x1)
theta=np.random.randn(2,1)
epochs=1000
m=len(x)
def learning_schedule(t):
 return 5/(50+t)

for epoch in range(epochs):
 for iteration in range(m):
  ri=np.random.randint(m)
  xi=x1[ri:ri+1]
  yi=y[ri:ri+1]
  grad=2/m*xi.T@(xi@theta-yi)
  eta=learning_schedule(epoch*m+iteration)
  theta=theta-eta*grad
theta
```

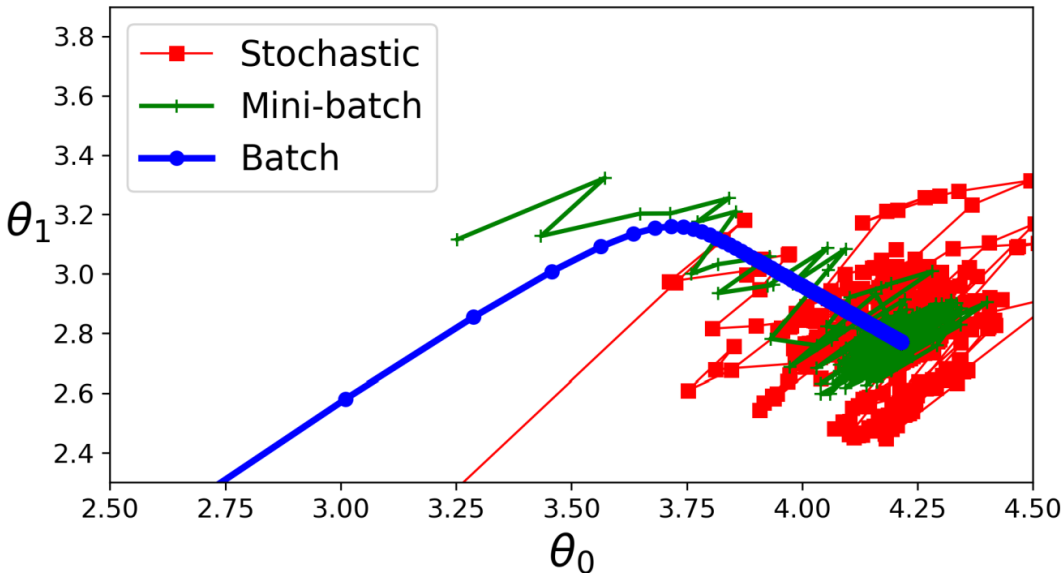
Performance of Stochastic Gradient Descent



Mini-batch Gradient Descent

We have seen the Batch Gradient Descent. We have also seen the Stochastic Gradient Descent. Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large. Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets. But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations. To tackle this problem, a mixture of Batch Gradient Descent and SGD is used. Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. Doing this helps us achieve the advantages of both the former variants.

Gradient Descent paths in parameter space



All algorithms end up near the minimum, but Batch GD's path actually stops at the minimum, while both Stochastic GD and Mini-batch GD continue to walk around.

However, don't forget that Batch GD takes a lot of time to take each step, and Stochastic GD and Mini-batch GD would also reach the minimum if you used a good learning schedule.

Polynomial Regression

Polynomial regression is a type of regression analysis used in statistics and machine learning when the relationship between the independent variable (input) and the dependent variable (output) is not linear. While simple linear regression models the relationship as a straight line, polynomial regression allows for more flexibility by fitting a polynomial equation to the data. When the relationship between the variables is better represented by a curve rather than a straight line, polynomial regression can capture the non-linear patterns in the data.

#Python code for polynomial Regression

```
m=100
import numpy as np
x=6*np.random.randn(m,1)-3
y=0.5*x**2+x+2+np.random.randn(m,1)
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
pf=PolynomialFeatures(degree=2,include_bias=False)
xpoly=pf.fit_transform(x)
lr=LinearRegression()
lr.fit(xpoly,y)
lr.intercept_,lr.coef_
```

Learning Curves

Learning curves are plots used to show a model's performance as the training set size increases. Another way it can be used is to show the model's performance over a defined period of time. We typically used them to diagnose algorithms that learn incrementally from data. It works by evaluating a model on the training and validation datasets, then plotting the measured performance.

Finding the right degree of a polynomial is a challenge and learning curves help in resolving it. Learning curves are the plots of training and validation error as a function of the training iteration.

#Python code to illustrate the use of Learning curves with Linear Regression

```
from sklearn.model_selection import learning_curve
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
tsize,tscore,vscore=learning_curve(LinearRegression(),x,y,
                   train_sizes=np.linspace(0.01,1,40),
                   cv=5,scoring="neg_root_mean_squared_error")
trainerr=-tscore.mean(axis=1)
validerr=-vscore.mean(axis=1)
import matplotlib.pyplot as plt
plt.plot(tsize,trainerr)
plt.plot(tsize,validerr)
plt.legend(["train","valid"])
```

```
plt.show()
```

#Python code to illustrate the use of Learning curves with Polynomial Regression

```
from sklearn.pipeline import make_pipeline
pr=make_pipeline(PolynomialFeatures(degree=2,include_bias=False),
            LinearRegression())
tsize,tscore,vscore=learning_curve(pr,x,y,
                    train_sizes=np.linspace(0.01,1,40),
                    cv=5,scoring="neg_root_mean_squared_error")
trainerr=-tscore.mean(axis=1)
validerr=-vscore.mean(axis=1)
import matplotlib.pyplot as plt
plt.plot(tsize,trainerr)
plt.plot(tsize,validerr)
plt.legend(["train","valid"])
plt.show()
```

Regularized Linear Models

Regularization is one of the most important concepts of machine learning. It is a technique to prevent the model from overfitting by adding extra information to it. Regularization works by adding a penalty or complexity term to the complex mode

There are three types of regularization techniques, which are given below:

- Ridge Regression
- Lasso Regression
- ElasticNet Regression

Ridge Regression

- Ridge regression is one of the types of linear regression in which a small amount of bias is introduced so that we can get better long-term predictions.
- Ridge regression is a regularization technique, which is used to reduce the complexity of the model. It is also called as L2 regularization.
- In this technique, the cost function is altered by adding the penalty term to it. The amount of bias added to the model is called Ridge Regression penalty. We can calculate it by multiplying with the lambda to the squared weight of each individual feature.
- The equation for the cost function in ridge regression will be:

$$J(\theta) = MSE(\theta) + \frac{\alpha}{m}\sum_{i=1}^{m}\theta_i^2$$

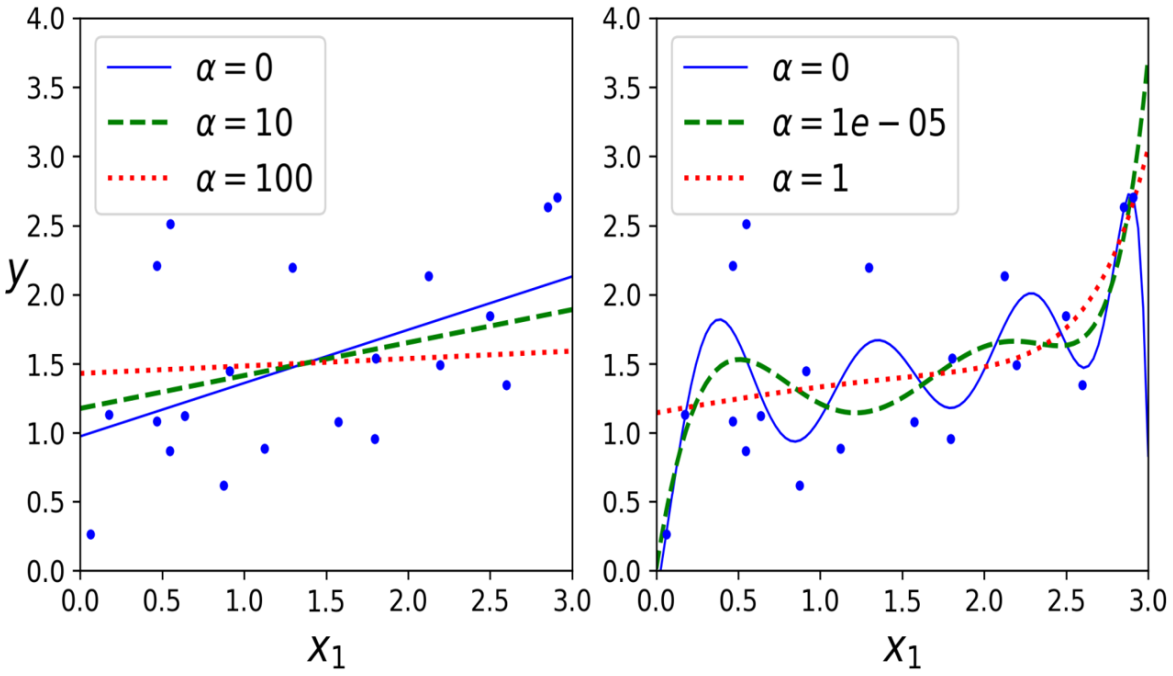
Normal Equation with Ridge regression

$$\hat{\theta} = (x^T x + \alpha A)^{-1}\, x^T y$$

#Python code for Ridge Regression

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1)
ridge_reg.fit(X, y)
OR
sgd_reg = SGDRegressor(penalty="l2")
```

Performance of Ridge regression for Linear and Non-linear cases:

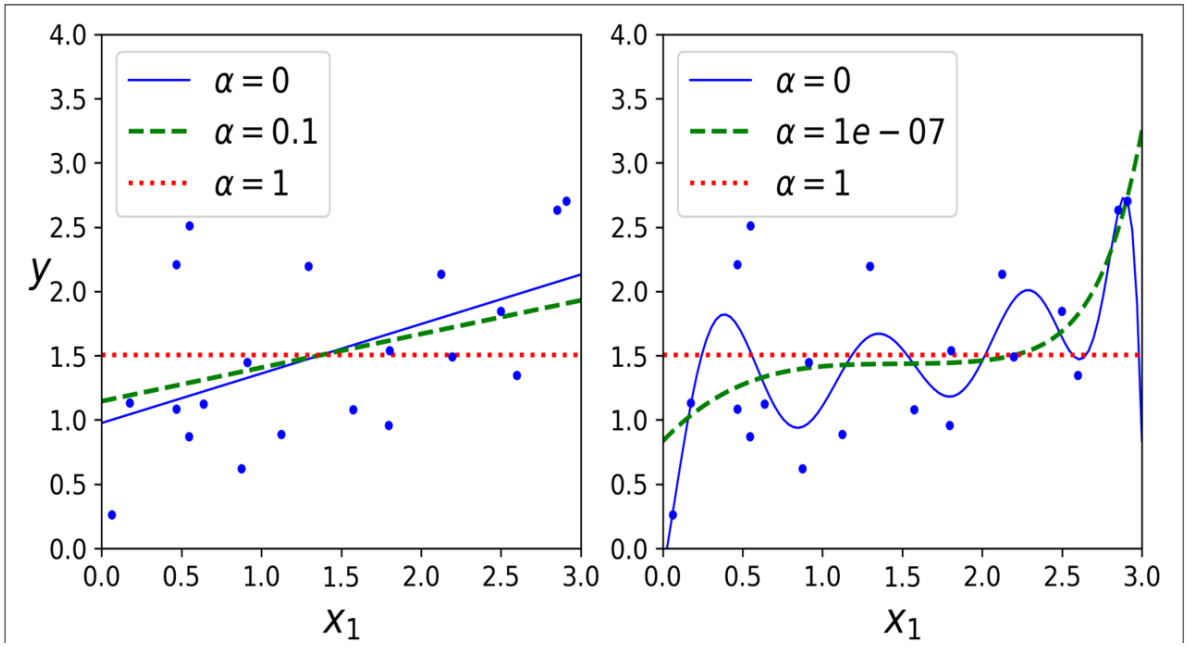


Lasso Regression

- Lasso regression is another regularization technique to reduce the complexity of the model. It stands for Least Absolute and Selection Operator.
- It is similar to the Ridge Regression except that the penalty term contains only the absolute weights instead of a square of weights.
- Since it takes absolute values, hence, it can shrink the slope to 0, whereas Ridge Regression can only shrink it near to 0.
- It is also called as L1 regularization. The equation for the cost function of Lasso regression will be:

$$J(\theta) = MSE(\theta) + 2\alpha \sum_{i=1}^{m} |\theta_i|$$

#Python code for Lasso Regression
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)
Performance of Lasso Regression



Elastic Net Regression

Elastic Net Regression is a powerful machine learning algorithm that combines the features of both Lasso and Ridge Regression. It is a regularized regression technique that is used to deal with the problems of multicollinearity and overfitting, which are common in high-dimensional datasets. This algorithm works by adding a penalty term to the standard least-squares objective function

Cost function of Elastic net regression:

$$J(\theta) = MSE(\theta) + (1 - r).\frac{\alpha}{m}\sum_{i=1}^{m}\theta_i^2 + r.2\alpha\sum_{i=1}^{m}|\theta_i|$$

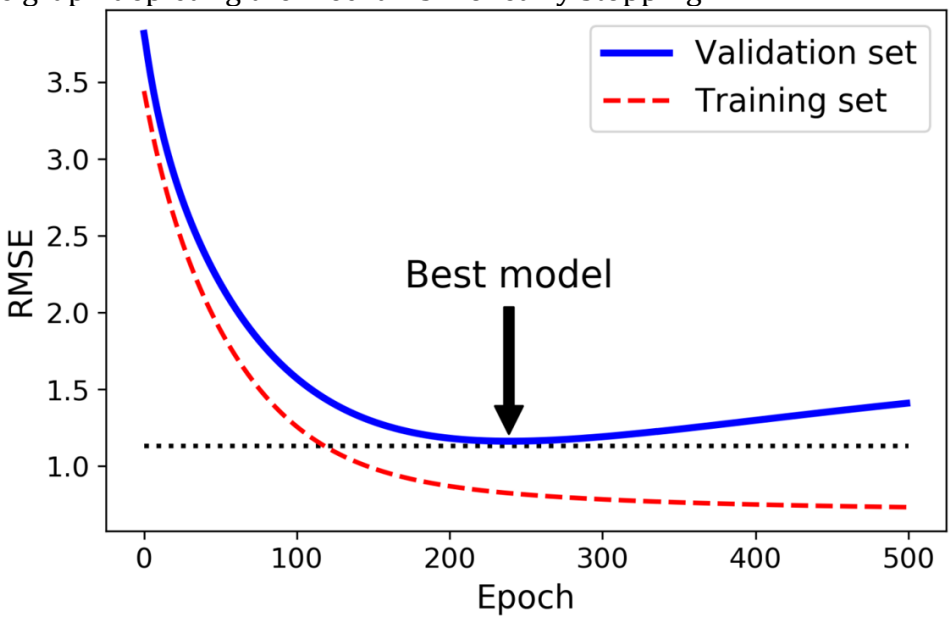
#Python code for Elastic net regression:

```
from sklearn.linear_model import ElasticNet
el_reg = ElasticNet(alpha=0.1,l1_ratio=0.5)
el_reg.fit(X, y)
```

Regularization by Early Stopping

In Regularization by Early Stopping, we stop training the model when the performance on the validation set is getting worse- increasing loss decreasing accuracy, or poorer scores of the scoring metric. By plotting the error on the training dataset and the validation dataset together, both the errors decrease with a number of iterations until the point where the model starts to overfit. After this point, the training error still decreases but the validation error increases. So, even if training is continued after this point, early stopping essentially returns the set of parameters that were used at this point and so is equivalent to stopping training at that point. So, the final parameters returned will enable the model to have low variance and better generalization. The model at the time the training is stopped will have a better generalization performance than the model with the least training errors.

The graph depicting the mechanism of early stopping:



Early stopping can be best used to prevent overfitting of the model, and saving resources. It would give best results if taken care of few things like – parameter tuning, preventing the model from overfitting, and ensuring that the model learns enough from the data.

Logistic Regression

Logistic regression is a supervised machine learning algorithm used for classification tasks where the goal is to predict the probability that an instance belongs to a given class or not. Logistic regression is used for binary classification where we use sigmoid function, that takes input as independent variables and produces a probability value between 0 and 1. Logistic regression predicts the output of a categorical dependent variable. Therefore, the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, true or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1. In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function, which predicts two maximum values (0 or 1).

Estimating probabilities in Logistic Regression:

$$\hat{p} = \sigma(\theta^T x)$$

$$\sigma(t) = \frac{1}{1 + exp(-t)}$$

$$\hat{y} = \begin{cases} 0 \ if \ \hat{p} < 0.5 \\ 1 \ if \ \hat{p} \geq 0.5 \end{cases}$$

Cost function of a single training instance of Logistic Regression

$$c(\boldsymbol{\theta}) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1-\hat{p}) & \text{if } y = 0 \end{cases}$$

Logistic Regression cost function (log loss)

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}log\left(\hat{p}^{(i)}\right) + \left(1-y^{(i)}\right)log\left(1-\hat{p}^{(i)}\right)\right]$$

Decision Boundaries

The fundamental application of logistic regression is to determine a decision boundary for a binary classification problem. Although the baseline is to identify a binary decision boundary, the approach can be very well applied for scenarios with multiple classification classes or multi-class classification.

#Python code to compute decision boundary for iris dataset

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
iris=load_iris(as_frame=True)
x=iris.data[['petal width (cm)']].values
y=iris.target_names[iris.target]=="virginica"
xtrain,xtest,ytrain,ytest=train_test_split(x,y,test_size=0.30)
lr=LogisticRegression()
lr.fit(xtrain,ytrain)
xnew=np.linspace(0,3,1000).reshape(-1,1)
yp=lr.predict_proba(xnew)
decisionboundary=xnew[yp[:,1]>=0.5][0,0]
print(decisionboundary)
```

Softmax regression

Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to the case where we want to handle multiple classes in the target column.

Softmax score for class k

$$s_k(\mathbf{x}) = \mathbf{x}^T\boldsymbol{\theta}^{(k)}$$

Softmax function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K}\exp\left(s_j(\mathbf{x})\right)}$$

K is the number of classes.

$\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance $\mathbf{x}$.

$\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance $\mathbf{x}$ belongs to class k given the scores of each class for that instance.

Softmax Regression classifier prediction

$$\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\ \left(\left(\boldsymbol{\theta}^{(k)}\right)^T \mathbf{x}\right)$$

The argmax operator returns the value of a variable that maximizes a function.
Cross entropy is frequently used to measure how well a set of estimated class probabilities match the target classes
Cross entropy cost function

$$J(\boldsymbol{\Theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$