Perceptron

September 17, 2024

1 Perceptron

1.0.1 Context of the Dataset

- The model is trained on the **Iris dataset**, which includes samples from three types of iris flowers: *Iris-setosa*, *Iris-versicolor*, and *Iris-virginica*.
- This implementation converts the problem into a **binary classification task**:
  - **Class 0:** *Iris-setosa*
  - **Class 1:** "Not Iris-setosa" (either *Iris-versicolor* or *Iris-virginica*)

1.0.2 How the Perceptron Model Makes Predictions

1. **Training Process:**
   - The perceptron model is trained on a subset of the dataset (`X_train`, `y_train`), adjusting its weights and bias to minimize errors in classifying samples as either *Iris-setosa* (0) or "not Iris-setosa" (1).
2. **Prediction:**
   - The `predict` method uses the learned weights and bias to classify new samples in `X_test`:
     - **0:** If the sample is predicted to be *Iris-setosa*.
     - **1:** If the sample is predicted to be either *Iris-versicolor* or *Iris-virginica*.

1.0.3 How the `fit` Method Works

The `fit` method trains the perceptron by adjusting the weights and bias to minimize the classification error over a specified number of iterations (`self.n_iters`).

Step-by-Step Breakdown

1. **Initialization:**
   - The weights (`self.weights`) are initialized to zeros (or small random values) with a length equal to the number of features in your dataset.
   - The bias (`self.bias`) is initialized to zero. These initial values provide a starting point for learning.
2. **Outer Loop (`for i in range(self.n_iters)`):**
   - This loop runs for a specified number of iterations (`self.n_iters`). Each iteration represents one complete pass over the entire training dataset.
   - The purpose of this loop is to train the model over multiple passes to ensure it converges to a solution that correctly separates the classes.

3. **Inner Loop (`for idx, x_i in enumerate(X)`):**
   - The inner loop iterates over each sample (`x_i`) in the training dataset (`X`) during each iteration of the outer loop.
   - `x_i` represents a single training sample (a vector of feature values), and `idx` is its corresponding index, which is used to access the correct target value (`y[idx]`).
4. **Calculate the Linear Combination (`z`):**
   - The perceptron computes a weighted sum of the input features plus the bias:

$$z = \text{self.weights} \cdot x_i + \text{self.bias}$$

   - Here, `self.weights @ x_i` is the dot product between the weight vector (`self.weights`) and the feature vector (`x_i`), resulting in a scalar value (`z`) representing the linear combination of inputs weighted by the current model weights.
   - Adding the bias (`self.bias`) shifts this linear combination to adjust the decision boundary.
5. **Apply the Step Function to Predict Output (`output`):**
   - The step function (`self._step_function(z)`) is applied to `z` to generate the predicted class label (`output`):
     - If `z >= 0`, the output is `1`.
     - If `z < 0`, the output is `0`.
   - This step function determines which side of the decision boundary the input `x_i` falls on.
6.

1.0.4 Calculate the Error and Update the Weights and Bias

The error is computed as the difference between the actual class label (`y[idx]`) and the predicted class label (`output`):

$$\text{error} = y[\text{idx}] - \text{output}$$

The weights are updated proportionally to the input feature vector and the error:

$$\text{self.weights}+ = \text{self.learning\_rate} \times \text{error} \times x_i$$

If the prediction (`output`) is incorrect, the weights are adjusted to reduce future errors.

The bias is updated similarly:

$$\text{self.bias}+ = \text{self.learning\_rate} \times \text{error}$$

$$

7. **Repeat Until Convergence or Completion of Iterations:**
   - The process repeats for all samples (`x_i`) in each iteration (`self.n_iters`), adjusting the weights and bias to minimize classification error.

Summary

- The `fit` method aims to adjust the weights and bias so that the perceptron correctly classifies as many samples as possible from the training dataset.

[35]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the dataset from the specified file path
df = pd.read_csv(r'C:\Users\Machine-Learning\Downloads\iris\iris.data',␣
↪header=None)

# Assign column names to the dataset
df.columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',␣
↪'class']

# Convert class labels to numerical values
df['class'] = df['class'].map({'Iris-setosa': 0, 'Iris-versicolor': 1,␣
↪'Iris-virginica': 2})

# Prepare the data for binary classification (e.g., "Iris-setosa" vs. "not␣
↪Iris-setosa")
# You can also extend this to a multi-class OvR setup
X = df[['sepal_length', 'sepal_width', 'petal_length', 'petal_width']].values
y = (df['class'] == 0).astype(int).values  # 1 for "Iris-setosa", 0 for "not␣
↪Iris-setosa"

# Split the dataset 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)

# Perceptron Implementation
class Perceptron:
    def __init__(self, learning_rate=0.01, n_iters=1000):
        self.learning_rate=learning_rate
        self.n_iters = n_iters
        self.weights = np.zeros(4)
        self.bias = 0



    def fit(self, X, y):
        i=0
        for i in range(self.n_iters): # Each iteration represents one complete␣
↪pass over the entire training dataset.
```

```python
            for idx, x_i in enumerate(X): # This loop will run over all samples␣
 ↪in the dataset idx=index, x_i=ith sample
                z=self.weights@x_i+self.bias # Calculation of z by calculating␣
 ↪weights and x and adding bias term
                output = self._step_function(z) # Calling the step function to␣
 ↪decide to output 1/0

                self.weights += self.learning_rate*(y[idx]-output)*x_i #␣
 ↪updating the weights
                self.bias += self.learning_rate*(y[idx]-output) #updating the␣
 ↪bias


    def predict(self, X):
        z=X@self.weights+self.bias
        return self._step_function(z)



    def _step_function(self, x):
        return np.where(x >= 0, 1, 0)

# Initialize and train the Perceptron model
perceptron = Perceptron(learning_rate=0.01, n_iters=1000) #Instantiate a␣
 ↪perceptron
perceptron.fit(X_train, y_train) #Fit to the dataset
# Make predictions
y_pred = perceptron.predict(X_test)
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Display weights and bias
print(f"Weights: {perceptron.weights}")
print(f"Bias: {perceptron.bias}")
```

```
Accuracy: 100.00%
Weights: [ 0.007  0.049 -0.066 -0.036]
Bias: 0.01
```

1.1 Explanation of the Perceptron Model's Predictions

1.1.1 Steps for Prediction

1. **Compute the Linear Combination (z):**

For each test sample, the perceptron calculates a linear combination (z) of the input features and the learned weights, plus the bias:

$$z = X \cdot \text{self.weights} + \text{self.bias}$$

This linear combination represents the perceptron's decision boundary: if `z` is greater than or equal to 0, the perceptron predicts class 1; if `z` is less than 0, it predicts class 0.

2. **Apply the Step Function:**

The step function converts this linear combination into a binary decision: - **1** ("not Iris-setosa") if $z \geq 0$ - **0** (*Iris-setosa*) if $z < 0$

3. **Return Predicted Labels:**

The predicted class labels (`y_pred`) for each sample in `X_test` are the outputs of this step function.

1.1.2 Interpretation of Results

- **Accuracy:**
  - The accuracy score calculated using `accuracy_score(y_test, y_pred)` represents the percentage of correct predictions the model made on the test data. For example, an accuracy of 85% means that 85% of the samples in the test set were correctly classified.
- **Weights and Bias:**
  - The **weights** (`self.weights`) show the importance assigned to each feature when making predictions. Higher absolute values indicate greater influence of that feature on the classification decision.
  - The **bias** (`self.bias`) shifts the decision boundary to better fit the training data, helping the perceptron decide the classification threshold.

1.1.3 Summary of Predictions

- The perceptron model learns to classify iris flowers as either *Iris-setosa* or "not Iris-setosa" based on the provided features (sepal length, sepal width, petal length, petal width).
- The predicted labels (`y_pred`) represent the model's classification of the test set samples, and the accuracy score indicates the model's performance in making these predictions.

1.1.4 Next Steps

- **Evaluate Model Performance:** If the accuracy is high, the model is performing well. If not, consider adjusting hyperparameters like the learning rate or number of iterations, or experiment with different features.
- **Experiment with More Data:** Use different subsets or perform cross-validation to assess the model's generalization.
- **Extend to Multi-class Classification:** Consider extending the perceptron to handle all three classes using a one-vs-rest approach, where multiple perceptrons are trained, one for each class.

[ ]:
```

```