Project 2 — Coordinate Descent

Gayatri Kharche
gkharche@ucsd.edu

Abstract

Abstract: This project delves into the utilization of coordinate descent optimization techniques for addressing logistic regression tasks, particularly in binary classification scenarios. By focusing on the wine dataset, this study examines two variations of coordinate descent: cyclic coordinate descent and random feature selection. Cyclic coordinate descent sequentially updates one coordinate at a time, iterating through all coordinates in each cycle, while random feature selection randomly selects a coordinate for update at each iteration, offering a stochastic approach to optimization. These methods provide alternative strategies for efficiently navigating the optimization landscape in logistic regression problems. Emphasizing the critical aspects of coordinate selection and update strategies, the research establishes a comprehensive framework for their implementation. Logistic regression models are trained using both cyclic and random feature selection coordinate descent methods, with a standard logistic regression approach serving as a benchmark. The experimental findings demonstrate the efficacy of cyclic coordinate descent and random feature selection methods in minimizing the logistic loss function during training. Furthermore, the study scrutinizes the convergence patterns and computational efficiency of these methods, highlighting their competitive performance relative to the standard logistic regression approach. Additionally, the project explores a sparse coordinate descent method, which aims to find solutions with a limited number of non-zero coefficients, providing insights into its effectiveness and performance in logistic regression tasks.

1 Introduction

In this project, we introduce a systematic coordinate descent method tailored for addressing unconstrained optimization problems, particularly focusing on the optimization of logistic regression models for binary classification tasks. Our approach addresses the pivotal challenges of selecting coordinates to update and determining the updated values of model parameters (w_i) effectively.

We begin by presenting a structured framework for coordinate selection, opting for a cyclic strategy over random selection to ensure systematic parameter updates. By cycling through the coordinates sequentially, our method guarantees comprehensive coverage of parameter updates, potentially leading to faster convergence compared to random selection strategies. Subsequently, we detail the update process for w_i after selecting a coordinate i, where we utilize gradient information to minimize the loss function $L(w)$. This update rule, based on the gradient of the loss function with respect to the selected coordinate, facilitates continuous progress towards the optimal solution, gradually reducing the loss until convergence is achieved.

Our proposed coordinate descent method is versatile and applicable to a wide range of cost functions, accommodating both smooth and non-smooth functions. While differentiability of the cost function is advantageous for gradient computation, our method does not strictly require it, ensuring adaptability to various optimization scenarios.

Expanding our investigation, we delve into specific variants of coordinate descent for logistic regression, namely cyclic coordinate descent and random feature selection. The former operates by iteratively updating one coordinate at a time in a structured manner, while the latter introduces

stochasticity by randomly selecting coordinates for update at each iteration. By exploring these methods alongside traditional logistic regression approaches, we aim to provide a comprehensive understanding of different optimization strategies in binary classification scenarios.

Furthermore, our study extends to include sparse coordinate descent methods, aiming to identify solutions with a limited number of non-zero coefficients. Through rigorous experimentation and analysis, we seek to elucidate the strengths and limitations of each optimization technique in terms of convergence patterns, computational efficiency, and overall performance, ultimately providing valuable insights for logistic regression optimization in binary classification tasks.

2 Methodology

2.0.1 Standard Logistic Regression

The methodology for standard logistic regression starts with loading and preparing the wine dataset. Specifically, only data points from the first two classes are retained to form a binary classification problem. Additionally, a constant term is added to the feature matrix by appending a column of ones, ensuring the model can learn a bias term. The features are then standardized using the 'StandardScaler()' function to center them around zero and scale them to unit variance, which helps stabilize optimization.

After data preprocessing, the logistic regression model is initialized with specific parameters: no regularization penalty and a maximum of 1000 iterations. These settings are selected to create a baseline logistic regression model without regularization, allowing for direct comparison with subsequent optimization methods. Next, the initialized logistic regression model is trained on the standardized training data. Through an iterative optimization process, the model learns optimal parameters (weights) that minimize the logistic loss function. This process typically involves using optimization algorithms such as gradient descent or its variants to iteratively update the model parameters. Once trained, the model is used to predict labels for the training data. These predicted labels are then compared against the true labels to calculate the logistic loss using the 'logloss()' function. This loss value measures the difference between predicted probabilities and actual labels, providing insights into the model's performance in fitting the training data.

Finally, the training loss is reported, offering an indication of the standard logistic regression model's effectiveness. This methodology serves as the basis for subsequent evaluations and comparisons with alternative optimization techniques, such as cyclic coordinate descent and random feature selection.

2.0.2 Cyclic Coordinate Descent Method

The methodology for Cyclic Coordinate Descent (CCD) optimization in logistic regression involves several key steps. Initially, the logistic loss function is defined to quantify the difference between predicted probabilities and actual labels, aiding in model assessment. Following this, the gradient of the logistic loss function is calculated to determine the direction and magnitude of parameter adjustments required for optimization. The CCD algorithm is then implemented to update model parameters iteratively, selecting one feature at a time in a cyclic manner and adjusting the parameter based on the computed gradient. Throughout this process, the logistic loss is continuously monitored to assess optimization progress. The logistic regression model is trained using the CCD algorithm on standardized training data, allowing for iterative refinement of parameters to minimize loss. Finally, the trained model's performance is evaluated by computing the logistic loss on the training data, providing insights into the effectiveness of the CCD method in optimizing logistic regression models for binary classification tasks. This comprehensive methodology offers a systematic approach to parameter optimization, essential for achieving optimal performance in logistic regression-based binary classification scenarios.

2.0.3 Cyclic Coordinate Descent Random method

The methodology for the Cyclic Coordinate Descent Random (CCDR) method involves a structured approach to optimizing logistic regression models for binary classification tasks. Initially, the logistic loss function is defined to measure the difference between predicted probabilities and actual labels. Then, the gradient of this function is computed to determine the necessary adjustments to model parameters for optimization. In the CCDR algorithm, iterations are performed over a set number of epochs. During each epoch, a random coor-

dinate index is selected for parameter update. This involves computing the gradient and adjusting the corresponding parameter accordingly. This iterative process continues, with the logistic loss continually monitored to track optimization progress. Using standardized training data, the logistic regression model is trained using the CCDR method, allowing for the iterative refinement of parameters to minimize loss. Finally, the model's performance is evaluated by computing the logistic loss on the training data, providing insights into the effectiveness of the CCDR method in optimizing logistic regression models for binary classification tasks. This methodology presents a systematic approach to parameter optimization, essential for achieving optimal performance in logistic regression-based binary classification scenarios.

2.0.4 Sparse coordinate descent

The methodology for sparse coordinate descent begins by initializing the weight vector $\mathbf{w}$ with zeros and establishing a list to record loss values during optimization. Randomly selecting k coordinate indices from the feature space, without repetition, follows. The process involves iterations over a defined maximum number of epochs, with each epoch iterating through the chosen coordinate indices. At each selected coordinate index, the gradient of the logistic loss function relative to the corresponding parameter is computed, and the parameter is updated using a predetermined step size α. Subsequently, the logistic loss is calculated to evaluate model performance, with the resulting loss appended to the list for tracking optimization progress. Post-optimization, the trained logistic regression model's performance is assessed using the computed loss values, offering insights into the attainment of sparsity in the solution vector $\mathbf{w}$. Additionally, a revised method can be suggested where k serves as an input parameter, enabling the specification of the desired sparsity level. However, it's essential to note that this method might not always discover the optimal k-sparse solution in convex cost functions due to factors such as problem nuances and optimization parameters. Application to the wine dataset involves initializing the logistic regression model with sparse coordinate descent, varying k values, and evaluating method performance by computing loss values for selected k values to gauge success in achieving sparsity in the solution vector.

3 Implementation Details

The implementation details for the three methods, cyclic coordinate descent, random feature coordinate descent, and sparse coordinate descent, followed a consistent approach. Initially, functions were defined for each method, incorporating essential parameters like the input data X and labels y, maximum epochs, and learning rate α. In cyclic coordinate descent, an iterative loop sequentially traversed each coordinate index within each epoch, while random feature coordinate descent randomly selected a coordinate index for update in each iteration. Both methods adjusted the weight vector $\mathbf{w}$ based on the gradient of the logistic loss function. Sparse coordinate descent introduced an additional parameter k to denote the desired sparsity level. This method initially selected k random coordinate indices and updated only those indices within each epoch. Then the trained models underwent evaluation using pertinent performance metrics, such as loss values, to gauge their efficacy in logistic regression tasks.

The algorithms presented encompass diverse optimization methods tailored for logistic regression. The coordinate descent method optimizes logistic regression by iteratively updating individual parameters, cycling through each parameter until reaching convergence. This ensures that with each step, the loss function is minimized along the chosen parameter's direction. Through this sequential refinement of parameters, the method progressively approaches the optimal solution.

In contrast, the random feature coordinate descent method introduces randomness by selecting parameters for update randomly at each iteration. Instead of adhering to a fixed sequence, this approach randomly selects parameters, offering a stochastic alternative to the deterministic cyclic method. This randomness can offer benefits, especially when certain parameters significantly impact the loss function.

Extending coordinate descent to handle sparse solutions, the sparse coordinate descent method promotes solutions with fewer non-zero parameters. It achieves this by iteratively updating a predefined number of coordinates, selected randomly during each iteration. This random selection encourages exploration of the parameter space, potentially leading to simpler and more interpretable models.

Algorithm 1 Cyclic Coordinate Descent Method

Input:

- X: Input feature matrix of size $n \times d$, where n is the number of samples and d is the number of features.
- y: True label vector of size n.
- `max_epochs`: Maximum number of epochs for training.
- `alpha`: Learning rate.

Output:

- w: Weight vector of size d.
- `losses`: List of logistic loss values at each iteration.

Procedure:

1. Initialize the weight vector w with zeros.
2. Initialize an empty list `losses` to track loss values.
3. For each epoch $t = 1, 2, \ldots,$ `max_epochs`:
   (a) Compute the logistic loss $L(w)$ using the current weight vector w:
   $$L(w) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$
   where $\hat{y}_i = \frac{1}{1+\exp(-X_i \cdot w)}$.
   (b) Append $L(w)$ to the list `losses`.
   (c) For each feature $j = 1, 2, \ldots, d$:
      i. Compute the gradient of the logistic loss with respect to the j-th feature:
      $$\frac{\partial L}{\partial w_j} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i) \cdot x_{ij}$$
      where x_{ij} is the j-th feature of sample i.
      ii. Update the j-th weight w_j using the cyclic coordinate descent update rule:
      $$w_j \leftarrow w_j - \texttt{alpha} \cdot \frac{\partial L}{\partial w_j}$$
4. Return the final weight vector w and the list of losses `losses`.

Algorithm 2 Cyclic Coordinate Descent Method with Random Feature Selection

Input:

- X: Input feature matrix of size $n \times d$, where n is the number of samples and d is the number of features.
- y: True label vector of size n.
- `max_epochs`: Maximum number of epochs for training.
- `alpha`: Learning rate.

Output:

- w: Weight vector of size d.
- `losses`: List of logistic loss values at each iteration.

Procedure:

1. Initialize the weight vector w with zeros.
2. Initialize an empty list `losses` to track loss values.
3. For each epoch $t = 1, 2, \ldots,$ `max_epochs`:
   (a) Compute the logistic loss $L(w)$ using the current weight vector w:
   $$L(w) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$
   where $\hat{y}_i = \frac{1}{1+\exp(-X_i \cdot w)}$.
   (b) Append $L(w)$ to the list `losses`.
   (c) For each epoch $t = 1, 2, \ldots,$ `max_epochs`:
      i. Select a random coordinate index j from 1 to d.
      ii. Compute the gradient of the logistic loss with respect to the j-th feature:
      $$\frac{\partial L}{\partial w_j} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i) \cdot x_{ij}$$
      where x_{ij} is the j-th feature of sample i.
      iii. Update the j-th weight w_j using the cyclic coordinate descent update rule:
      $$w_j \leftarrow w_j - \texttt{alpha} \cdot \frac{\partial L}{\partial w_j}$$
4. Return the final weight vector w and the list of losses `losses`.

Algorithm 3 Sparse Coordinate Descent for Logistic Regression

Input:

- X: Input feature matrix of size $n \times d$, where n is the number of samples and d is the number of features.
- y: True label vector of size n.
- k: Desired sparsity level.
- `max_epochs`: Maximum number of epochs for training.
- `alpha`: Learning rate.

Output:

- w: Weight vector of size d.
- `losses`: List of logistic loss values at each iteration.

Procedure:

1. Initialize the weight vector w with zeros.
2. Initialize an empty list `losses` to track loss values.
3. Randomly select k coordinate indices from 1 to d without replacement.
4. For each epoch $t = 1, 2, \ldots,$ `max_epochs`:
   (a) Compute the logistic loss $L(w)$ using the current weight vector w:
   $$L(w) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$
   where $\hat{y}_i = \frac{1}{1+\exp(-X_i \cdot w)}$.
   (b) Append $L(w)$ to the list `losses`.
   (c) For each coordinate index j in the randomly selected k coordinate indices:
      i. Compute the gradient of the logistic loss with respect to the j-th feature:
      $$\frac{\partial L}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i) \cdot x_{ij}$$
      where x_{ij} is the j-th feature of sample i.
      ii. Update the j-th weight w_j using the coordinate descent update rule:
      $$w_j \leftarrow w_j - \texttt{alpha} \cdot \frac{\partial L}{\partial w_j}$$
5. Return the final weight vector w and the list of losses `losses`.

4 Convergence

The convergence of the cyclic coordinate selection method to the optimal loss relies on the convexity of the optimization problem and specific criteria and strong convexity. Within convex optimization, this method sequentially updates each coordinate, ensuring a monotonic decrease in the objective function until it reaches either a local or global minimum, thereby achieving convergence towards the optimal loss. However, if the objective function lacks convexity or the optimization problem doesn't meet the necessary conditions for convergence, convergence may not be guaranteed. At the maximum iteration, the loss approaches a local minimum.

5 Experimental results

5.1 Comparison between Logistic Regression and Coordinate Descent methods

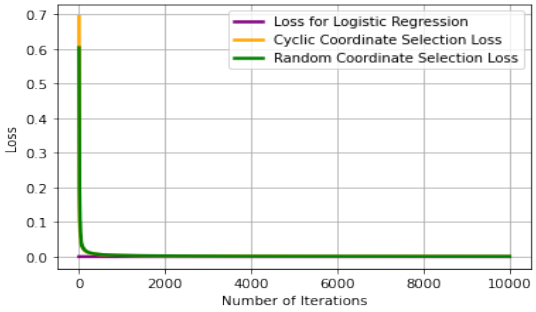


Figure 1: Plot

The comparison between Logistic Regression and Coordinate Descent methods as shown in Figure 1 reveals that despite their differences in convergence speed, all approaches eventually reach similar loss values. Logistic Regression demonstrates the quickest convergence, followed by Coordinate Descent with Strategic Selection, and then Coordinate Descent with Random Selection. A closer examination of the loss curves shows that Logistic Regression achieves convergence within the initial epochs. On the other hand, Coordinate Descent with Strategic Selection converges after a few epochs, closely matching the loss of Logistic Regression. However, Coordinate Descent with Random Selection, while initially showing similar convergence, diverges in later epochs, resulting in a slightly higher final loss. In summary, Logistic Regression and Coordinate Descent with Strategic Selection produce comparable results, while Co-

ordinate Descent with Random Selection exhibits a slightly higher loss in the end.

The differences observed in the convergence rates and final loss values between Logistic Regression and Coordinate Descent methods stem from various contributing factors.

Firstly, the choice of optimization algorithm plays a significant role. Logistic Regression often employs algorithms like gradient descent, which are tailored for smooth, continuous functions, resulting in faster convergence. In contrast, Coordinate Descent methods update parameters sequentially, which may lead to slower convergence, especially in datasets with highly correlated features.

Moreover, the selection strategy for updating coordinates in Coordinate Descent is crucial. Strategic Selection prioritizes updating coordinates with a more significant impact on reducing loss, potentially facilitating faster convergence. Conversely, Random Selection updates coordinates indiscriminately, which can lead to less efficient optimization and slower convergence.

The choice of step size (learning rate) in Coordinate Descent methods also influences convergence. Well-tuned step sizes can expedite convergence, while poorly chosen ones may result in oscillations or slow convergence.

Furthermore, the correlation among features in the dataset affects optimization performance. Highly correlated features may cause Coordinate Descent methods to converge slower due to redundant updates and oscillations.

Additionally, the initial values of the weight vector in Coordinate Descent methods can impact convergence. If the initial values are distant from the optimal solution, it may take longer for the algorithm to converge.

Overall, while Logistic Regression benefits from specialized optimization algorithms, Coordinate Descent methods offer flexibility and scalability, albeit with potentially slower convergence. The selection between these methods depends on the dataset's characteristics and the trade-offs between convergence speed and computational efficiency.

5.2 Sparse Coordinate Descent for Logistic Regression

The results demonstrate the impact of varying the sparsity level k on the logistic loss function in the context of sparse coordinate descent applied to logistic regression. It's essential to note that in the sparse coordinate descent function, the k coordinate indices are randomly selected without replacement.

k	Train Loss
1	8.712537905800993
2	4.122658708601228
4	2.1516759755725117
10	0.27036416234905775

Table 1: Loss values for different values of k

As k increases, indicating a higher degree of sparsity in the solution vector w, the training loss generally decreases. For $k = 1$, the loss value is the highest, indicating that the model with only one nonzero entry in w may not capture enough information from the data, leading to higher loss. As k increases to 2, 4, and 10, the loss values decrease successively, suggesting that increasing sparsity allows the model to capture more relevant information from the data, resulting in lower loss.

However, it's important to note that there may be a trade-off between sparsity and model performance. While increasing sparsity can lead to lower loss values, excessively sparse models may sacrifice predictive accuracy by discarding useful information. Therefore, the choice of k should be carefully balanced to achieve the desired level of sparsity without compromising model performance.

6 Critical Evaluation

While the study provided valuable insights, there are areas for further improvement. Fine-tuning hyperparameters such as the learning rate α and regularization strength could enhance convergence speed and overall performance. Additionally, exploring alternative coordinate selection strategies, such as Gauss-Southwell or greedy coordinate descent, may offer improvements in optimization efficiency and convergence properties. Incorporating regularization techniques like L1 or L2 regularization could help mitigate overfitting and improve model generalization capabilities.

Furthermore, exploring the application of coordinate descent methods in conjunction with advanced optimization techniques, such as stochastic gradient descent with momentum or Adam, could provide further performance gains. Additionally, considering the impact of feature scaling

and preprocessing techniques on optimization performance warrants attention in future studies.

In conclusion, while coordinate descent methods show promise for optimizing logistic regression models, further research and improvement are needed. Addressing these areas and exploring innovative optimization strategies can advance the state-of-the-art in logistic regression optimization and facilitate the development of more accurate and efficient binary classification models.

7 Conclusion

In summary, this project explores the application of coordinate descent optimization methods for logistic regression in binary classification scenarios. Through an investigation using the wine dataset, which includes chemical analysis data from two distinct wine cultivars, we analyze cyclic coordinate descent, random feature selection, and sparse coordinate descent techniques. These methods offer different ways to efficiently optimize logistic regression models.

Our results indicate that logistic regression achieves the fastest convergence, while coordinate descent methods, particularly with strategic selection, can closely approximate its performance. However, random selection during coordinate descent may result in slightly higher final loss values due to divergence in later epochs. Furthermore, our exploration of sparse coordinate descent underscores the trade-off between achieving sparsity and maintaining model performance, highlighting the importance of carefully selecting the sparsity level (k).

Our critical evaluation suggests several areas for further improvement, including fine-tuning hyperparameters, exploring alternative coordinate selection strategies, and incorporating regularization techniques. Additionally, integrating coordinate descent methods with advanced optimization techniques and considering the impact of feature scaling and preprocessing methods could enhance optimization performance.

In conclusion, while coordinate descent methods offer promising avenues for optimizing logistic regression, ongoing research and refinement are necessary to fully leverage their potential. By addressing these areas and advancing optimization strategies, we can develop more accurate and efficient logistic regression models for binary classification tasks.

251aproject2

February 27, 2024

1 Project 2 - Coordinate Descent

1.0.1 Utilizes coordinate descent optimization for logistic regression in binary classification.

1.0.2 Analyzes cyclic and random feature selection methods on the wine dataset.

- Cyclic: updates one coordinate at a time sequentially.
- Random: selects coordinates randomly for update. ### It also:
- Benchmarks against standard logistic regression.
- Explores sparse coordinate descent for solutions with limited non-zero coefficients.

[1]:
```
import numpy as np
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, log_loss
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt
```

[2]:
```
pip install ucimlrepo
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: ucimlrepo in
/home/gkharche/.local/lib/python3.9/site-packages (0.0.3)
Note: you may need to restart the kernel to use updated packages.
```

1.1 Importing the dataset into the code

[3]:
```
from ucimlrepo import fetch_ucirepo

# fetch dataset
wine = fetch_ucirepo(id=109)

# data (as pandas dataframes)
X = wine.data.features
y = wine.data.targets

# metadata
```

```
print(wine.metadata)

# variable information
print(wine.variables)
```

```
{'uci_id': 109, 'name': 'Wine', 'repository_url':
'https://archive.ics.uci.edu/dataset/109/wine', 'data_url':
'https://archive.ics.uci.edu/static/public/109/data.csv', 'abstract': 'Using
chemical analysis to determine the origin of wines', 'area': 'Physics and
Chemistry', 'tasks': ['Classification'], 'characteristics': ['Tabular'],
'num_instances': 178, 'num_features': 13, 'feature_types': ['Integer', 'Real'],
'demographics': [], 'target_col': ['class'], 'index_col': None,
'has_missing_values': 'no', 'missing_values_symbol': None,
'year_of_dataset_creation': 1992, 'last_updated': 'Mon Aug 28 2023',
'dataset_doi': '10.24432/C5PC7J', 'creators': ['Stefan Aeberhard', 'M. Forina'],
'intro_paper': {'title': 'Comparative analysis of statistical pattern
recognition methods in high dimensional settings', 'authors': 'S. Aeberhard, D.
Coomans, O. Vel', 'published_in': 'Pattern Recognition', 'year': 1994, 'url': 'h
ttps://www.semanticscholar.org/paper/83dc3e4030d7b9fbdbb4bde03ce12ab70ca10528',
'doi': '10.1016/0031-3203(94)90145-7'}, 'additional_info': {'summary': 'These
data are the results of a chemical analysis of wines grown in the same region in
Italy but derived from three different cultivars. The analysis determined the
quantities of 13 constituents found in each of the three types of wines.
\r\n\r\nI think that the initial data set had around 30 variables, but for some
reason I only have the 13 dimensional version. I had a list of what the 30 or so
variables were, but a.)  I lost it, and b.), I would not know which 13 variables
are included in the set.\r\n\r\nThe attributes are (dontated by Riccardo Leardi,
riclea@anchem.unige.it )\r\n1) Alcohol\r\n2) Malic acid\r\n3) Ash\r\n4)
Alcalinity of ash  \r\n5) Magnesium\r\n6) Total phenols\r\n7) Flavanoids\r\n8)
Nonflavanoid phenols\r\n9) Proanthocyanins\r\n10)Color
intensity\r\n11)Hue\r\n12)OD280/OD315 of diluted wines\r\n13)Proline \r\n\r\nIn
a classification context, this is a well posed problem with "well behaved" class
structures. A good data set for first testing of a new classifier, but not very
challenging.           ', 'purpose': 'test', 'funded_by': None,
'instances_represent': None, 'recommended_data_splits': None, 'sensitive_data':
None, 'preprocessing_description': None, 'variable_info': 'All attributes are
continuous\r\n\t\r\nNo statistics available, but suggest to standardise
variables for certain uses (e.g. for us with classifiers which are NOT scale
invariant)\r\n\r\nNOTE: 1st attribute is class identifier (1-3)', 'citation':
None}}
                          name     role         type demographic  \
0                        class   Target  Categorical        None
1                      Alcohol  Feature   Continuous        None
2                    Malicacid  Feature   Continuous        None
3                          Ash  Feature   Continuous        None
4              Alcalinity_of_ash  Feature   Continuous        None
5                      Magnesium  Feature      Integer        None
```

```
6                  Total_phenols  Feature   Continuous        None
7                     Flavanoids  Feature   Continuous        None
8           Nonflavanoid_phenols  Feature   Continuous        None
9                Proanthocyanins  Feature   Continuous        None
10               Color_intensity  Feature   Continuous        None
11                           Hue  Feature   Continuous        None
12  OD280_OD315_of_diluted_wines  Feature   Continuous        None
13                       Proline  Feature      Integer        None

   description units missing_values
0         None  None             no
1         None  None             no
2         None  None             no
3         None  None             no
4         None  None             no
5         None  None             no
6         None  None             no
7         None  None             no
8         None  None             no
9         None  None             no
10        None  None             no
11        None  None             no
12        None  None             no
13        None  None             no
```

[4]:
```python
# Load the wine dataset and preprocess the data
wine = load_wine()
X, y = wine.data[:130, :], wine.target[:130]  # Using only the first two classes
X = np.concatenate((np.ones((X.shape[0], 1)), X), axis=1)  # Add intercept term
X_train, y_train = X, y  # Assign preprocessed data to training variables
scaler = StandardScaler()  # Initialize StandardScaler
X_train_scaled = scaler.fit_transform(X)  # Scale feature matrix
```

1.2 1. Standard Logistic Regression

- The methodology for standard logistic regression involves initial data preparation by loading and preprocessing the wine dataset, focusing on binary classification.
- After standardizing the features and initializing the logistic regression model with specific parameters, training commences using iterative optimization methods.
- The trained model predicts labels for the training data, and the logistic loss is computed to assess its performance.
- This methodology serves as a baseline for comparing alternative optimization techniques like cyclic coordinate descent and random feature selection.

[5]:
```python
# Initialize the logistic regression model
binary_model = LogisticRegression(penalty="none", max_iter=1000)
```

```
# Fit the model to the training data
binary_model.fit(X_train_scaled, y_train)

# Predict the training labels
y_pred = binary_model.predict(X_train_scaled)

# Calculate the training loss
loss = log_loss(y_true=y_train, y_pred=y_pred)

# Print the training loss
print(f'Training loss for {loss}')
```

```
Training loss for 9.992007221626415e-16
```

1.3 2. Coordinate descent method

The methodology for Cyclic Coordinate Descent (CCD) optimization in logistic regression involves several key steps:

- Define the logistic loss function to quantify the difference between predicted probabilities and actual labels.
- Calculate the gradient of the logistic loss function to determine the direction and magnitude of parameter adjustments.
- Implement the CCD algorithm to iteratively update model parameters, selecting one feature at a time in a cyclic manner and adjusting the parameter based on the computed gradient.
- Continuously monitor the logistic loss to assess optimization progress.
- Train the logistic regression model using the CCD algorithm on standardized training data for iterative parameter refinement.
- Evaluate the trained model's performance by computing the logistic loss on the training data to assess the effectiveness of the CCD method in optimizing logistic regression models for binary classification tasks.

[6]:
```
def sigmoid(z):
    """Sigmoid function."""
    return 1 / (1 + np.exp(-z))
```

[7]:
```
def logistic_loss(X, y, w):
    """Logistic loss function."""
    n = len(y)
    y_pred = sigmoid(X.dot(w))
    loss = -np.sum(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred)) / n
    return loss
```

[8]:
```
def logistic_gradient(X, y, w):
    """Gradient of logistic loss function."""
    n = len(y)
    y_pred = sigmoid(X.dot(w))
    gradient = X.T.dot(y_pred - y) / n
```

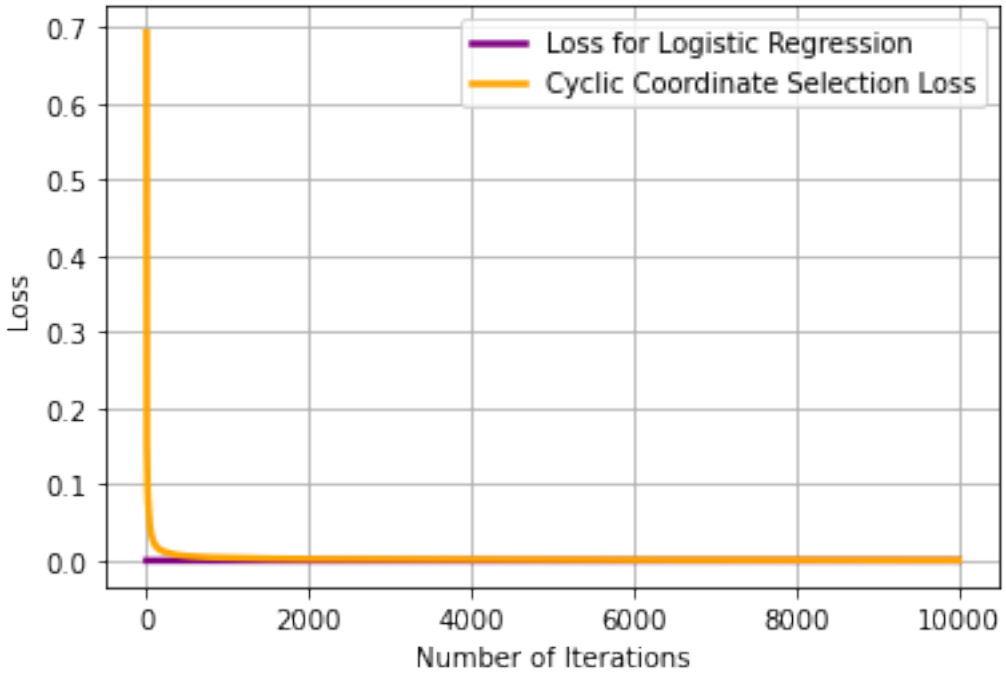
```
    return gradient
```

[9]:
```
def cyclic_coordinate_descent_logistic(X, y, max_epochs=10000, alpha=10):
    """Coordinate descent for logistic regression using cyclic method."""
    n, d = X.shape
    w = np.zeros(d)  # Initialize weight vector
    losses = []  # Track loss at each iteration

    for epoch in range(max_epochs):
        prev_loss = logistic_loss(X, y, w)
        # Select one coordinate for update in this epoch
        coordinate_index = epoch % d
        gradient_i = logistic_gradient(X, y, w)[coordinate_index]
        w[coordinate_index] -= alpha * gradient_i  # Update selected coordinate
        # Compute loss after updating one coordinate
        curr_loss = logistic_loss(X, y, w)
        losses.append(curr_loss)


    return w, losses

# Train logistic regression model using cyclic coordinate descent
w_cyclic, losses_cyclic = cyclic_coordinate_descent_logistic(X_train_scaled,␣
 ↪y_train)

# Compute loss for cyclic method
loss_train_cyclic = logistic_loss(X_train_scaled, y_train, w_cyclic)

print("\nCyclic Coordinate Descent - Train Loss:", loss_train_cyclic)
```

```
Cyclic Coordinate Descent - Train Loss: 0.0005210669127488455
```

[10]:
```
plt.plot([loss]*10000, linewidth=2.5, label='Loss for Logistic Regression',␣
 ↪color='purple')
plt.plot(losses_cyclic, linewidth=2.5, label='Cyclic Coordinate Selection␣
 ↪Loss', color='orange')
plt.xlabel('Number of Iterations')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
```

[10]:
```
<matplotlib.legend.Legend at 0x7ff813dba700>
```



1.4 3. Random feature Coordinate Descent

The CCDR method involves these steps:

- Define the logistic loss function.
- Compute the gradient.
- Perform iterations over epochs.
- Randomly select a coordinate for parameter update.
- Adjust parameters based on the gradient.
- Monitor the logistic loss.
- Train the model using standardized data.
- Evaluate performance by computing logistic loss on training data.

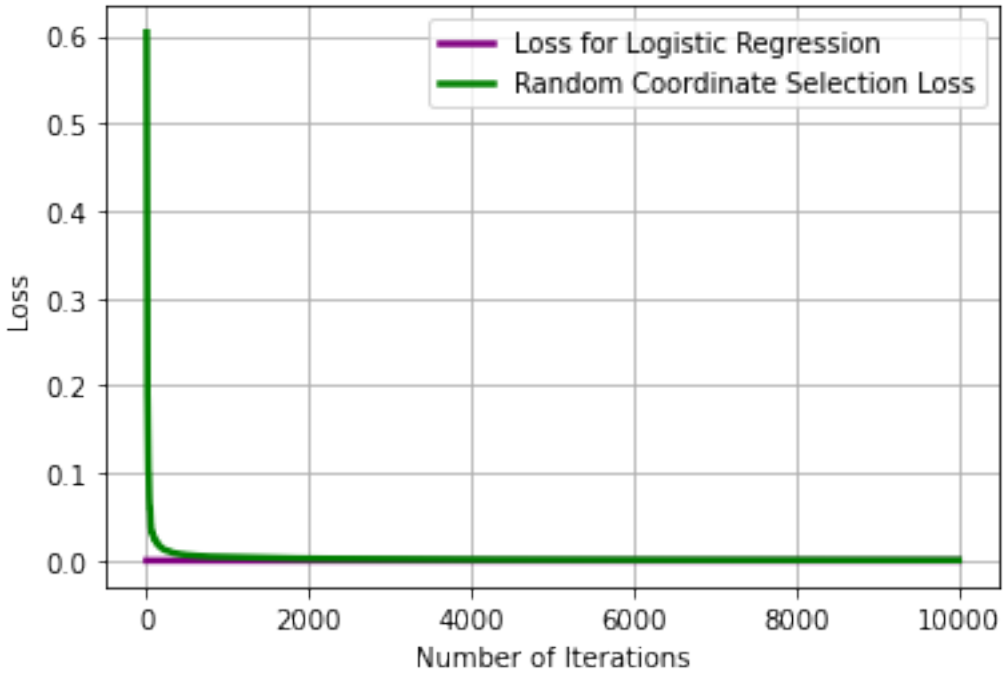
[11]:
```
import random

def cyclic_coordinate_descent_logistic_random(X, y, max_epochs=10000, alpha=10):
    """Coordinate descent for logistic regression using cyclic method."""
    n, d = X.shape
    w = np.zeros(d)  # Initialize weight vector
    losses = []  # Track loss at each iteration

    for epoch in range(max_epochs):
        prev_loss = logistic_loss(X, y, w)
        # Select one coordinate for update in this epoch
        coordinate_index = random.randint(0,X.shape[1]-1)
```

```
        gradient_i = logistic_gradient(X, y, w)[coordinate_index]
        w[coordinate_index] -= alpha * gradient_i  # Update selected coordinate
        # Compute loss after updating one coordinate
        curr_loss = logistic_loss(X, y, w)
        losses.append(curr_loss)


    return w, losses

# Train logistic regression model using cyclic coordinate descent
w_cyclic_random, losses_cyclic_random =␣
 ↪cyclic_coordinate_descent_logistic_random(X_train_scaled, y_train)

# Compute loss for cyclic method
loss_train_cyclic_random = logistic_loss(X_train_scaled, y_train,␣
 ↪w_cyclic_random)

print("\nCyclic Coordinate Descent Random - Train Loss:",␣
 ↪loss_train_cyclic_random)
```

```
Cyclic Coordinate Descent Random - Train Loss: 0.0005318171991449021
```

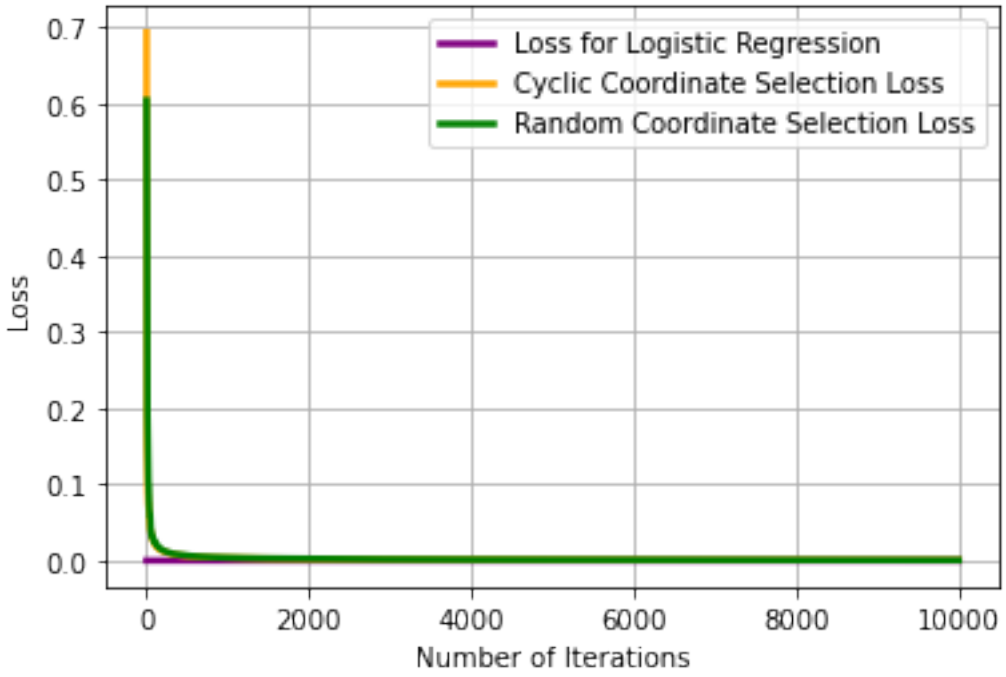
[12]:
```
plt.plot([loss]*10000, linewidth=2.5, label='Loss for Logistic Regression',␣
 ↪color='purple')
plt.plot(losses_cyclic_random, linewidth=2.5, label='Random Coordinate␣
 ↪Selection Loss', color='green')
plt.xlabel('Number of Iterations')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
```

[12]:
```
<matplotlib.legend.Legend at 0x7ff813d31970>
```



[13]:
```
plt.plot([loss]*10000, linewidth=2.5, label='Loss for Logistic Regression',␣
 ↪color='purple')
plt.plot(losses_cyclic, linewidth=2.5, label='Cyclic Coordinate Selection␣
 ↪Loss', color='orange')
plt.plot(losses_cyclic_random, linewidth=2.5, label='Random Coordinate␣
 ↪Selection Loss', color='green')
plt.xlabel('Number of Iterations')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
```

[13]: <matplotlib.legend.Legend at 0x7ff813d1d940>



1.5 Sparse Coordinate Descent

- Sparse coordinate descent initializes w with zeros and randomly selects k coordinates without repetition.
- It iterates over epochs, updating parameters based on selected coordinates.
- At each index, it computes the gradient of the logistic loss function and updates the parameter using step size .
- The logistic loss assesses model performance, tracking progress.
- Post-optimization, the model's performance is evaluated using loss values to gauge sparsity in w.
- It can specify the desired sparsity level with input k.
- Achieving the optimal k-sparse solution varies in convex cost functions.
- Applied to the wine dataset, it initializes the model with sparse coordinate descent, varying k, and evaluates performance using loss values.

[14]:
```
def sparse_cyclic_coordinate_descent_logistic(X, y, k, max_epochs=10000,␣
↪alpha=0.1):
    """Sparse coordinate descent for logistic regression."""
    n, d = X.shape
    w = np.zeros(d)  # Initialize weight vector
    losses = []  # Track loss at each iteration

    coordinate_indices = np.random.choice(d, k, replace=False)

    for epoch in range(max_epochs):
```

```
        prev_loss = logistic_loss(X, y, w)
        for coordinate_index in coordinate_indices:
            gradient_i = logistic_gradient(X, y, w)[coordinate_index]
            w[coordinate_index] -= alpha * gradient_i  # Update selected␣
 ↪coordinate
        # Compute loss after updating k coordinates
        curr_loss = logistic_loss(X, y, w)
        losses.append(curr_loss)

    return w, losses
```

[15]:
```
# Define values of k to try
k_values = [1, 2, 4, 10]

# Dictionary to store loss values for different k
losses_dict = {}

# Try sparse coordinate descent for each value of k
for k in k_values:
    w_sparse, losses_sparse =␣
 ↪sparse_cyclic_coordinate_descent_logistic(X_train_scaled, y_train, k)
    loss_train_sparse = log_loss(y_train, X_train_scaled.dot(w_sparse))
    losses_dict[k] = loss_train_sparse

# Print loss values for different k
print("Loss values for different values of k:")
for k, loss1 in losses_dict.items():
    print(f"k = {k}: Train Loss = {loss1}")
```

```
Loss values for different values of k:
k = 1: Train Loss = 8.712537905800993
k = 2: Train Loss = 4.122658708601228
k = 4: Train Loss = 2.1516759755725117
k = 10: Train Loss = 0.27036416234905775
```