Theory and Mathematics of Recursive Least Squares (RLS)

Theory

Recursive Least Squares (RLS) is an adaptive filter algorithm that recursively finds the coefficients θ that minimize a weighted linear least squares cost function relating to the input signals. This method is particularly useful for real-time signal processing where the system model parameters need to be updated continuously as new data comes in.

The RLS algorithm assumes a linear relationship between the input vector b^T and the output $f(x)$, modeled as:

$$f(x) = b^T\theta$$

Objective

The objective of the RLS algorithm is to minimize the sum of weighted squared errors:

$$J_p = \sum_{j=1}^{p} \lambda^{t-i}(f({x_0}^j) - {y_0}^j)^2$$

where λ (0 < λ ≤ 1) is the forgetting factor that gives exponentially less weight to older data.

Mathematical Steps

1. **Initialization:**

   - Initialize the weight vector θ_0 (often starting with zeros).
   - Initialize the inverse correlation matrix P_0 (often starting with a large value times the identity matrix).

   $$\theta_0 = 0$$
   $$P_0 = \alpha I$$

   where α is a large positive constant and I is the identity matrix.

2. **For each step p:**

   - Compute the a priori error:

   $$y_p = f_p(x) = b^T(x_0^p)\theta_p$$
   $$e_p = y_p - b^T(x_0^p)\theta_{p-1}$$

   - Compute the gain vector K_p (the Kamau gain):

   $$K_p = \frac{P_{p-1}b(x_0^p)}{\lambda + b(x_0^p)^T P_{p-1} b(x_0^p)}$$

   - Update the weight vector:

   $$\theta_p = \theta_{p-1} + K_p e_p$$

   - Update the inverse correlation matrix:

   $$P_p = \frac{1}{\lambda}\left(P_{p-1} - K_p b(x_0^p)^T P_{p-1}\right)$$

3. **Compute the cost function $J_p(\theta)$:**

   $$J_p(\theta) = \sum_{i=1}^{t} \lambda^{t-i}(y_i - x_i^T\theta)^2$$

Code Explanation

1. **Import necessary libraries and load the data**:

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

data = pd.read_csv('data/rls_data_unsortted.csv')
```

2. **Initialize parameters**:

```
n_features = 4 # Number of inputs
w = np.zeros(n_features) # The theta matrix
```

```
P = np.eye(n_features) * 1000  # Large initial covariance, alpha = 1000
lambda_ = 1  # Forgetting factor, Default 1

y_pred = np.zeros(len(data)) # Initialize array for predictions values
J_p = np.zeros(len(data)) # Initialize array for cost function values
```

3. **RLS algorithm with cost function computation**:

```
for t in range(len(data)):
    x_p = data.iloc[t, :-1].values # The value of "b"
    y_p = data.iloc[t, -1]

    y_pred[t] = np.dot(x_p, w)
    error = y_p - y_pred[t]

    K_p = np.dot(P, x_p) / (lambda_ + np.dot(x_p.T, np.dot(P, x_p)))
    w += K_p * error
    P = (P - np.outer(K_p, np.dot(x_p.T, P))) / lambda_

    # Compute the cost function J_p(w)
    J_p[t] = np.sum([lambda_**(t-i) * (data.iloc[i, -1] - np.dot(data.iloc[i, :-1].values, w))**2 for i in range(t+1)])
```

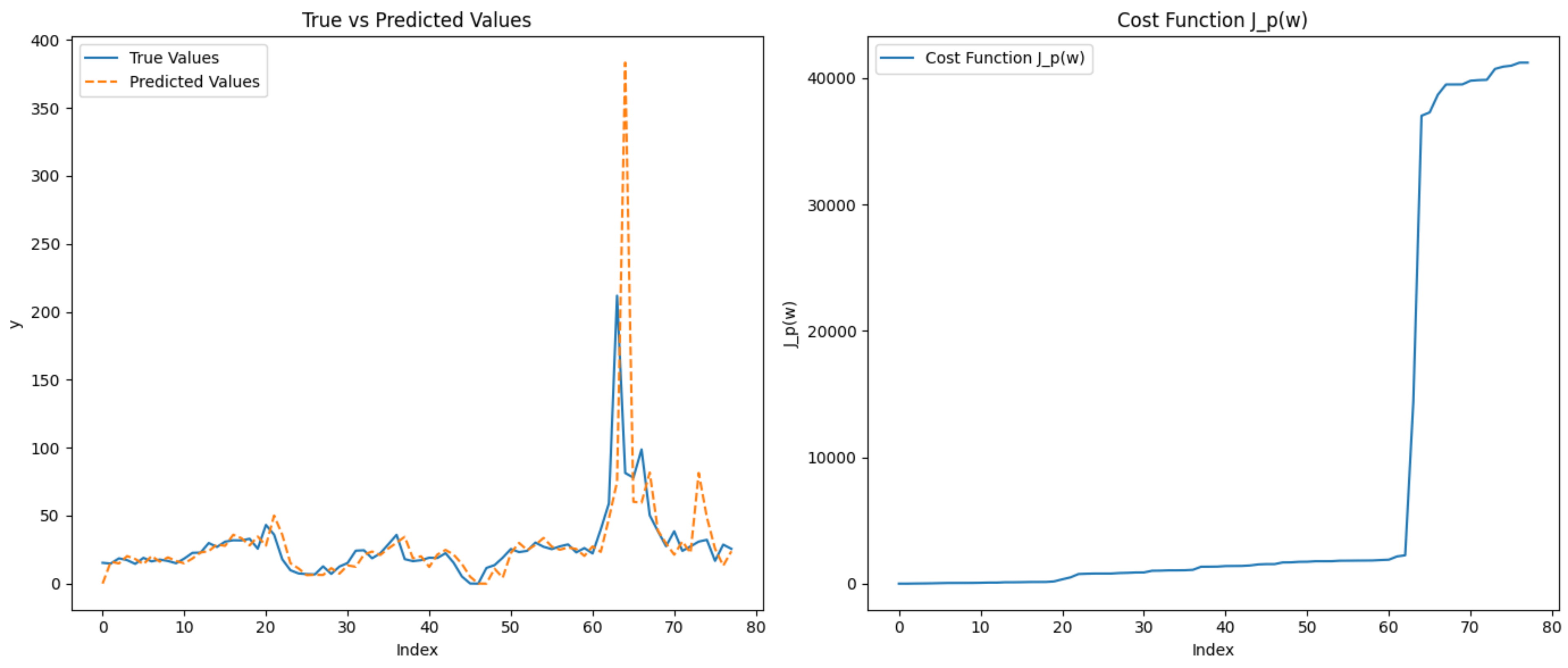
4. **Plotting true vs predicted values and cost function**:

```
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
plt.plot(data.index, data['y'], label='True Values')
plt.plot(data.index, y_pred, label='Predicted Values', linestyle='--')
plt.xlabel('Index')
plt.ylabel('y')
plt.legend()
plt.title('True vs Predicted Values')

plt.subplot(1, 2, 2)
plt.plot(range(len(data)), J_p, label='Cost Function J_p(w)')
plt.xlabel('Index')
plt.ylabel('J_p(w)')
plt.legend()
plt.title('Cost Function J_p(w)')

plt.tight_layout()
plt.show()
```



By Amirreza Yarahmadi