Perceptron Algorithm

A perceptron is a type of artificial neural network that is commonly used in machine learning for binary classification tasks. It was developed in the 1950s and is one of the simplest forms of neural networks.

At its core, a perceptron consists of a single neuron that takes in input data and produces an output. The input data is first transformed by a set of weights, and then passed through an activation function which produces the output. The activation function is typically a step function that produces a binary output (0 or 1) based on the input.

Perceptrons can be used for a variety of binary classification tasks, such as predicting whether an email is spam or not, or whether a tumor is benign or malignant. While perceptrons are limited in their capabilities compared to more complex neural networks, they are still a useful tool in many machine learning applications.

How Perceptron works ?

A perceptron is a type of artificial neural network used for binary classification tasks. It works by taking in a set of input features and assigning a weight to each feature. The perceptron then computes the weighted sum of the inputs and compares it to a threshold value. If the sum is greater than the threshold, the perceptron outputs a positive class label; otherwise, it outputs a negative class label.

During training, the perceptron adjusts the weights assigned to each feature based on the error it makes on the training data. The weights are updated in such a way that the error on the training data is minimized. This process continues until the perceptron has converged to a set of weights that classify the training data accurately.

Once trained, the perceptron can be used to make predictions on new data by taking in the input features and computing the weighted sum as before. The output of the perceptron can then be used to classify the new input as either positive or negative.

Difference in taking normalized and unnormalized data :

Normalizing the data helps to ensure that each feature contributes equally to the learning process, regardless of its original scale. This is particularly important when features have different scales or units, as it can prevent the model from being dominated by a single feature. In the case of the perceptron, normalization can improve the accuracy and convergence of the algorithm.

On the other hand, when using unnormalized data in a perceptron, the algorithm may be biased towards features with larger values. This can result in slower convergence and reduced accuracy, particularly if some features have a much larger range than others.

Fischer Discriminant Analysis

Tasks

Learning Task 1: Build Fisher's linear discriminant model (FLDM1) on the training data and thus reduce 32 dimensional problem to univariate dimensional problem. Find out the decision boundary in the univariate dimension using a generative approach. You may assume gaussian distribution for both positive and negative classes in the univariate dimension.

Learning Task 2: Change the order of features in the dataset randomly. Equivalently speaking,
for an example of feature tuple (f1, f2, f3, f4, . . , f32), consider a random permutation (f3, f1, f4,
f2, f6, ..., f32) and build the Fisher's linear discriminant model (FLDM2) on the same training
data as in the learning task 1. Find out the decision boundary in the univariate dimension using
generative approach and you may assume gaussian distribution for both positive and negative
classes in the univariate dimension. Outline the difference between the models – FLDM1 and
FLDM2 - and their respective performances

Concepts Used in solving the tasks

- Fisher's Linear Discriminant Analysis: a method used in statistics and other fields, to find a linear combination of features that characterizes or separates two or more classes of objects or events. The resulting combination may be used as a linear classifier, or, more commonly, for dimensionality reduction before later classification.
- Gaussian Distribution: Gaussian distribution (also known as normal distribution) is a bell-shaped curve, and it is assumed that during any measurement values will follow a normal distribution with an equal number of measurements above and below the mean value.

Approach

1. The Data is read from a csv file ("**data.csv**")
2. The Data is then processed to map the outputs to 1 (Benign) and 0 (Malignant) and remove all missing values from the dataset
3. A Random 67% of the data is loaded into a training dataset and the remaining 33% into a testing dataset
4. A model "**FLDM1**" is created using sklearn's "**LinearDiscriminantAnalysis**" with number of components as 1 and the training data is made to fit the model
5. A Gaussian Generative Model is then used fit the transformed data from the above model and this is then used to predict the results.
6. The accuracy of this model is stored in a list "**results1**"
7. The steps 4-6 are repeated for a new model "**FLDM2**" using a dataset with the features/columns shuffled. The accuracies of this model are stored in "**results2**"
8. The decision boundaries for both "**FLDM1**" and "**FLDM2**" are calculated and plotted and are stored in a directory named "**plots/**"
9. This is done for multiple splits (> 10) (We used 10 and 50)
10. The average accuracy is then calculated for all the splits and displayed to the user

Loading and Processing of dataset

```python
# read data from csv file
df = pd.read_csv("data.csv")
df['diagnosis'] = pd.Series(np.where(df.diagnosis.values == 'B', 1, 0), df.index)

for column in df.columns:
    if column != 'diagnosis':
        _mean = df[column].mean()
        df[column].fillna(_mean, inplace=True)
```

Gaussian Generative Model

```python
from scipy.stats import multivariate_normal

class GaussianGenerativeModel:
    def __init__(self):
        self.phi = None
        self.mu_0 = None
        self.mu_1 = None
        self.sigma = None

    def fit(self, X, y):
        # Compute phi, mu_0, mu_1, and sigma
        self.phi = sum(y) / len(y)
        self.mu_0 = X[y == 0].mean(axis=0)
        self.mu_1 = X[y == 1].mean(axis=0)
        self.sigma = ((X[y == 0] - self.mu_0).T @ (X[y == 0] - self.mu_0) +
                      (X[y == 1] - self.mu_1).T @ (X[y == 1] - self.mu_1)) / len(y)

    def predict(self, X):
        # Compute the probability of each class for each data point in X
        p_0 = multivariate_normal.pdf(X, mean=self.mu_0, cov=self.sigma)
        p_1 = multivariate_normal.pdf(X, mean=self.mu_1, cov=self.sigma)

        # Assign the class with higher probability to each data point
        return (p_1 > p_0).astype(int)

```

Usage of the Generative Model and FLDA

```
# Fisher's Linear Discriminant Model
    FLDM1 = LinearDiscriminantAnalysis(n_components=1)
    X_train_lda = FLDM1.fit_transform(X_train, y_train)
    X_test_lda = FLDM1.transform(X_test)

    model = GaussianGenerativeModel()
    model.fit(X_train_lda, y_train)
    y_pred = model.predict(X_test_lda)
    accuracy = accuracy_score(y_test, y_pred)
    results1.append(accuracy)
```

Plotting of the Decision Boundary

```
# plot decision boundary
if len(w) > 1:
    x1 = np.linspace(X_train_lda[:, 0].min(), X_train_lda[:, 0].max(), 100)
    x2 = (-w0 - w[0] * x1) / w[1]
    plt.plot(x1, x2, 'k--', label='Decision boundary')
else:
    x = np.linspace(X_train_lda[:, 0].min(), X_train_lda[:, 0].max(), 100)
    plt.axvline(x=w0 / w[0], color='k', linestyle='--', label='Decision boundary')
plt.scatter(X_train_lda[:, 0], y_train, c=y_train, cmap='bwr', alpha=0.5)
plt.xlabel('LD1')
plt.legend()
plt.savefig(f'plots/FLDM1-{i}.png')
plt.clf()
```

Logistic Regression

Logistic regression is a statistical method used for binary classification problems, where the goal is to predict the probability of an outcome based on a set of predictor variables.

The basic idea behind logistic regression is to fit a logistic function to the data, which maps the input features to the probability of the binary outcome. This function takes the form of the sigmoid curve, which has an S-shaped curve that maps any real-valued input to a probability value between 0 and 1.

The logistic regression model is trained by minimizing a cost function, here the log-likelihood function, using gradient descent. The resulting model can then be used to predict the probability of the binary outcome for new data points, and a threshold can be applied to classify them into one of the two classes.

Batch gradient descent:
In this approach, the entire dataset is used to compute the gradients of the model parameters. The model is updated once per epoch, which is a full pass through the entire dataset. Batch gradient descent is generally slower than other approaches since it needs to process the entire dataset at once, but it can lead to a more accurate estimate of the gradient.

Stochastic gradient descent:
In this approach, the model parameters are updated after processing each training example. The gradient is estimated using a single randomly selected training example, and the model is updated after each example. Stochastic gradient descent is generally faster than batch gradient descent since it only processes one example at a time, but it can be more noisy and lead to a less accurate estimate of the gradient.

Mini-batch gradient descent:
This approach is a compromise between batch gradient descent and stochastic gradient descent. In this approach, a small subset of the training data (a "mini-batch") is used to estimate the gradient, and the model is updated after processing each mini-batch. Mini-batch gradient descent is faster than batch gradient descent, since it only processes a small subset of the data at a time, but it is less noisy than stochastic gradient descent, since it processes multiple examples at once.

Difference in LT-1 and LT-2
The main difference between using normalized and unnormalized data is that normalizing the data can help the algorithm converge faster and improve the overall performance of the model. This is because normalization ensures that all input features have a similar scale and magnitude, which can prevent certain features from dominating others during the training process.

On the other hand, using unnormalized data can sometimes lead to slower convergence and poorer model performance, especially if the input features have different scales or magnitudes.

Comparing Outputs of models

Perceptron:

```
Mean accuracy for task1 : 91.07526881720432
Variance for task1 : 5.7925771765522125
Mean accuracy for task2 : 94.0583554376658
Variance for task2 : 19.82987286197751
Mean accuracy for task3 : 91.0215053763441
Variance for task3 : 5.321424442132047
```

Fischer:

```
Results of 50 random training/testing splits:
FLDM1
Average accuracy: 0.9635106382978722
Accuracy variance: 0.00011544816659121758
FLDM2
Average accuracy: 0.9607446808510638
Accuracy variance: 0.00014700090538705298
```

Gradient Descent:

Batch:

```
Mean  96.91489361702128
Variance  2.4784970574920666
```

Mini Batch:

```
Mean  96.22340425531915
Variance  1.8928248076052465
```

Stochastic:

```
Mean  94.46808510638297
Variance  0.9732910819375238
```