FADML Scribe

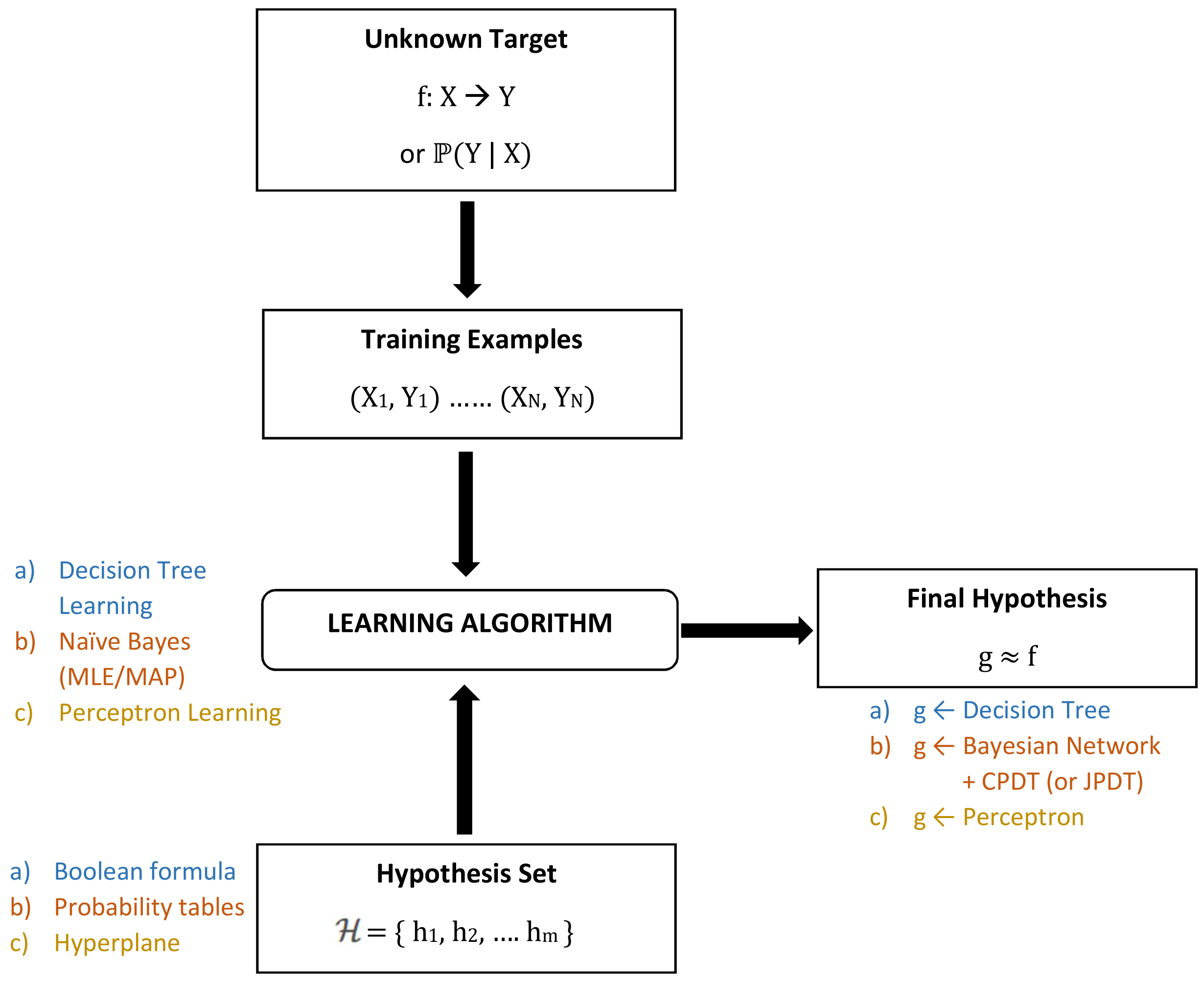
13th March, 2025

Pranjal Chakraborty

LINEAR MODELS

Classification and Regression

1. Supervised Learning Framework



Given a set of training examples, we do not explicitly know the function f, but we observe corresponding pairs of (X,Y). We approximate the **unknown target function** either in a deterministic form, where we directly map inputs to outputs, or in a probabilistic form, where we model the conditional probability $\mathbb{P}(Y \mid X)$.

Using this data, we develop a **learning algorithm** that produces a hypothesis, which serves as an approximation of f. The goal is to derive this hypothesis in a structured and generalizable manner.

To design an effective learning algorithm, we first define a **hypothesis set** H, which represents the space of possible functions the model can learn. If we take a generic approach, we might consider including every possible hypothesis. However, this comes with constraints on the size of the hypothesis space, as the **theory of generalization** attempts to balance the number of hypotheses we consider with the bounds we derive. This, in turn, influences the generalization ability of our model on unseen data.

1.1. Common Supervised Learning Frameworks

- ***Decision Tree Learning*** – We use it to learn Boolean formulas, which produces a structured representation of the data. The decision tree can be interpreted as a likelihood model or a Boolean formula, where each path from the root to a leaf forms a conjunctive clause, and the overall tree represents a disjunction of these clauses.
- ***Bayesian Methods*** – For probabilistic outcomes, we use models such as **Naïve Bayes**, **Gaussian Naïve Bayes**, and other conditional probability-based algorithms. We find probability distributions through **smart estimations** like Maximum Likelihood Estimation **(MLE)** or Maximum A Posteriori **(MAP)** estimation, or by incorporating domain knowledge in the form of **Bayesian Networks**.
  In Bayesian methods, we can represent dependencies between variables using a **Conditional Probability Distribution Table (CPDT)** or a **Joint Probability Distribution Table (JPDT).**
- ***Perceptron Learning*** – Here we use simple linear models for classification. It creates **hyperplanes** (2D it is a line, in 3D a plane, and in higher dimensions, a hyperplane). It serves as a fundamental building block, where multiple linear boundaries can be combined to form more complex models, eventually leading to **Artificial Neural Networks** and **Deep Learning**.

1.2. Objectives

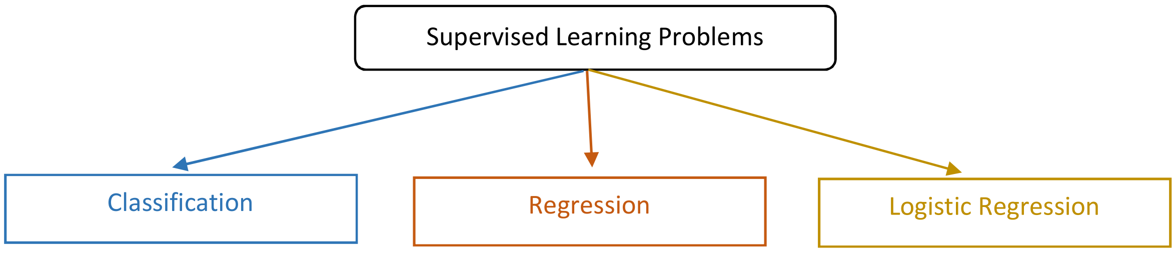
- **Algorithmic**: $\mathrm{E_{in}(g)} \approx 0$

  If we can reduce the in-sample error close to zero by using practical algorithms, the model is likely to track well or generalize well on out-of-sample data. Given the number of input examples, our objective is to make the least error over those examples to achieve the best learning performance.

- **Theory:** $\mathbb{P}[\,|\mathrm{E_{in}(g)} - \mathrm{E_{out}(g)}| > \epsilon\,] \leq 2\mathrm{M}e^{-2\epsilon^2 \mathrm{N}}$ (Hoeffding) (1.1)

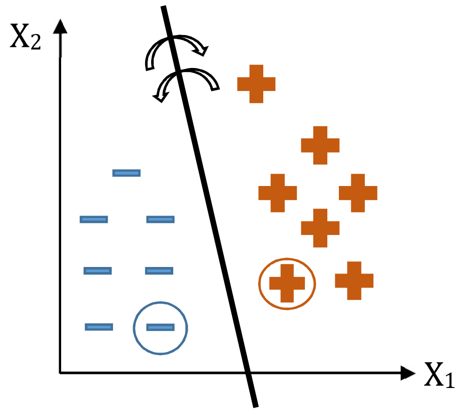
  The probability that in-sample error $\mathrm{E_{in}(g)}$ deviates from out-sample error $\mathrm{E_{out}(g)}$ by more than **ε** is at most $2\mathrm{M}e^{-2\epsilon^2\mathrm{N}}$, where:
  M = Number of hypotheses (models) considered
  N = Number of training examples
  ϵ = Allowed error

1.3. Supervised Learning Problems



2. Perceptron Learning Problem (or Hyperplane)

2.1. Classification Problem and the Perceptron Model



The figure represents a **binary classification problem**, where data points are categorized as plus (+) and minus (−)**.** The black line acts as a **decision boundary (classifier)** separating the two groups. The curved arrows suggest that the line's position might be adjustable during the learning process.

Our algorithm determines a linear separator (classifier) to separate the two classes. The equation of this line is:

$$\begin{aligned} & w_1x_1 + w_2x_2 \;\geq Threshold \\ \Rightarrow\; & w_1x_1 + w_2x_2 - Threshold \;\geq 0 \\ \Rightarrow\; & w_1x_1 + w_2x_2 - w_0 \geq 0 \end{aligned} \tag{2.1}$$

A new point $(\mathrm{x}_1, \mathrm{x}_2)$ is classified based on its position relative to this boundary:

- $w_1x_1 + w_2x_2 \;\geq Threshold$, classify as plus $(+)$.
- $w_1x_1 + w_2x_2 \;\geq Threshold$, classify as minus $(-)$.

This means that to classify any new point, we only need to compute the expression (2.1) and check if it is less than or exceeds zero.

This generalizes to multi-dimensional data with **d features**, where given $x = (x_1, x_2, \dots, x_3)$ the classification boundary is represented as:

$$\sum_{i-0}^{d} w_i\, x_i \geq 0 \tag{2.2}$$

Where $\mathrm{x}_0 = 1$ is an assumption that allows for a bias term

Classification Rule:

To classify a new point x**,** we evaluate the **sign** of the linear formula $h \in \mathrm{H}$

$$h(x) = sign\left(\sum_{i-0}^{d} w_i\, x_i\right) \tag{2.3}$$

- If $h(x)$ is positive → classify as plus $(+1)$
- If $h(x)$ is negative → classify as minus (-1)

Linear Algebra Formulation:

Using vector notation, we get (2.3) simplifies to a **dot product**:

$$h(x) = \; sign\;(w^T x) \tag{2.4}$$

Where $w = \begin{bmatrix} w_0 \\ \vdots \\ w_d \end{bmatrix}$ is a $(d{+}1)$ X 1 weight vector

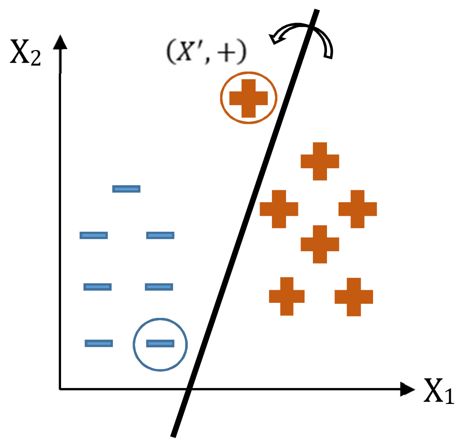
$x = \begin{bmatrix} x_0 \\ \vdots \\ x_d \end{bmatrix}$ is a $(d{+}1)$ X 1 feature vector

2.2. Classification Steps

We are not given the decision boundary beforehand. Instead, our task is to find this line based on the given training examples. The only **parameters** that determine the positioning of this line are the weights (w_i). During learning, we adjust these weights to position the line correctly, ensuring it separates the data accurately.

a) How to find the line?
We start with an **arbitrary vector** (initial line). Given all the points, we check whether they are classified correctly. If a point is **misclassified**, we **adjust** the line accordingly.

b) How to adjust the line?
Adjustment can be done in two ways:
- **Rotating** the line → Changing the slope by modifying $w_1, w_2, \ldots, w_d$.
- **Shifting** the line → Adjusting the bias term w_0.

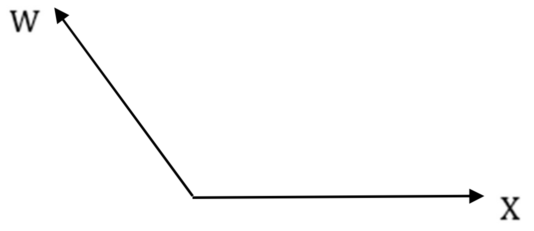


c) When do we update?
We update the weights **whenever a misclassification occurs**. For example, consider the misclassified point X' in the figure above. If **X' is actually positive**, but our current decision boundary incorrectly classifies it as negative, an update is needed.

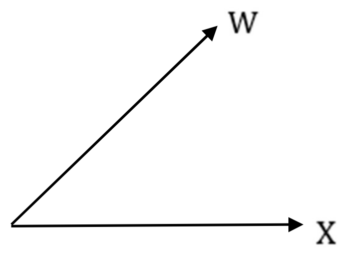
d) Why is the sign incorrect?
Classification is based on the dot product $w^T x$. A misclassification happens when $w^T x$ has the wrong sign. In this case, X' is actually positive (+1), but the dot product is negative. This occurs because w and x form an **obtuse angle**, making their dot product negative.

The other situation could be, w and x formed **acute angle** but y was -1. Both of these situations can be seen in the figures below.

y = + 1 and (X' , +)



y = -1 and (X' , +)



e) How to update?
Since the weights (parameters) determine classification, we need to update them to correct the misclassification. Our goal is to adjust w so that:

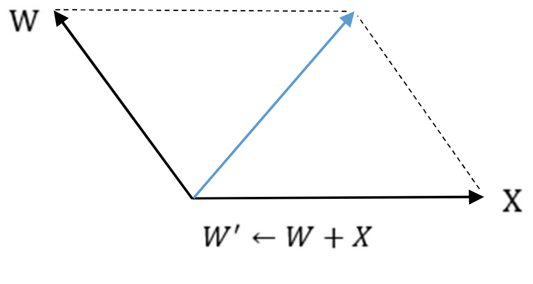
- In the first case, instead of an obtuse angle, w and x form an acute angle making $w^T x$ positive.

$$W' \leftarrow W + X$$

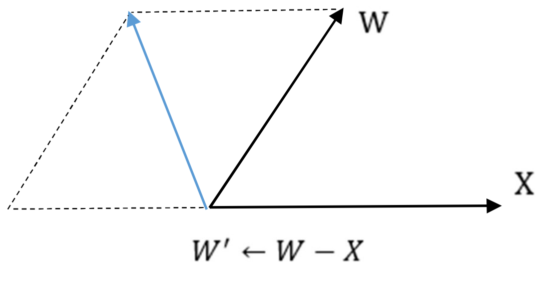
- Similarly, in second situation, if $w^T x$ was positive, but the actual label y was negative (-1), we need to reverse the effect, making the angle obtuse:

$$W' \leftarrow W - X$$

y = + 1



y = -1

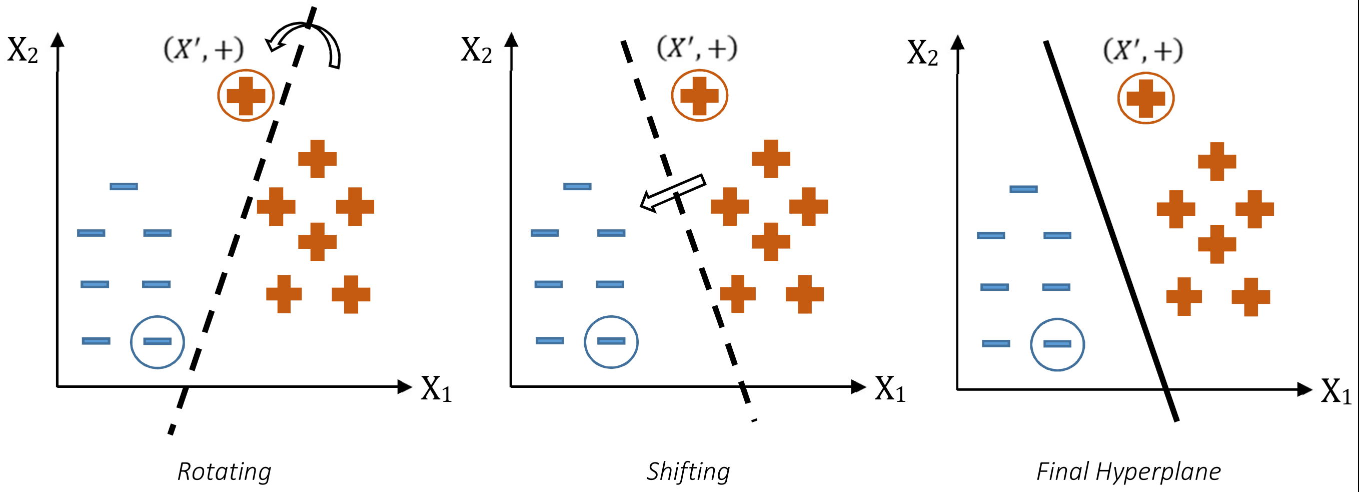


f) Delta Update Rule

Combining both cases, the weight update rule is done only for the misclassified Xs:

$$W' \leftarrow W + yX \quad or$$
$$W' \leftarrow W + \Delta W \tag{2.5}$$

where y is the true label (+1 or −1), ensuring that misclassified points adjust the boundary in the correct direction.



3. Linear Model

3.1. Pocket Algorithm

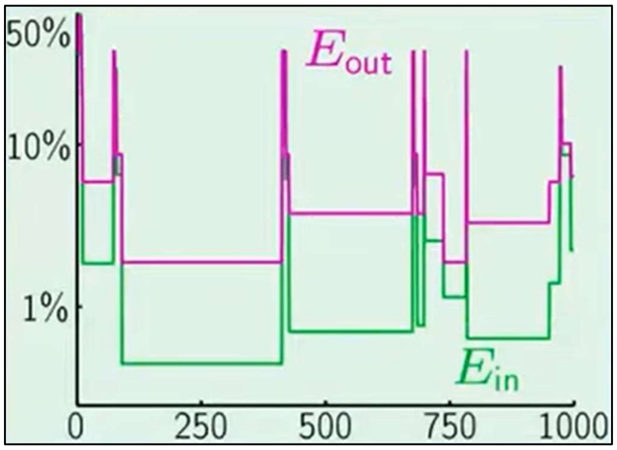


Fig. 3.1. Perceptron Learning Algorithm

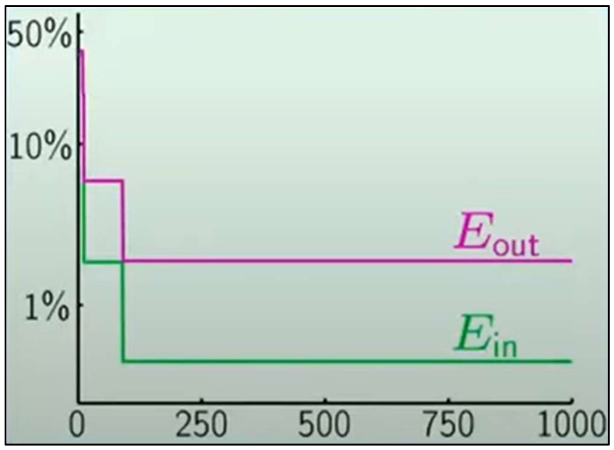


Fig. 3.2. Pocket Algorithm

In the previously discussed classification problem, the classifier makes some errors during training. We take one example, rotate or shift the decision boundary slightly, and try to improve the result. Each example gives us a certain **in-sample error** E_{in}, but we can only observe this error on the training set.

In each iteration, we compute E_{in} based on the training examples which **may increase or decrease till it converges.** The basic learning assumption is that **in-sample error tracks the out-of-sample error** and the E_{out} will be within a bound.

$$E_{in} \approx E_{out}$$

So, given the sample, if we classify in such a way that we reduce E_{in} , we are in a good position to estimate and say that our out-of-sample performance will also improve**.** This is the idea behind Perceptron Learning Algorithm (PLA), as shown in Fig. 3.1.

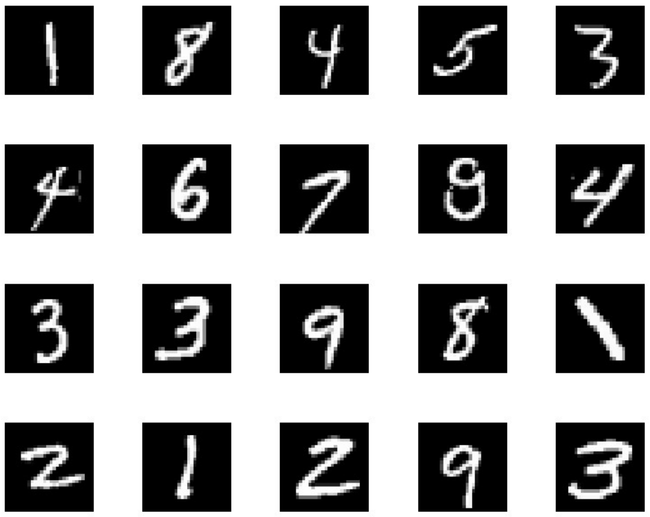
Fig. 3.2 shows a simple extension or improvement over PLA, known as the **Pocket Algorithm**. Instead of simply updating the hypothesis at every step, we keep **track of the best solution** i.e. $\min(E)$ and corresponding W_is , seen so far and keep the in-sample performance in our pocket:

- We compute E_{in} at each step and check whether the new model improves classification.
- If the new model reduces E_{in} , we replace the stored hypothesis with the new one.
- If not, we keep the best solution in our pocket until a better one appears.

This strategy ensures that we do not switch to a worse classifier, leading to significant improvements in classification performance compared to standard PLA.

If we do not use the Pocket Algorithm, we can sometimes **randomize** the starting points and perform the learning process. Randomization can help in reducing bias and can assist in cross-validation by providing different initial conditions.

3.2. Hand Digit Recognition



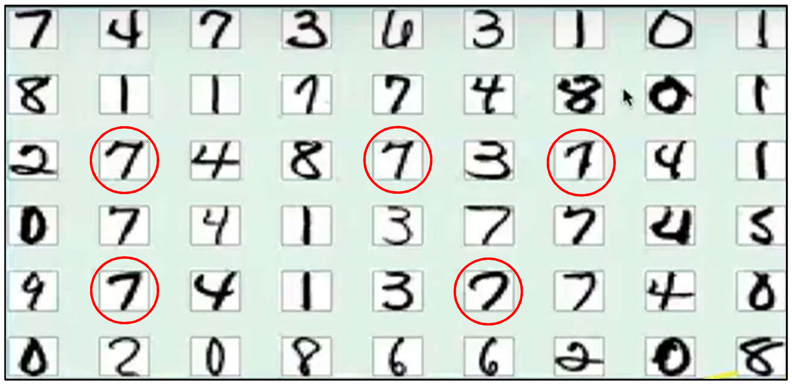


Fig. 3.3. Handwritten Digits

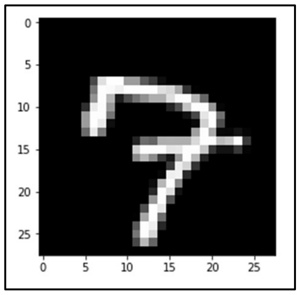


Fig. 3.4. 16 x 16 pixel image

Suppose in handwritten digit recognition, we have a pincode like **721302**. People may write the digit '7' in different ways as shown in Fig. 3.3. , leading to significant variations. To process this, we take an input of 16×16 pixels $(x_1, x_2, \ldots x_{256})$ and identify which pixels are bolded as depicted in Fig. 3.4. However, with so many attributes, the complexity increases.

To handle this, we extract meaningful features using an algorithm. For example, **symmetry**—the digit '1' is more symmetric than '5'. Similarly, in terms of **intensity**, '5' has more black pixels than '1'. If we plot **intensity vs. symmetry**, we get a distribution of handwritten digits, where some '1's may appear bolder than usual, and some '5's might be more symmetric than expected. A sample plot is shown below:

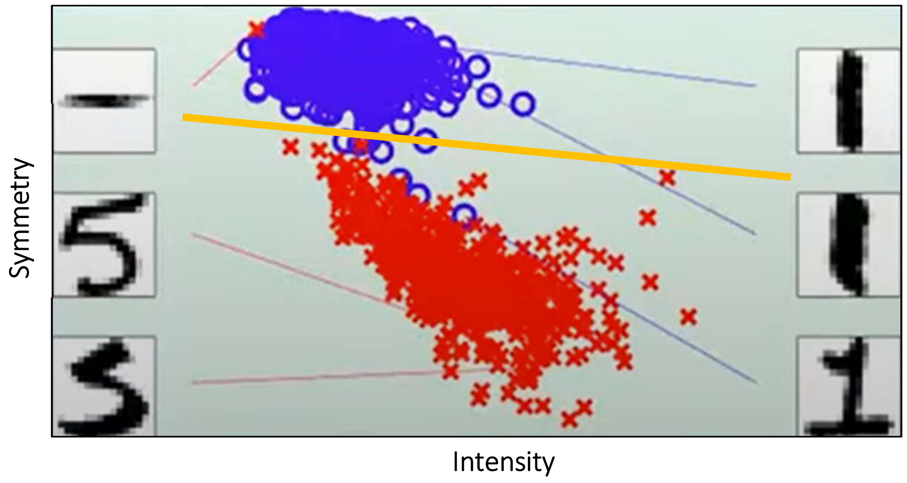


Fig. 3.5. Intensity vs Symmetry Plot of handwritten digits 5s and 1s

Since some digits may not be perfectly separable, so we do not aim for $\mathrm{E_{in}} = 0$ (zero in-sample error). Instead, we seek a decision boundary (represented by the yellow line in Fig.3.5.) in the form:

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

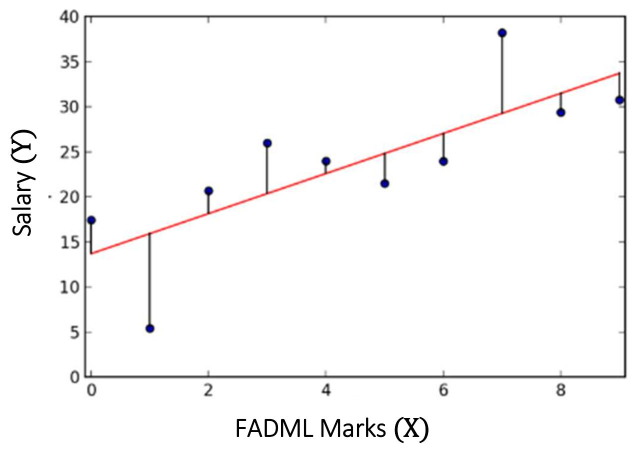
where $x_1 and\ x_2$ represent intensity and symmetry, respectively. This defines the problem space we are targeting for classification. So, here our aim is to reduce the number of attributes rather than working in a 256 dimensional domain.

How do we get the features?

In traditional approaches, feature extraction is done manually based on domain knowledge. However, in modern approaches, we use deep networks where features are automatically extracted. For this, we use **Convolutional Neural Networks (CNNs)**—we convolute and figure out the features i.e. apply convolution operations to detect patterns like edges, curves, and textures at different layers. After extracting features, we reduce the dimensionality because generalization improves in lower-dimensional spaces. In higher dimensions, the degrees of freedom increase and the possibility of doing bad in higher dimensions is more than in lower dimensions.

4. Regression

4.1. Regression Problem



Above figure illustrates a regression approach where the outcome is not binary (+1 or -1) but a real number. We have data points representing FADML marks vs. salary, and our goal is to fit the best line (hypothesis) that minimizes the sum of all errors in the training set.

In regression problems, we focus only on predicting the value, so we concentrate on the expression $w \bullet x$. Mathematically, for N data points, our prediction is $w^T x_i$ while y_i is the actual value. To measure error, we compute the **Mean Squared Error (MSE)**:

$$E_{\text{in}}(w) = \frac{1}{N}\sum_{i=1}^{N}(w^T x_i - y_i)^2$$
$$= \frac{1}{N}||Xw - Y||^2 \tag{4.1}$$

Where $\mathrm{X} = \begin{bmatrix} — x_1^{\mathrm{T}} — \\ — x_2^{\mathrm{T}} — \\ \vdots \\ — x_n^{\mathrm{T}} — \end{bmatrix}$, $\mathrm{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$ $and\ w = \begin{bmatrix} w_0 \\ \vdots \\ w_d \end{bmatrix}$

Our objective is to determine w that minimizes E_{in}, ensuring the best possible fit for the given data. Thus, we take partial derivative of $\mathrm{E_{in}}$ with respect to w and equate it to 0.

$$\frac{\partial E_{in}}{\partial w} = 0$$

$$\Rightarrow \frac{2}{N} X^T (Xw - Y) = 0$$

$$\Rightarrow X^T X w = X^T Y$$

$$\Rightarrow w = (X^T X)^{-1} X^T Y \tag{4.2}$$

Dimensions:

- $X : \big(N \times (d+1)\big)$
- $X^T X : \big((d+1) \times (d+1)\big)$
- $X^T Y : \big((d+1) \times 1\big)$
- $w : \big((d+1) \times 1\big)$

(4.2) gives us a **closed form solution** and sometimes the term $(X^T X)^{-1} X^T$ is called **pseudo-inverse** of X because if we multiply it by X we get an identity matrix.

4.2. Gradient Descent Methods

Gradient Descent is an **iterative optimization algorithm** used to minimize the error function in machine learning, especially when a **closed-form solution** (e.g. pseudoinverse) is computationally expensive or infeasible.

4.2.1. One-step Batch Gradient Descent

The closed-form solution $\boldsymbol{w = (X^T X)^{-1} X^T Y}$ feels like a one-step batch update since it directly jumps to the solution, but it is not referred to as gradient descent

4.3.2. Batch Gradient Descent

Computes the gradient using the entire dataset and updates weights in one step per iteration.

$$w \leftarrow w - \eta \frac{1}{N} \sum_{i=1}^{N} \nabla E_{\text{in}}(w)$$

4.3.3. Stochastic Gradient Descent

Updates weights after computing the gradient from a single random sample per iteration.

$$w \leftarrow w - \eta \nabla E_{\text{in}}\left(w^{(i)}\right)$$

4.3.4. Mini-Batch Gradient Descent

Updates weights using a small batch of data, balancing efficiency and stability.

$$w \leftarrow w - \eta \frac{1}{B} \sum_{j=1}^{B} \nabla E_{\text{in}}\left(w^{(j)}\right)$$

where:

$$w\ is\ the\ weight\ vector,$$

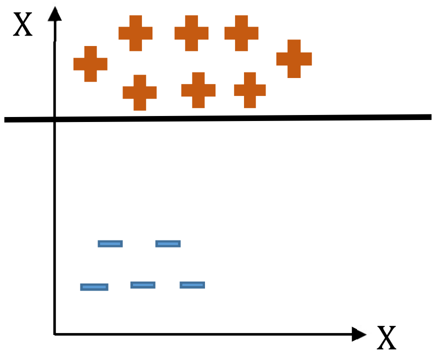
$$\eta\ is\ the\ learning\ rate,$$

$$N\ is\ the\ total\ number\ of\ samples,$$

$$\nabla E_{\text{in}}(w) is the gradient of the error function.$$

4.3. Demerits and Merits of using Linear Regression for Classification

Demerits:



Consider the figure above: since the density of positives is higher than negatives, the regression line shifts closer to the positive cloud to minimize error. However, this can lead to **poor generalization** and does not necessarily provide the best classification boundary. It may **overfit one side** and fail to achieve the optimal class-separating line.

Merits:

- It is a **simple algorithm** that can be solved with a single matrix multiplication. If the data is not perfectly separable, it still provides a line that fits the data as best as possible.

- Regression aims to find the best-fit line by starting with arbitrary weights and solving the regression problem. This can be used as an **initialization for the classification problem** where the regression line is then refined using the Perceptron Learning Algorithm (PLA) to obtain a better classification boundary, thereby reducing the number of iterations required.