Supervised Learning

Linear Regression

Linear Regression is a fundamental supervised learning algorithm used to predict continuous numerical values. The goal is to establish a relationship between one or more independent variables (features) and a dependent variable (target) by fitting a linear equation to the observed data.

Implementation in Python

Below is a simple implementation of Linear Regression using the Ordinary Least Squares (OLS) method without relying on any machine learning libraries.

```
import numpy as np
import pandas as pd

class LinearRegressionScratch:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.lr = learning_rate
        self.n_iter = n_iterations
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        # Initialize weights and bias
```

```
        self.weights = np.zeros(n_features)
        self.bias = 0

        # Gradient Descent
        for _ in range(self.n_iter):
            y_pred = np.dot(X, self.weights) + self.bias
            # Compute gradients
            dw = (1/n_samples) * np.dot(X.T, (y_pred - y))
            db = (1/n_samples) * np.sum(y_pred - y)
            # Update parameters
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias

# Example Usage
# Assuming you have a dataset loaded into pandas DataFrame `df`
# X = df[['feature1', 'feature2']].values
# y = df['target'].values

# model = LinearRegressionScratch(learning_rate=0.01, n_iterations=1000)
# model.fit(X, y)
# predictions = model.predict(X)
```

Explanation:

1. Initialization: We start by initializing the weights (coefficients) and bias to zero.

2. Gradient Descent: The algorithm iteratively updates the weights and bias to minimize the cost function (Mean Squared Error).

Prediction: Compute the predicted values using the current weights and bias.

Gradient Calculation: Calculate the gradients of the cost function with respect to the weights and bias.

Parameter Update: Adjust the weights and bias in the opposite direction of the gradients to minimize the cost.

Practical Applications

Predicting Housing Prices: Estimating the price of a house based on features like size, location, and number of bedrooms.

Sales Forecasting: Predicting future sales based on historical sales data and other influencing factors.

Logistic Regression

Logistic Regression is a supervised learning algorithm used for binary classification tasks. It predicts the probability that a given input belongs to a particular class.

Implementation in Python

Below is a simple implementation of Logistic Regression using gradient descent without relying on any machine learning libraries.

```
import numpy as np

class LogisticRegressionScratch:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.lr = learning_rate
        self.n_iter = n_iterations
        self.weights = None
        self.bias = None

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

    def fit(self, X, y):
        n_samples, n_features = X.shape
        # Initialize weights and bias
        self.weights = np.zeros(n_features)
        self.bias = 0
```

```
        # Gradient Descent
        for _ in range(self.n_iter):
            linear_model = np.dot(X, self.weights) + self.bias
            y_pred = self.sigmoid(linear_model)

            # Compute gradients
            dw = (1/n_samples) * np.dot(X.T, (y_pred - y))
            db = (1/n_samples) * np.sum(y_pred - y)

            # Update parameters
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict_prob(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        return self.sigmoid(linear_model)

    def predict(self, X, threshold=0.5):
        probabilities = self.predict_prob(X)
        return np.where(probabilities >= threshold, 1, 0)

# Example Usage
# Assuming you have a dataset loaded into pandas DataFrame `df`
# X = df[['feature1', 'feature2']].values
# y = df['target'].values
```

```
# model = LogisticRegressionScratch(learning_rate=0.01, n_iterations=1000)
# model.fit(X, y)
# predictions = model.predict(X)
```

Explanation:

1. Sigmoid Function: Converts the linear combination of inputs into a probability between 0 and 1.

2. Initialization: Initialize weights and bias to zero.

3. Gradient Descent: Iteratively update weights and bias to minimize the binary cross-entropy loss.

Prediction: Compute the predicted probabilities using the sigmoid function.

Gradient Calculation: Calculate the gradients of the loss function with respect to the weights and bias.

Parameter Update: Adjust the weights and bias to minimize the loss.

Practical Applications

Spam Detection: Classifying emails as spam or not spam.

Medical Diagnosis: Predicting the presence or absence of a disease based on patient data.

Evaluation Metrics

Evaluating the performance of machine learning models is essential to understand their effectiveness. Depending on the type of problem (regression or classification), different metrics are used.

Regression Metrics

1. Mean Squared Error (MSE)

2. Root Mean Squared Error (RMSE)

3. R-squared ()

Implementation in Python

```
def mean_squared_error(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

def root_mean_squared_error(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))

def r2_score(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    return 1 - (ss_res / ss_tot)

# Example Usage
# mse = mean_squared_error(y, predictions)
# rmse = root_mean_squared_error(y, predictions)
# r2 = r2_score(y, predictions)
```

Explanation:

MSE: Calculates the average of the squared differences between actual and predicted values.

RMSE: The square root of MSE, providing error in the same units as the target variable.

R-squared: Indicates the proportion of variance in the dependent variable that is predictable from the independent variables.

Classification Metrics

1. Precision

2. Recall

3. F1-Score

Implementation in Python

```
def precision_score(y_true, y_pred):
    true_positive = np.sum((y_true == 1) & (y_pred == 1))
    false_positive = np.sum((y_true == 0) & (y_pred == 1))
    return true_positive / (true_positive + false_positive) if (true_positive + false_positive) != 0 else 0

def recall_score(y_true, y_pred):
    true_positive = np.sum((y_true == 1) & (y_pred == 1))
    false_negative = np.sum((y_true == 1) & (y_pred == 0))
    return true_positive / (true_positive + false_negative) if (true_positive + false_negative) != 0 else 0

def f1_score(y_true, y_pred):
```

```
    prec = precision_score(y_true, y_pred)
    rec = recall_score(y_true, y_pred)
    return 2 * (prec * rec) / (prec + rec) if (prec + rec) != 0 else 0

# Example Usage
# precision = precision_score(y_true, y_pred)
# recall = recall_score(y_true, y_pred)
# f1 = f1_score(y_true, y_pred)
```

Explanation:

Precision: Measures the accuracy of positive predictions.

Recall: Measures the ability of the model to find all the relevant cases.

F1-Score: The harmonic mean of precision and recall, providing a balance between the two.

Unsupervised Learning

K-Means Clustering

K-Means Clustering is an unsupervised learning algorithm used to partition a dataset into distinct, non-overlapping clusters. The algorithm aims to minimize the variance within each cluster.

Implementation in Python

Below is a simple implementation of the K-Means algorithm from scratch.

```
import numpy as np

class KMeansScratch:
    def __init__(self, n_clusters=3, max_iters=100, tolerance=1e-4):
        self.k = n_clusters
        self.max_iters = max_iters
        self.tol = tolerance
        self.centroids = None

    def fit(self, X):
        n_samples, n_features = X.shape
        # Initialize centroids randomly from the data points
        random_indices = np.random.choice(n_samples, self.k, replace=False)
        self.centroids = X[random_indices]

        for _ in range(self.max_iters):
            # Assign clusters
            distances = self._compute_distances(X)
            labels = np.argmin(distances, axis=1)

            # Compute new centroids
```

```
        new_centroids = np.array([X[labels == i].mean(axis=0) if len(X[labels == i]) > 0 else self.centroids[i] for i in range(self.k)])

        # Check for convergence
        if np.all(np.abs(new_centroids - self.centroids) < self.tol):
            break
        self.centroids = new_centroids

    def _compute_distances(self, X):
        distances = np.zeros((X.shape[0], self.k))
        for i in range(self.k):
            distances[:, i] = np.linalg.norm(X - self.centroids[i], axis=1)
        return distances

    def predict(self, X):
        distances = self._compute_distances(X)
        return np.argmin(distances, axis=1)

# Example Usage
# Assuming you have a dataset loaded into numpy array `X`
# model = KMeansScratch(n_clusters=3, max_iters=100)
# model.fit(X)
# labels = model.predict(X)
```

Explanation:

1. Initialization: Randomly select data points as initial centroids.

2. Assignment Step: Assign each data point to the nearest centroid based on Euclidean distance.

3. Update Step: Recompute the centroids as the mean of all data points assigned to each cluster.

4. Convergence: Repeat the assignment and update steps until the centroids stabilize within a specified tolerance or until the maximum number of iterations is reached.

Applications of K-Means Clustering

Customer Segmentation: Grouping customers based on purchasing behavior and demographics to tailor marketing strategies.

Image Compression: Reducing the number of colors in an image by clustering similar colors and representing them with cluster centroids.

Anomaly Detection: Identifying outliers by finding data points that do not belong to any cluster or belong to small clusters.

Document Clustering: Organizing a large set of documents into topics based on content similarity.

Practical Machine Learning Workflow

A robust machine learning workflow ensures that models are built efficiently and generalize well to new, unseen data. Two critical components of this workflow are Cross-Validation and Hyperparameter Tuning.

Cross-Validation

Cross-Validation is a technique for assessing how a machine learning model will generalize to an independent dataset. It involves partitioning the original dataset into training and testing subsets multiple times to ensure the model's reliability.

Implementation in Python

Below is an implementation of K-Fold Cross-Validation from scratch.

```
import numpy as np

def k_fold_cross_validation(X, y, k=5):
    n_samples = X.shape[0]
    indices = np.arange(n_samples)
    np.random.shuffle(indices)
    fold_sizes = np.full(k, n_samples // k, dtype=int)
```

```
    fold_sizes[:n_samples % k] += 1
    current = 0
    folds = []
    for fold_size in fold_sizes:
        start, stop = current, current + fold_size
        folds.append(indices[start:stop])
        current = stop
    return folds

# Example Usage
# X = np.array([...])
# y = np.array([...])
# folds = k_fold_cross_validation(X, y, k=5)
# for i in range(5):
#     test_idx = folds[i]
#     train_idx = np.hstack([folds[j] for j in range(5) if j != i])
#     X_train, X_test = X[train_idx], X[test_idx]
#     y_train, y_test = y[train_idx], y[test_idx]
#     # Train your model here
```

Explanation:

1. Shuffle Data: Randomly shuffle the dataset to ensure that each fold is representative of the whole.

2. Divide into Folds: Split the data into equally sized folds.

3. Training and Testing: For each fold, use it as the test set and the remaining folds as the training set.

4. Performance Aggregation: Train the model on the training set and evaluate it on the test set. Aggregate the performance metrics across all folds to obtain an overall performance estimate.

Hyperparameter Tuning

Hyperparameter Tuning involves finding the optimal set of hyperparameters for a machine learning model to enhance its performance. Common techniques include Grid Search and Random Search.

Grid Search Implementation in Python

Grid Search exhaustively searches through a predefined set of hyperparameter combinations.

```
import itertools

def grid_search(X, y, model_class, param_grid, k=5):
    folds = k_fold_cross_validation(X, y, k)
```

```
    param_combinations = list(itertools.product(*param_grid.values()))
    best_score = -np.inf
    best_params = None

    for params in param_combinations:
        param_dict = dict(zip(param_grid.keys(), params))
        scores = []
        for i in range(k):
            test_idx = folds[i]
            train_idx = np.hstack([folds[j] for j in range(k) if j != i])
            X_train, X_test = X[train_idx], X[test_idx]
            y_train, y_test = y[train_idx], y[test_idx]

            model = model_class(**param_dict)
            model.fit(X_train, y_train)
            predictions = model.predict(X_test)
            score = evaluate_metric(y_test, predictions)  # Define your evaluation metric
            scores.append(score)
        avg_score = np.mean(scores)
        if avg_score > best_score:
            best_score = avg_score
            best_params = param_dict
    return best_params, best_score

# Example Usage
# Define a parameter grid
```

```
# param_grid = {
#     'learning_rate': [0.01, 0.1],
#     'n_iterations': [1000, 5000]
# }
# best_params, best_score = grid_search(X, y, LinearRegressionScratch, param_grid, k=5)
```

Explanation:

1. Parameter Combinations: Generate all possible combinations of hyperparameters from the parameter grid.

2. Cross-Validation: For each combination, perform K-Fold Cross-Validation and compute the average performance metric.

3. Selection: Select the hyperparameter combination with the best average performance.

Random Search Implementation in Python

Random Search samples a fixed number of hyperparameter combinations from a specified distribution.

```
import random
```

```
def random_search(X, y, model_class, param_distributions, n_iter=10, k=5):
    folds = k_fold_cross_validation(X, y, k)
    best_score = -np.inf
    best_params = None

    param_keys = list(param_distributions.keys())
    for _ in range(n_iter):
        param_dict = {key: random.choice(param_distributions[key]) for key in param_keys}
        scores = []
        for i in range(k):
            test_idx = folds[i]
            train_idx = np.hstack([folds[j] for j in range(k) if j != i])
            X_train, X_test = X[train_idx], X[test_idx]
            y_train, y_test = y[train_idx], y[test_idx]

            model = model_class(**param_dict)
            model.fit(X_train, y_train)
            predictions = model.predict(X_test)
            score = evaluate_metric(y_test, predictions)  # Define your evaluation metric
            scores.append(score)
        avg_score = np.mean(scores)
        if avg_score > best_score:
            best_score = avg_score
            best_params = param_dict
    return best_params, best_score
```

```
# Example Usage
# Define parameter distributions
# param_distributions = {
#     'learning_rate': [0.001, 0.01, 0.1],
#     'n_iterations': [1000, 5000, 10000]
# }
# best_params, best_score = random_search(X, y, LinearRegressionScratch, param_distributions, n_iter=20, k=5)
```

Explanation:

1. Random Sampling: Randomly select hyperparameter combinations from the specified distributions.

2. Cross-Validation: For each sampled combination, perform K-Fold Cross-Validation and compute the average performance metric.

3. Selection: Select the hyperparameter combination with the best average performance.

Choosing Between Grid Search and Random Search:

Grid Search is exhaustive and guarantees finding the optimal combination within the specified grid but can be computationally expensive.

Random Search is more efficient, especially with large hyperparameter spaces, and often finds a sufficiently good solution without exploring the entire grid.

Applying Machine Learning Techniques

Building Regression and Classification Models

Let's apply the implemented Linear and Logistic Regression models on structured datasets.

Example: Linear Regression on a Synthetic Dataset

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

# Generate synthetic data for Linear Regression
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X.flatten() + np.random.randn(100)

# Initialize and train the model
model = LinearRegressionScratch(learning_rate=0.1, n_iterations=1000)
model.fit(X, y)
predictions = model.predict(X)
```

```
# Plot the results
plt.scatter(X, y, color='blue', label='Actual')
plt.plot(X, predictions, color='red', label='Predicted')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()

# Calculate evaluation metrics
mse = mean_squared_error(y, predictions)
rmse = root_mean_squared_error(y, predictions)
r2 = r2_score(y, predictions)
print(f"MSE: {mse}, RMSE: {rmse}, R2: {r2}")
```

Explanation:

1. Data Generation: Creates a synthetic dataset with a linear relationship.

2. Model Training: Fits the Linear Regression model to the data.

3. Visualization: Plots the actual data points and the regression line.

4. Evaluation: Computes MSE, RMSE, and to assess model performance.

Example: Logistic Regression on a Synthetic Dataset

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

# Generate synthetic data for Logistic Regression
np.random.seed(42)
X_pos = np.random.randn(50, 2) + np.array([2, 2])
X_neg = np.random.randn(50, 2) + np.array([-2, -2])
X = np.vstack((X_pos, X_neg))
y = np.array([1]*50 + [0]*50)

# Initialize and train the model
model = LogisticRegressionScratch(learning_rate=0.1, n_iterations=1000)
model.fit(X, y)
predictions = model.predict(X)

# Plot the results
plt.scatter(X[y==1][:,0], X[y==1][:,1], color='blue', label='Class 1')
plt.scatter(X[y==0][:,0], X[y==0][:,1], color='red', label='Class 0')
plt.scatter(X[:,0], X[:,1], c=predictions, cmap='bwr', alpha=0.3, label='Predicted')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
```

```
plt.legend()
plt.show()

# Calculate evaluation metrics
precision = precision_score(y, predictions)
recall = recall_score(y, predictions)
f1 = f1_score(y, predictions)
print(f"Precision: {precision}, Recall: {recall}, F1-Score: {f1}")
```

Explanation:

1. Data Generation: Creates a synthetic binary classification dataset.

2. Model Training: Fits the Logistic Regression model to the data.

3. Visualization: Plots the data points with their predicted classes.

4. Evaluation: Computes Precision, Recall, and F1-Score to assess classification performance.

Applying K-Means Clustering

Let's apply the implemented K-Means algorithm on a synthetic dataset.

Example: K-Means Clustering on a Synthetic Dataset

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

# Generate synthetic data for K-Means
from sklearn.datasets import make_blobs

X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Initialize and fit K-Means
kmeans = KMeansScratch(n_clusters=4, max_iters=100)
kmeans.fit(X)
labels = kmeans.predict(X)

# Plot the clusters
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(kmeans.centroids[:, 0], kmeans.centroids[:, 1], s=300, c='red', marker='X')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('K-Means Clustering')
plt.show()
```

Explanation:

1. Data Generation: Creates a synthetic dataset with four distinct clusters.

2. Model Training: Fits the K-Means model to the data.

3. Visualization: Plots the data points colored by their assigned cluster and marks the centroids.