**Visual Representation:**

**Supervised Learning:**

* **Input (X)** → **Algorithm** → **Output (Y)**
  + Example: [Features: Email Content] → Predict whether it's [Spam/Not Spam]

**Unsupervised Learning:**

* **Input (X)** → **Algorithm** → **Clusters/Patterns**
  + Example: [Features: Customer Transactions] → Identify [Groups of Similar Customers]

In neural network programming (NNP), evaluating and fine-tuning models is crucial for achieving optimal performance. Here are some common evaluation metrics and fine-tuning methods used in neural network programming:

**Evaluation Metrics**

1. **Accuracy:**
   * Measures the proportion of correctly predicted instances out of the total instances.
   * Commonly used for classification problems.
2. **Precision:**
   * Measures the proportion of true positive predictions out of all positive predictions made by the model.
   * Useful when the cost of false positives is high.
3. **Recall (Sensitivity):**

* Measures the proportion of true positive predictions out of all actual positive instances.
* Important when the cost of false negatives is high.

1. **F1-Score:**

* The harmonic mean of precision and recall.
* Provides a balanced measure when there is an uneven class distribution.

1. **Mean Squared Error (MSE):**

* Measures the average of the squares of the errors between predicted and actual values.
* Commonly used for regression problems.

1. **Mean Absolute Error (MAE):**

* Measures the average of the absolute differences between predicted and actual values.
* Less sensitive to outliers compared to MSE.

1. **Confusion Matrix:**

* A table that shows the true positive, true negative, false positive, and false negative predictions.
* Helps in understanding the performance of a classification model.

1. **ROC-AUC (Receiver Operating Characteristic - Area Under Curve):**

* Measures the model's ability to distinguish between classes.
* The higher the AUC, the better the model.

1. **Log Loss (Cross-Entropy Loss):**
   * Measures the performance of a classification model by penalizing false classifications.
   * Used for probabilistic classification models.

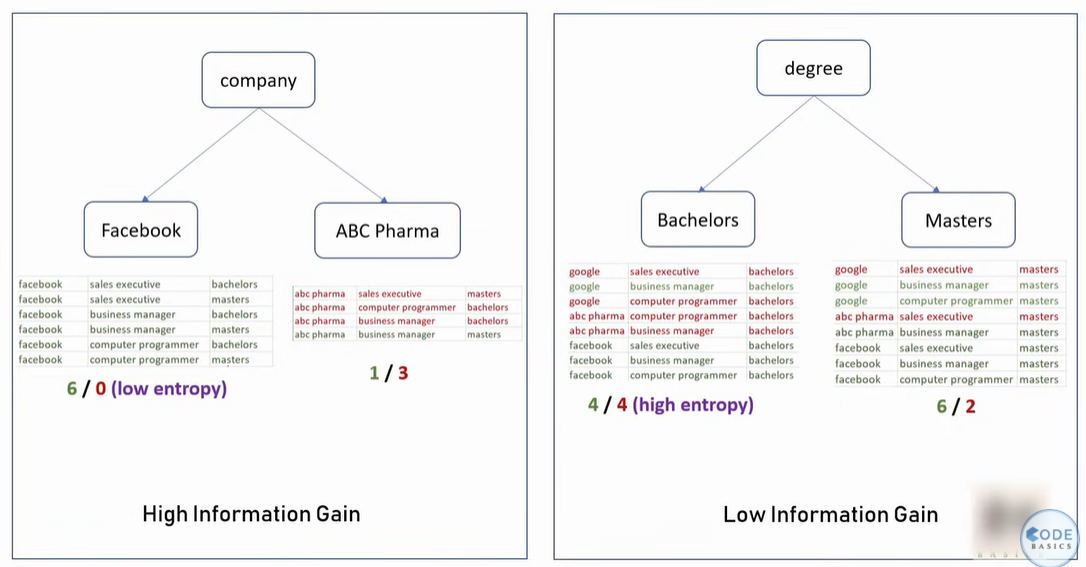
**Fine-Tuning Methods**

1. **Learning Rate Adjustment:**
   * Adjusting the learning rate helps in finding the optimal step size for updating weights.
   * Techniques include learning rate scheduling, learning rate annealing, and adaptive learning rates (e.g., Adam optimizer).
2. **Regularization:**
   * Techniques like L1 and L2 regularization help in preventing overfitting by adding a penalty to the loss function.
   * Dropout regularization randomly drops neurons during training to prevent overfitting.
3. **Early Stopping:**
   * Stops the training process when the model's performance on the validation set stops improving.
   * Helps in avoiding overfitting.
4. **Hyperparameter Tuning:**
   * Systematically searching for the best hyperparameters (e.g., learning rate, batch size, number of layers).
   * Techniques include Grid Search, Random Search, and Bayesian Optimization.
5. **Data Augmentation:**
   * Increasing the diversity of the training data by applying transformations like rotation, scaling, and flipping to prevent overfitting.
6. **Batch Normalization:**
   * Normalizes the inputs of each layer to speed up training and improve stability.
7. **Transfer Learning:**
   * Using pre-trained models on large datasets and fine-tuning them on a smaller, domain-specific dataset.
8. **Ensembling:**
   * Combining predictions from multiple models to improve overall performance.

**Summary**

Evaluation metrics and fine-tuning methods are essential components of neural network programming. Metrics like accuracy, precision, recall, and F1-score help assess the model's performance, while fine-tuning methods like learning rate adjustment, regularization, and hyperparameter tuning help in optimizing the model.

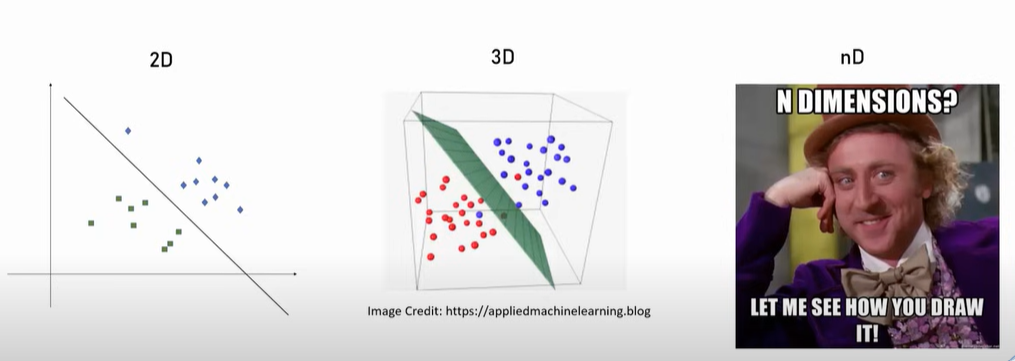
**Decision Tree Classification:**



Hence we should choose an approach that gives us a high information at every split

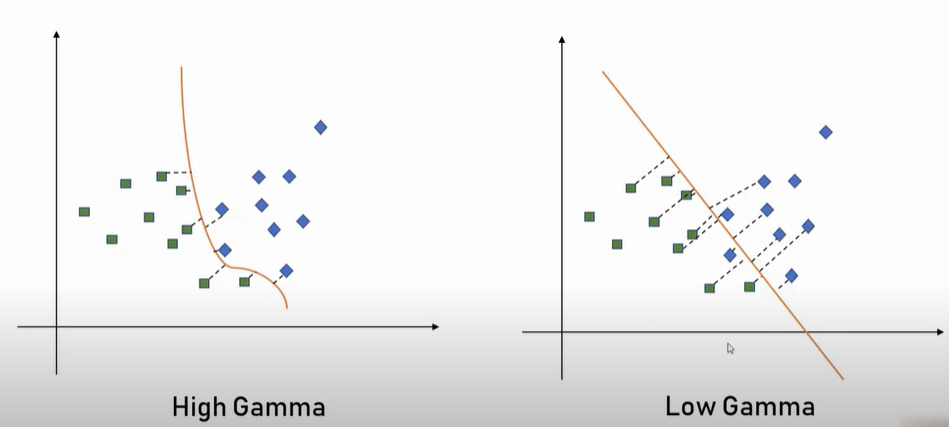
Hence in the above example choosing the feature company to split will give the high information gain since it has one low entropy where the samples are pure

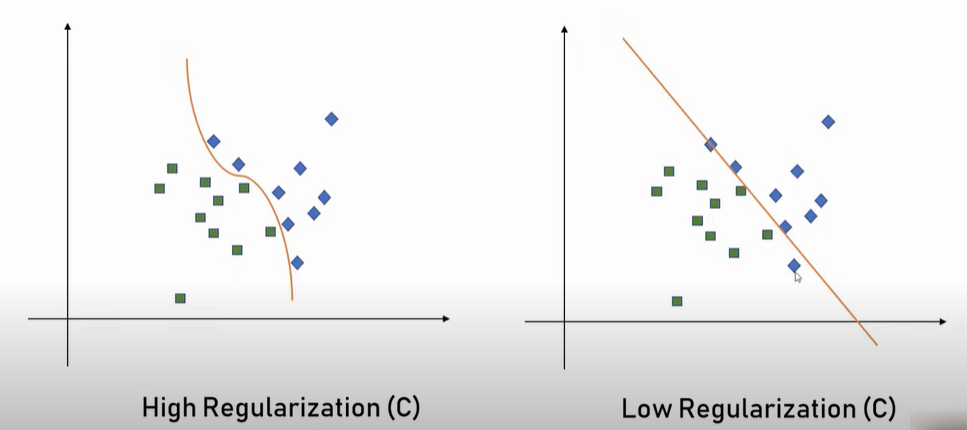
**Support Vector Machine (SVM):**

****

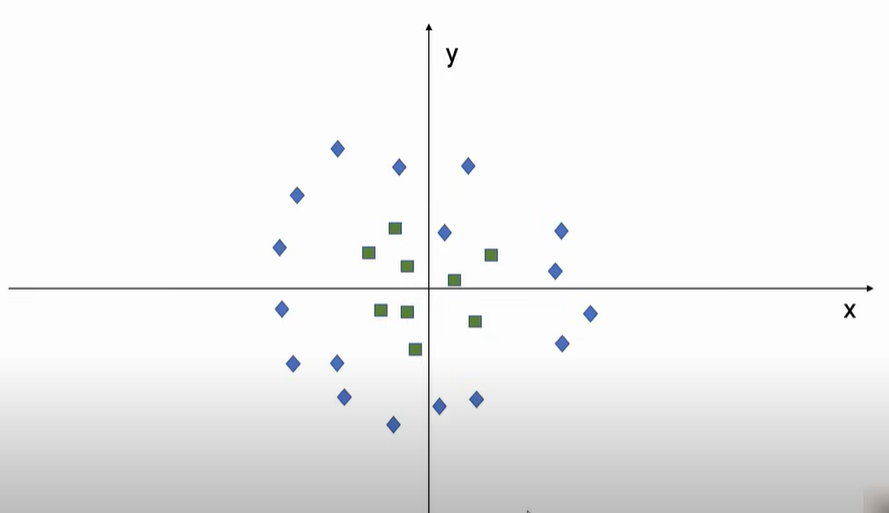
Support Vector Machine draws a hyper plane in n dimensional space such that it maximises margin between classification groups

Gamma & Regularization:

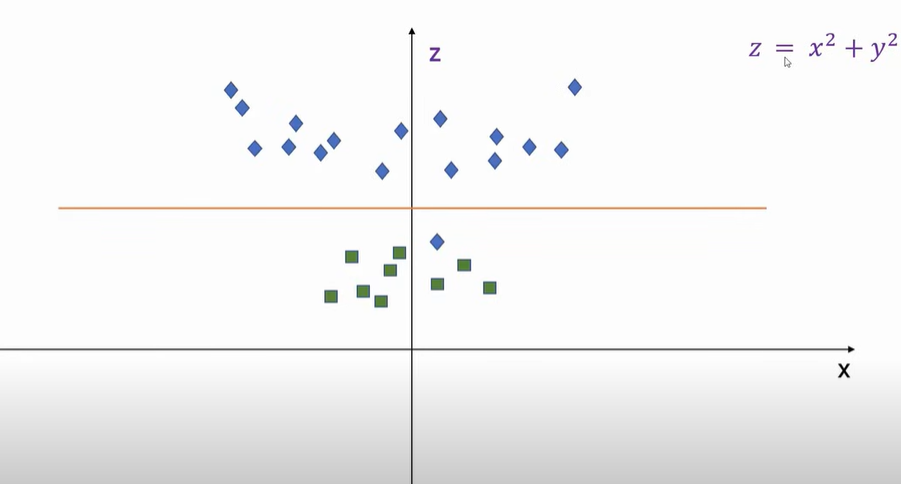




Kernel:



When the data points are like shown in the above fig its hard to draw a line between the datapoint so we can use the kernel technique here by transforming the data



By transforming the existing features and creating the new feature it is possible to draw the decision boundary. In the above picture z is the transformation that happened to the existing x and y features. After that we can super impose it on the x and y axis.

In a Support Vector Machine (SVM) model, the gamma and kernel parameters are crucial for controlling the behavior of the model and its decision boundary.

**Gamma Parameter**

The gamma parameter defines how far the influence of a single training example reaches. In simpler terms, it determines the curvature of the decision boundary:

* **Low gamma:** The model considers points far from the decision boundary, resulting in a smoother and more generalized boundary.
* **High gamma:** The model focuses on points closer to the decision boundary, leading to a more complex and specific boundary.

Choosing the right gamma value is essential to avoid overfitting (too high gamma) or underfitting (too low gamma).

In the context of Support Vector Machines (SVM), **auto** and **scale** are options for the gamma parameter that determine how the kernel coefficient is calculated:

**Auto**

* When gamma is set to **auto**, the value is calculated as: {gamma} = 1/{n\_features} where nfeatures is the number of features in the dataset.
* This means the influence of each training example is inversely proportional to the number of features. It can be useful for datasets with a small number of features.

**Scale**

* When gamma is set to **scale**, the value is calculated as: {gamma} = 1/{n\_features\*Var(X)} where n\_features is the number of features and Var(X) is the variance of the input data.
* This option adjusts gamma based on the variance of the features, making it more robust for datasets with varying feature scales. It is the default setting in many SVM implementations.

Both options aim to balance the influence of training examples on the decision boundary, but **scale** is generally preferred for datasets with features that have different ranges or units. You can explore more about these settings here. Let me know if you'd like further clarification!

**Math behind the auto and scale gamma values**

Certainly! Let's break down the mathematical calculations for **auto** and **scale** gamma values in Support Vector Machines (SVM):

**Auto Gamma**

When gamma is set to **auto**, the formula is: {gamma} = 1/n\_features

* Here, nfeaturesn\_{\text{features}} is the number of features (or dimensions) in your dataset.
* This means the gamma value is inversely proportional to the number of features. If you have more features, the gamma value becomes smaller, which results in a smoother decision boundary.

For example:

* If your dataset has 5 features, then: {gamma} = 1/5 = 0.2$$

**Scale Gamma**

When gamma is set to **scale**, the formula is: {gamma} = 1/(n\_features\*variance)

* Here:
  + nfeaturesn\_{\text{features}} is the number of features.
  + Var(X)\text{Var}(X) is the variance of the input data XX, which measures how spread out the feature values are.

This formula adjusts gamma based on both the number of features and the variance of the data, making it more robust for datasets with varying feature scales.

For example:

* If your dataset has 5 features and the variance of the data is 2, then: {gamma} = 1/(5\*2)= 0.1$$

**Kernel Parameter**

The kernel parameter specifies the type of transformation applied to the input data to make it linearly separable in a higher-dimensional space. Common kernel types include:

1. Linear Kernel: Suitable for linearly separable data. It doesn't transform the data but uses the original feature space.
2. Polynomial Kernel: Maps the data into a higher-dimensional space using polynomial functions. It has additional parameters like degree and coefficient.
3. Radial Basis Function (RBF) Kernel: The most commonly used kernel. It maps the data into an infinite-dimensional space and works well for non-linear data.
4. Sigmoid Kernel: Similar to a neural network activation function. It's less commonly used but can be effective in specific scenarios.

The choice of kernel depends on the nature of the data and the problem you're solving.

**Logistic Regression Parameters:**

**Solver Parameter**

The **solver** parameter specifies the algorithm used for optimization. Here are the common solvers and their characteristics:

1. **lbfgs**: Stands for Limited-memory Broyden–Fletcher–Goldfarb–Shanno. It's efficient for small datasets and supports both binary and multinomial classification.
2. **liblinear**: Suitable for small datasets and supports only binary classification. It uses a coordinate descent algorithm.
3. **newton-cg**: Uses the Newton method with conjugate gradient. It's computationally expensive but works well for multinomial classification.
4. **sag**: Stochastic Average Gradient Descent. Efficient for large datasets but supports only L2 regularization.
5. **saga**: A variant of SAG that supports L1, L2, and ElasticNet regularization. It's ideal for very large datasets.

**Multi\_class Parameter**

The **multi\_class** parameter determines how the model handles multi-class classification problems. Here are the options:

1. **auto**: Automatically selects the strategy based on the solver. For example, it uses "ovr" for liblinear and "multinomial" for lbfgs, newton-cg, sag, and saga.
2. **ovr** (One-vs-Rest): Fits one classifier per class and is suitable for binary classification solvers like liblinear.
3. **multinomial**: Uses a single classifier for all classes and is supported by solvers like lbfgs, newton-cg, sag, and saga.

Each combination of solver and multi\_class has specific use cases depending on the dataset size, regularization type, and classification problem. You can find more details here. Let me know if you'd like further clarification!

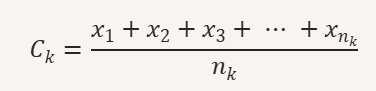
**K-Means Clustering:**

The goal of K-means is to divide the data into KK clusters, where each cluster contains data points that are closer to the center (called the "centroid") of that cluster than to any other cluster.

**Steps in the Algorithm**

1. **Initialization**:
   * Choose the number of clusters K (a predefined value).
   * Initialize K centroids randomly or based on a heuristic.
2. **Assign Points to Clusters**:
   * Each data point is assigned to the nearest centroid based on a distance metric, typically the Euclidean distance: distance = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}
3. **Update Centroids:**

* After assigning points to clusters, recompute the centroids by taking the mean of all points in each cluster:



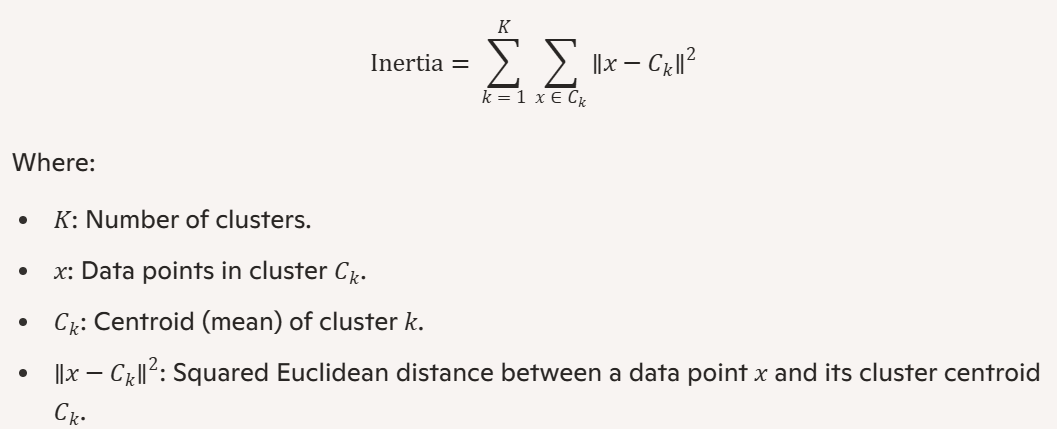
This represents the centroid Ck as the arithmetic mean of all the points x1,x2,…,xnk within the cluster k, where nk is the total number of points in that cluster.

1. **Iterate:**

* Repeat the assignment and update steps until the centroids no longer change significantly or a maximum number of iterations is reached.

**The Math Behind K-means**

* 1. K-means minimizes the Within-Cluster Sum of Squares (WCSS), which is also called the inertia



* 1. This formula calculates the sum of squared distances between all data points and their respective cluster centroids, across all K clusters. The algorithm minimizes this inertia to form well-separated clusters.

**How It Identifies Clusters**

The algorithm groups data based on the following principles:

* Points are assigned to clusters in such a way that the intra-cluster distance is minimized.
* The centroids are updated to reflect the new "center of gravity" of their cluster after re-assignment.

By repeatedly recalculating centroids and assigning points, the algorithm converges to stable clusters that represent the underlying structure of the data.

**Limitations of K-means**

1. **Choice of K**: You need to specify the number of clusters KK beforehand, which might not always be obvious.
2. **Sensitivity to Initialization**: Random initialization of centroids can lead to different results (though methods like "K-means++" address this issue).
3. **Cluster Shape**: K-means assumes clusters are spherical and equally sized, which might not suit all datasets.
4. **Scaling**: K-means relies on distance metrics, so features need to be scaled properly.