22MAT220

Mathematics For Computing

A THESIS

Logistic regression with L1 regularization

Submitted by

K.NANDINI	**CB.EN.U4AIE22030**
SUBASHREE.M	**CB.EN.U4AIE22048**
VALLETI MAHI VIGNESH	**CB.EN.U4AIE22056**
T.LAKSHMAN	**CB.EN.U4AIE22067**

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGYIN
Artificial Intelligence Engineering

Centre for Computational Engineering and

Networking

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAMCOIMBATORE - 641 112 (INDIA)
DECEEMBER- 2023

AMRITA
VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY UNDER SECTION 3 OF THE UGC ACT, 1956

BONAFIDE CERTIFICATE

This is to certify that the thesis entitled “Logistic regression with L1 regularization” submitted by **K.NANDINI-CB.EN.U4AIE22030, SUBASHREE.M-CB.EN.U4AIE22048, VALLETI MAHI VIGNESH -CB.EN.U4AIE22056, T.LAKSHMAN- CB.EN.U4AIE22067**, for the award of the Degree of Bachelorof Technology in the “CSE(AI) ” is a bonafide record of the work carried out by her under our guidance and supervision at Amrita School of Artificial Intelligence, Coimbatore.

Dr. Neethu Mohan
Project Guide

Dr. K.P. Soman
Professor and Head CEN

Submitted for the university examination held on Tuesday 21 December 2023

DECLARATION

We**,**

K.NANDINI-CB.EN.U4AIE22030, SUBASHREE.M-CB.EN.U4AIE22048, VALLETI MAHI VIGNESH -CB.EN.U4AIE22056, T.LAKSHMAN-CB.EN.U4AIE22067 hereby declare that this thesis entitled "Mathematics for Computing 3 report", is the record of the original work done by us under the guidance of Dr Neethu Mohan, Assistant Professor (SG), Centre for Computational Engineering and Networking, Amrita School of Artificial Intelligence, Coimbatore. To the best of our knowledge, this work has not formed the basis for the award of any degree/diploma/ associateship/fellowship/or a similar award to any candidate in any University.

Place: Coimbatore

Date:20-12-2023

ACKNOWLEDGMENT

This project has been possible due to the sincere and dedicated efforts of many. We thank everyone who has specifically or by implication helped and helped us in growing our psyches and investigating the learning which lies in the profundities of room. Our team would like to express our gratitude towards the Department of Centre for Excellence in Computational Networking (CEN) for giving us this chance to show our abilities and to make ourselves mindful of our potential. This project has given us part of learning about the various hardware and software components of computers and really made us consider something we could just envision. We would also like to express our gratitude towards our Mentor, Dr Neethu Mohan, for giving every one of us vital data about the project. Our folks have given their best by involving us fiscally and giving us moral support. We healthily express gratitude toward them for their hard work and assistance. We likewise thank our establishment Amrita Vishwa Vidyapeetham, for giving us this chance to showcase our abilities and teamwork.

Contents:

Abstract:

Logistic regression is widely used in machine learning for classification problems. It is well-known that regularization is required to avoid over-fitting, especially when there is a only small number of training examples, or when there are a large number of parameters to be learned. In particular, L1 regularized logistic regression is often used for feature selection, and has been shown to have good generalization performance in the presence of many irrelevant features

Applications

1. **Customer Churn Prediction:** Addressing customer retention challenges, the Customer Churn Prediction subtopic employs logistic regression with L1 regularization to predict customer churn. By emphasizing feature selection, we aim to identify the critical factors contributing to customer attrition, providing businesses with actionable insights to implement retention strategies.

2. **Credit Card Fraud Detection:** In the context of Credit Card Fraud Detection, logistic regression with L1 regularization aids in identifying the most influential features related to fraudulent transactions. By penalizing non-contributing features, the model becomes more efficient in distinguishing between genuine and fraudulent activities, thereby improving fraud detection accuracy.

3. **Spam Email Classification:** In the domain of communication, the Spam Email Classification subtopic leverages logistic regression with L1 regularization to improve the accuracy of spam detection. The regularization helps identify significant words or phrases, optimizing the model's ability to differentiate between spam and legitimate emails and thereby enhancing the overall performance of email filtering systems.

4. **Diabetes Prediction:** Moving to the realm of healthcare, the Medical Diagnosis - Diabetes Prediction subtopic utilizes logistic regression with L1 regularization to predict the likelihood of diabetes based on relevant health metrics. The regularization technique not only enhances the model's interpretability but also facilitates the identification of key indicators for diabetes prediction, offering valuable insights for early disease diagnosis.

1. Chapter 1: Customer Churn Prediction

1.1 Introduction

Customer churn prediction is a crucial task for businesses aiming to retain their customer base. Churn refers to the phenomenon where customers discontinue using a service or product. In this context, logistic regression is employed as a predictive modeling tool to assess the likelihood of a customer churning based on various features.

Logistic regression is a statistical method suitable for binary classification problems, making it apt for predicting customer churn (Yes/No). In the presented code, we utilize logistic regression and extend it to incorporate L1 regularization, enhancing the model's ability to generalize and prevent overfitting. L1 regularization helps in feature selection by penalizing and shrinking the less informative features.

1.2 Methodology:

- **Data Exploration:** The code begins with exploring the dataset, checking for missing values, and providing descriptive statistics.

- **Data Visualization:** Visualizations are presented to understand the distribution of churn across different categorical features.

- **Preprocessing:** Unnecessary columns are dropped, and non-numeric columns are encoded. The data is then scaled for uniformity.

- **Model Training:** The logistic regression model is trained on the preprocessed data. The dataset is split into training and testing sets.

- **Model Evaluation:** Classification metrics such as precision, recall, and F1-score are used to evaluate the model's performance.

1.3 Implementation In Python

```
import pandas as pd
import numpy as np
import sklearn
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

df = pd.read_csv('Customer_Churn.csv')

cln_df = df.drop('customerID', axis=1)
for c in cln_df:
    if np.issubdtype(cln_df[c].dtype, np.number):
        continue
    cln_df[c] = LabelEncoder().fit_transform(cln_df[c])

X = cln_df.drop('Churn', axis=1)
y = cln_df['Churn']
X = StandardScaler().fit_transform(X)

x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
model = LogisticRegression(max_iter=1000)
model.fit(x_train, y_train)

predictions = model.predict(x_test)
print(classification_report(y_test, predictions))

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def compute_cost(X, y, theta):
    m = len(y)
    h = sigmoid(X.dot(theta))
    cost = (-1 / m) * (y.T.dot(np.log(h)) + (1 - y).T.dot(np.log(1 - h)))
    return cost

def gradient_descent(X, y, theta, alpha, num_iterations):
    m = len(y)
    cost_history = []

    for _ in range(num_iterations):
        h = sigmoid(X.dot(theta))
        gradient = (1 / m) * (X.T.dot(h - y))
        theta -= alpha * gradient
        cost = compute_cost(X, y, theta)
        cost_history.append(cost)

    return theta, cost_history

X_with_intercept = np.c_[np.ones((X.shape[0], 1)), X]
theta_init = np.zeros(X_with_intercept.shape[1])
alpha = 0.01
num_iterations = 1000

theta, cost_history = gradient_descent(X_with_intercept, y, theta_init, alpha,
num_iterations)

X_test_with_intercept = np.c_[np.ones((x_test.shape[0], 1)), x_test]
predictions = (sigmoid(X_test_with_intercept.dot(theta)) >= 0.5).astype(int)

print(classification_report(y_test, predictions))

def compute_cost(X, y, theta, lambda_reg):
    m = len(y)
    h = sigmoid(X.dot(theta))
    cost = (-1 / m) * (y.T.dot(np.log(h)) + (1 - y).T.dot(np.log(1 - h)))
    reg_term = (lambda_reg / (2 * m)) * np.sum(np.abs(theta[1:]))
    return cost + reg_term

def gradient_descent(X, y, theta, alpha, lambda_reg, num_iterations):
    m = len(y)
    cost_history = []

    for _ in range(num_iterations):
        h = sigmoid(X.dot(theta))
        gradient = (1 / m) * (X.T.dot(h - y))
        reg_term = (lambda_reg / m) * np.sign(theta[1:])
        gradient[1:] += reg_term
        theta -= alpha * gradient
        cost = compute_cost(X, y, theta, lambda_reg)
        cost_history.append(cost)

    return theta, cost_history

X_train_with_intercept = np.c_[np.ones((x_train.shape[0], 1)), x_train]
theta_init = np.zeros(X_train_with_intercept.shape[1])
alpha = 0.04
lambda_reg = 0.3
num_iterations = 1000
```

```python
theta, cost_history = gradient_descent(X_train_with_intercept, y_train,
theta_init, alpha, lambda_reg, num_iterations)

X_test_with_intercept = np.c_[np.ones((x_test.shape[0], 1)), x_test]
predictions = (sigmoid(X_test_with_intercept.dot(theta)) >= 0.5).astype(int)

print(classification_report(y_test, predictions))

def compute_cost(X, y, theta, lambda_reg):
    m = len(y)
    h = sigmoid(X.dot(theta))
    cost = (-1 / m) * (y.T.dot(np.log(h)) + (1 - y).T.dot(np.log(1 - h)))
    reg_term = (lambda_reg / (2 * m)) * np.sum(theta[1:]**2)
    return cost + reg_term

def gradient_descent(X, y, theta, alpha, lambda_reg, num_iterations):
    m = len(y)
    cost_history = []

    for _ in range(num_iterations):
        h = sigmoid(X.dot(theta))
        gradient = (1 / m) * (X.T.dot(h - y))
        reg_term = (lambda_reg / m) * theta[1:]
        gradient[1:] += reg_term
        theta -= alpha * gradient
        cost = compute_cost(X, y, theta, lambda_reg)
        cost_history.append(cost)

    return theta, cost_history

X_train_with_intercept = np.c_[np.ones((x_train.shape[0], 1)), x_train]
theta_init = np.zeros(X_train_with_intercept.shape[1])
alpha = 0.01
lambda_reg = 0.1
num_iterations = 1000

theta, cost_history = gradient_descent(X_train_with_intercept, y_train,
theta_init, alpha, lambda_reg, num_iterations)

X_test_with_intercept = np.c_[np.ones((x_test.shape[0], 1)), x_test]
predictions = (sigmoid(X_test_with_intercept.dot(theta)) >= 0.5).astype(int)

print(classification_report(y_test, predictions))
```

1.4 Code Explanation

The code starts by loading the dataset ('Customer_Churn.csv') and performs basic data exploration, including checking for missing values and displaying descriptive statistics. Visualizations using seaborn and matplotlib are employed to understand the distribution of churn across different categorical features, offering insights into potential patterns in customer behavior.

In the preprocessing step, unnecessary columns are dropped, and non-numeric columns are encoded using label encoding. The data is then scaled to ensure uniformity. Logistic regression is applied to the preprocessed data to create a predictive model. The dataset is split into training and testing sets, and the logistic regression model is trained on the training set. The model's performance is evaluated using the classification report, providing metrics such as precision, recall, and F1-score for both classes (churn and non-churn).

The code further explores logistic regression with L1 regularization. The regularization term is incorporated into the cost function, penalizing less informative features. The gradient descent optimization algorithm is adapted to include the regularization term. The model is trained using the training set, and its performance is evaluated on the testing set. The results are presented in a classification report, allowing a comparison with the standard logistic regression model. The extension of the analysis to include regularization provides insights into whether feature selection improves the model's predictive capabilities for customer churn.

1.5 Conclusion

In conclusion, the logistic regression model employed in predicting customer churn exhibits a satisfactory level of accuracy. However, the integration of L1 regularization does not bring about a substantial improvement in performance for this specific case. To refine the model further, future analyses could explore alternative machine learning algorithms or delve into additional feature engineering techniques. A deeper understanding of the underlying patterns in customer behavior may be crucial for enhancing predictive capabilities.

Comparing logistic regression, logistic regression with L1 regularization, and logistic regression with L2 regularization sheds light on the trade-offs inherent in precision, recall, and overall accuracy. The selection of the regularization method hinges on the unique requirements and characteristics of the dataset. In this scenario, logistic regression with L1 regularization emerges as a potentially favorable choice due to its capacity for feature selection, which proves beneficial in situations with a high number of features.

2. Chapter 2: Credit Card Fraud Prediction

2.1 Introduction

Credit card fraud detection is a critical task in the financial industry to identify and prevent fraudulent transactions. In this context, machine learning, specifically logistic regression, is employed to build a predictive model. Logistic regression is a statistical method that is well-suited for binary classification problems, making it applicable for distinguishing between legitimate and fraudulent credit card transactions. The dataset contains features such as time, transaction amount, and various anonymized numerical features (V1 to V28). The target variable is binary, indicating whether a transaction is fraudulent (Class = 1) or legitimate (Class = 0).

2.2 Methodology

- **Data Exploration:** The code begins with exploring the dataset, checking for missing values, and providing descriptive statistics.
- **Data Visualization:** Visualizations are presented to understand the distribution of churn across different categorical features.
- **Preprocessing:** Unnecessary columns are dropped, and non-numeric columns are encoded. The data is then scaled for uniformity.
- **Model Training:** The logistic regression model is trained on the preprocessed data. The dataset is split into training and testing sets.
- **Model Evaluation:** Model accuracy is evaluated on both the training and testing sets.

2.3 Implementation in Python

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler

data = pd.read_csv('creditcard.csv')

legit = data[data['Class'] == 0]
fraud = data[data['Class'] == 1]

legit_sample = legit.sample(n=492)
new_dataset = pd.concat([legit_sample, fraud], axis=0)

X = new_dataset.drop(columns='Class', axis=1)
Y = new_dataset['Class']

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
stratify=Y, random_state=2)
def linear_cost_function(X, Y, theta):
    m = len(Y)
    h = np.dot(X, theta)
    cost = (1/(2*m)) * np.sum((h - Y)**2)
    return cost

def linear_gradient_descent(X, Y, theta, learning_rate, iterations):
    m = len(Y)
    cost_history = np.zeros(iterations)

    for i in range(iterations):
        h = np.dot(X, theta)
        gradient = (1/m) * np.dot(X.T, (h - Y))
        theta = theta - learning_rate * gradient
        cost_history[i] = linear_cost_function(X, Y, theta)

    return theta, cost_history

def feature_scaling(X):
    mean = np.mean(X, axis=0)
    std_dev = np.std(X, axis=0)
    X_scaled = (X - mean) / std_dev
    return X_scaled

X_train_scaled = feature_scaling(X_train.values)
X_test_scaled = feature_scaling(X_test.values)
```

```
X_train_linear_scaled_bias = np.c_[np.ones((X_train_scaled.shape[0], 1)),
X_train_scaled]
X_test_linear_scaled_bias = np.c_[np.ones((X_test_scaled.shape[0], 1)),
X_test_scaled]

theta_init_linear_scaled = np.zeros(X_train_linear_scaled_bias.shape[1])

learning_rate_linear_scaled = 0.1
iterations_linear_scaled = 1000

theta_trained_linear_scaled, cost_history_linear_scaled =
linear_gradient_descent(X_train_linear_scaled_bias, Y_train.values,
theta_init_linear_scaled, learning_rate_linear_scaled,
iterations_linear_scaled)

train_predictions_linear_scaled = np.dot(X_train_linear_scaled_bias,
theta_trained_linear_scaled)

training_accuracy_linear_scaled =
np.sqrt(np.mean((train_predictions_linear_scaled - Y_train.values) ** 2))
print('Root Mean Squared Error on training data:',
training_accuracy_linear_scaled)

test_predictions_linear_scaled = np.dot(X_test_linear_scaled_bias,
theta_trained_linear_scaled)

test_accuracy_linear_scaled = np.sqrt(np.mean((test_predictions_linear_scaled
- Y_test.values) ** 2))
print('RMSE on Test Data :', test_accuracy_linear_scaled)

model = LogisticRegression()
model.fit(X_train, Y_train)
X_train_prediction = model.predict(X_train)
training_data_accuracy = accuracy_score(X_train_prediction, Y_train)
print('Accuracy on Training data : ', training_data_accuracy)

X_test_prediction = model.predict(X_test)
test_data_accuracy = accuracy_score(X_test_prediction, Y_test)
print('Accuracy score on Test Data : ', test_data_accuracy)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def cost_function(X, Y, theta):
    m = len(Y)
    h = sigmoid(np.dot(X, theta))
    epsilon = 1e-10
    cost = (-1/m) * np.sum(Y * np.log(h + epsilon) + (1 - Y) * np.log(1 - h +
epsilon))
    return cost

def gradient_descent(X, Y, theta, learning_rate, iterations):
    m = len(Y)
    cost_history = np.zeros(iterations)

    for i in range(iterations):
        h = sigmoid(np.dot(X, theta))
        gradient = np.dot(X.T, (h - Y)) / m
        theta = theta - learning_rate * gradient
        cost_history[i] = cost_function(X, Y, theta)

    return theta, cost_history

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

X_train_bias = np.hstack((np.ones((X_train_scaled.shape[0], 1)),
X_train_scaled))
X_test_bias = np.hstack((np.ones((X_test_scaled.shape[0], 1)), X_test_scaled))

theta_init = np.zeros(X_train_bias.shape[1])

learning_rate = 0.01
iterations = 1000

theta_trained, cost_history = gradient_descent(X_train_bias, Y_train.values,
theta_init, learning_rate, iterations)

train_predictions = predict(X_train_bias, theta_trained)

training_accuracy = accuracy_score(train_predictions, Y_train)
print('Accuracy on Training data (logistic regression):', training_accuracy)

test_predictions = predict(X_test_bias, theta_trained)

test_accuracy = accuracy_score(test_predictions, Y_test)
print('Accuracy on Test Data (logistic regression):', test_accuracy)

def cost_function(X, Y, theta, lambda_reg):
    m = len(Y)
    h = sigmoid(np.dot(X, theta))
    epsilon = 1e-10
    regularization_term = (lambda_reg / (2 * m)) * np.sum(theta[1:]**2)
    cost = (-1/m) * np.sum(Y * np.log(h + epsilon) + (1 - Y) * np.log(1 - h +
epsilon)) + regularization_term
    return cost

def gradient_descent(X, Y, theta, learning_rate, iterations, lambda_reg):
    m = len(Y)
    cost_history = np.zeros(iterations)

    for i in range(iterations):
        h = sigmoid(np.dot(X, theta))
        regularization_term = (lambda_reg / m) * theta[1:]
        gradient = (np.dot(X.T, (h - Y)) + np.concatenate(([0],
regularization_term))) / m
        theta = theta - learning_rate * gradient
        cost_history[i] = cost_function(X, Y, theta, lambda_reg)

    return theta, cost_history

learning_rate = 0.05
iterations = 1000
lambda_reg = 0.1

theta_trained_reg, cost_history_reg = gradient_descent(X_train_bias,
Y_train.values, theta_init, learning_rate, iterations, lambda_reg)

train_predictions_reg = predict(X_train_bias, theta_trained_reg)

training_accuracy_reg = accuracy_score(train_predictions_reg, Y_train)
print('Accuracy on Training data (L1 regularization):', training_accuracy_reg)

test_predictions_reg = predict(X_test_bias, theta_trained_reg)

test_accuracy_reg = accuracy_score(test_predictions_reg, Y_test)
print('Accuracy on Test Data (L1 regularization):', test_accuracy_reg)
```

2.4 Code Explanation

Data Preprocessing and Standard Logistic Regression:

The code starts by loading a credit card dataset, balancing the classes, and splitting it into training and testing sets. It then uses the scikit-learn library to create a logistic regression model and trains it on the training data. The accuracy of the model is evaluated on both the training and testing datasets. Standardization is applied to the input features using StandardScaler to improve the convergence of the logistic regression algorithm.

Logistic Regression from Scratch:

The code defines a sigmoid function, cost function, and gradient descent function for logistic regression from scratch. It scales the input features, adds a bias term, initializes parameters, and applies gradient descent to optimize the logistic regression model. The accuracy of the model is then evaluated on both the training and testing datasets. This part of the code demonstrates a manual implementation of logistic regression, providing insights into the underlying mathematics.

Logistic Regression with L1 Regularization:

The code extends the logistic regression from scratch by incorporating L1 regularization. It modifies the cost function and gradient descent to include a regularization term. The regularization helps prevent overfitting by penalizing large coefficients. The model is trained with the regularized logistic regression, and its accuracy is evaluated on both training and testing datasets.

2.5 Conclusion

In conclusion, the logistic regression models, both the standard implementation using scikit-learn and the manual implementation from scratch, demonstrate their effectiveness in detecting credit card fraud. The models provide a reliable means of classification, showcasing the utility of logistic regression in binary classification tasks. The extension to logistic regression with L1 regularization introduces a valuable technique for preventing overfitting and improving the model's generalization. However, the careful selection of the regularization parameter is crucial, as it significantly influences the model's performance.

The comparative analysis of accuracy and performance across different logistic regression approaches emphasizes their suitability for credit card fraud detection. The results highlight the importance of considering regularization techniques to enhance the model's robustness. Future work may involve further exploration of hyperparameters and the investigation of alternative methodologies to continue refining the fraud detection model for practical applications in real-world scenarios.

3. Chapter 3: Spam Mail Prediction

3.1 Introduction

In the realm of email classification, where distinguishing between spam (unwanted) and ham (wanted) emails is crucial, logistic regression serves as a powerful tool for predictive modeling. Logistic regression is a supervised learning algorithm commonly used for binary classification problems. It is particularly suitable for this scenario as it models the probability that a given input belongs to a certain category.

This code demonstrates the application of logistic regression, both standard and with L1 regularization, for classifying emails as spam or ham based on their content. The benefits of L1 regularization, which introduces a penalty term for the absolute values of coefficients, include feature selection and improved model generalization.

3.2 Methodology

- **Data Exploration:** The code begins with exploring the dataset, checking for missing values, and providing descriptive statistics.

- **Data Visualization:** Visualizations are presented to understand the distribution of churn across different categorical features.

- **Preprocessing:** Unnecessary columns are dropped, and non-numeric columns are encoded. The data is then scaled for uniformity.

- **Model Training:** The logistic regression model is trained on the preprocessed data. The dataset is split into training and testing sets.

- **Model Evaluation:** Model accuracy is evaluated on both the training and testing sets.

3.3 Implementation in Python

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Data Collection & Pre-Processing
raw_mail_data = pd.read_csv('mail_data.csv')
mail_data = raw_mail_data.where((pd.notnull(raw_mail_data)), '')

mail_data.loc[mail_data['Category'] == 'spam', 'Category'] = 0
mail_data.loc[mail_data['Category'] == 'ham', 'Category'] = 1

X = mail_data['Message']
Y = mail_data['Category'].astype(int)

# Splitting the Data
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
random_state=3)

# Feature Extraction
feature_extraction = TfidfVectorizer(min_df=1, stop_words='english',
lowercase=True)
X_train_features = feature_extraction.fit_transform(X_train)
X_test_features = feature_extraction.transform(X_test)

# Training the Model - Logistic Regression
model = LogisticRegression()
model.fit(X_train_features, Y_train)

# Evaluating the trained model
prediction_on_training_data = model.predict(X_train_features)
accuracy_on_training_data = accuracy_score(Y_train,
prediction_on_training_data)

prediction_on_test_data = model.predict(X_test_features)
accuracy_on_test_data = accuracy_score(Y_test, prediction_on_test_data)

# Building a Predictive System
input_mail = ["I've been searching for the right words to thank you for this
breather. I promise i wont take your help for granted and will fulfil my
promise. You have been wonderful and a blessing at all times"]

input_data_features = feature_extraction.transform(input_mail)
prediction = model.predict(input_data_features)
```

```
if prediction[0] == 1:
    print('Ham mail')
else:
    print('Spam mail')

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import accuracy_score

# Function to sigmoid activation
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Function to calculate cost and gradient
def compute_cost_and_gradient(X, y, theta, lambda_val=0):
    m = len(y)
    h = sigmoid(np.dot(X, theta))
    regularization_term = (lambda_val / (2 * m)) *
np.sum(np.square(theta[1:]))

    cost = (-1 / m) * (np.dot(y, np.log(h)) + np.dot((1 - y), np.log(1 - h)))
+ regularization_term

    # Gradient with regularization
    gradient = (1 / m) * np.dot(X.T, (h - y))
    gradient[1:] += (lambda_val / m) * theta[1:]

    return cost, gradient

def train_logistic_regression(X, y, learning_rate, num_iterations,
lambda_val=0):
    m, n = X.shape
    X = np.hstack((np.ones((m, 1)), X.toarray()))  # Add a bias term

    theta = np.zeros(n + 1)

    for i in range(num_iterations):
        cost, gradient = compute_cost_and_gradient(X, y, theta, lambda_val)
        theta -= learning_rate * gradient

        if i % 1000 == 0:
            print(f'Iteration {i}, Cost: {cost}')

    return theta

# Function to predict using trained logistic regression model
```

```python
def predict(X, theta):
    X = np.hstack((np.ones((X.shape[0], 1)), X.toarray()))
    probabilities = sigmoid(np.dot(X, theta))
    return (probabilities >= 0.5).astype(int)

# Load data
raw_mail_data = pd.read_csv('mail_data.csv')

# Replace null values with an empty string
mail_data = raw_mail_data.where((pd.notnull(raw_mail_data)), '')

# Label encoding
mail_data.loc[mail_data['Category'] == 'spam', 'Category'] = 0
mail_data.loc[mail_data['Category'] == 'ham', 'Category'] = 1

X = mail_data['Message']
Y = mail_data['Category'].astype(int)

# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
random_state=3)

# Feature extraction
feature_extraction = TfidfVectorizer(min_df=1, stop_words='english',
lowercase=True)

X_train_features = feature_extraction.fit_transform(X_train)
X_test_features = feature_extraction.transform(X_test)

# Train logistic regression model
learning_rate = 0.1
num_iterations = 5000

theta = train_logistic_regression(X_train_features, Y_train, learning_rate,
num_iterations)

# Evaluate the model
prediction_on_training_data = predict(X_train_features, theta)
accuracy_on_training_data = accuracy_score(Y_train,
prediction_on_training_data)
print('Accuracy on training data:', accuracy_on_training_data)

prediction_on_test_data = predict(X_test_features, theta)
accuracy_on_test_data = accuracy_score(Y_test, prediction_on_test_data)
print('Accuracy on test data:', accuracy_on_test_data)

# Logistic Regression with L1 Regularization
learning_rate = 0.2
```

```
num_iterations = 5000

def compute_cost_and_gradient_l1(X, y, theta, lambda_val=0):
    m = len(y)
    h = sigmoid(np.dot(X, theta))
    regularization_term = (lambda_val / (2 * m)) * np.sum(np.abs(theta[1:]))

    cost = (-1 / m) * (np.dot(y, np.log(h)) + np.dot((1 - y), np.log(1 - h)))
+ regularization_term

    gradient = (1 / m) * np.dot(X.T, (h - y))
    gradient[1:] += (lambda_val / m) * np.sign(theta[1:])

    return cost, gradient

def train_logistic_regression_l1(X, y, learning_rate, num_iterations,
lambda_val=0):
    m, n = X.shape
    X = np.hstack((np.ones((m, 1)), X.toarray()))

    theta = np.zeros(n + 1)

    for i in range(num_iterations):
        cost, gradient = compute_cost_and_gradient_l1(X, y, theta, lambda_val)
        theta -= learning_rate * gradient

        if i % 1000 == 0:
            print(f'Iteration {i}, Cost: {cost}')

    return theta

theta_l1 = train_logistic_regression_l1(X_train_features, Y_train,
learning_rate, num_iterations, lambda_val=0.1)

prediction_on_training_data_l1 = predict(X_train_features, theta_l1)
accuracy_on_training_data_l1 = accuracy_score(Y_train,
prediction_on_training_data_l1)

prediction_on_test_data_l1 = predict(X_test_features, theta_l1)
accuracy_on_test_data_l1 = accuracy_score(Y_test, prediction_on_test_data_l1)

print('Accuracy on training data with L1 regularization:',
accuracy_on_training_data_l1)
print('Accuracy on test data with L1 regularization:',
accuracy_on_test_data_l1)
```

3.4 Code Explanation

The code implements logistic regression and logistic regression with L1 regularization for spam classification in emails. The dataset is loaded and pre-processed, with null values handled appropriately. The text data is transformed into numerical features using TF-IDF vectorization, and the dataset is split into training and test sets. Logistic regression is employed to train a model on the training data, and its accuracy is evaluated on both the training and test datasets.

A predictive system is built, allowing users to input new emails for classification. The code then introduces logistic regression with L1 regularization, a technique that includes a penalty term for the absolute values of coefficients. This regularization improves the model's accuracy on the test data compared to standard logistic regression, showcasing the benefits of L1 regularization in enhancing generalization and mitigating overfitting.

The implementation details include functions for sigmoid activation, cost and gradient calculation, and training logistic regression models with and without L1 regularization. The code demonstrates the iterative training process, showing the cost at regular intervals during training. The final accuracy metrics for both logistic regression and logistic regression with L1 regularization highlight the latter's superior performance, emphasizing the importance of regularization techniques in enhancing the robustness of machine learning models for spam classification.

3.5 Conclusion

In conclusion, the implemented logistic regression models showcase their efficacy in accurately classifying emails as spam or ham. The standard logistic regression model demonstrates high accuracy on both the training and test datasets, indicating its ability to generalize well to unseen data. Additionally, the introduction of logistic regression with L1 regularization further improves model performance, particularly on the test data, underscoring the regularization's role in preventing overfitting and enhancing the model's ability to discern patterns in new email data.

While logistic regression proves to be a robust approach for email classification, ongoing monitoring and exploration of advanced methods could further refine the model's performance in the dynamic landscape of email content. The comparative analysis between standard logistic regression and its L1-regularized counterpart highlights the latter's superiority, suggesting that incorporating regularization techniques can be instrumental in developing more resilient and accurate models for spam detection in real-world applications.

4. Chapter 4: Diabetes Prediction

4.1 Introduction

We are implementing logistic regression and logistic regression with L1 regularization to predict the onset of diabetes based on certain features. In the context of diabetes prediction, the dataset includes features such as Pregnancies, Glucose level, BMI, and Age, with the target variable being 'Outcome' indicating whether an individual has diabetes (1) or not (0).
L1 regularization is incorporated to the logistic regression model to prevent overfitting and encourage sparsity in feature selection. L1 regularization introduces a penalty term to the cost function, promoting the reduction of coefficients of less important features to zero.

4.2 Methodology

- **Data Exploration:** The code begins with exploring the dataset, checking for missing values, and providing descriptive statistics.

- **Data Visualization:** Visualizations are presented to understand the distribution of churn across different categorical features.

- **Preprocessing:** Unnecessary columns are dropped, and non-numeric columns are encoded. The data is then scaled for uniformity.

- **Model Training:** The logistic regression model is trained on the preprocessed data. The dataset is split into training and testing sets.

- **Model Evaluation:** Model accuracy is evaluated on both the training and testing sets.

4.3 Implementation in Python

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
import warnings

warnings.filterwarnings("ignore", category=UserWarning)

df = pd.read_csv('diabetes.csv')
x = df[["Pregnancies", "Glucose", "BMI", "Age"]]
y = df["Outcome"]

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def calculate_cost(y, y_pred):
    m = len(y)
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    cost = (-1/m) * np.sum(y * np.log(y_pred) + (1-y) * np.log(1-y_pred))
    return cost

def gradient_descent(X, y, y_pred, alpha):
    m = len(y)
    gradient = np.dot(X.T, (y_pred - y)) / m
    return alpha * gradient

def logistic_regression(X, y, alpha, epochs):
    m, n = X.shape
    X = np.column_stack((np.ones(m), X))
    theta = np.zeros(n + 1)

    for _ in range(epochs):
        z = np.dot(X, theta)
        y_pred = sigmoid(z)
        cost = calculate_cost(y, y_pred)
        gradient = gradient_descent(X, y, y_pred, alpha)
        theta -= gradient

    return theta

def predict(X, theta):
    X = np.column_stack((np.ones(X.shape[0]), X))
    y_pred = sigmoid(np.dot(X, theta))
    return (y_pred >= 0.5).astype(int)

def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)

def l1_regularization(theta, alpha):
    return alpha * np.sign(theta)

def logistic_regression_l1_regularization(X, y, alpha, epochs):
    m, n = X.shape
    X = np.column_stack((np.ones(m), X))
    theta = np.zeros(n + 1)

    for _ in range(epochs):
        z = np.dot(X, theta)
        y_pred = sigmoid(z)
        cost = calculate_cost(y, y_pred)
        gradient = gradient_descent(X, y, y_pred, alpha)
        regularization_term = l1_regularization(theta[1:], alpha)
        theta[0] -= gradient[0]
        theta[1:] -= gradient[1:] + regularization_term

    return theta

x_train_np = x_train.values
x_test_np = x_test.values

# Logistic Regression
alpha = 0.04
epochs = 1000
theta_lr = logistic_regression(x_train_np, y_train, alpha, epochs)
y_pred_lr = predict(x_test_np, theta_lr)
accuracy_lr = accuracy(y_test, y_pred_lr)
print("Logistic Regression Accuracy:", accuracy_lr)

# Logistic Regression with L1 Regularization
alpha_l1 = 0.04
theta_lr_l1 = logistic_regression_l1_regularization(x_train_np, y_train,
alpha_l1, epochs)
y_pred_lr_l1 = predict(x_test_np, theta_lr_l1)
accuracy_lr_l1 = accuracy(y_test, y_pred_lr_l1)
print("Logistic Regression with L1 Regularization Accuracy:", accuracy_lr_l1)

# Scikit-Learn Logistic Regression
model = LogisticRegression()
model.fit(x_train, y_train)
x_test
i1 = model.predict_proba([[4,141,27.6,40]])
```

```
print(i1)
i_pred = model.predict([[4,141,27.6,40]])
print(i_pred)

i2 = model.predict_proba([[13,106,34.2,52]])
print(i2)

accuracy = model.score(x_test, y_test)
print("Scikit-Learn Logistic Regression Accuracy:", accuracy)
```

4.4 Code Explanation

In this code, logistic regression is applied to predict the onset of diabetes based on certain features. The dataset is loaded, and unnecessary columns are removed. The feature set (x) includes variables such as Pregnancies, Glucose level, BMI, and Age, while the target variable (y) is the binary outcome indicating whether an individual has diabetes or not. The dataset is split into training and testing sets using scikit-learn's train_test_split.

The logistic regression model is implemented from scratch, with functions for the sigmoid activation, cost calculation, gradient descent, and the logistic regression algorithm itself. Additionally, a logistic regression model with L1 regularization is implemented to prevent overfitting and promote sparsity in feature selection. The code evaluates the accuracy of both models on the testing set, showcasing that the logistic regression model with L1 regularization performs slightly better in this particular case.

Finally, scikit-learn's logistic regression model is used for comparison purposes. The model is trained and evaluated on the same testing set, demonstrating a comparable accuracy. The code includes predictions for sample inputs using the scikit-learn model and outputs the accuracy of this model. This threefold approach provides a comprehensive understanding of logistic regression, its regularization, and a benchmark comparison with a well-established library.

4.5 Conclusion

In conclusion, the implemented logistic regression model, both with and without L1 regularization, demonstrates its capability to predict the onset of diabetes based on the provided features. While the logistic regression model with L1 regularization slightly outperforms the regular logistic regression in terms of accuracy on the test set, both models exhibit reasonable predictive capabilities. The scikit-learn logistic regression model, employed for comparison, yields a comparable accuracy, emphasizing the practicality and efficiency of using well-established libraries for such tasks. Further optimization and exploration of alternative models may be warranted for enhancing predictive performance, depending on the specific characteristics of the dataset.

References

Dataset links:

1. **Churn Prediction:** https://www.kaggle.com/datasets/blastchar/telco-customer-churn
2. **Credit card fraud prediction:** https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud
3. **Spam mail prediction:** https://drive.google.com/file/d/1uzbhec5TW_OjFr4UUZkoMm0rpyvYdhZw/view
4. **Diabetes prediction:** https://www.dropbox.com/s/uh7o7uyeghqkhoy/diabetes.csv?dl=0