# NumpyAIModels

Timothy Pearson

June 2024

## Introduction

This document details the mathematics of a collection of models that are implemented using Numpy. These models are part of my studies on various machine learning algorithms and their applications.

1. **Principal Component Analysis**
2. **Decision Trees**
3. **K-Means Clustering**
4. **Gradient Boosting**
5. **Feed Forward Neural Network**

## Principal Component Analysis (PCA)

Principal Component Analysis is a dimensionality reduction technique that transforms high-dimensional data into a lower-dimensional form while preserving as much variance as possible. This implementation in NumPy focuses on the following steps:

- Standardizing the data.
- Computing the covariance matrix.
- Calculating the eigenvalues and eigenvectors.
- Sorting the eigenvalues and eigenvectors.
- Transforming the data to the new basis.

### Standardize and Normalize the Data

- **Mean normalization:** Centers the dataset around zero by subtracting the mean of each feature, removing bias.
- **Standardization:** Scales each feature to have a mean of 0 and standard deviation of 1, ensuring similar scales.
- **Prevent numerical instability:** Improves stability of optimization algorithms, leading to faster convergence.
- **Enhance interpretability:** Makes model coefficients more interpretable for comparing feature importance.

## Calculate Covariance Matrix and Eigenvectors

- **Covariance Matrix:** Calculated using the formula:

  $$\text{covariance}(\mathbf{X}) = \frac{1}{N}(\mathbf{X} - \bar{\mathbf{X}})^T(\mathbf{X} - \bar{\mathbf{X}})$$

  Where:

  - $\mathbf{X}$ is the data matrix (after standardization),
  - $\bar{\mathbf{X}}$ is the mean vector of the data matrix,
  - $N$ is the number of samples.

- **Eigenvalues and Eigenvectors:** Obtained by performing eigendecomposition on the covariance matrix:

  $$\mathbf{C} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}$$

  Where:

  - $\mathbf{C}$ is the covariance matrix,
  - $\mathbf{Q}$ is the matrix of eigenvectors,
  - $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues.

## Calculate Explained Variance Ratio

- **Total Variance:** Sum of all eigenvalues obtained from the eigendecomposition of the covariance matrix:

  $$\text{Total Variance} = \sum_{i=1}^{n} \lambda_i$$

  Where $\lambda_i$ represents the $i$-th eigenvalue.

- **Explained Variance Ratio:** Ratio of each individual eigenvalue to the total variance:

  $$\text{Explained Variance Ratio} = \frac{\lambda_i}{\text{Total Variance}}$$

  Where $\lambda_i$ represents the $i$-th eigenvalue.

- **Cumulative Explained Variance Ratio:** Cumulative sum of explained variance ratios:

  $$\text{Cumulative Explained Variance Ratio} = \sum_{i=1}^{n} \text{Explained Variance Ratio}_i$$

  Starting from the first principal component ($i = 1$) up to the $n$-th principal component.

## Select Principal Components

- **Threshold Selection:** Choose a threshold to determine the amount of variance to retain.

- **Determine Number of Components:** Find the number of principal components needed to exceed the chosen threshold of cumulative explained variance.

- **Selection of Principal Components:** Select the principal components corresponding to the determined number.

- **Projection of Data:** Project the standardized data onto the selected principal components to obtain the reduced-dimensional form.

# Decision Trees

Decision Trees are a type of supervised learning algorithm used for classification and regression tasks. This implementation includes testing different criteria for splitting the nodes:

- **Gini Impurity**
- **Entropy**
- **Misclassification Error**

## Gini Impurity

The Gini impurity for a given set of labels $y$ is calculated using the following formula:

$$\mathrm{Gini}(y) = 1 - \sum_{i=1}^{C} p_i^2$$

This formula computes the Gini impurity by summing the squared proportions of each class and subtracting the sum from 1. The result represents how impure or mixed the classes are within the dataset.

## Entropy

The entropy for a given set of labels $y$ is calculated using the following formula:

$$\mathrm{Entropy}(y) = -\sum_{i=1}^{C} p_i \cdot \log_2(p_i)$$

This formula computes the entropy by summing the product of each class proportion $p_i$ and its logarithm to the base 2, then taking the negative of the sum. Entropy measures the uncertainty or randomness in the distribution of classes within the dataset.

## Misclassification Error

The misclassification error for a given set of labels $y$ is calculated using the following formula:

$$\text{Misclassification Error}(y) = 1 - \max_{i=1}^{C} p_i$$

This formula computes the misclassification error by finding the proportion of the majority class (the class with the highest proportion) and subtracting it from 1. The result represents the error rate or the proportion of misclassified samples within the dataset.

## Findings on the Iris Dataset

Through experimentation with the Iris dataset, I found that using the Misclassification Error as the criterion provided the best performance compared to Gini Impurity and Entropy. The results indicated that Misclassification Error resulted in higher accuracy and better generalization for this specific dataset.

# K-Means Clustering

K-Means Clustering is a method for partitioning data into $K$ distinct clusters. This model aims to minimize the variance within each cluster by iteratively assigning data points to clusters and updating cluster centroids.

## Initialization of Centroids

Centroids are initialized using the $k$-means++ algorithm to improve the quality of initial centroids. This ensures a more even spread across the data, contributing to better clustering results. The method initializes centroids by selecting the first centroid randomly and subsequently choosing each subsequent centroid based on the probability proportional to the square of the distance from the nearest existing centroid.

$$C_0 = \mathbf{X}[i], \quad \text{for } i \sim \text{Uniform}(1, n)$$

$$C_j = \arg\min_{x \in \mathbf{X}} \left( \min_{c \in C} \|x - c\|^2 \right)$$

## Assignment of Labels

Data points are assigned to the nearest centroid based on the Euclidean distance. This assignment is updated iteratively as centroids are recalculated.

$$\text{Label}(x_i) = \arg\min_{c_j \in C} \|x_i - c_j\|^2$$

## Distance Matrix

The distance matrix calculates the Euclidean distance between each pair of points and centroids.

$$D(i, j) = \sqrt{\sum_{k=1}^{d} (x_{ik} - y_{jk})^2}$$

Where $x_i$ and $y_j$ are data points and $d$ is the number of dimensions.

## Model Training

The model is trained by iteratively assigning labels to data points and updating centroids until convergence.

$$C_{new} = \frac{1}{|S|} \sum_{x \in S} x$$

Where $S$ is the set of points assigned to centroid $c$.

## Silhouette Score

The silhouette score evaluates the quality of the clustering. It is defined as:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

Where:

$$a(i) = \frac{1}{|S_i| - 1} \sum_{x \in S_i} \|x - c_i\|$$

is the average distance between point $i$ and all other points in the same cluster, and

$$b(i) = \min_{j \neq y} \left( \frac{1}{|S_j|} \sum_{x \in S_j} \|x - c_j\| \right)$$

is the average distance between point $i$ and all points in the nearest cluster $j$.

# Gradient Boosting

Gradient Boosting is an ensemble learning technique that builds a model in a stage-wise fashion from weak learners, typically decision trees. This implementation modifies the decision tree algorithm to allow the leaf nodes to return residuals instead of predictions. This modification enables the model to correct errors from previous iterations, improving overall performance.

## Algorithm Overview

The Gradient Boosting algorithm involves the following steps:

- **Initialize Model:** Start with an initial prediction, typically the mean of the target values for regression or the mode for classification.
- **Compute Residuals:** Calculate the residuals, which are the differences between the actual target values and the current predictions.
- **Fit Weak Learner:** Train a weak learner (e.g., a decision tree) on the residuals.
- **Update Model:** Update the model by adding the predictions from the weak learner to the previous predictions, scaled by a learning rate.
- **Iterate:** Repeat the steps of computing residuals, fitting a weak learner, and updating the model for a predefined number of iterations or until convergence.

## Objective Function

The objective function in Gradient Boosting is to minimize the loss function $L(y, F(x))$ over the training data, where $y$ are the actual target values and $F(x)$ are the predictions from the model. Common loss functions include:

- **Mean Squared Error (MSE):** For regression tasks.
- **Logistic Loss:** For binary classification tasks.
- **Multinomial Deviance:** For multi-class classification tasks.

## Learning Rate

The learning rate $\eta$ is a hyperparameter that controls the contribution of each weak learner to the final model. A smaller learning rate requires more iterations but can lead to better generalization.

$$F_m(x) = F_{m-1}(x) + \eta h_m(x)$$

Where $F_m(x)$ is the updated prediction, $F_{m-1}(x)$ is the previous prediction, $\eta$ is the learning rate, and $h_m(x)$ is the prediction from the $m$-th weak learner.

## Regularization

Regularization techniques can be applied to prevent overfitting in Gradient Boosting models. Common methods include:

- **Shrinkage:** Scaling the contribution of each weak learner by the learning rate.
- **Tree Constraints:** Limiting the depth of the trees, the number of leaves, or the minimum number of samples required to split a node.
- **Stochastic Gradient Boosting:** Introducing randomness by subsampling the data points or the features used for training each weak learner.

Gradient Boosting is a powerful and flexible method that often achieves state-of-the-art performance in various machine learning tasks. However, it requires careful tuning of hyperparameters and can be computationally intensive.

## Decision Tree as Weak Learner

The weak learner used in this implementation of Gradient Boosting is a modified decision tree. Instead of predicting the target values directly, each leaf node of the tree returns the residuals (difference between actual and predicted values). This allows subsequent trees to correct the errors made by previous trees. This modified decision tree is trained to minimize the loss function by recursively partitioning the data based on feature splits that minimize the residual errors at each node.

# Feed Forward Neural Network

The Classification Model is designed to predict class probabilities for multi-class classification tasks using deep neural networks.

## Initialization

The model is initialized with the input dimension $D_{\text{input}}$ and the number of output classes $K$.

$$\begin{aligned}\text{input\_dim} &= D_{\text{input}} \\ \text{output\_dim} &= K\end{aligned}$$

## Activation Functions

### ReLU (Rectified Linear Unit)

$$\text{ReLU}(x) = \max(0, x)$$

ReLU introduces non-linearity in the model, applied element-wise.

### ReLU Derivative

$$\text{ReLU\_Derivative}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Gradient of ReLU used in backpropagation.

### Softmax Function

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Softmax outputs a probability distribution over classes, ensuring sum of probabilities equals one.

## Forward Propagation

### Forward Pass through Layers

$$z^{(l)} = W^{(l)} \cdot a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = \text{ReLU}(z^{(l)})$$

Linear transformation $z^{(l)}$ followed by ReLU activation $a^{(l)}$ for hidden layers.

### Final Layer (Softmax Activation)

$$z^{(L)} = W^{(L)} \cdot a^{(L-1)} + b^{(L)}$$

$$\hat{y} = \text{Softmax}(z^{(L)})$$

Final layer computes class probabilities $\hat{y}$ using softmax activation.

## Loss Calculation

### Cross-Entropy Loss

$$\text{Loss} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{i,k} \log(\hat{y}_{i,k})$$

Cross-entropy loss measures the difference between predicted probabilities $\hat{y}$ and true labels $y$.

## Backward Propagation

### Gradient Descent Update

Weights $W$ and biases $b$ are updated using gradients computed from loss.

$$\frac{\partial \text{Loss}}{\partial W^{(l)}} = dz^{(l)} \cdot a^{(l-1)T}$$

$$\frac{\partial \text{Loss}}{\partial b^{(l)}} = \text{mean}(dz^{(l)}, \text{axis} = 1)$$

Backpropagates gradients through layers to minimize loss using gradient descent.