Comprehensive Machine Learning Course Review

3. Supervised Learning and Linear Regression

Supervised learning is a type of machine learning in which we train a model using labeled data, meaning the data includes both input features and the corresponding target output. Linear regression is a foundational algorithm in supervised learning, used for predicting continuous values by modeling the relationship between input features and a numeric target.

3.1 Linear Regression Model

Linear regression assumes a linear relationship between input variables (features) and the output variable (target). The simplest form of linear regression, called simple linear regression, involves only one feature and is represented by:

$$y = wx + b$$

- **y:** Predicted output, also known as the dependent variable.
- **w:** Weight or coefficient for the feature, representing the slope of the line.
- **x:** Input feature, also known as the independent variable.
- **b:** Bias term or intercept, which allows the line to shift up or down.

3.2 Example Problem: Predicting House Prices

Consider a problem where we want to predict house prices based on their square footage. Our data might look like this:

Square Footage	Price ($)
1500	200,000
1600	210,000
1700	230,000
1800	240,000

In this scenario, `Square Footage` is our feature x, and `Price` is the target variable y. Our task is to find the best-fit line that predicts house prices based on the square footage.

3.3 Cost Function (Mean Squared Error)

The cost function is a measure of how well the model's predictions match the actual data. For linear regression, the cost function is typically the Mean Squared Error (MSE), which calculates the average squared difference between predicted and actual values. The formula is:

$$J(w, b) = (1/2m) \Sigma (f_{w,b}(x^{(i)}) - y^{(i)})^2$$

where:

- **m:** Number of training examples
- **$f_{w,b}(x^{(i)})$:** Model's prediction for example i
- **$y^{(i)}$:** Actual output for example i

By minimizing this cost function, we aim to find values for w and b that allow the model to make accurate predictions.

3.4 Calculating the Gradient

The gradient is a vector that points in the direction of the greatest increase of the cost function. To minimize the cost, we move in the opposite direction of the gradient. For each parameter, the partial derivative with respect to w and b is:

$$\partial J(w, b)/\partial w = (1/m) \Sigma (f_{w,b}(x) - y) * x$$
$$\partial J(w, b)/\partial b = (1/m) \Sigma (f_{w,b}(x) - y)$$

3.5 Applications of Linear Regression

- **Real Estate:** Predicting housing prices based on features like square footage, location, and number of bedrooms.

- **Finance:** Estimating stock prices based on historical data and other economic indicators.
- **Healthcare:** Predicting patient health outcomes based on medical history and other features.

---

4. Gradient Descent and Multiple Linear Regression

In this section, we expand upon the concepts of linear regression and gradient descent, introducing models with multiple features (multiple linear regression) and advanced techniques like feature engineering.

4.1 Multiple Linear Regression

In many real-world problems, the target variable depends on more than one feature. For instance, house prices might depend on square footage, location, number of rooms, and age of the property. Multiple linear regression allows us to use multiple features in our model:

$$y = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

Each feature x_i has an associated weight w_i, which represents the feature's influence on the target variable.

4.2 Feature Engineering

Feature engineering is the process of transforming raw data into meaningful features that improve model performance. Examples include:

- **Scaling:** Transforming features to a standard scale, such as normalizing values between 0 and 1.
- **Polynomial Features:** Adding squared or cubic terms to capture non-linear relationships, e.g., x^2 or x^3.
- **Interaction Terms:** Creating new features by multiplying pairs of features, capturing interactions between them.

4.3 Polynomial Regression

Polynomial regression is an extension of linear regression where we add polynomial terms to the model. For example, to capture a quadratic relationship, we could use:

$$y = w_0 + w_1x + w_2x^2$$

Polynomial regression is useful when the relationship between features and the target is non-linear, but we want to use a linear model for simplicity.

4.4 Gradient Descent for Multiple Features

For multiple features, the gradient descent algorithm is similar to that for a single feature but updates each weight based on its partial derivative:

$$w_i := w_i - \alpha * \partial J(w, b)/\partial w_i$$
$$b := b - \alpha * \partial J(w, b)/\partial b$$

This process iteratively updates all weights to minimize the cost function.

---

5. Classification and Logistic Regression

Classification is a type of supervised learning used for categorizing data into discrete classes. Logistic regression is a powerful classification algorithm commonly used for binary classification tasks (i.e., where the output is one of two classes).

5.1 The Logistic Function (Sigmoid Function)

Logistic regression uses the sigmoid function to map predictions to probabilities, allowing us to interpret the output as the likelihood of a data point belonging to a particular class. The sigmoid function, $\sigma(z)$, is defined as:

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

where $z = wx + b$. The sigmoid function outputs values in the range [0, 1], so values close to 1 indicate a high probability of belonging to the positive class, while values close to 0 indicate the opposite.

5.2 Logistic Regression Model

In logistic regression, we model the probability of the positive class (e.g., class 1) as a function of the features:

$$P(y=1|x) = \sigma(w \cdot x + b)$$

where $w \cdot x$ represents the dot product of the weights and features. The decision rule for logistic regression is to classify a data point as the positive class if $P(y=1|x) > 0.5$.

5.3 Logistic Regression Cost Function

The cost function for logistic regression, called the cross-entropy loss or log-loss, is designed to penalize incorrect predictions more heavily. For a single training example, the cost is:

$$J(w, b) = -[y \log(f_{w,b}(x)) + (1 - y) \log(1 - f_{w,b}(x))]$$

where:

- y is the actual class label (0 or 1)
- $f_{w,b}(x)$ is the predicted probability of the positive class

Minimizing this cost function ensures that the model's predicted probabilities align closely with the actual labels.

5.4 Applications of Logistic Regression

Logistic regression is widely used in various fields for binary classification tasks:

- **Medical Diagnosis:** Classifying patients as having a disease or not based on medical records and test results.
- **Spam Detection:** Predicting whether an email is spam or not based on word frequencies and metadata.
- **Customer Churn Prediction:** Determining if a customer will churn based on their usage and interaction history.

5.5 Example Calculation of Sigmoid Output

Let's say our model's prediction for a specific data point is $z = wx + b = 1.5$. Using the sigmoid function:

$$\sigma(1.5) = 1 / (1 + e^{-1.5}) \approx 0.82$$

This indicates an 82% probability that the data point belongs to the positive class.

---

6. Multi-Class Classification

While logistic regression works well for binary classification, many real-world tasks involve more than two classes. Multi-class classification extends binary classification to problems where the target variable has more than two categories, such as classifying images of different animals (e.g., cats, dogs, birds).

6.1 One-vs-All (OvA) Strategy

One-vs-All, also known as One-vs-Rest, is a strategy for handling multi-class classification by decomposing the problem into multiple binary classification tasks. Here's how it works:

- For each class, train a binary classifier that distinguishes that class from all other classes.
- If there are K classes, K binary classifiers are trained, each predicting whether a data point belongs to its respective class.
- During prediction, the model selects the class with the highest confidence score across all binary classifiers.

For example, in a three-class problem with labels A, B, and C:

- Classifier 1: Predicts whether a sample is class A or not.
- Classifier 2: Predicts whether a sample is class B or not.
- Classifier 3: Predicts whether a sample is class C or not.

6.2 Softmax Regression

Softmax regression, also called multinomial logistic regression, is a direct extension of logistic regression for multi-class classification. It uses the softmax function to assign probabilities to each class, ensuring they sum to 1.

$$\sigma(z_i) = e^{z_i} / \Sigma e^{z_j}$$

where:

- z_i: The score for class i.
- Σe^{z_j}: Sum of exponentiated scores across all classes.

This formula ensures that the output probabilities are between 0 and 1 and add up to 1. The model predicts the class with the highest probability.

6.3 Applications of Multi-Class Classification

- **Image Classification:** Identifying objects in images, such as distinguishing among types of animals or types of vehicles.
- **Text Categorization:** Classifying documents into categories like news, sports, entertainment, or politics.
- **Medical Diagnosis:** Classifying patient conditions into multiple possible diagnoses based on medical data.

6.4 Example Calculation with Softmax

Suppose a model computes the following scores for three classes (A, B, and C):

- $z_A = 2.0$
- $z_B = 1.0$
- $z_C = 0.1$

To calculate the probability of each class, we first compute the exponentials of each score and normalize them:

$$P(A) = e^{2.0} / (e^{2.0} + e^{1.0} + e^{0.1}) \approx 0.58$$
$$P(B) = e^{1.0} / (e^{2.0} + e^{1.0} + e^{0.1}) \approx 0.26$$
$$P(C) = e^{0.1} / (e^{2.0} + e^{1.0} + e^{0.1}) \approx 0.16$$

The model would classify the sample as class A, as it has the highest probability.

---

7. K-Nearest Neighbor (KNN) Classifier

K-Nearest Neighbor (KNN) is a simple, non-parametric algorithm that can be used for both classification and regression. It makes predictions based on the K nearest data points in the feature space, making it easy to understand and implement.

7.1 How KNN Works

To make a prediction, KNN identifies the K closest points to a given input. In classification, it assigns the class that is most common among these neighbors. In regression, it predicts the average of the target values of the nearest neighbors.

1. Choose the number of neighbors K.
2. Calculate the distance between the new point and all points in the training set (e.g., Euclidean distance).
3. Select the K closest points.
4. For classification, assign the most frequent class among these points as the predicted class.

7.2 Hyperparameter K

The choice of K can significantly affect the model's performance:

- **Small K:** High variance and possible overfitting, as the model becomes sensitive to noise in the training data.
- **Large K:** High bias and possible underfitting, as the model may ignore relevant local patterns.

To determine the optimal value of K, we often test different values using a validation set.

7.3 Distance Metrics

Different distance metrics can be used in KNN, depending on the nature of the data:

- **Euclidean Distance:** Most commonly used for continuous features, calculated as:

$$d = \sqrt{\Sigma (x_i - y_i)^2}$$

- **Manhattan Distance:** Suitable for grid-based data, calculated as:

$$d = \Sigma |x_i - y_i|$$

- **Cosine Similarity:** Measures the angle between vectors, often used for text data.

7.4 Example Calculation with K=3

Suppose we have a dataset with three classes: A, B, and C. For a new data point, the three nearest neighbors are from classes A, A, and B. Since class A is the most common among the neighbors, KNN would classify the new data point as class A.

7.5 Applications of KNN

- **Image Recognition:** Classifying images based on visual similarity to labeled images.
- **Recommender Systems:** Recommending items based on user similarity, where neighbors represent similar users.
- **Medical Diagnosis:** Predicting diseases by comparing patient data with records of previous patients.

---

8. Bayesian Classifier

Bayesian classifiers are probabilistic classifiers based on Bayes' theorem, which relates the conditional probability of an event given prior knowledge. Naive Bayes is a popular type of Bayesian classifier that assumes independence between features, making it efficient and easy to implement.

8.1 Bayes' Theorem

Bayes' theorem provides a way to update the probability of a hypothesis based on new evidence. Given two events A and B, Bayes' theorem states:

$$P(A|B) = (P(B|A) * P(A)) / P(B)$$

where:

- **P(A):** Prior probability of event A.
- **P(B|A):** Likelihood of event B given that A is true.
- **P(B):** Marginal probability of event B.
- **P(A|B):** Posterior probability of event A given evidence B.

8.2 Naive Bayes Classifier

The Naive Bayes classifier applies Bayes' theorem with the "naive" assumption that features are conditionally independent, given the class. Despite this strong assumption, Naive Bayes performs well on many tasks, especially with text data.

The probability of a data point belonging to a class C is given by:

$$P(C|X) \propto P(C) * \Pi\ P(X_i|C)$$

where $P(C)$ is the prior probability of the class, and $P(X_i|C)$ is the likelihood of feature X_i given class C.

8.3 Types of Naive Bayes

There are different types of Naive Bayes classifiers, each suited to different types of data:

- **Gaussian Naive Bayes:** Assumes continuous features follow a Gaussian (normal) distribution. Often used for numerical data.
- **Multinomial Naive Bayes:** Suitable for discrete data, particularly word counts in text classification.
- **Bernoulli Naive Bayes:** Assumes binary features, commonly used for text classification with binary term frequencies (e.g., word presence/absence).

8.4 Example Calculation with Naive Bayes

Suppose we want to classify a document as either sports-related or politics-related. Given the words in the document, Naive Bayes calculates the probability of each class based on the presence or absence of certain keywords, such as "goal" or "election." It then assigns the class with

the highest probability.

8.5 Applications of Naive Bayes

- **Spam Filtering:** Classifying emails as spam or not spam based on word frequencies.
- **Sentiment Analysis:** Determining whether a text expresses positive, negative, or neutral sentiment.
- **Medical Diagnosis:** Estimating the likelihood of diseases based on symptoms.

---

9. Neural Networks

Neural networks are inspired by the structure of the human brain, where neurons process information and send signals to one another. In machine learning, neural networks consist of layers of artificial neurons that transform input data through learned weights and biases, enabling the model to capture complex patterns and relationships.

9.1 Perceptron: The Basic Unit of Neural Networks

The perceptron is the fundamental building block of neural networks. It performs a linear transformation on input data and applies an activation function to produce an output:

```
output = activation(w • x + b)
```

where:

- **w:** Weight vector, representing the importance of each feature.
- **x:** Input feature vector.
- **b:** Bias term, allowing the perceptron to shift the decision boundary.

If the output is greater than a threshold, the perceptron produces a positive class; otherwise, it produces a negative class.

9.2 Multi-Layer Perceptron (MLP)

Multi-Layer Perceptrons (MLPs) are neural networks that consist of multiple layers of perceptrons, or neurons, organized into three types of layers:

- **Input Layer:** Receives the raw data.
- **Hidden Layers:** Layers between the input and output layers that enable the network to learn complex patterns.
- **Output Layer:** Produces the final prediction. For classification, it often includes a softmax or sigmoid activation to generate probabilities.

MLPs are powerful because they can approximate non-linear functions by stacking multiple layers and introducing non-linear activation functions.

9.3 Feedforward Propagation

Feedforward propagation is the process of moving data forward through the network, from the input layer to the output layer. At each layer, the following steps occur:

1. Calculate the weighted sum of inputs.
2. Add a bias term.
3. Apply an activation function to introduce non-linearity.

This process transforms the data as it passes through each layer, enabling the network to capture complex patterns and relationships within the data.

9.4 Activation Functions

Activation functions are essential in neural networks because they introduce non-linearity, allowing the network to model complex, non-linear patterns. Common activation functions include:

9.4.1 Sigmoid Activation

The sigmoid function outputs values between 0 and 1, making it suitable for binary classification. It is defined as:

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

The sigmoid function is useful in the output layer for binary classification problems. However, it can lead to vanishing gradients during backpropagation, making training deep networks challenging.

9.4.2 ReLU (Rectified Linear Unit)

ReLU is the most commonly used activation function in hidden layers due to its computational efficiency and ability to alleviate the vanishing gradient problem. It is defined as:

$$ReLU(z) = \max(0, z)$$

ReLU outputs 0 for negative values and the input itself for positive values, allowing the network to learn faster and achieve better performance in deeper architectures.

9.4.3 Tanh (Hyperbolic Tangent)

The tanh function outputs values between -1 and 1, making it zero-centered and often preferable to sigmoid in hidden layers:

$$\tanh(z) = (e^{z} - e^{-z}) / (e^{z} + e^{-z})$$

While tanh addresses some issues of the sigmoid function, it can still suffer from vanishing gradients in deep networks.

9.4.4 Softmax Activation

The softmax function is used in the output layer for multi-class classification problems. It converts raw scores into probabilities, ensuring that all outputs sum to 1:

$$\text{softmax}(z_i) = e^{z_i} / \Sigma\ e^{z_j}$$

9.5 Types of Neural Networks

- **Feedforward Neural Networks (FNN):** Data flows in one direction, from the input layer to the output layer. Often used for simple classification tasks.
- **Convolutional Neural Networks (CNN):** Primarily used for image data, CNNs apply filters to capture spatial relationships and patterns in the data.
- **Recurrent Neural Networks (RNN):** Designed for sequential data, such as time series or natural language, RNNs have connections that cycle, allowing them to retain memory of previous inputs.

9.6 Applications of Neural Networks

- **Image Recognition:** Classifying objects in images, detecting faces, and recognizing handwriting.
- **Natural Language Processing:** Translating languages, analyzing sentiment, and generating text.
- **Medical Diagnosis:** Identifying diseases based on medical images or patient data.

---

10. Neural Networks Training

Training neural networks involves updating the network's weights and biases to minimize the prediction error on the training data. This process is achieved through backpropagation and gradient descent.

10.1 Loss Functions

The loss function quantifies the error between the predicted output and the actual output, guiding the network during training. Common loss functions include:

- **Mean Squared Error (MSE):** Used for regression tasks, it calculates the average squared difference between predictions and actual values.
- **Cross-Entropy Loss:** Commonly used for classification, it measures the distance between predicted probabilities and true labels.

10.2 Backpropagation

Backpropagation is an algorithm that computes the gradient of the loss function with respect to each weight in the network, allowing the model to learn by adjusting weights in the direction that reduces the loss. The process includes:

1. **Forward Pass:** Calculate the output by passing data through the network.
2. **Loss Calculation:** Compare the prediction with the actual output to compute the loss.
3. **Backward Pass:** Calculate gradients of the loss with respect to each weight using the chain rule.
4. **Weight Update:** Adjust weights based on the calculated gradients and learning rate.

10.3 Gradient Descent Variants

Several variants of gradient descent are used in training neural networks to improve convergence speed and stability:

10.3.1 Batch Gradient Descent

In batch gradient descent, the gradient is calculated over the entire dataset, and weights are updated once per epoch. While stable, it can be computationally expensive for large datasets.

10.3.2 Stochastic Gradient Descent (SGD)

SGD updates weights for each training example, leading to faster updates but more variance in the optimization path. This variance can help the model escape local minima but may cause instability in training.

10.3.3 Mini-Batch Gradient Descent

A compromise between batch and stochastic gradient descent, mini-batch gradient descent divides the data into small batches, updating weights after each batch. It is widely used for training neural networks as it combines the stability of batch and the speed of SGD.

10.4 Learning Rate and its Importance

The learning rate controls the step size of each update. If it's too high, the model may oscillate and fail to converge. If it's too low, the model may converge too slowly. Often, techniques like learning rate decay or adaptive learning rates are used to improve training.

10.5 Regularization Techniques

Regularization helps prevent overfitting by discouraging the model from relying too heavily on any one feature or subset of features:

- **L2 Regularization (Ridge):** Adds a penalty proportional to the square of the weights, encouraging smaller weights.
- **L1 Regularization (Lasso):** Adds a penalty proportional to the absolute values of the weights, encouraging sparsity (some weights become zero).
- **Dropout:** Randomly drops neurons during training, preventing the network from becoming overly reliant on specific neurons.

10.6 Optimization Algorithms

Advanced optimization algorithms improve upon standard gradient descent, especially for training deep networks:

- **Momentum:** Accelerates gradient descent by adding a fraction of the previous update, helping to overcome local minima.
- **AdaGrad:** Adjusts learning rates for each parameter based on historical gradients, allowing more movement in less frequently updated parameters.
- **Adam (Adaptive Moment Estimation):** Combines momentum and AdaGrad, making it one of the most popular optimizers for deep learning due to its adaptive learning rate and efficient updates.

10.7 Example Training Process

Consider a neural network trained to classify handwritten digits from the MNIST dataset. The training process involves:

1. Initial weight and bias assignment.
2. Using forward propagation to obtain predictions.
3. Calculating cross-entropy loss between predicted and actual labels.

4. Applying backpropagation to compute gradients for each weight and bias.
5. Updating weights using a variant of gradient descent (e.g., mini-batch gradient descent with Adam optimizer).
6. Repeating steps 2-5 for multiple epochs until the model achieves satisfactory accuracy on validation data.

10.8 Applications of Neural Network Training Techniques

- **Image Classification:** Training CNNs on large datasets like ImageNet to classify images into thousands of categories.
- **Language Translation:** Training RNNs or Transformers on large text corpora for machine translation tasks.
- **Speech Recognition:** Training neural networks on audio data to transcribe spoken language into text.

Homework Summaries

HW1: Linear Regression

In this assignment, you implemented linear regression to predict restaurant profits based on city populations. Key tasks included computing cost, performing gradient descent, and visualizing the regression line.

1. Importing Libraries and Loading Data

The assignment begins by importing essential libraries such as `numpy` for numerical calculations and `matplotlib` for data visualization.

```
import numpy as np
import matplotlib.pyplot as plt
```

Loading Data

To load the data, we use `numpy` to load it into arrays, separating features (X) and target values (y). Here's an example of loading and displaying the data:

```
data = np.loadtxt('data.txt', delimiter=',')
X = data[:, 0]
y = data[:, 1]
m = len(y)  # Number of training examples

print(f"First 5 examples:\nX: {X[:5]}\ny: {y[:5]}")
```

After loading the data, we can visualize it to understand the relationship between the input (e.g., population) and the output (e.g., profit).

Data Visualization

The code below uses `matplotlib` to plot the data points:

```
plt.scatter(X, y, marker='x', color='red')
plt.xlabel("Population (10,000s)")
plt.ylabel("Profit ($10,000s)")
plt.title("Profit vs. Population")
plt.show()
```

This scatter plot helps us understand the linear relationship between population and profit, which linear regression aims to model.

2. Cost Function

The cost function, or Mean Squared Error (MSE), quantifies the error between predicted and actual values. It is defined as:

$$J(w, b) = (1/2m) * \Sigma (f_{w,b}(x^{(i)}) - y^{(i)})^2$$

In Python, we implement this as a function that calculates the cost based on current weights and bias:

```
def compute_cost(X, y, w, b):
    m = len(y)
    cost = (1 / (2 * m)) * np.sum((X * w + b - y) ** 2)
    return cost
```

The `compute_cost` function calculates the average squared error of predictions, giving a single value that represents model performance.

3. Gradient Descent Function

Gradient descent is an optimization algorithm that updates the weights and bias iteratively to minimize the cost function. For linear regression, the update rules are:

$$w := w - \alpha * (1/m) * \Sigma (f_{w,b}(x) - y) * x$$
$$b := b - \alpha * (1/m) * \Sigma (f_{w,b}(x) - y)$$

In Python, the gradient descent function calculates gradients and updates `w` and `b` iteratively:

```
def gradient_descent(X, y, w, b, alpha, num_iters):
    m = len(y)
    for i in range(num_iters):
        # Compute predictions
        predictions = X * w + b
        # Calculate gradients
        dw = (1 / m) * np.sum((predictions - y) * X)
        db = (1 / m) * np.sum(predictions - y)
        # Update parameters
        w -= alpha * dw
        b -= alpha * db
        # Print cost for every 100 iterations
        if i % 100 == 0:
            print(f"Iteration {i}: Cost {compute_cost(X, y, w, b)}")
    return w, b
```

Here, `alpha` is the learning rate, which controls the step size of each update. `num_iters` is the number of iterations for training. The function prints the cost periodically to track progress.

4. Model Training and Parameter Optimization

To train the model, we initialize the weights and bias and call the `gradient_descent` function with suitable parameters. Example:

```
initial_w = 0
initial_b = 0
alpha = 0.01
num_iters = 1500

# Train the model
w, b = gradient_descent(X, y, initial_w, initial_b, alpha, num_iters)
print(f"Optimized parameters: w = {w}, b = {b}")
```

After training, we get optimized parameters `w` and `b` that minimize the cost function and fit the data well.

5. Prediction and Visualization of the Regression Line

With the optimized weights and bias, we can make predictions for new data points. Here's a function to predict values and visualize the regression line:

```
def predict(X, w, b):
    return X * w + b

# Plot data and regression line
plt.scatter(X, y, marker='x', color='red')
plt.plot(X, predict(X, w, b), color='blue')
plt.xlabel("Population (10,000s)")
plt.ylabel("Profit ($10,000s)")
plt.title("Linear Regression Fit")
plt.show()
```

The function `predict` returns predicted values based on the learned parameters, and the plot shows the regression line over the data points, visualizing the model's fit.

6. Analysis Summary

- **Cost Function:** Computes the error between predictions and actual values, helping to evaluate model performance.
- **Gradient Descent:** Optimizes the weights and bias by iteratively reducing the cost function.
- **Visualization:** Data and regression line visualizations provide insights into the linear relationship and model fit.

This linear regression implementation provides a foundation for understanding supervised learning and optimization in machine learning.

Homework 2: Logistic Regression Analysis

This section covers the key concepts, Python code, and implementation details from the logistic regression homework assignment. The main tasks include implementing the sigmoid function, defining the cost function (cross-entropy loss), applying gradient descent, and incorporating regularization to prevent overfitting.

1. Importing Libraries and Loading Data

The assignment begins by importing essential libraries such as `numpy` for numerical calculations and `matplotlib` for data visualization. Logistic regression requires data preprocessing, so additional libraries like `scikit-learn` can also be helpful for splitting data and scaling features.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Loading Data and Data Preprocessing

We load and split the data into training and testing sets. Standard scaling is often applied to features to improve convergence during gradient descent:

```
data = np.loadtxt('data_logistic.txt', delimiter=',')
X = data[:, :-1]
y = data[:, -1]

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Here, we split the data into training and testing sets to evaluate model performance on unseen data. Standardization helps gradient descent converge more efficiently by giving each feature a similar scale.

2. Sigmoid Function

The sigmoid function is a key component in logistic regression as it converts the linear combination of inputs into a probability value between 0 and 1. It is defined as:

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

In Python, we implement the sigmoid function as follows:

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

The sigmoid function is applied to the linear combination of weights and features, allowing the model to interpret the output as a probability.

3. Cost Function (Cross-Entropy Loss)

The cost function for logistic regression, known as cross-entropy loss or log-loss, quantifies the error between the predicted probabilities and the true labels. It is defined as:

$$J(w, b) = -(1/m) * \Sigma [y \log(f_{w,b}(x)) + (1 - y) \log(1 - f_{w,b}(x))]$$

In Python, the cost function is implemented as follows:

```
def compute_cost(X, y, w, b):
    m = len(y)
    predictions = sigmoid(np.dot(X, w) + b)
    cost = -(1/m) * np.sum(y * np.log(predictions) + (1 - y) * np.log(1 - predictions))
    return cost
```

This function calculates the cost by summing the error over all training examples. Minimizing this cost will improve the model's accuracy on the training data.

4. Gradient Descent for Logistic Regression

Gradient descent updates the weights and bias iteratively to minimize the cross-entropy loss. For logistic regression, the gradients for the weights and bias are calculated as follows:

$$dw = (1/m) * \Sigma (f_{w,b}(x) - y) * x$$
$$db = (1/m) * \Sigma (f_{w,b}(x) - y)$$

In Python, the gradient descent function calculates gradients and updates the parameters:

```
def gradient_descent(X, y, w, b, alpha, num_iters):
    m = len(y)
    for i in range(num_iters):
        # Compute predictions
        predictions = sigmoid(np.dot(X, w) + b)
        # Calculate gradients
        dw = (1 / m) * np.dot(X.T, (predictions - y))
        db = (1 / m) * np.sum(predictions - y)
        # Update parameters
        w -= alpha * dw
        b -= alpha * db
        # Print cost for every 100 iterations
        if i % 100 == 0:
            print(f"Iteration {i}: Cost {compute_cost(X, y, w, b)}")
    return w, b
```

Here, `alpha` is the learning rate, and `num_iters` is the number of iterations. This function optimizes the weights and bias by reducing the cost function.

5. Regularization (L2 Regularization)

Regularization helps prevent overfitting by adding a penalty term to the cost function, discouraging large weights. L2 regularization, also known as Ridge regularization, is commonly used in logistic regression and adds a penalty proportional to the sum of the squared weights:

$$J(w, b) = -(1/m) * \Sigma [y \log(f_{w,b}(x)) + (1 - y) \log(1 - f_{w,b}(x))] + \lambda/(2m) * \Sigma w^2$$

In Python, we modify the cost function to include regularization:

```
def compute_cost_reg(X, y, w, b, lambda_):
    m = len(y)
    predictions = sigmoid(np.dot(X, w) + b)
    cost = -(1/m) * np.sum(y * np.log(predictions) + (1 - y) * np.log(1 - predictions))
    reg_cost = cost + (lambda_ / (2 * m)) * np.sum(w ** 2)
    return reg_cost
```

The `lambda_` parameter controls the regularization strength. Higher values of `lambda_` result in stronger regularization, which can help reduce overfitting but may also decrease model flexibility.

6. Model Training and Prediction

To train the model, we initialize the weights and bias, set the learning rate, and call the gradient descent function with regularization. Here's an example:

```
initial_w = np.zeros(X_train.shape[1])
initial_b = 0
alpha = 0.01
num_iters = 1000
lambda_ = 0.1

# Train the model
w, b = gradient_descent(X_train, y_train, initial_w, initial_b, alpha, num_iters)
print(f"Optimized parameters: w = {w}, b = {b}")
```

After training, we can use the optimized weights and bias to make predictions on new data:

```
def predict(X, w, b):
    probability = sigmoid(np.dot(X, w) + b)
    return [1 if p >= 0.5 else 0 for p in probability]

# Example prediction
y_pred = predict(X_test, w, b)
```

7. Model Evaluation

To evaluate model performance, we calculate accuracy by comparing predicted labels with actual labels:

```
accuracy = np.mean(y_pred == y_test) * 100
print(f"Accuracy: {accuracy:.2f}%")
```

Accuracy provides a simple measure of how well the model performs on the test data.

8. Analysis Summary

- **Sigmoid Function:** Converts linear outputs into probabilities, allowing binary classification.
- **Cost Function:** Cross-entropy loss quantifies prediction error and is minimized to improve model performance.
- **Gradient Descent:** Optimizes weights and bias by iteratively reducing the cost function.
- **Regularization:** Prevents overfitting by adding a penalty for large weights, enhancing model generalization.

This logistic regression implementation provides insight into classification tasks, optimization techniques, and methods for improving model robustness through regularization.

Homework 3: K-Nearest Neighbor and Naive Bayes Analysis

This section covers the code, key concepts, and implementation details for training K-Nearest Neighbor (KNN) and Naive Bayes classifiers on the Iris dataset. The primary tasks include implementing the classifiers, performing hyperparameter tuning, and visualizing decision boundaries.

1. Importing Libraries and Loading Data

The assignment starts by importing libraries such as `numpy`, `matplotlib`, and `scikit-learn` for handling data, creating models, and visualizing results.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
```

Loading and Preprocessing Data

The Iris dataset is used in this assignment, containing features such as sepal and petal lengths and widths for three flower species. The data is split into training and testing sets, and feature scaling is applied to standardize the data:

```
# Load Iris dataset
iris = load_iris()
X = iris.data
```

```
y = iris.target

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Standardizing data helps models like KNN perform better by ensuring that each feature contributes equally to distance calculations.

2. K-Nearest Neighbor (KNN) Classifier

KNN is a non-parametric, instance-based learning algorithm. Given a data point, it finds the `K` closest points in the training set and assigns the most frequent class among these neighbors as the prediction.

2.1 Training the KNN Model

The `KNeighborsClassifier` from `scikit-learn` is used to implement KNN. After creating the model, it's trained on the dataset:

```
# Create and train the KNN model
k = 5  # Initial choice for K
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)
```

2.2 Prediction and Evaluation

Once trained, the model can be used to predict on the test set, and accuracy can be calculated to evaluate performance:

```
# Make predictions on the test set
y_pred = knn.predict(X_test)

# Calculate accuracy
accuracy = np.mean(y_pred == y_test) * 100
print(f"KNN Accuracy with K={k}: {accuracy:.2f}%")
```

2.3 Hyperparameter Tuning (Choosing Optimal K)

The value of `K` is a critical hyperparameter in KNN, as a small `K` can lead to overfitting, while a large `K` may cause underfitting. To find the optimal `K`, we can test various values and evaluate accuracy:

```
k_values = range(1, 21)
train_accuracies = []
test_accuracies = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    train_accuracies.append(knn.score(X_train, y_train))
    test_accuracies.append(knn.score(X_test, y_test))

# Plot accuracy for different K values
plt.plot(k_values, train_accuracies, label='Train Accuracy')
plt.plot(k_values, test_accuracies, label='Test Accuracy')
plt.xlabel('K Value')
plt.ylabel('Accuracy')
plt.legend()
plt.title('KNN Accuracy vs. K Value')
plt.show()
```

This plot helps visualize how the choice of `K` affects the model's accuracy, allowing us to choose an optimal `K` that balances bias and variance.

3. Naive Bayes Classifier

Naive Bayes is a probabilistic classifier based on Bayes' theorem, with the naive assumption that features are independent given the class. We use Gaussian Naive Bayes, which assumes that feature distributions are Gaussian (normal).

3.1 Training the Naive Bayes Model

We use `GaussianNB` from `scikit-learn` to implement Naive Bayes, and train it on the dataset:

```
# Create and train the Naive Bayes model
nb = GaussianNB()
nb.fit(X_train, y_train)
```

3.2 Prediction and Evaluation

Similar to KNN, we use the trained Naive Bayes model to make predictions and calculate accuracy:

```
# Make predictions on the test set
y_pred_nb = nb.predict(X_test)

# Calculate accuracy
accuracy_nb = np.mean(y_pred_nb == y_test) * 100
print(f"Naive Bayes Accuracy: {accuracy_nb:.2f}%")
```

The accuracy provides insight into the model's performance on the test set.

4. Decision Boundary Visualization

Visualizing decision boundaries helps understand how each model classifies regions of the feature space. Here's an example of plotting decision boundaries for KNN and Naive Bayes with only two features for simplicity:

Preparing Data for Visualization

We select the first two features of the dataset to reduce dimensionality and enable visualization in 2D.

```
X_train_2d = X_train[:, :2]
X_test_2d = X_test[:, :2]

# Retrain models on 2D data
knn_2d = KNeighborsClassifier(n_neighbors=k)
knn_2d.fit(X_train_2d, y_train)

nb_2d = GaussianNB()
nb_2d.fit(X_train_2d, y_train)
```

Plotting Decision Boundaries

Using a mesh grid, we plot the decision boundaries for KNN and Naive Bayes to visualize their classifications:

```
# Create mesh grid
x_min, x_max = X_train_2d[:, 0].min() - 1, X_train_2d[:, 0].max() + 1
y_min, y_max = X_train_2d[:, 1].min() - 1, X_train_2d[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

# Plot KNN decision boundaries
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
Z = knn_2d.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, alpha=0.3)
plt.scatter(X_train_2d[:, 0], X_train_2d[:, 1], c=y_train, edgecolor='k', marker='o')
plt.title("KNN Decision Boundary")

# Plot Naive Bayes decision boundaries
plt.subplot(1, 2, 2)
Z = nb_2d.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, alpha=0.3)
```

```
plt.scatter(X_train_2d[:, 0], X_train_2d[:, 1], c=y_train, edgecolor='k', marker='o')
plt.title("Naive Bayes Decision Boundary")
plt.show()
```

This visualization shows how each model divides the feature space and predicts class labels, helping us understand the different decision-making processes of KNN and Naive Bayes.

5. Analysis Summary

- **KNN:** A distance-based classifier, where tuning `K` helps balance bias and variance. KNN is sensitive to feature scaling.
- **Naive Bayes:** A probabilistic classifier with the assumption of feature independence. It performs well on text data and certain structured datasets.
- **Hyperparameter Tuning:** Choosing optimal `K` for KNN improves model performance by balancing overfitting and underfitting.
- **Decision Boundaries:** Visualization reveals how each classifier segments the feature space, highlighting differences in their classification strategies.

This assignment provides insights into instance-based and probabilistic classifiers, the impact of hyperparameter tuning, and techniques for evaluating model performance on complex datasets.

This comprehensive review covers core topics in machine learning, providing explanations, equations, and practical examples for each concept.