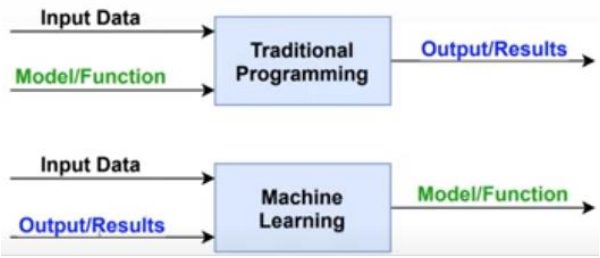
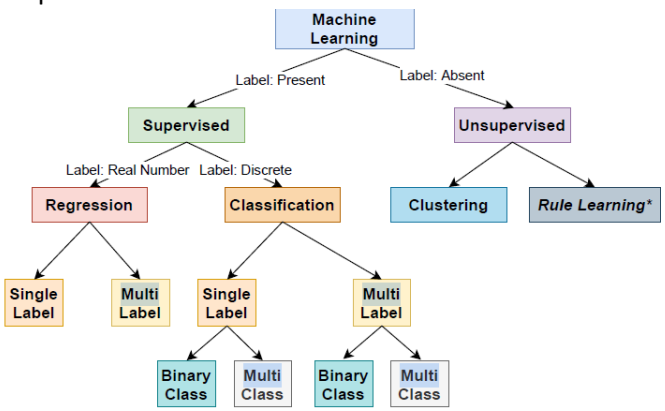
Basic intro to ML and its metrics

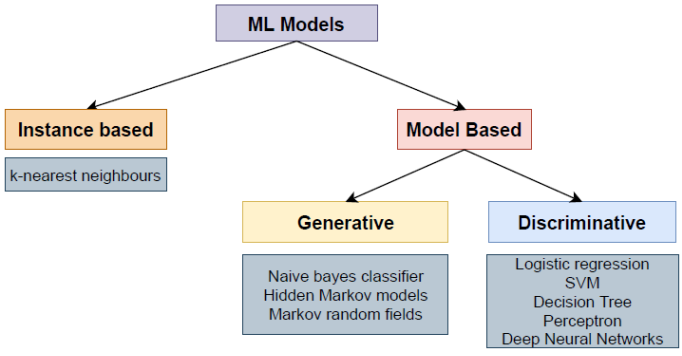
- Machine learning is subset of AI.
- Supervised and Unsupervised learning are two broad classes of machine learning algorithms.
- Linear Regression, Logistic Regression, Decision Trees, Support Vector Machines, Neural Networks are examples of supervised learning, while K-means clustering is unsupervised.
- Training Data, Model, Loss function, Optimization, Evaluation are the 5 steps followed by all machine learning algorithms.
- Traditional programming works on inputs and pre-defined model/function, and generates outputs. Machine learning works on inputs and outputs, and generates models/function that fits. Once the model/function has been learnt, traditional programming can be employed to make predictions (generate outputs)



- Broad categorization of machine learning algorithms based on data. Typically, multi-class/single label classification problems are simply called *multi-class*, where are multi-class/multi-label classification problems are simply called *multi-label*. In the case of regression, multi-label problems are called multi-output.



- Broad categorization of machine learning algorithms based on model.



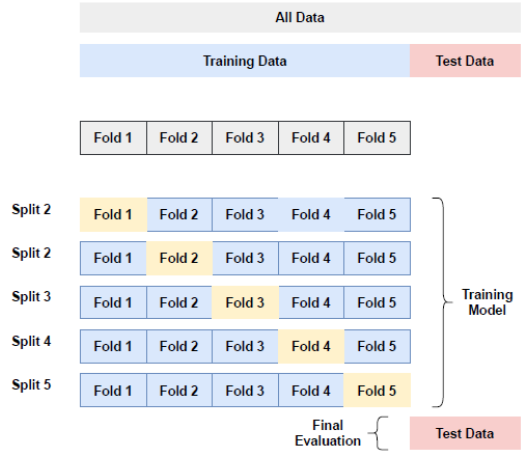
- Machine learning algorithms are categorized into Batch learning and Online learning, based on learning style.
- **D** typically represents the set of training examples, a set of ordered pairs of (features, labels).

$\boldsymbol{D} = \{(x^{(i)}, y^{(i)})\}$

- In a vectorized representation, we use matrix **X** to represent the input.

$\mathbf{X} = [feature^{(1)^T}, feature^{(2)^T}, \ldots, feature^{(n)^T}]$ and use vector **y** or matrix **Y** to represent the labels.

- *Training data* is used for training the model, *validation data* is used for tuning hyper-parameters (like regularization rate), and *test data* is used to evaluate the model performance.
- In cross-validation, we have several (say, *k*) sets of training-test data. Use (k - 1) partitions for training and 1 for validation. In an extreme case, k = n, each example in its own partition. Finally, compute average of evaluation metrics across *k* training/validation sets.



- Other techniques employed during preparing the data pre-processing.
  - Features are normalized to ensure that the convergence occurs faster. Discrete attributes are converted to numbers using one-hot encoding, hashing or embeddings.
  - Relationships between features and labels are examined using appropriate visualization techniques, or statistical methods like chi-square tests.
  - Data is subjected to cleansing (missing values) before using.
  - Class-imbalance is removed, by performing oversampling of minority classes.
- Example of a linear model, where there are m features is

  Output = $weight_0 + weight_1 * feature_1 + weight_2 * feature_2 + \ldots + weight_m * feature_m$
- More generically, a model is mathematically represented as h_w: X -> Y.
- Weight parameters are estimated from the training data. The set of weight parameters is called a weight vector.
- Weight parameters learnt from the training data *generally* works well with real-world data, since both sets of data are assumed to originate from the same distribution.
- When the output is a real number, we choose regression models. When the output is a discrete quantity, we choose classification models.
- Loss function is used to measure the difference between the predicted label and the actual label

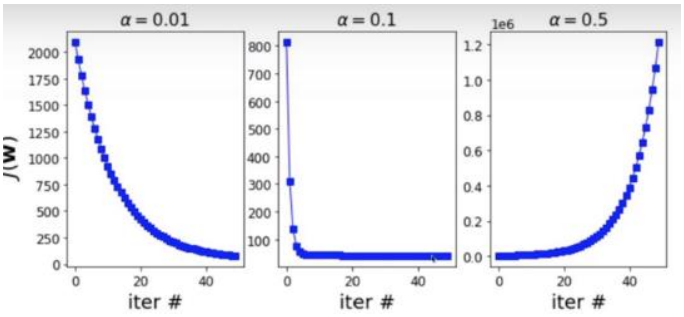
$$J(W) = \sum_{i=1}^{n} [predicted^{(i)} - actual^{(i)}]^2$$

- Loss is a function of the weight vector. This is mathematically represented as *J: W -> R*
- Loss function is *convex* in the case of linear/non-linear regression, support vector machines, and *non-convex* in the case of neural networks.
- Optimizing the loss function yields the optimal set of weights for generating the output closest to the actual, with minimal loss. This is mathematically represented as $W = argmin_W J(W)$
- Best way to reach the minimal value of loss is by equating derivative of loss to 0, mathematically represented as $\frac{d}{dW} J(W) = 0$
- Hence, one of the techniques used for optimization is (batch) gradient descent, where weight vector is updated in multiple short iterations. This can be mathematically represented as

$$w_i[t+1] \longleftarrow w_i[t] - \alpha \cdot \frac{\partial J(w)}{\partial w_i[t]}$$

In the above equation, [t+1] is the iteration that follow [t], α is the learning rate, and $\frac{\partial J(w)}{\partial w_i[t]}$ is the partial derivative of the loss with respect to the weight vector.

- To speed up the (batch) gradient descent process, we use mini-batch gradient descent, where we work on a small set of examples, instead of the entire dataset. Batch size is chosen to be a power of 2, to optimize the disk read/write during computations. In the case of stochastic gradient descent (SGD), we use a batch-size of 1.
- Learning curves are used to measure the efficacy of the learning process. It plots iterations on X-axis and loss on the Y-axis, and expects loss to steeply climb down. If the loss *isn't reducing (but, increasing)*, reduce the learning rate α and retry. If the loss *is reducing, but isn't reducing enough* in each iteration, increase the learning rate α and retry. Look at learning curves plotted for different values of α, in a linear regression problem.

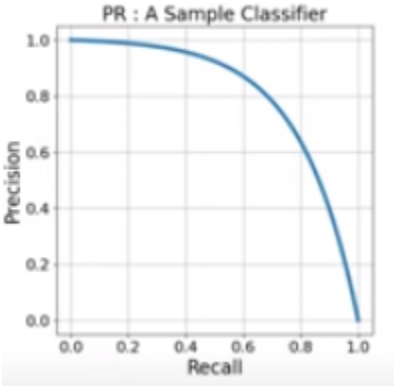


Note that in the first case (α=0.01), the loss reduces but not as quickly as in the second case (α=0.1). When α is too high at say 0.5, the loss increases instead.

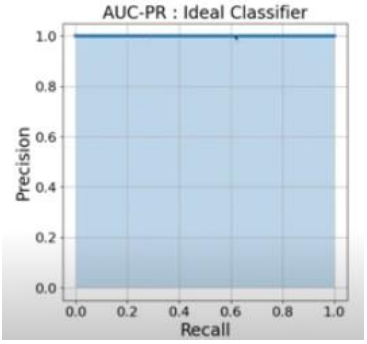
- When training and validation loss are low, it's the right fit. When training loss is low, and validation loss is high, it's an overfit. When the training and validation loss are high, it's an underfit.
- Overfitting can be avoided by learning from more data, or by using regularization (penalty)
- Underfitting can be avoided by increasing the model capacity by including the polynomial features, or by reducing regularization rate.
- Efficacy of the model can be measured by precision, recall, F1 score, AUC-ROC, AUC-PR. Accuracy isn't the best metric for measuring efficacy because it won't work well when there's class imbalance.
- *Confusion matrix* is used to calculate the above-mentioned metrics

Predicted / Actual	False(0)	True(1)
False(0)	True Negative (TN)	False Positive (FP)
True(1)	False Negative (FN)	True Positive (TP)

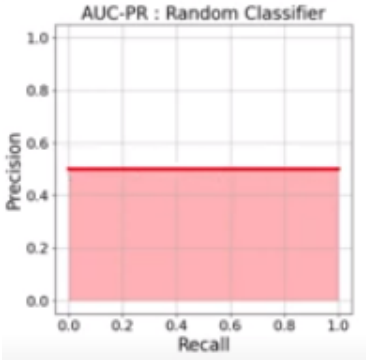
- Precision = TP / (TP + FP); Recall = TP / (TP + FN); Accuracy = (TP + TN) / (TP + TN + FN + FP)
- F1-Score = 2 * Precision * Recall/(Precision + Recall)
- PR curve can be produced by plotting the precision-recall values at different thresholds of probability.
- A typical PR curve looks like this



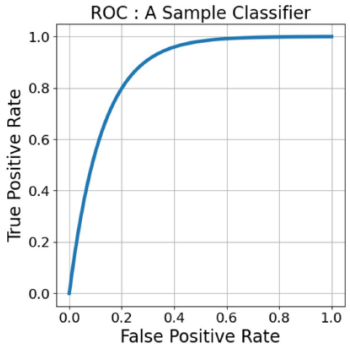
- PR curve for an ideal classifier looks like this.



- PR curve for a no-skills/random classifier with 0 class-imbalance looks like this. PR curve for a classifier with 70-30 class imbalance, has the line at 0.7 instead.



- AUC-PR is calculated by computing the total area under the PR curve. This is the preferred metric, when there's moderate to large class imbalance. In the ideal case, this area should be 1.
- ROC is plotted with False Positive Rate (FPR) on X-axis and True Positive Rate (TPR) on Y-axis, at different thresholds. Here's how it looks.

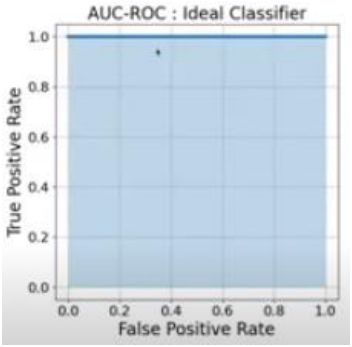


- Here's how FPR and TPR are calculated.

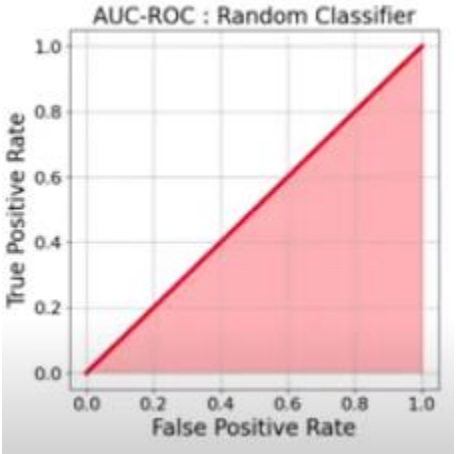
$$FPR = 1 - Specificity = \frac{FP}{FP + TN} \; ; \; TPR = Sensitivity = \frac{TP}{FN + TP}$$

NOTE: TPR is equal to Recall.

- ROC curve for an ideal classifier looks like this



- ROC curve for a no-skills/random classifier looks like this, and has an area of 0.5. Anything below this area is worse than the random classifier. AUC-ROC is preferred as a metric when there's no class imbalance.



Linear Regression and Gradient descent (intro)

- In the case of linear regression, input is (n x m) feature matrix comprising of n examples each with m features. Label matrix is n x 1, where each label is a scalar.
- Linear regression model is represented mathematically as $y = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_m x_m$, where the w_0, $w_1 \ldots w_m$ are the weights, and x_1, $x_2 \ldots x_m$ are the features. Since no feature is associated with w_0, we add a dummy feature x_0 with value 1. This is compactly represented as $y = \sum_{i=0}^{m} w_i x_i$. In a vectorized form, this can be rewritten as $\mathbf{y} = w^T \mathbf{x}$. In the case of multiple examples, this can be rewritten as $\mathbf{y} = \mathbf{X}w$, where $\mathbf{X}$ is a matrix with shape *n x (m + 1)*, w has a shape *(m + 1) x 1* and $\mathbf{y}$ has a shape *n x 1*
- In the case of a single feature, the geometry of the model is that of a line; with 2 features, it's a plane; in general, with m features, it's a hyper-plane with (m + 1) dimensions.
- Loss function (SSE) is represented mathematically as $J(W) = \sum_{i=1}^{n} [predicted^{(i)} - actual^{(i)}]^2$
- We usually divide the above value by a factor of 0.5, for mathematical convenience. This can be rewritten as $\frac{1}{2} \sum_{i=1}^{n} \left(\mathbf{w}^T \mathbf{x}^{(i)} - y^{(i)} \right)^2$ or as (in vectorized notation) $\frac{1}{2} \left(\mathbf{Xw} - \mathbf{y} \right)^T \left(\mathbf{Xw} - \mathbf{y} \right)$
- For fairly large number of training examples, the time of execution for vectorized form as compared to non vectorized form is less.
- Visualization of loss surface. As the yellow ball is dragged toward the bottom of the bowl-shaped loss surface, the predicted linear model matches the actual.
- Partial derivative of the loss is $\mathbf{X}^T \left(\mathbf{Xw} - \mathbf{y} \right)$. In the case of **normal equation** (fit), we'll equate this to zero, to give $w = X^{-1}y$.
- In the case of gradient descent, we'll start with an arbitrary weight vector, and keep updating the weights over multiple iterations (epochs). Weight update rule is given as $\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$ $:= \mathbf{w}_k - \alpha \mathbf{X}^T \left(\mathbf{Xw} - \mathbf{Y} \right)$, where α represents the learning rate. Note that the weight is updated with the product of (α, error, feature)
- Learning rate and number of iterations (epochs) are the two hyper-parameters.
- Lower learning rates takes much longer to reach the optimum loss, but very high rates could cause the loss to increase. Use learning curves to find the optimum learning rate.
- Number of iterations must be high enough so that lowest loss could be reached, and must be low enough so that computational cycles aren't wasted after reaching the lowest loss.
- In Gradient descent algorithm, weights are updated simultaneously for all examples in the data. But, in mini-batch gradient descent, weights are updated simultaneously for a small subset of examples (in a mini-batch) from the data. In stochastic descent algorithm, weights are updated simultaneously only for single example at a time from the data. Thus, GD performs 1 weight update for all examples, MBGD performs n/k times (where n is the total number of examples and k is the mini-batch size), and SGD performs n times.

- Evaluation is done using the root-mean square error (RMSE) measure. RMSE is calculated as $\sqrt{\frac{2}{n} \cdot J(w)}$

- When you fit polynomial data, say, generated by Sin(2πx) using linear model, it underfits. In this case, we'll need to make a *polynomial transformation* of the single input feature x to the k^{th} degree, and **generate** new features x^2, x^3, …x^k. Note that k is an arbitrary number. Once we've *k* features, we'll use these features to fit into a linear model. This will *closely approximate* the polynomial fit of the single original feature.
- Denoting these transformations to the input feature using the label ϕ, we can mathematically represent the fitment as $y = \mathbf{w}^T \phi(\mathbf{x})$
- Lower degree polynomial transformations tend to underfit, while higher degree tend to overfit.
- Here's a pictorial representation of the process of *generating* the polynomial features.

Input features	[[1,2], [3,4]]			**Input features**	[[2, 3], [4, 5]]	
Column-wise	[[1,3], [2,4]]			**Column-wise**	[[2,4], [3,5]]	
Combinations				**Combinations**		
Degree	**Combinations**	**Dot Product**		**Degree**	**Combinations**	**Dot Product**
0	([1,1])	[1,1]		0	([1,1])	[1,1]
1	([1,3])	[1,3]		1	([2,4])	[2,4]
1	([2,4])	[2,4]		1	([3,5])	[3,5]
2	([1,3], [1,3])	[1,9]		2	([2,4],[2,4])	[4,16]
2	([2,4],[2,4])	[4,16]		2	([3,5],[3,5])	[9,25]
2	([1,3], [2.4])	2, 12		2	([2,4],[3,5])	[6,20]
				3	([2,4],[2,4],[2,4])	[8,64]
				3	([3,5],[3,5],[3,5])	[27,125]
				3	([2,4],[3,5],[3,5])	[18,100]
				3	([3,5],[2,4],[2,4])	[12,80]

- In order to find the number of features after applying a 4^{th} degree polynomial transformation to ***n*** data samples with 5 features?

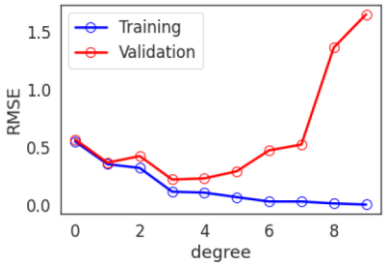
Formula to calculate the number of features including the dummy feature is as follows: If there are n features, while fitting a N^{th} degree polynomial, the number of features after transformation is

$$= \frac{(N+1)(N+2)...(N+n)}{n!}$$

$$= \frac{(4+1)(4+2)(4+3)(4+4)(4+5)}{5!}$$

$$= \frac{5*6*7*8*9}{5*4*3*2} = 126$$

- RMSE vs degree plots are typically used to identify overfit/underfit tendency. Thus, in following graph, the model tends to overfit more than degree 6.



- In the case of higher degree polynomials, the weights tend to be arbitrarily large for the features. This is a symptom of overfit. This is also called high variance problem.
- In order to avoid overfit, use larger training sets, or use regularization to control the model complexity.
- Regularization process adds a penalty to the loss, leading to a change in its derivative, and hence the weight update.
- Regularization is controlled by two terms – the penalty function (function of weight vector) and the regularization rate.
- 3 types of regularization are used in the industry – L1 (Lasso), L2 (Ridge), and a combination of L1 and L2.
- Ridge regularization uses the second norm of the weight vector, and takes the following vectorized form

$$J(\mathbf{w}) = \frac{1}{2}(\mathbf{X}\mathbf{w} - \mathbf{y})^T(\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

- Derivative of the loss after applying the ridge regularization is $\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\mathbf{w}$.

Equating this to 0, the normal equation takes the form $\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$

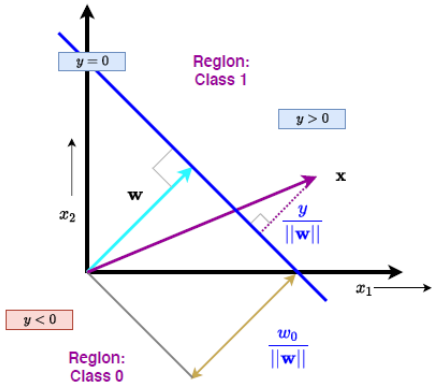
- In the case of gradient descent, we'll start with an arbitrary weight vector, and keep updating the weights over multiple iterations (epochs). Weight update rule is given as $\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$ $:= \mathbf{w}_k - \alpha\{\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\mathbf{w}\}$, where α represents the learning rate.
- With increasing rates of regularization λ, the overfit reduces; but beyond a certain threshold of the λ, it causes the model to underfit.
- To find out the right value of λ, train the model and for each value of λ, calculate the cross-validation error (loss or rmse) on validation set. The regularization rate that gives the least cross-validation error is the right choice of λ.
- λ Vs cross-validation error typically takes a bowl shape when plotted. Most appropriate value of the λ is at the lowest point of the bowl.
- Lasso regularization uses the first norm of the weight vector, and takes the following form.

$$J(\mathbf{w}) = \frac{1}{2}(\mathbf{X}\mathbf{w} - \mathbf{y})^T(\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{\lambda}{2}\sum_{j=1}^{m}|w_j|$$

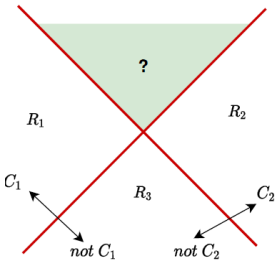
- Lasso regression assigns sparse weights. It assigns non zero weights to only important features and zero weights to unimportant features.
- Since the above form is not differentiable, we use special methods (from sklearn library) to carry out this process.

- Class label is a discrete quantity, unlike a real number in the regression setup.
- Under single-label classification (aka *binary/multi-class classification*), each example gets assigned with a single label, whereas under *multi-label classification*, each example gets assigned with multiple labels. Note that multi-class, multi-label problems are typically referred to simply as *multi-label* problems.

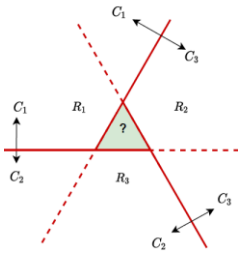
- Classification methods include
    a. Discriminant functions that learn direct mapping between feature matrix x and labels.
    b. Generative models that learn conditional probability distribution P(y|x) using Bayes theorem and prior probabilities of classes, and assign labels based on that.
    c. Discriminative models that use parameters to build models based on conditional probability, and assign labels on that.
    d. Instance-based models that compare training and test examples, and assign class labels based on certain measures of similarity.
- Multi-class (single-label) and multi-label classifications use one-hot encoding to represent classes assigned to an example. While in the former case, exactly one entry in each row is 1, in the latter, multiple entries on each row could be 1. In the case of binary classification, the classes are represented as -1 and +1 and doesn't use any encoding.
- In the case of binary-class classification, feature matrix X is represented by *(n x m)* matrix, weights by *(m x 1)* vector and labels by *(n x 1)* vector. In the case of multi-class (and multi-label) classification (with k classes), feature matrix X is represented by *(n x m)* matrix, weights by *(m x k)* matrix and labels by *(n x k)* matrix.
- Discriminant function has the same mathematical representation as the linear regression $\mathbf{y = w_0 + w^T x}$, where $\mathbf{w_0}$ is the bias. It has the geometry of a hyper-plane with *(m-1)* dimensions, where *m* is the number of features.
- The decision boundary of the classes class-0 and class-1 is y = 0. When y > 0, it belongs to the class-1, else it belongs to the class-0.
- The weight vector is orthogonal to every vector lying within the decision surface; hence it determines the orientation of the decision surface. The location of the decision boundary is $w_0/||w||$ and y is the signed measure of the perpendicular distance of the point x from the decision surface. This is represented in the figure below.



- When there are multiple classes (>2), we could use One-Vs-rest (where we build *k-1* discriminant functions) or one-Vs-one (where we build kC2 discriminant functions), both of which have their faults.

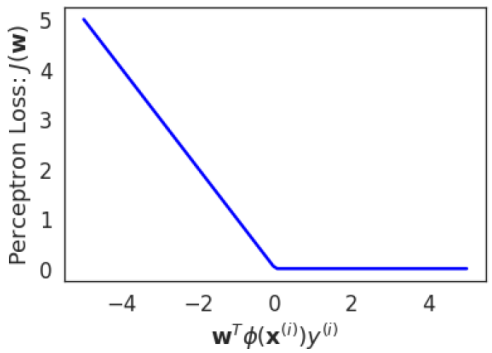


(One Vs Rest)



(One Vs One)

- Hence, we develop a single k-class discriminant function that carries the form $y_k = w_{k0} + \mathbf{w_k}^T\mathbf{x}$ (Note that subscript 0 represents the bias). In the case of k-class discriminant function, the correct classification is done using majority vote. Thus, label y_k is assigned to an example, if $y_k > y_j$ for all $j \neq k$
- To learn parameters of model, we may use *Least Squares Classification* or *Perceptron*.
- Calculations of loss, optimization and weight update rule in the case of *Least Squares Classification* remains the same as in the case of Linear Regression, which computes J(w) = $\frac{1}{2}(\mathbf{Xw}-\mathbf{y})^T(\mathbf{Xw}-\mathbf{y})$; w = $X^{-1}y$ is the normal equation; and $\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} := \mathbf{w}_k - \alpha\mathbf{X}^T(\mathbf{Xw}-\mathbf{Y})$ is the weight update rule. The equations used in the case of regularization also remain unchanged.
- From a code perspective, the only change required is to change the *predict* function from `return X @ self.w` to `return np.argmax(X @ self.w, axis=-1)`. However, note that the *loss* and *optimization* (*calculate_gradient* method) procedures use *predict_internal* method that continues to use `return X @ self.w`
- Perceptron, a basic classification algorithm was invented by Frank Rosenblatt in 1958.
- Perceptron can solve only binary classification problems, and hence label matrix has n x 2 shape. Labels can be {-1,+1}
- Perceptron model can be mathematically represented as $y = \text{sign}(\mathbf{w}^T\phi(\mathbf{x}))$, where w is the weight vector, and **X** is the transformed feature matrix**.** Hence, this is essentially a step function. Generalizing this, $y = f(\mathbf{w}^T\phi(\mathbf{x}))$ is the equation used to represent the model, where *f* is a non-linear activation function. Note that $\mathbf{w}^T\phi(\mathbf{x})$ is sometimes written as $h_w(\mathbf{X})$
- Loss of each example in the perceptron model can be represented as $J(\mathbf{w}) = \sum_{i=1}^{n}\max(0, -y^{(i)}h_{\mathbf{w}}(x^{(i)}))$. Losses for all examples are summed up to get the cumulative loss; Loss function is not differentiable and can be geometrically represented as



- In the misclassified region, loss is a linear function of the weight vector, and can be reduced by reducing the weight vector. In the correctly classified region, leave the weight vector unchanged.
- While linearly separable examples lead to zero loss ultimately and convergence of the algorithm, non-separable examples lead to oscillating loss and hence don't converge.
- Similar to the loss, the weight update rule for each example can be represented as $\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} + \alpha\left(y^{(i)} - \hat{y}^{(i)}\right)\phi(\mathbf{x}^{(i)})$, and must be summed up to get the cumulative weight update. This is similar to weight update rule in all models where weights are updated with the product of (α, error, feature).
- Models learnt in week1-4 can be summarized using the following table:

	Model	Loss [J(w)]	Opt-Normal equation (w)	Opt-Weight update
Linear Reg	$\mathbf{w}^T\mathbf{x}$	$\frac{1}{2}(\mathbf{Xw}-\mathbf{y})^T(\mathbf{Xw}-\mathbf{y})$	$\mathbf{X}^{-1}\mathbf{y}$	$\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha\mathbf{X}^T(\mathbf{Xw}-\mathbf{Y})$
Poly Reg	$\mathbf{w}^T\phi(\mathbf{x})$	$\frac{1}{2}(\mathbf{Xw}-\mathbf{y})^T(\mathbf{Xw}-\mathbf{y})$	$\mathbf{X}^{-1}\mathbf{y}$	$\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha\mathbf{X}^T(\mathbf{Xw}-\mathbf{Y})$
Ridge		$\frac{1}{2}(\mathbf{Xw}-\mathbf{y})^T(\mathbf{Xw}-\mathbf{y}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$	$(\mathbf{X}^T\mathbf{X}+\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$	$\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha\{\mathbf{X}^T(\mathbf{Xw}-\mathbf{y}) + \lambda\mathbf{w}\}$
Lasso		$\frac{1}{2}(\mathbf{Xw}-\mathbf{y})^T(\mathbf{Xw}-\mathbf{y}) + \frac{\lambda}{2}\sum_{j=1}^{m}\lvert w_j\rvert$	Use sklearn	Use sklearn
Least squares		$\frac{1}{2}(\mathbf{Xw}-\mathbf{y})^T(\mathbf{Xw}-\mathbf{y}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$	$(\mathbf{X}^T\mathbf{X}+\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$	$\mathbf{w}_{k+1} := \mathbf{w}_k - \alpha\{\mathbf{X}^T(\mathbf{Xw}-\mathbf{y}) + \lambda\mathbf{w}\}$
Perceptron	$\text{sign}(\mathbf{w}^T\phi(\mathbf{x}))$	$\sum_{i=1}^{n}\max(0, -y^{(i)}h_{\mathbf{w}}(x^{(i)}))$	$(\mathbf{X}^T\mathbf{X}+\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$	$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} + \alpha\left(y^{(i)} - \hat{y}^{(i)}\right)\phi(\mathbf{x}^{(i)})$, [For one example, sum up for all such examples]

- Logistic Regression is an important building block in the construction of Neural Networks.
- It is a *discriminative* classifier, and can be applied for binary/multi-class/multi-label classifications.
- It obtains probability of sample belonging to a specific class by computing sigmoid (aka logistic function) of linear combination of features.
- The training model can be mathematically represented as

$D = \{(\mathbf{X}, \mathbf{y})\} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$ (binary classification)

where **X** is the (n x m) matrix containing training samples, **y** is a (n,) size label vector, and $y^{(i)}$ is a scalar

Or

$D = \{(\mathbf{X}, \mathbf{Y})\} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{n}$ (multi-class/multi-label classification)

where **X** is the (n x m) matrix containing training samples, **Y** is the (n x k) label matrix and $\mathbf{y}^{(i)}$ is a (k,) size label vector

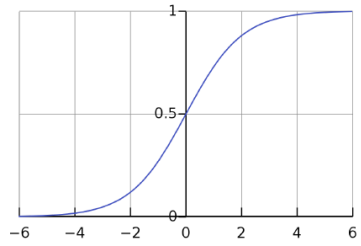
- In the case of binary classification, for each feature vector **x,** the probability of the label y belonging to class-1, is given by the following equation, where **g** represents the sigmoid function.

$$\Pr(y = 1|\mathbf{x}) = g(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$$

- In the case of samples that're not linearly separable, we'll use the following formula instead. Here ϕ represents the transformation, say polynomial based. This leads to many more parameters, and thus more #weights.

$$\Pr(y = 1|\mathbf{x}) = g(\mathbf{w}^T\phi(\mathbf{x})) = \frac{1}{1+\exp(-\mathbf{w}^T\phi(\mathbf{x}))}$$

- A sigmoid is graphically represented as follows, where X axis is $z = \mathbf{w}^T\mathbf{x}$ and Y axis is the probability. Thus, as z tends to +∞, g(z) tends to 1; as z tends to -∞, g(z) tends to 0; when z=0, g(z)=0.5.



- In the case of multi-class classification, *softmax* function is used instead of *sigmoid*.
- Thus, in the case of logistic regression, the learning problem is to estimate the weight vector based on the training data by minimizing the loss function through appropriate optimization procedure.
- In the case of a binary classification problem, for one sample x, the probability is represented using a single equation as follows

$\Pr(y|\mathbf{x};\mathbf{w}) = (h_\mathbf{w}(\mathbf{x}))^y(1 - h_\mathbf{w}(\mathbf{x}))^{(1-y)}$, where $h_\mathbf{w}(\mathbf{x}) = \Pr(y = 1|\mathbf{x};\mathbf{w})$

- In the case of n samples, we can write this as

$$\prod_{i=1}^{n} \left(h_\mathbf{w}(\mathbf{x}^{(i)})\right)^{y^{(i)}} \left(1 - h_\mathbf{w}(\mathbf{x}^{(i)})\right)^{1-y^{(i)}}$$

- Negative log likelihood (cross-entropy loss) is often used to represent the loss, and written as

J(w) = $= -\sum_{i=1}^{n} y^{(i)} \log\left(h(\mathbf{x}^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - h(\mathbf{x}^{(i)})\right)$

- With L2 regularization, the binary cross-entropy loss is represented as

$$J(\mathbf{w}) = -\sum_{i=1}^{n} y^{(i)} \log\left(h(\mathbf{x}^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - h(\mathbf{x}^{(i)})\right) + \frac{\lambda}{2}||\mathbf{w}||^2$$

- With L1 regularization, the binary cross-entropy loss is represented as

$$J(\mathbf{w}) = -\sum_{i=1}^{n} y^{(i)} \log\left(h(\mathbf{x}^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - h(\mathbf{x}^{(i)})\right) + \frac{\lambda}{2}||\mathbf{w}||$$

- In order to optimize the loss, we could use a gradient descent $\mathbf{w} := \mathbf{w} - \alpha \frac{\partial}{\partial \mathbf{w}} (J(\mathbf{w}))$, which when substituted with the derivative of the loss function is rewritten as follows.

$$w_j := w_j - \alpha \left(\sum_{i=1}^{n} \left(h_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)}\right) x_j^{(i)}\right)$$

Note that the weight vector combines the individual w_js (for each feature)

- The above can be vectorized using the following equation:

$$\mathbf{w} := \mathbf{w} - \alpha \left(\mathbf{X}^T \left(h_{\mathbf{w}}(\mathbf{X}) - \mathbf{y}\right)\right) := \mathbf{w} - \alpha \left(\mathbf{X}^T \left(g(\mathbf{X}\mathbf{w}) - \mathbf{y}\right)\right)$$

NOTE: This is similar to weight update rule in all models where weights are updated with the product of (α, error, feature).

- Inference can be represented mathematically as

$$y = \begin{cases} 1, & \text{if } \Pr(y = 1|\mathbf{x}) > 0.5 \\ 0, & \text{otherwise.} \end{cases}$$

- Naïve Bayes classifier is a *generative* counter-part of logistic regression.
- It uses Bayes theorem to calculate the probability of a sample belonging to a class, while assuming that the features are conditionally independent, given a label. This is mathematically represented as follows.

$$p(x_1, x_2, \ldots, x_m|y) = p(x_1|y)\, p(x_2|y) \ldots p(x_m|y) = \prod_{j=1}^{m} p(x_j|y)$$

- It's used in document classification and spam filtering.
- The training model can be mathematically represented as

$D = \{(\mathbf{X}, \mathbf{y})\} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$ (binary classification)

where **X** is the (n x m) matrix containing training samples, **y** is a (n,) size label vector, and $y^{(i)}$ is a scalar

Or

$D = \{(\mathbf{X}, \mathbf{Y})\} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{n}$ (multi-class/multi-label classification)

where **X** is the (n x m) matrix containing training samples, **Y** is the (n x k) label matrix and $\mathbf{y}^{(i)}$ is a (k,) size label vector

NOTE: This is the same as in the case with logistic regression

- Probability that the sample belongs to class *yc* is given by the following formula.

$$\frac{p(y_c) \prod_{j=1}^{m} p(x_j|y_c)}{\sum_{r=1}^{k} p(y_r) \prod_{j=1}^{m} p(x_j|y_r)}.$$

NOTE: *xj* represents each feature in the input sample x; *yr* represents each of the multiple classes. Also note that there are ***k*** prior probabilities represented as *p(yr)*, ***m*** conditional densities per class (and there are k such classes) represented as *p(xj|yr)*. Terms in the numerator is one of these ***k*** probabilities.

- Sum of all prior probabilities is equal to 1.
- The parameters for the model depends on the class distribution used to model the situation.
  a. If each sample has only 2 (binary) features, we'll use Bernoulli distribution, in which we'll use the *mean* as parameter. $p(x_j|y_c) \sim \text{Bernoulli}(\mu_{jc}) = \mu_{jc}^{x_j}(1-\mu_{jc})^{(1-x_j)}$
  b. If each sample has *e* features, we'll use Categorical distribution, in which we'll use *e* and *mean* for each category as parameters.

$$p(x_j|y_c) \sim \text{Cat}(e, \mu_{j1c}, \mu_{j2c}, \ldots, \mu_{jec}) = \mu_{j1c}^{1(x_j=v_1)}\mu_{j2c}^{1(x_j=v_2)}\ldots\mu_{jec}^{1(x_j=v_e)}$$

  c. If each sample has continuous features, we'll use Normal distribution, in which we'll use *mean* and *variance* as parameters. $p(x_j|y_c) \sim \mathcal{N}(\mu_{jc}, \sigma_{jc}) = \frac{1}{\sqrt{2\pi}\sigma_{jc}} e^{-\frac{1}{2}(\frac{x_j-\mu_{jc}}{\sigma_{jc}})^2}$ in the case of a 1D distribution, and $= \frac{1}{\sqrt{(2\pi)^m|\Sigma|}}\exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right)$ otherwise, where ∑ is the covariance matrix.
  d. If each sample has multinomial features where the sequence length is l, we'll use multinomial distribution, in which we'll use l, *mean* and *variance* for each feature as parameters. $p(\mathbf{x}|y_c) \sim \text{Multinomial}(l, \mu_{1c}, \mu_{2c}, \ldots, \mu_{mc}) = \frac{n!}{x_1!\ldots x_m!}\prod_{j=1}^{m}\mu_{jc}^{x_j}$
- In order to calculate the loss, first compute the likelihood of the observed data, given weight w.

$$L(\mathbf{w}) = p(D;\mathbf{w}) = p(\mathbf{X},\mathbf{y};\mathbf{w}) = \prod_{i=1}^{n} p(\mathbf{x}^{(i)}, y^{(i)};\mathbf{w})$$

.

Taking log on both sides, we've the log likelihood represented as

$$l(\mathbf{w}) = \sum_{i=1}^{n} \log\left(p(\mathbf{x}^{(i)}, y^{(i)};\mathbf{w})\right)$$

and can be rewritten as

$$l(\mathbf{w}) = \sum_{i=1}^{n} \log p(y^{(i)};\mathbf{w}) + \sum_{i=1}^{n}\sum_{j=1}^{m} \log p(x_j^{(i)}|y^{(i)};\mathbf{w})$$

NOTE: Take the negative of this quantity to get the J(w)

- Thus, $\max l(w) \equiv \max \sum_{i=1}^{n} \log\left(P(x^{(i)} \mid y^{(i)})\right)$
- Prior is the ratio of the number of labels that match the given class from the training dataset, and represented mathematically as

$$P(y=c) = \frac{\sum_{i=1}^{n} 1(y^{(i)}=c)}{n}$$

- In the case of a Gaussian Naïve Bayes model, the parameters are:
  a. Mean $\hat{\mu_k} = \frac{1}{N_k}\sum_{i=1}^{N} 1(y^{(i)}=y_k)\mathbf{x_i}$

b. Variance $\hat{\text{var}}_k = \frac{1}{N_k}\sum_{i=1}^{N} 1(y^{(i)} = y_k)(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k) \odot (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k)$

NOTE: N_k represents the number of data points that belong to class ***k***

- Inference is represented mathematically as $y_{\text{pred}} = \underset{c \in \{1,\cdots,k\}}{\arg\max} \quad P(c) \cdot P(x \mid y = c)$

Here is a summary of the implementation, for each of the class distributions.

Bernoulli NB

Posterior = (Prior * Likelihood) / Evidence

Explanation:

Likelihood is given as $\prod_{j=1}^{m} p(x_j|y_c)$

So, posterior probability,

$$p(y = y_c|\mathbf{x}) = \frac{p(y_c)\prod_{j=1}^{m} p(x_j|y_c)}{\sum_{r=1}^{k} p(y_r)\prod_{j=1}^{m} p(x_j|y_r)}$$

In code, this is the method named **_predict_proba**

predict_proba = exp(log_likelihood_prior_prod) / sum(log_likelihood_prior_prod))

Numerator is calculated as follows:

log_likelihood_prior_prod = fit + _calc_pdf

Explanation: Label ***y*** that results in the highest value for numerator is assigned to the given example.

Thus, $y = \text{argmax}_y \log p(y) + \sum_{j=1}^{m} \log p(x_j|y)$.

First part of the equation is the fraction of examples with label c among all training examples.

In code, the first part is **self.w_priors** calculated by method **fit;** the second part of the equation is calculated by method **_calc_pdf**

Instead of using the **_calc_pdf** method**,** it's hardcoded as

X @ (np.log(self.w).T) + (1 - X) @np.log((1 - self.w).T)

and represented mathematically as

$$\mathbf{X} \log \mathbf{w}^T + (1 - \mathbf{X}) \log (1 - \mathbf{w})^T + \log \mathbf{w}_{\text{prior}}$$

The denominator is calculated as follows:

Explanation: Converting log back into probability, we have

$$p(\mathbf{x}, y_r) = \exp(\log p(\mathbf{x}, y_r))$$

Summing up these probabilities across all classes, we get

$$p(\mathbf{x}) = \sum_{r=1}^{k} p(\mathbf{x}, y_r) = \sum_{r=1}^{k} \exp(\log p(\mathbf{x}, y_r))$$

In code, this is **sum(exp(log_likelihood_prior_prod))**

Prediction is done using **argmax(log_likelihood_prior_prod)**

Gaussian NB

Posterior = (Prior * Likelihood) / Evidence

Explanation:

Likelihood is given as $\prod_{j=1}^{m} p(x_j|y_c)$

$$p(y = y_c|\mathbf{x}) = \frac{p(y_c) \prod_{j=1}^{m} p(x_j|y_c)}{\sum_{r=1}^{k} p(y_r) \prod_{j=1}^{m} p(x_j|y_r)}$$

In code, this is the method named **_predict_proba**

_predict_proba = exp(_calc_prod_likelihood_prior) / sum(exp(_calc_prod_likelihood_prior))

Numerator is calculated as follows:

_calc_prod_likelihood_prior = fit + _calc_pdf

Explanation: Label ***y*** that results in the highest value for numerator is assigned to the given example.

Thus, $$y = \text{argmax}_y \log p(y) + \sum_{j=1}^{m} \log p(x_j|y)$$.

The first part of the equation is the fraction of examples with label c among all training examples.

In code, the first part is **self._priors** calculated by method **fit;** the second part of the equation is calculated by method **_calc_pdf**

The denominator is calculated as follows:

Converting log back into probability, we have $p(\mathbf{x}, y_r) = \exp(\log p(\mathbf{x}, y_r))$

Summing up these probabilities across all classes, we've $p(\mathbf{x}) = \sum_{r=1}^{k} p(\mathbf{x}, y_r) = \sum_{r=1}^{k} \exp(\log p(\mathbf{x}, y_r))$

In code, this is **sum(exp(_calc_prod_likelihood_prior))**

Prediction is done using **argmax((_calc_prod_likelihood_prior)**

- Regression and Classification models are special cases of broader family of models called *Generalized Linear Models (GLMs)* and can be represented mathematically as

$$p(y;\eta) = b(y)\exp(\eta^T T(y) - a(\eta))$$

In the above formula, ***η*** is the natural or canonical parameter of distribution, ***T(y)*** is the sufficient statistic (in most cases, it's equal to y) and ***a*** is the log partition function. The quantity $e^{-a(\eta)}$ essentially plays the role of a *normalization constant,* that makes sure that the distribution $p(y;\eta)$ sums/integrates over y to 1.

- Many including Bernoulli, Gaussian, Multinomial, Poisson, Gamma, Exponential, Beta, Dirichlet distributions can be written in the above form, with appropriate values for the parameters.
- To derive GLM for the above distribution, we make the following assumptions or design choices:
  a. $y|\mathbf{x};\mathbf{w} \sim \text{ExponentialFamily}(\eta)$
  b. $h(\mathbf{x}) = E[y|\mathbf{x}]$ where E represents the expected value of the sufficient statistic $T(y) = y$
  c. Linear relationship between ***η*** and ***x;*** $\eta = \mathbf{w}^T\mathbf{x}$

For example, in the case of ordinal least squares modelled as a Gaussian distribution $y|\mathbf{x};\mathbf{w} \sim \mathcal{N}(\mu, \sigma^2)$, where μ = η

$$\begin{aligned} h_{\mathbf{w}}(\mathbf{x}) &= E[y|\mathbf{x};\mathbf{w}] && \text{(by Assumption 2)} \\ &= \mu && \text{(Since , } y|\mathbf{x} \sim N(\mu,\sigma^2)) \\ &= \eta && \text{(by Assumption 1)} \\ &= \mathbf{w}^T\mathbf{x} && \text{(by Assumption 3)} \end{aligned}$$

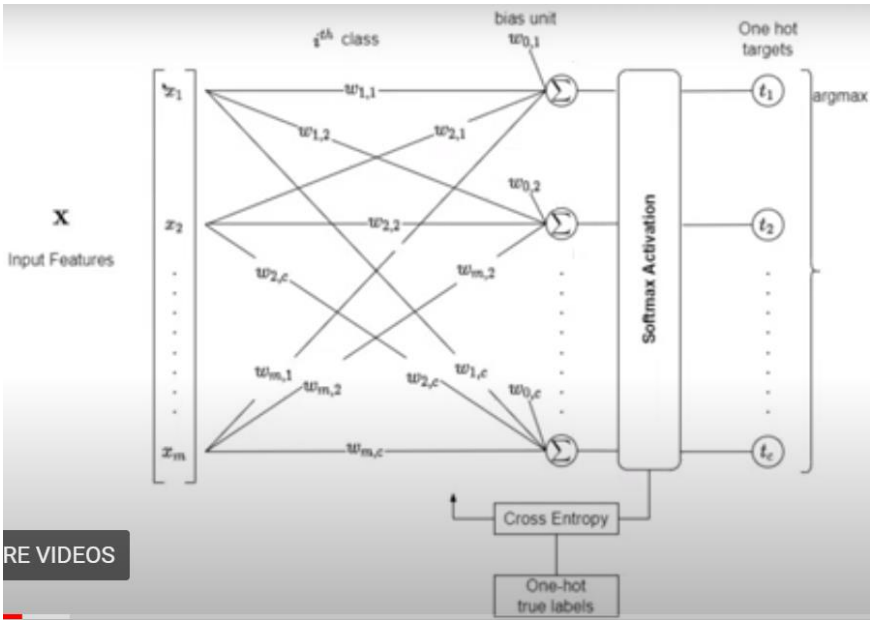
- Softmax function is represented mathematically as

$$\phi_i = \frac{e^{\eta_i}}{\sum_{j=1}^{k} e^{\eta_j}}$$

- Softmax regression is often to used to model a multi-class, multi-label classification problem, and can be represented mathematically as follows:

$$h_{\mathbf{w}}(\mathbf{x}) = E[T(y)|\mathbf{x};\mathbf{w}]$$
$$= \begin{bmatrix} \frac{\exp(\mathbf{w}_1^T\mathbf{x})}{\sum_{j=1}^{k}\exp(\mathbf{w}_j^T\mathbf{x})} \\ \frac{\exp(\mathbf{w}_2^T\mathbf{x})}{\sum_{j=1}^{k}\exp(\mathbf{w}_j^T\mathbf{x})} \\ \vdots \\ \frac{\exp(\mathbf{w}_{k-1}^T\mathbf{x})}{\sum_{j=1}^{k}\exp(\mathbf{w}_j^T\mathbf{x})} \end{bmatrix}$$

- Here's a pictorial representation of softmax regression.



- In order to construct the model, use the following steps:
  a. Linear combination: $Z_{nxk} = X_{nxm}\, W_{mxk} + b_{kx1}$
  b. Softmax activation: $\exp(Z) / \sum\exp(Z)$

NOTE1: Sum of softmax across all classes for a given sample is 1.
NOTE2: The class label with the highest value of softmax is assigned to the sample.

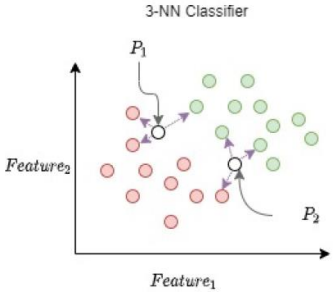
Now, the weight update rule is given by the following formulae:

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \alpha\frac{\partial J(\mathbf{W},\mathbf{b})}{\partial \mathbf{W}} \leftarrow \mathbf{W}_t - \alpha\,\mathbf{X}^T(\hat{\mathbf{Y}} - \mathbf{Y})$$
$$\mathbf{b}_{t+1} \leftarrow \mathbf{b}_t - \alpha\frac{\partial J(\mathbf{W},\mathbf{b})}{\partial \mathbf{b}} \leftarrow \mathbf{b}_t - \alpha\,\hat{\mathbf{Y}} - \mathbf{Y}$$

Watch https://www.youtube.com/watch?v=8ps_JEW42xs for a lucid explanation about softmax regression.

- K-nearest neighbors (KNN) is a supervised learning algorithm that can be used with both regression and classification tasks. It's an instance based technique.

- KNN compares a new example with existing training examples, obtains nearest neighbors and assigns an output label based on their labels.
- Two hyper-parameters = #neighbors, distance metric (Euclidean/Manhattan)
- Consider the following pictorial representation of datapoints.



In the above picture, for P1, 2 out of 3 neighbors are red, therefore, it is predicted to be in class *Red*. Similarly, for P2, 2 out of 3 neighbors are green, therefore, it is predicted to be in class *Green.*

- Euclidean distance in vectorized format is represented mathematically as follows:

$$\left(\left(\mathbf{x}^{(1)}-\mathbf{x}^{(2)}\right)^{T}\left(\mathbf{x}^{(1)}-\mathbf{x}^{(2)}\right)\right)^{\frac{1}{2}}$$

NOTE: x^1 and x^2 are vectors with m components each.

- Manhattan distance in vectorized format is represented mathematically as follows:

$$\sum_{j=1}^{m}\left|x_j^{(1)}-x_j^{(2)}\right|$$

- For classification task, the neighbors take part in voting. The class that receives highest number of votes is the predicted class
- For regression task, the output/prediction is calculated as average of the outputs/labels of k neighbors.

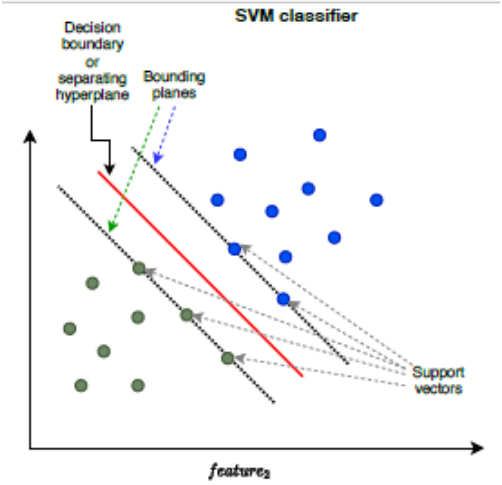
$$\hat{y}=\frac{1}{k}\sum_{i=1}^{k}y_i$$

- If #neighbors ***k*** is too small, the decision boundary will be jagged and will result in overfit. As ***k*** increases, the decision boundary gets smoother.
- If #neighbors ***k*** is too large, the model will be biased and will result in underfit.
- As #neighbors ***k*** comes close to total number of points in the dataset, the model will predict label of majority class for all samples.
- Advantages of KNN model:
  a. Quite easy to understand and implement the algorithm.
  b. The output of a prediction can be explained based on its neighbors. This adds to interpretability of the K-NN model.
- Disadvantages of KNN model:
  a. For large training set, K-NN can be time consuming, since all computations are performed at runtime.
  b. K-NN is sensitive to redundant or irrelevant features since all features are used to compute distance between two points.

c. On significantly difficult tasks, it can be out performed by other techniques such as SVM, Neural Networks.

SVM

- SVM is a supervised ML algorithm that can be used for both classification and regression tasks
- It's a discriminative classifier (like perceptron and logistic regression) and works for both binary and multi-class/multi-label classification.
- Model is represented as $y = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$; here the label ***y*** is assumed to be +1 or -1
- The separating hyperplane (in red) is the classifier. It's at equal distance from both classes and separates them such that there's maximum margin between the two classes.

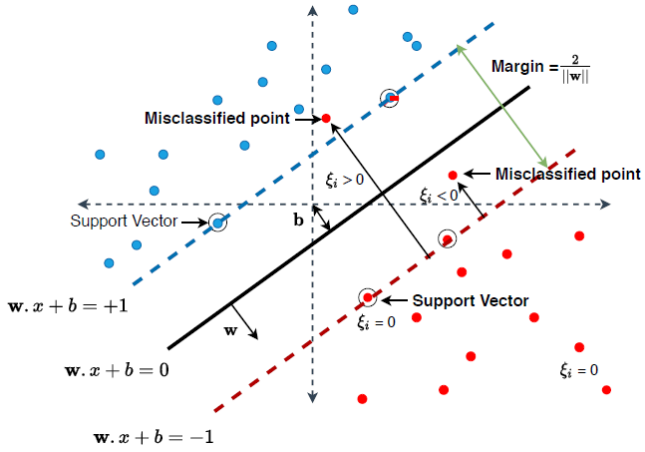


Let's look initially at *hard-margin SVM*, wherein the classes are assumed to be linearly separable.

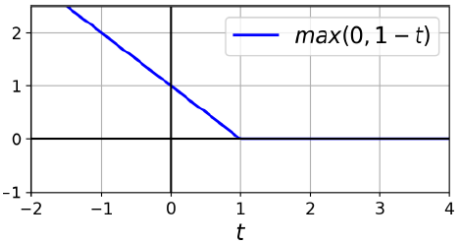
- Bounding planes are parallel to separating hyperplanes on the either sides and pass through support vectors.
- Support vectors are subset of training points closer to the separating hyperplane and influence its position and orientation. All support vectors are points on the bounding planes.
- Bounding planes can be represented mathematically as $y(\mathbf{w}^T\mathbf{x} + b) = 1$
- All correctly classified points are represented mathematically as $y(\mathbf{w}^T\mathbf{x} + b) \geq 1$
- Separating hyperplane can be represented mathematically as $\mathbf{w}^T\mathbf{x} + b = 0$
- Width between the bounding planes (margin) is given by the formula $\frac{2}{\|\mathbf{w}\|}$
- Optimization problem of linear SVM is maximize the above quantity, or (alternatively) minimizing w. This is written mathematically as $\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$. This is called the *primal problem* and is guaranteed to have a global minimum. It can be converted into its dual using Lagrange multipliers, $\sum_{i=1}^{n} \alpha^{(i)} - \frac{1}{2}\sum_{i=1}^{n}\sum_{k=1}^{n}\langle \alpha^{(i)} y^{(i)} x^{(i)}, \alpha^{(k)} y^{(k)} x^{(k)} \rangle$, which is guaranteed to have a global maximum. This implies that the dual problem depends on the inner product of the training data.

NOTE: ***α*** is the Lagrange multiplier.

- In the case of soft-margin SVM, the classes are largely linearly separable, but there are a few misclassifications that lie inside the margin. Hence, we're unable to find a perfect hyperplane that maximizes the margin.



- In the case of soft-margin SVM, we introduce a *slack* variable $\xi^{(i)}$ > 0 for each training point, such that $y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi^{(i)}$. The constraints are now a less-strict because each training point need only be at a distance of (1 - $\xi^{(i)}$) from the separating hyperplane instead of a hard distance of 1. The slack allows the input to be closer to the hyperplane or even be on the wrong side.
- In order to prevent slack variable becoming too large, we penalize it in the objective function thus giving us the primal problem that can be stated as $\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi^{(i)}$, which when optimized using a dual problem will yield $\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\max\left(1 - \mathbf{y}^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b), 0\right)$. The second term in this formula is called hinge loss. Hinge loss is represented graphically as follows



NOTE: X-axis represents $y^{(i)}(\mathbf{w}.\mathbf{x}^{(i)} + b)$ and Y-axis represents the misclassification cost (loss)

- Thus, the weight update rule is given as

$\mathbf{w}^{(\text{new})} = \mathbf{w}^{(\text{old})} - \text{learning rate} \times \left(\mathbf{w} + C\sum_{i=0}^{n} \mathbf{1}\left(1 - \mathbf{y}^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) > 0\right)\mathbf{y}^{(i)}\mathbf{x}^{(i)}\right)$ and

$b^{(\text{new})} = b^{(\text{old})} - \text{learning rate} \times C\sum_{i=0}^{n} \mathbf{1}\left(1 - \mathbf{y}^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) > 0\right)\mathbf{y}^{(i)}$

- In the case of non-linearly separable data, Kernel SVMs simplify computations since we don't have to perform explicit transformations, but instead performs dot product between input features ***in a higher dimensional space*** using special functions called Kernel functions.
- Following kernel functions are typically used:
  a. Linear Kernel - $K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \mathbf{x}^{(i)^T}\mathbf{x}^{(j)}$
  b. Polynomial Kernel - $K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \left(1 + \mathbf{x}^{(i)^T}\mathbf{x}^{(j)}\right)^d$
  c. Radial Basis Functions (RBF) Kernel - $K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\frac{(\mathbf{x}^{(i)} - \mathbf{x}^{(j)})^2}{\sigma^2}\right)$

- If ***K(x,y)*** and ***K`(x,y)*** are kernels, then ***K+K`***, ***αK*** and ***α1K + α2K`***, are also kernels.

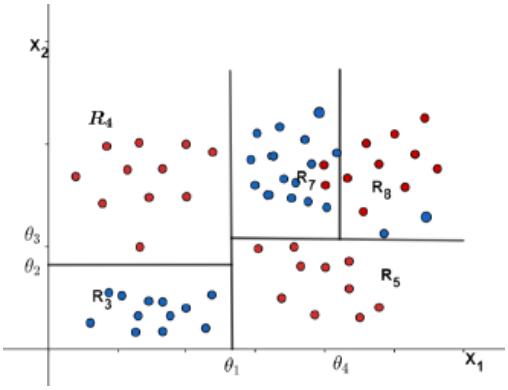
Decision Trees

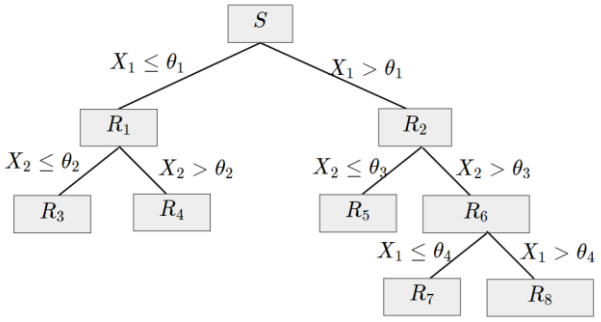
- Decision trees are non-parametric supervised learning methods that can be used for modelling classification and regression problems.
- Decision trees partition feature space into a set of rectangles (or cuboids) and then fit a model on each.
- The training model can be mathematically represented as

$$D = \{(\mathbf{X}, \mathbf{y})\} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$$

$y^{(i)}$ is an element from a discrete set in the case of a classification problem, and is a real number in the case of regression problem.

- ID3 (Iterative Dichotomizer 3) is the most common algorithm to solve such problems. It uses a top-down, greedy search (without backtracking) through a space of possible decision trees.
- This can be pictorially represented as follows.





- In the case of classification, prediction is the label in the leaf node and in the case of regression, prediction is the sample mean of all labels that belong to the subset of the training data, in the leaf node.
- Proportion of data points in node ***i*** that belong to the class ***k*** is represented mathematically as

$$p_{i,k} = \frac{1}{N_i} \sum_{x^{(i)} \in R_i} 1(y^{(i)} = k)$$

, where N_i is the number of samples in region R_i

- When all examples in a subset belong to the same class, then entropy is 0.
- When all examples in a subset are equally divided between two classes, then entropy is 1.
- Following are some of the impurity measures commonly used.
    a. Proportion of misclassified examples in node ***i*** is given by $Q_i(T) = 1 - p_{i,k(i)}$, where ***k***(***i***) is the class prediction for this node.
    b. If we code each example as 1 for class k and 0 otherwise, the variance over the node/attribute for this 0-1 response is $p_{i,k}(1-p_{i,k})$. Summing over all classes, we get the Gini index. Thus, $G_i = \sum_{k=1}^{K} p_{i,k}(1 - p_{i,k})$ which is the same as $Gini(D) = 1 - \sum_{i=1}^{k} p_i^2$

c. Entropy of a node is given by $H_i = -\sum_{k=1}^{n} p_{i,k} \log_2 p_{i,k}$. Node with the least entropy is used to split the tree.

d. Information gain of attribute A is the expected reduction in the entropy caused by partitioning the examples according to the attribute. It's mathematically represented as $IG(S, A) = \text{Entropy}(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} \text{Entropy}(S_v)$. Here, ***S*** represents the entire set of examples and $\mathbf{S_v}$ represents the subset that has the attribute ***A*** set to value ***v***. Node with the highest information gain is used to split the tree.

- Gini index is always between 0 and 0.5, both inclusive.
- Recursion on a subset will stop under any of the following conditions.
  a. Every element in the subset belongs to the same class. In this case, node becomes a leaf node, and labelled with the class of the examples.
  b. There are no more features, but examples don't still belong to the same class. In this case, the node is made a leaf node and labelled with the most common class of all examples in the subset.
  c. There are no more examples, when no example in the parent set matches a specific value of the selected attribute.
- Classification And Regression Trees (CART) can model both classification and regression problems
  - Essentially, this technique tries to find a splitting variable ***j*** and split point ***s***, among two regions R1 and R2, such that c1 and c2 are the respective classes for the two regions, by minimizing the sum of squared error loss

$$\min_{j,s} \left[\min_{c_1} \sum_{i;\mathbf{x}^{(i)} \in R_1} (\mathbf{y}^{(i)} - c_1)^2 + \min_{c_2} \sum_{i;\mathbf{x}^{(i)} \in R_2} (\mathbf{y}^{(i)} - c_2)^2 \right]$$

While solving the inner minimization problem, we get

$$\hat{c}_1 = \frac{1}{N_1} \sum_{i;x^{(i)} \in R_1} y^{(i)} \quad \text{and} \quad \hat{c}_2 = \frac{1}{N_2} \sum_{i;x^{(i)} \in R_2} y^{(i)}$$

NOTE: N1 is the #samples in region R1, N2 is the #samples in region R2

  - This process of splitting of regions is repeated for each pair of regions.

- Overfitting in decision trees happens typically when the trees grow bigger. This implies that tree size is one of the hyperparameters in decision trees.
- How to avoid overfitting?
  a. Pre-pruning
     a. Stop if the number of samples is less than some user specified threshold.
     b. Stop if expanding the current node does not improve impurity.
  b. Post-pruning
     a. Grow the tree until it reaches a minimum #nodes or #datapoints covered

b. Prune back the tree to reach a balance between the residual error and #nodes. The pruning criteria is mathematically represented as follows.

$C(T) = \sum_{i=1}^{|T|} Q_i(T) + \alpha|T|$, where $Q_i(T) = \sum_{x^{(i)} \in R_i} (y^{(i)} - c_i)^2$ and $c_i = \frac{1}{N_i} \sum_{x^{(i)} \in R_m} y^{(i)}$

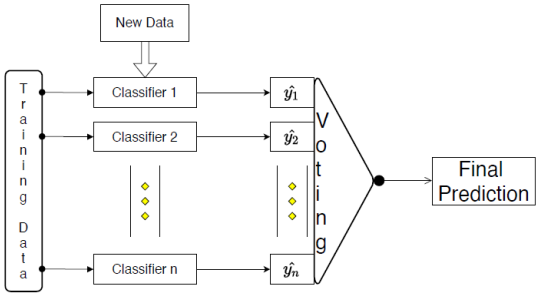
NOTE: The regularization parameter ***α*** determines the trade-off between the overall residual sum-of-squares error and the complexity of the model as measured by the number **|T|** of leaf nodes, and its value is chosen by cross-validation.

- Large value of ***α*** result in smaller trees , and conversely for smaller values of ***α***
- Decision trees suffer from the problem of high variance, which can be mitigated by ensembles dealt in the Week-10.

Ensemble methods

- Combining predictions of multiple models. Better performance, than the individual model
- Methods include
  - Majority Voting
  - Bagging – Random Forest is a popular bagging algorithm.
  - Boosting - Gradient Boosting, Adaboost, XGBoost
- Two categories of Ensemble learning

a. Voting (among high variance models to prevent outlier predictions and overfitting)



NOTE: In the case of a regression model, replace the *classifiers* by *regressors*, and *voting among predictions* by *average of all predictions*.

b. Boosting (Weak learners, combined to form Strong learners)

- Bagging and voting differs in the types classifiers used for different subsets of the samples. Bagging uses same type of classifiers, and voting might employ different types.
- In bagging, the predictions of the individual models won't depend on each other.
- Bagging reduces variance of the classifier, and can help build robust classifiers from unstable classifiers.
- Majority is one way of combining outputs from various classifiers which are being bagged
- While bagging (among different sections of the training set) can be performed in parallel, boosting is inherently a sequential process.

- In boosting, weak learners typically have low variance (and hence don't cause overfit) and high bias, since they work very well on a specific part of the problem.
- Both bagging and boosting can be used on classification as well as regression problems.
- Boosting can result in an increase in error over a base classifier due to over-emphasis on existing noise data points in later iterations, whereas error will not increase in the case of bagging.
- Probability of making a wrong prediction via the ensemble is $\binom{q}{r}\epsilon^r(1-\epsilon)^{q-r}$, assuming that r out of total q classifiers predict incorrectly. Here, ε is the error rate of each of the q classifiers. Thus, if q = 11, ε = 0.3, error of ensemble is 6.696 * 10^{-6}, which is significantly smaller than the error rate for each classifier. That's the reason, majority voting is useful.
- Class label that receives the highest votes is chosen as the final prediction, in the case of hard voting.
- In the case of soft voting, take average of probability vectors produced by different classifiers. Class with the highest probability is the final prediction.
- Bagging/bootstrap aggregation starts by voting with same classifier on different datasets, and then combines multiple such prediction functions to improve accuracy. It is useful when the predictors tend to overfit, such as decision trees, where the tree structure is highly sensitive (has high variance) to the input data.

Here's the algorithm. Let's say there're *n* datapoints x1…x10, we'll form *q* classifiers (bootstraps), with each *bootstrap* formed by dividing them among *training* and *test* sets. *Training* set is constructed by making *combinations with replacement* on the original dataset (keeping *n* datapoints in each), and rest are set aside for the *test* set. Now, for a new datapoint x, the prediction is given as $\hat{y} = \frac{1}{q}\sum_{j=1}^{q} h_j(\mathbf{x})$

- Random forest is a bagging algorithm that uses decision trees. In each of the q classifiers, *u/m* features are randomly selected, without replacement. Note that only the selected *u* features are used for split.
- In Random forest you can generate hundreds of trees (say T1, T2 . . . ..Tn) and then aggregate the results of these trees. Individual tree is built on a subset of features and observations (datapoints).
- Random forest has two hyperparameters - #classifiers *q* and #features *u*. The algorithm is insensitive to its hyperparameters. Set *q* as high as possible, and u to either $\sqrt{m}$ or $\log m + 1$,
- Boosting is an iterative process, where we build a model on the training dataset, then another to rectify errors present in the previous one, until and unless the errors are minimized. In particular, we start with a weak model, and each new model is fit on a modified version of the original dataset.
- Adaptive and Gradient boosting techniques differ mainly regarding how the weights are updated on the datapoints, and how the classifiers are combined.

- Here's the detailed algorithm for Gradient Boosting.

1. Make a first guess for y_train and y_test, using the average of y_train

$$y_{train_{p0}} = \frac{1}{n}\sum_{i=1}^{n} y_{train_i} \quad + \quad y_{test_{p0}} = y_{train_{p0}}$$

2. Calculate the residuals from the training data set → $r_0 = y_{train} - y_{train_{p0}}$
3. Fit a weak learner to the residuals minimizing the loss function. → $r_0 = f_0(X_{train})$
4. Increment the predict y's

$$y_{train_{p1}} = y_{train_{p0}} + \alpha f_0(X_{train}) \quad + \quad y_{test_{p1}} = y_{test_{p0}} + \alpha f_0(X_{test})$$

where α is the learning rate.

5. Repeat steps 2-4, until the desired accuracy is reached.

- In *Adaboost*, decision trees with only one level (decision stumps) are used, whereas in Gradient Boost, decision trees contain normally 3-7 levels. Multiple decision stumps can be combined to make a strong classifier.
- Here's the detailed algorithm for *Adaboost*

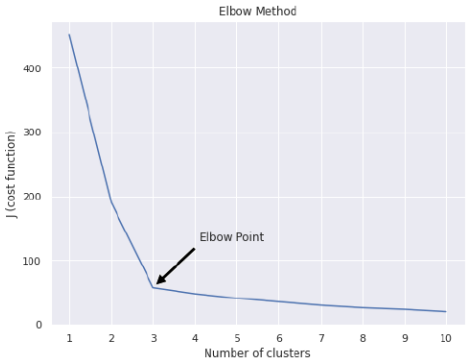
1. It builds an initial model while giving equal weights to all the data points.
2. It then assigns higher weights to points that were misclassified.
3. Now all the points which have higher weights are given more importance in the next model.
4. In other words, each model compensates the weaknesses of its predecessors.
5. In this way, it will keep training models.
6. The final model uses the weighted average of individual models.

- Performance of *Adaboost* ***α*** is calculated as $\alpha = \frac{1}{2}\ln\frac{1-\text{Total Error}}{\text{Total Error}}$ where Total Error is the sum of weights of misclassified samples (always between 0 and 1).
- ***α*** is used to update weights for the next model using the following formulae
  - For misclassified samples $\text{weight}^{(\text{new})} = \text{weight}^{(\text{old})} \times e^{\alpha}$
  - For correctly classified samples $\text{weight}^{(\text{new})} = \text{weight}^{(\text{old})} \times e^{-\alpha}$
- *XGBoost* has an in-built capability to handle missing values

---

- Clustering is the process of grouping similar samples in the training set into the same cluster. It's an example of unsupervised ML algorithm
- Some of the applications of clustering can be found in customer profiling, anomaly detection, image segmentation, image compression, geo-statistics, astronomy
- Training data can be mathematically represented as $D = \{\mathbf{x}^{(i)}\}_{i=1}^{n}$
- Clustering could follow either of
  a. Hard-clustering, where each point in the training set is assigned to one of the ***k*** clusters
  b. Soft clustering, where each point has a probability of membership to ***k*** clusters, such that the sum of such probabilities is 1.

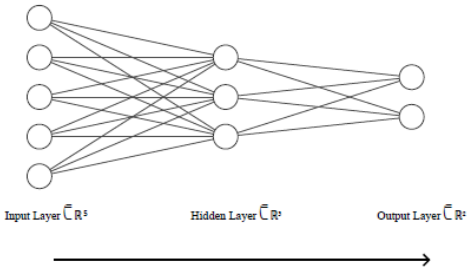
- *k-means clustering* is an example of hard-clustering. In this model, each cluster c_r (1 <= r <= k) is represented by its centroid, which is calculated as the average of the vector of points in that cluster; mathematically represented as $\mu^{(r)} = \frac{1}{|\mathbf{x}^{(i)} \in c_r|} \sum_{i=1}^{n} \mathbf{1}(\mathbf{x}^{(i)} \in c_r)\mathbf{x}^{(i)}$
- Clusters get assigned to the datapoints based on a distance measure (typically Euclidean distance), represented as $d = \sqrt{\sum_{j=1}^{m} \left(\mathbf{x}_j^{(i)} - \mu_j^{(r)}\right)^2}$
- Loss is represented mathematically as $J(c) = \sum_{r=1}^{k} \sum_{i=1}^{n} \mathbf{1}(\mathbf{x}^{(i)} \in c_r)(||\mathbf{x}^{(i)} - \mu^{(r)}||)^2$
- Here's the detailed algorithm used in k-mean clustering.
    1. Start off with *k* initial (randomly assigned) cluster centers.
    2. Assign each point to the closest cluster center.
    3. For each cluster, recompute its center as the average of all its assigned points.
    4. Repeat above two steps until centroids don't move or certain number of iterations have been performed.
- Disadvantages of k-means clustering include
    1. Poor performance due to incorrect initialization
    2. Inability to converge since datapoints don't form a spherical shape.
    3. With more number of data points, the algorithm could be very slow to converge.
    4. *k* is unknown in the beginning; computing this (using elbow method) could be expensive.
- In the elbow method of computing k, loss is calculated at different values of k. Beyond the elbow point, the loss doesn't reduce further significantly. In this specific case, the ideal #clusters ***k*** is 3.



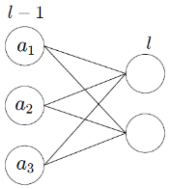
- Another method of computing k is the Silhouette coefficient, which is computed as $s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$, if $|C_I| > 1$, where a is the mean of the distance between the sample ***i*** and other samples in the cluster, and b is the minimum of mean distance between sample ***i*** and each sample in another cluster. Note that this number is always between -1 and 1 (including both). When the silhouette coefficient is at its peak, the corresponding X-value is the ideal #clusters ***k***
- If the features have a correlation coefficient of 1, the cluster centroids will be in a straight line.
- k-means is extremely sensitive to cluster center initialization.
- Bad initialization can lead to poor convergence speed and bad overall clustering.

NN

- Linear combination of inputs $z = w_1a_1 + w_2a_2 + w_3a_3 + b$ and non-linear activation $a = g(z)$ makes artificial neuron.
- Following diagram represents an forward pass ANN.



- The network has a sequence of layers, including the input layer, output layer and any number of hidden layers. The input to the network passes through these successive layers. Each layer transforms the input from the previous layer with the help of weights and activation functions. Output from the last layer is $\hat{\boldsymbol{Y}}$
- Number of layers ***L*** and the choice of activation function are hyperparameters.
- At layer l (0 < l <= L), the total number of weights is equal to $S_{l-1} * S_l$, which can be represented as a matrix.
- In the following ANN with two layers , $l-1$ and l ,the pre-activations at layer l is represented as



$$\begin{bmatrix} z_1 & z_2 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \end{bmatrix}$$

simply written as

$$\boldsymbol{Z_l} = \boldsymbol{A_{l-1}}\boldsymbol{W_l} + \boldsymbol{b_l}$$

$n \times S_l$ S_l $n \times S_{l-1}$ $S_{l-1} \times S_l$

and, the activations at layer l is represented by

$$\boldsymbol{A_l} = g(\boldsymbol{Z_l})$$

$n \times S_l$

- Without activation, ANN is a linear model.
- 3 activation functions commonly used for regression problems. However, in the case of multi-class classification problems, *Softmax* is used.

Function type	*Formula*	*Range*
Sigmoid	$g(z) = \frac{1}{1+e^{-z}}$	0 < g(z) < 1
Tanh	$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	-1 < g(z) < 1
ReLU (Rectified Linear Unit)	$g(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases}$	$g(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases}$

- Loss is computed as $L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{2} \cdot (\hat{\boldsymbol{y}} - \boldsymbol{y})^T (\hat{\boldsymbol{y}} - \boldsymbol{y})$ in the case of regression
- Loss is computed as $L(\boldsymbol{Y}, \hat{\boldsymbol{Y}}) = -\mathbf{1}_n^T \left(\boldsymbol{Y} \odot \log \hat{\boldsymbol{Y}} \right) \mathbf{1}_k$ in the case of classification

NOTE: Symbol $\odot$ represents element-wise multiplication)