University of California San Diego

DSC 148: Introduction to Data Mining

Final Individual Project

March 18, 2024

Author	*Student ID*
Minchan Kim	A17278478

Contents

1 Linear Regression

1.1 Linear Regression Background

Linear regression is a foundational statistical model used to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data. The core idea behind linear regression is to describe this relationship through a linear function that predicts the dependent variable based on the values of the independent variables.

Continuing from the overview of linear regression and its reliance on a linear function for prediction, we delve into the methodologies for determining the optimal weights (coefficients) that can be attached to the independent variables. These methodologies are fundamentally categorized into two approaches: the linear algebra approach and the gradient descent approach. Both methods aim to minimize the discrepancy between the predicted and actual values of the dependent variable, albeit through different computational mechanisms.

However, this implementation of linear regression will aim to answer the coefficient problem through the lens of gradient descent.

1.2 Mathematical Formulation

Linear regression generally has the form of

$$Y_i = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ...$$

The coefficients can be solved in two ways:

- Linear algebra $\hat{\theta} = (X^T X)^{-1} X^T Y$
- Gradient descent

The loss function used in this implementation of gradient descent will be squared error:

$$F(x) = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

Since $\hat{Y} = Xw$, we can derive the gradient (derivative) of this function after a few simple calculations.

$$F(w) = (y - Xw)^T (y - Xw)$$

$$\nabla_w F(w) = \frac{\partial}{\partial w} \left((y - Xw)^T (y - Xw) \right)$$

$$\nabla_w F(w) = -2X^T (y - Xw)$$

1.3 Implementation

The linear regression model is implemented in Python as a class `‘Linear_Regression‘`, encapsulating the functionality required for fitting the model to the data and making predictions. The implementation leverages the NumPy library for numerical computations and the pandas library for data handling, ensuring efficient matrix operations and data manipulation.

1.3.1 Class Initialization

The constructor of the class initializes with several parameters that define the behavior of the model during training:

- `‘alpha‘`: The learning rate for gradient descent, determining the step size at each iteration.
- `‘num_iter‘`: The maximum number of iterations to perform during gradient descent.
- `‘early_stop‘`: A threshold for early stopping, to halt training if the improvement in error becomes negligible.
- `‘intercept‘`: A boolean flag indicating whether to include an intercept term in the model.
- `‘init_weight‘`: Initial values for the model's weights, which can be used for testing or to set a starting point for optimization.

These parameters allow for customization of the model's training process, providing flexibility to accommodate different datasets and training requirements.

1.3.2 Model Fitting

The `‘fit‘` method is responsible for preparing the data and initiating the training process. It accepts the feature matrix `‘X_train‘` and target values `‘y_train‘`, performing necessary preprocessing such as reshaping and adding an intercept term if required. If no initial weights are provided, they are initialized to random values within a uniform distribution between -1 to 1.

1.3.3 Gradient Descent Optimization

The core of the model's training process is in the `‘gradient_descent‘` method, which iteratively updates the model's weights by performing gradient descent on the cost function. The gradient is computed with the `‘gradient‘` helper method, yielding $-2X^T(y - Xw)$, where X is the feature matrix, y is the vector of target values, and w is the vector of model weights.

The gradient descent algorithm iterates for a predefined number of iterations or until early stopping criteria are met. At each iteration, the algorithm updates the weights in the direction that reduces the cost function, with the magnitude of the update controlled by the learning rate. The learning rate is adjusted dynamically based on the progression of the error: it increases if the error decreases (1.3x), promoting faster convergence, and decreases if the error increases (0.9x), to prevent overshooting the minimum. Even if the model is initialized with an absurdly high learning rate, the model will auto-correct itself through the learning rate to ensure the minimum can be found.

The model keeps track of the cost function's value at each iteration, allowing the monitoring of the training process and potential diagnosis of issues such as non-convergence or overfitting.

1.3.4 Predictions

For making predictions, the `‘ind_predict‘` method computes the predicted value for a single instance, and the `‘predict‘` method extends this functionality to handle multiple instances. The prediction methods take into account whether an intercept term is included in the model and applies the current weights to the input features to produce the predictions.

1.4 Performance Analysis

In assessing the performance of my custom '`Linear_Regression`' class against scikit-learn's implementation on the UCI wine dataset, notable disparities are observed in their predictive outcomes. Specifically, the scikit-learn model yielded a total squared error of 800.668, compared to 805.228 from my model. A critical factor contributing to this discrepancy could be attributed to the difference in model configurations; the scikit-learn model was executed without fitting an intercept, whereas my custom implementation included an intercept term. This distinction in approach may account for the slight variation in prediction errors between the two models.

In [8]:
```
from sklearn.linear_model import LinearRegression
```

In [9]:
```
url_Wine = 'https://archive.ics.uci.edu/ml/machine-learning-database
wine = pd.read_csv(url_Wine, delimiter=';')
X = wine[['density','alcohol']]
y = wine.quality
```

In [10]:
```
lr = LinearRegression()
lr.fit(X,y)
## Squared Error with sklearn.
sum((lr.predict(X) - y)**2)
```

Out[10]:
```
800.6676988774321
```

In [13]:
```
lr.coef_
```

Out[13]:
```
array([34.82170159,  0.39144139])
```

In [11]:
```
clf = Linear_Regression(alpha = 1, num_iter = 5000000)
clf.fit(X,y)
```

...

In [12]:
```
sum((clf.predict(X) - y)**2)
```

Out[12]:
```
805.2283536684514
```

In [15]:
```
clf.coef
```

Out[15]:
```
matrix([[-0.35448323],
        [ 2.22057545],
        [ 0.3623942 ]])
```

2 Logistic Regression

2.1 Logistic Regression Background

Logistic regression is a statistical model used primarily for classification problems. Unlike linear regression which predicts a continuous output, logistic regression is designed for binary outcomes (0 or 1), predicting the probability that a given input point belongs to a particular category.

The crux of logistic regression is the logistic function - the sigmoid function - which maps any real-valued number into a value between 0 and 1, interpreted as the probability of the dependent variable belonging to a particular class.

2.2 Mathematical Formulation

The logistic regression model can be formulated as

$$\hat{y} = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + ... + \beta_n x_n)}}$$

$\hat{y}$ is the predicted probability, β_0 is the intercept, $\beta_1, \ldots, \beta)n$ are the coefficients, and $x_1, \ldots, x_n$ are the independent variables. The coefficients are determined using the maximum likelihood estimation (MLE) principle, aiming to maximinze the likelihood of the observed sample data.

2.2.1 Relationship to the Odds Ratio

The logistic regression model expresses the logarithm of the odds ratio of the positive class to the negative class as a linear combination of the input features

$$log(\frac{\hat{y}}{1 - \hat{y}}) = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n$$

This relationship is known as the logit function, which links the probability scale, bounded between 0 and 1, to the entire range of real numbers.

2.2.2 Loss Function for Optimization

The model's goodness-of-fit is assessed using the cross-entropy loss function, which for logistic regression is:

$$L(\beta) = -\sum_{i=1}^{n} [y_i log(\hat{y_i}) + (1 - y_i) log(1 - \hat{y_i})]$$

Minimizing this loss function equates to maximizing the likelihood of the data under the estimated model.

2.2.3 Optimization via Gradient Descent

Using the loss function for optimization, we can calculate the gradient after a few simple calculations.

$$\frac{\partial L(\beta)}{\partial \beta_j} = -\sum_{i=1}^{n} \left[y_i \frac{x_{ij}}{\hat{y_i}} - (1 - y_i) \frac{x_{ij}}{1 - \hat{y_i}} \right] (\hat{y_i}(1 - \hat{y_i}))$$

$$\frac{\partial L(\beta)}{\partial \beta_j} = \sum_{i=1}^{n} [x_{ij}(\hat{y_i} - y_i)]$$

$$\nabla_\beta L = X^T(\hat{y} - y)$$

2.3 Implementation

Logistic regression is implemented as a class `‘Logistic_Regression‘` in Python, capturing the essence of binary classification through the logistic function. It leverages NumPy for mathematical computations, ensuring efficient operations.

2.3.1 Initialization and Gradient Computation

Initialization sets up model parameters, including weights and bias, and prepares the model to fit the training data. Gradient computation is then used to calculate the derivatives of the cross-entropy loss function, which informs parameter updates.

2.3.2 Optimization and Prediction

Through gradient descent, the model iteratively adjusts its parameters to minimize the loss function until convergence, as determined by an early stopping condition. The `‘predict‘` function employs the sigmoid activation to estimate class probabilities, which are then mapped to binary outcomes for classification. As the central part of the model, `gradient_descent_logistic` is crucial to take note of.

```
def gradient_descent_logistic(self, alpha, num_pass,
    early_stop=0, standardized = True):
    if standardized:
        self.X = z_standardize(self.X)

    for i in range(num_pass):
        grad_theta, grad_b = self.gradient(self.X, self.y, self.theta, self.b)
        temp_theta = self.theta - alpha * grad_theta
        temp_b = self.b - alpha * grad_b

        y_hat = sigmoid(np.dot(self.X, temp_theta) + temp_b)
        temp_error = -np.mean(self.y * np.log(y_hat)
                        + (1 - self.y) * np.log(1 - y_hat))

        if i > 0 and abs(prev_error - temp_error) < early_stop:
            break

        self.theta = temp_theta
        self.b = temp_b
        prev_error = temp_error
    return self.theta, self.b
```

Initially, if the `‘standardized‘` argument is set to `‘True‘`, the input matrix `‘self.X‘` is standardized. This is an important preprocessing step to ensure that features contribute equally to the regression and avoid potential bias towards features with larger scales.

Then, it iteratively goes through a loop, updating weights and bias through the `‘self.gradient‘` function, as shown in Linear Regression. Once the stopping criterion is met or the specified number of iterations is reached (`‘num_pass‘`), the method finalizes the model's parameters.

2.4 Performance Analysis

The custom logistic regression model achieved a 58.68% accuracy rate after training with gradient descent, while scikit-learn's implementation yielded a higher accuracy of 67.85%. This discrepancy suggests the custom model may benefit from further tuning or the inclusion of regularization techniques that are standard in scikit-learn's approach.

3 Naïve Bayes

3.1 Naïve Bayes background

The Naive Bayes classifier is a probabilistic machine learning model used for classification tasks, which is based on Bayes' theorem with the assumption for independence among predictors. Despite its simplicity, Naive Bayes can outperform more sophisticated classification methods.

3.2 Mathematical Foundation

The Naive Bayes classifier is underpinned by Bayes' Theorem, a fundamental principle in probability theory that describes the relationship between the conditional probabilities of statistical quantities. For a classification problem, Bayes' Theorem can be expressed as

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)}$$

Where:

- $P(C|X)$ is the posterior probability of class C given predictor X.
- $P(C)$ is the prior probability of class C.
- $P(X|C)$ is the likelihood which is the probability of predictor X given class C.
- $P(X)$ is the prior probability of predictor X.

In the Naive Bayes classifier, we simplify the computation by assuming that the predictors (features) are independent given the class label. This assumption allows the joint probability model to be expressed as a product of individual probabilities:

$$P(X_1, X_2, ..., X_n|C) = P(X_1|C)P(X_2|C)...P(X_n|C)$$

Substituting this into Bayes' Theorem gives us:

$$P(C|X_1, X_2, ..., X_n) = \frac{P(C)\prod_{i=1}^{n} P(X_i|C)}{P(X_1, X_2, ..., X_n)}$$

As the denominator $P(X_1, X_2, \ldots, X_n)$ is constant for all classes, it can be ignored for classification purposes, leading to the simplified model:

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

The Naive Bayes classifier then selects the class with the highest posterior probability as the predicted class.

Estimating Probabilities:

- **Prior Probabilities**: $P(C)$ can be estimated by the relative frequency of class C in the training set.
- **Conditional Probabilities**: $P(X_i|C)$ can be estimated from the training data by considering the frequency of each feature X_i appearing in samples from class C.

3.3 Implementation

The Naive Bayes class, implemented in Python, consists of methods for fitting the method to the training data and predicting labels for new data. The `'fit'` method calculates the prior probabilities of the classes and the likelihoods of the features given the class labels. The `'predict'` method uses these probabilities to compute the posterior probabilities of the classes for new instances and predicts the class with the highest probability. To handle unseen feature values, a small probability factor of 10^{-9} is introduced, thus maintaining non-zero class probabilities even when an unfamiliar feature value appears across all classes.

```
def fit(self, X_train, y_train):
    # Compute the prior distribution of all y labels
    self.prior = dict()
    for y in y_train:
        prior_key = f'Y = {y}'
        if prior_key in self.prior.keys():
            self.prior[prior_key] += 1
        else:
            self.prior[prior_key] = 1

    # Normalize the prior distribution
    for key in self.prior.keys():
        self.prior[key] /= len(y_train)

    # Compute the likelihood distribution of all X_j given Y
    self.likelihood = dict()
    for x, y in zip(np.array(X_train), y_train):
        for j in range(len(x)):
            likelihood_key = f'X{j} = {x[j]} | Y = {y}'
            if likelihood_key in self.likelihood.keys():
                self.likelihood[likelihood_key] += 1
            else:
                self.likelihood[likelihood_key] = 1

    # Normalize the likelihood distribution
    for key in self.likelihood.keys():
        self.likelihood[key] /= len(y_train)
```

The `'fit'` method is the very backbone of this entire model as it calculates prior probabilities, estimates likelihood probabilities, normalizes likelihoods, and features the independence assumption as it multiplies individual feature probabilities to obtain the likelihood of the feature vector. At its core, the method encapsulates the fundamental Naive Bayes training algorithm and directly impacts the classifier's performance.

3.4 Performance Analysis

In the evaluation of classification models for the UCI balance scale dataset, the custom Naive Bayes classifier reached an accuracy of 82.61%, while the Gaussian Naive Bayes classifier from the scikit-learn library achieved an accuracy of 89.37%. The superior performance of the Gaussian Naive Bayes classifier may be attributed to its assumption of a Gaussian distribution for the likelihood of the features, which appears to align better with the underlying distribution of the dataset used. This suggests that for datasets where the normality assumption holds for features, Gaussian Naive Bayes may provide a more accurate classification model compared to a basic Naive Bayes classifier without such an assumption. Further investigation into feature distributions and additional parameter tuning could potentially enhance the custom Naive Bayes classifier's performance.

4 KMeans Clustering

4.1 Background

KMeans is a widely used clustering algorithm that partitions data into K number of distinct, non-overlapping clusters. It assumes cluster homogeneity and attempts to minimize the variance within each cluster. Usually, it is computationally cheap to do and easy to understand, meaning it is most often the first unsupervised machine learning model many people are exposed to.

4.2 Mathematical Foundation

The objective of KMeans is to find groupings $\{S_k\}_{k=1}^{K}$ in the data that minimizes the within-cluster sum of squares (WCSS), defined as

$$\text{WCSS} = \sum_{k=1}^{K} \sum_{x_i \in S_k} ||x_i - \mu_k||^2$$

Where x_i are the data points, S_k represents the clusters, and μ_k is the centroid of points in S_k.

The centroid μ_k of each cluster is computed as the mean of all points x_i assigned to cluster C_k, formulated as

$$\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$$

where $|C_k|$ is the number of data points in cluster $C_k k$.

4.3 Implementation

All the magic in the KMeans class occurs in the train method. It first randomly chooses the initial centers `centers = X[np.random.choice(r, self.k, replace=False)]`, then iterates through the for loop for `‘self.num_iter‘` times.

In each iteration of the for loop, it assigns data points to the nearest centroid, creates new centers, recalculates the centroids, and finally checks for convergence to exit the loop.

4.4 Performance Analysis

In evaluating the iris dataset from the scikit-learn datasets, my KMeans model had an average silhouette score of around 61%. Although it doesn't seem like the accuracy is that high, it's important to consider the nature of clustering and the silhouette score itself. The silhouette score is a measure of how similar an object is to its own cluster compared to other clusters, with a score of 1 indicating a perfect match, 0 indicating overlapping clusters, and negative values indicating potentially incorrect assignments. A score of 61% suggests that, on average, data points are more similar to their own cluster than to neighboring clusters, indicating a reasonable level of cluster separation and cohesion.

This score should be interpreted in the context of the iris dataset, known for its one cluster being distinct, while the other two are somewhat mixed. Given this inherent overlap, a silhouette score in this range is quite respectable. It reflects the model's ability to distinguish the separate cluster while managing the overlap between the more similar clusters reasonably well.

5 Gaussian Mixture

5.1 Gaussian Mixture Background

Gaussian Mixture is a probablistic model that assumes data is generated from a mixture of several Gaussian distributions with unknown parameters. Gaussian Mixture accommodates mixed membership of points to clusters.

5.2 Mathematical Foundation

5.2.1 Probability Distribution of a Mixture Model

A Gaussian Mixture Model is a weighted sum of M component Gaussian densities, given by the equation:

$$p(x|\lambda) = \sum_{i=1}^{M} \pi_i \mathcal{N}(x|\mu_i, \Sigma_i)$$

- x represents the data points.
- $\lambda = \{\pi_i, \mu_i, \Sigma_i\}_{i=1}^{M}$ denotes the model parameters, including the mixture weights π_i, means μ_i, and covariance matrices $\sum_i$ of each component i.
- π_i are the mixture weights for each component, subject to the constraint $\sum_{i=1}^{M} \pi_i = 1$, ensuring that the mixture model represents a valid probability distribution.
- $\mathcal{N}(x|\mu_i, \Sigma_i)$ is the probability density function of the Gaussian distribution for component i.

5.2.2 Expectation Step (E-Step)

Calculates the posterior probabilities (responsibilities) that each data point belongs to each of the mixture components, based on the current parameter estimates. This is given by

$$\gamma(z_{ik}) = \frac{\pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k)}{\sum_{j=1}^{M} \pi_j \mathcal{N}(x_i|\mu_j, \Sigma_j)}$$

where $\gamma(z_{ik})$ is the responsibility of component k for data point i.

5.2.3 Maximization Step (M-Step)

Re-estimate parameters $\{\pi_i, \mu_i, \Sigma_i\}$ using current responsibilities. The new parameter values are

$$\mu_k^{new} = \frac{1}{N_k} \sum_{i=1}^{N} \gamma(z_{ik}) x_i$$

$$\Sigma_k^{new} = \frac{1}{N_k} \sum_{i=1}^{N} \gamma(z_{ik})(x_i - \mu_k^{new})(x_i - \mu_k^{new})^T$$

$$\pi_k^{new} = \frac{N_k}{N}$$

5.2.4 Convergence of the EM Algorithm

The EM algorithm iterates between these two steps until the change in the log-likelihood function $\ln p(X|\lambda)$ or the change in parameters between iterations is below a predefined threshold, indicating convergence to a local maximum.

5.2.5 Log-Likelihood Function

Maximizing the following function with respect to the parameters in the EM algorithm allows the model to capture the underlying structure of the data.

$$\ln p(X|\lambda) = \sum_{i=1}^{N} \ln \left(\sum_{k=1}^{M} \pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k) \right)$$

5.3 Implementation

In this implementation, the Gaussian Mixture Model (GMM) begins its parameter initialization process leveraging the KMeans algorithm. This serves as a heuristic to provide a robust starting point for the iterative Expectation-Maximum process. Following the initialization, the model employs Gaussian probability densities to describe each cluster, allowing for a more nuanced representation of the data that accounts for both the central tendency and dispersion.

By initializing GMM components with KMeans and modeling each cluster with a Gaussian distribution, the implementation effectively captures the underlying structure in complex datasets, accommodating clusters of different sizes and shapes.

In other words, the code takes the following structure:

- Initialization with KMeans
- Initialization of other parameters (covariance matrices, mixture weights)
- Expectation Step (E-Step)
- Maximization Step (M-Step)
- Convergence Check
- Final Model

5.4 Performance Analysis

The custom GMM implementation exhibits cluster centers and sample log-likelihood scores that vary from scikit-learn's GMM results. While the differences in cluster means are relatively minor, there is a noticeable discrepancy in the log-likelihood scores assigned to individual samples. This suggests that the fitting process and the resulting parameters of the two models differ, likely due to distinct initialization processes, convergence thresholds, or computational precision.

```
Means by sklearn:
 [[ 6.6973974   1.5535085  -8.51537837]
 [-6.80711161 -2.15182238  5.09952672]
 [-9.94834157 -6.75207749 -5.8639604 ]]
Means by our implementation:
 [[ 6.75671816  1.23242737 -7.54452744]
 [-8.38121788 -4.45708075 -0.39448188]
 [ 6.6266031   1.93688846 -9.67457967]]
Scores by sklearn:
 [ -7.32774504  -8.7718697   -6.03675619  -6.79383387  -6.93562459
  -6.21635595 -10.36893116  -8.65132105  -8.26141278  -7.00359507
 -10.30637686 -10.77400816  -7.76403303  -7.9298516   -7.09962188
  -7.66406257  -8.52427873 -10.25370847  -6.78587063  -7.65425758]
Scores by our implementation:
 [ -9.00398012  -8.77090634  -7.66377294  -6.89067966  -7.85394744
  -6.17940312 -10.4835465   -7.70488038 -10.68910817  -8.61975374
 -10.77824968 -11.26331584  -7.89749276  -7.20351544  -6.99216467
  -7.78967911  -9.40198199 -11.46612034  -6.69244778  -8.02126374]
```

6 References and Code

6.1 Linear Regression

- https://github.com/m1nce/ml-models/tree/main/LinearRegression
- https://www.youtube.com/watch?v=4b4MUYve_U8&t=2340s
- https://en.wikipedia.org/wiki/Linear_regression

6.2 Logistic Regression

- https://github.com/m1nce/ml-models/tree/main/LogisticRegression
- https://www.youtube.com/watch?v=het9HFqo1TQ&t=1940s
- https://en.wikipedia.org/wiki/Logistic_regression

6.3 Naïve Bayes

- https://github.com/m1nce/ml-models/tree/main/NaiveBayes
- https://www.youtube.com/watch?v=nt63k3bfXS0
- https://en.wikipedia.org/wiki/Naive_Bayes_classifier

6.4 KMeans and GMM

- https://github.com/m1nce/ml-models/tree/main/KMeansGMM
- https://www.youtube.com/watch?v=rVfZHWTwXSA&t=1599s