# DSC 148 Final Project - Implementation of ML Models from Scratch

Chanyoung Park - Data Science at UC San Diego

March 2025

# Contents

# 1 Linear Regression

## 1.1 Linear Regression Background

The linear regression model is one of the most fundamental machine learning models that finds the best line that represents the relationships between 2 or more variables. This supervised model assumes a linear relationship between the input features and the target. The model fits the data by minimizing the sum of squared errors between the predicted and actual values. Applications of this model include problems such as text classification in sentiment analysis and spam email filtering.

## 1.2 Linear Regression Algorithm

Linear regression generally has the form of $Y_i = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ...$

There are several ways to find the coefficients of the regression:

1. Linear Algebra: $\hat{\theta} = (X^T X)^{-1} X^T Y$ (When X is invertible)
2. Gradient Descent: In this case, we need to write out the loss function and try to minimize the loss.

$$L(x) = \text{Loss Function} = \text{SE} = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

## 1.3 Linear Regression Implementation

Gradient Descent will be used to minimize the loss of the implemented model. The class initialization takes into account the following parameters: alpha, num_iter, early_stop, intercept, and init_weight.

alpha: Learning Rate, default 1e-10

num_iter: Number of Iterations to update coefficient with training data

early_stop: Constant control early_stop.

intercept: Bool, If we are going to fit a intercept, default True.

init_weight: Matrix (n x 1), input init_weight for testing.

For each step of gradient descent the equation

$$\nabla_\theta = -\frac{2}{n} \cdot \mathbf{X}^T \cdot (\mathbf{y} - \mathbf{X} \cdot \theta)$$

will be used to update the weights which in turn will take steps to minimize the mean squared error using the previous weights and alpha step value.

$$\theta \leftarrow \theta - \alpha \nabla_\theta$$

Here $n$ represents the number of observations in the dataset, $\mathbf{X}$ represents the feature matrix, $\mathbf{y}$ represents the prediction labels, and $\theta$ represents the model weights.

Using the wine-quality dataset for model comparison between the custom and sklearn implementation, the squared error for the custom model was 805 while sklearn had a squared error of 800. This is due to the custom model using gradient descent to get the optimal model weights in comparison to sklearn, which uses the normal equations. Gradient Descent is dependent on the step size and initial weights which may cause a non optimal convergence of weights for the given dataset.

### 1.4 Linear Regression Notebook

The full implementation of the Linear Regression model using Python can be accessed here: Access the Jupyter Notebook

# 2 Logistic Regression

## 2.1 Logistic Regression Background

The logistic regression model is a supervised machine learning model used for binary classification tasks. The model predicts the probability of a class label using the sigmoid function, which maps a predicted output into the range between 0 and 1.

## 2.2 Logistic Regression Algorithm

Logistic regression is a statistical method for binary classification. The model takes the form:

$$P(y = 1|\mathbf{X}) = \sigma(\mathbf{X}\theta + b)$$

where $\mathbf{X}$ represents the feature matrix, $\theta$ represents the model weights, $b$ represents the model intercept, and $\sigma(z)$ is the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The parameters $\theta$ and $b$ are optimized to minimize the logistic loss function:

$$L(\theta, b) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

## 2.3 Logistic Regression Implementation

Gradient Descent will be used to minimize the logistic loss function and takes into account the following parameters:

alpha: Learning rate, controls step size for weight updates

num_pass: Number of iterations for weight updates

early_stop: Threshold for minimal improvement in loss before stopping

standardized: Boolean to determine if feature standardization is applied

For each step of gradient descent, the weight updates are computed as:

$$\nabla_\theta = X^T(\sigma(\mathbf{X}\theta + b) - \mathbf{y})$$

$$\nabla_b = \sum(\sigma(\mathbf{X}\theta + b) - \mathbf{y})$$

Weights are updated using:

$$\theta \leftarrow \theta - \alpha \nabla_\theta$$

$$b \leftarrow b - \alpha \nabla_b$$

For a given feature vector $x$, the probability of class 1 is given by:

$$p = \sigma(x\theta + b)$$

For multiple test instances, predictions are obtained by applying the same function across the dataset.

Again using the wine-quality dataset for model comparison between the custom and sklearn implementation of logistic regression, the custom implementation performed better than the sklearn implementation. Using a probability threshold of 0.5 to classify the given feature vector as 0 or 1, the custom implementation was able to get 928/1319 predictions correct while sklearn was only able to get 897/1319 correct.

## 2.4 Logistic Regression Notebook

The full implementation of the Logistic Regression model using Python can be accessed here: Access the Jupyter Notebook

# 3 Naive Bayes

## 3.1 Naive Bayes Background

The logistic regression model is a supervised machine learning model and probabilistic classifier based on Bayes' theorem, which calculates the probability of a class given the input features. The model assumes that all features are conditionally independent given the class label. With this assumption, Naive Bayes is a straightforward and computationally efficient model to implement.

## 3.2 Naive Bayes Algorithm

Naive Bayes is a conditional probability model, with the formula:

$$P(C|x_1, x_2, x_3, ...) = \frac{P(C)P(X|C)}{P(X)}$$

It is naive because we have naive assumption such that every pair of features are independent from each other given C. So we can rewrite the formula as:

$$P(C|x_1, x_2, x_3, ...) = P(C)P(x_1|C)P(x_2|C)... = P(C)\prod_{i=1}^{n} P(x_i|C)$$

For each feature vector, the probability will be calculated for each unique class within the dataset. The feature vector will be classified by the highest probability for a specific class.

## 3.3 Naive Bayes Implementation

The Naive Bayes class has a fit function where the probabilities of each class from the given dataset is calculated. This involves getting the count of each class in the dataset and dividing by the total number of observations.

The ind_predict function predicts the most likely class label of one test instance based on its feature vector x. To prevent numerical underflow due to many multiplications of probabilities, log-probabilities are used instead in the custom implementation of Naive Bayes. To prevent conditionally independent probabilities that do not have a observation under a specific class to be 0, which in turn will set the entire probability to 0 regardless of the other conditionally independent probabilities, Laplace smoothing was implemented by adding 1 to all observation counts and adding the number of unique classes to the denominator.

For each feature vector, this probability can be represented:

$$\log P(C|\mathbf{x}) = \log P(C) + \sum_{i=1}^{d} \log P(X_i = x_i|C)$$

Where $d$ is the number of features, $X_i$ is the current feature, $x_i$ is the current observation, and

$$\log P(X_i = x_i|C) = \log\left(\frac{\text{Count}(X_i = x_i \text{ and } C) + 1}{\text{Count of instances with } C + \text{Count of unique classes}}\right)$$

The predict function goes through each feature vector in the dataset and returns the class prediction for the given vector.

Using the balance-scale data, the custom model was able to achieve 88% accuracy on the test set making Naive Bayes a strong and efficient model for many classification tasks.

### 3.4 Naive Bayes Notebook

The full implementation of the Naive Bayes model using Python can be accessed here: Access the Jupyter Notebook

# 4 K-means

## 4.1 K-means Background

The K-means model is an unsupervised clustering machine learning algorithm that groups a dataset into K clusters by minimizing the sum of squared distances between data points and their cluster centroids. The algorithm iterates between assigning each data point to the nearest centroid and updating centroids as the mean of their assigned points. The model is computationally efficient, but depends to the choice of K clusters, is sensitive the presence of outliers, and assumes spherical, equally-sized clusters.

## 4.2 K-means Algorithm

K-Means algorithm is a clustering algorithm. It is also a classic Expectation-Maximization algorithm.

Given a set of observations $(x_1, x_2, ..., x_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k sets S = $S_1, S_2, ..., S_k$ so as to minimize the within-cluster sum of squares (WCSS) (i.e. variance)

The steps are as follows:
1. Randomly select k "cluster centers" from the data set
2. For each iteration (iterate through all points):

1. Find the nearest center for each point, and store the cluster for each center
2. Calculate the new center for each cluster
3. If there are no change of the means, end the loop; otherwise iterate

## 4.3 K-means Implementation

The K-means class initialization takes into account the following parameters: k, num_iter, centers, and RM

k: Number of clusters we are trying to classify

num_iter: Number of iterations we are going to loop

centers: Initial centers of each cluster

RM: Relation matrix that stores which cluster the observation belongs to; Initially set to None and has a dimension of $n$ x $k$ where n is the number of observations in the dataset

Within the fit function of the KMeans class, if the initial centers are not speci-

fied by the user, KMeans++ initialization is used in the custom implementation. Spread out starting points are used instead of randomly selected points, ensuring faster convergence and better clustering results.

$$C_1 = X_{\text{random}}$$

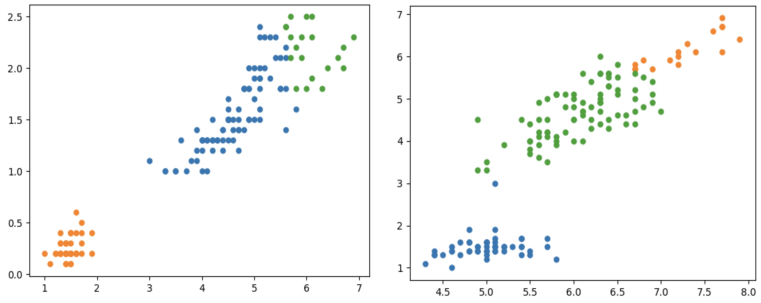
$$C_{i+1} = \arg\max_{x \in X} \left( \min_{c \in C} ||x - c|| \right)$$

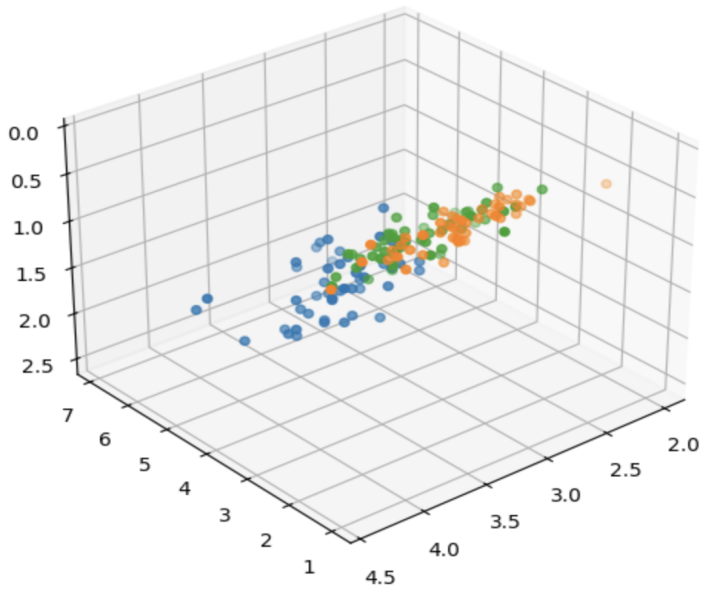
Where $C$ is a cluster and $X_{\text{random}}$ a random feature vector from the dataset. Each cluster is determined by getting minimum distances of points using all of the previous determined clusters then extracting the maximum distance point from these distances. Once the centers are defined, each point is assigned to its closest center using the Euclidean distance where it is assigned a 1 for its row and cluster column within the relation matrix. Else it is assigned a 0 in the relation matrix.

For each cluster the mean position of all the points which are assigned to that specific cluster is set to the new center for the cluster. All the step above are repeated until the change of means is insignificant or num_iter is finished.

$$C_k = \frac{1}{|S_k|} \sum_{X_i \in S_k} X_i$$

Where $S$ is the set of points within a specific class and $k$ is the specific cluster. Looking at the computed $k$ clusters using the iris dataset, the custom implementation does a good job of determining clusters that are well separated.






## 4.4 K-means Notebook

The full implementation of the K-means model using Python can be accessed here (first half of notebook): Access the Jupyter Notebook

# 5 Gaussian Mixture Model (GMM)

## 5.1 GMM Background

The Gaussian Mixture Model or GMM, is a unsupervised probabilistic clustering machine learning algorithm that models the given data as a mixture of multiple Gaussian distributions. These mixtures will each represent a different cluster in which the number of clusters is specified by the user. The GMM provides soft assignments, meaning each point has a probability of belonging to multiple clusters, unlike in K-means which assigns hard cluster labels. The model creates these clusters using the Expectation-Maximization (EM) algorithm, which iteratively updates point to cluster probabilities in the E-step and optimizes the Gaussian parameters (mixture weight, mean, covariance) in the M-step.

## 5.2 GMM Algorithm

The GMM algorithm assumes that each observation in the data is generated from one of $k$ Gaussian distributions. The algorithm aims to estimate the parameters of these distributions: the mean ($\mu_k$), covariance matrix ($\Sigma_k$), and the mixture weight ($w_k$) for each cluster.

The steps of the GMM algorithm are as follows:
Initialize cluster parameters: Assign initial values to the mean vectors $\mu_k$, covariance matrices $\Sigma_k$, and weights $w_k$.

E-Step: Compute the posterior probabilities of each data point belonging to each Gaussian cluster.

M-Step: Update the parameters ($\mu_k$, $\Sigma_k$, and $w_k$) using the computed posteriors.

Compute the log-likelihood repeat the EM steps until convergence is reached.

## 5.3 GMM Implementation

The GMM custom implementation consists of several functions.

The gaussian function computes the probability density of a given data point under a Gaussian distribution:

$$p(x|\mu, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

where $\mu$ is the mean vector, $\Sigma$ is the covariance matrix, and $d$ is the dimensionality of the feature matrix.

The initialize_clusters function initializes the clusters using the K-means algorithm to determine the initial means ($\mu_k$). The probability weights are equally distributed by the number of clusters $k$ and the covariance matrices are set as identity matrices:

$$w_k = \frac{1}{k}, \quad \Sigma_k = I$$

In the expectation_step function, the posterior probability for each data point belonging to each cluster is computed as:

$$p(C_k|x_i) = \frac{w_k p(x_i|\mu_k, \Sigma_k)}{\sum_{j=1}^{k} w_j p(x_i|\mu_j, \Sigma_j)}$$

where $C_k$ represents the cluster assignment for $x_i$.

In the maximization_step function, the model parameters for each cluster are updated using the computed posteriors where $N$ is the number of observations for a specific cluster:

$$N_k = \sum_{i=1}^{N} p(C_k|x_i)$$

$$w_k = \frac{N_k}{N}, \quad \mu_k = \frac{1}{N_k} \sum_{i=1}^{n} p(C_k|x_i) x_i$$

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{N} p(C_k|x_i)(x_i - \mu_k)(x_i - \mu_k)^T$$

In the get_likelihood function, the log-likelihood of the dataset is computed as:

$$\log L = \sum_{i=1}^{N} \log \sum_{j=1}^{k} w_k p(x_j|\mu_j, \Sigma_j)$$

The algorithm iterates through the E-step and M-step until the maximum number of epochs specified is reached. The GMM can be used to assign probabilities to new data points and perform clustering based on the highest posterior probability. The GMM is more flexible than the K-means model as it can model elliptical clusters and overlapping distributions, making it useful for applications such as underlying density estimation and anomaly detection. Using a sample dataset, the custom implementation of the GMM performed on par with the sklearn implementation.

## 5.4 GMM Notebook

The full implementation of the K-means model using Python can be accessed here (second half of notebook): Access the Jupyter Notebook