Code Analysis Report

Analyzed Code:

```
################################################################################
########
## PROBLEM1: Gradient Descent
## Gradient descent is a popular optimization technique to solve many
## machine learning problems. In this case, we will explore the gradient
## descent algorithm to fit a line for the given set of 2-D points.
## ref: https://tinyurl.com/yc4jbjzs
## ref:
https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression/
##
##
## input: directory of faces in ./data/1_points.csv/
## function for reading points is provided
##
##
## your task: fill the following functions:
## evaluate_cost
## evaluate_gradient
## udpate_params
## NOTE: do NOT change values of 'init_params' and 'max_iterations' in
optimizer
##
##
## output: cost after convergence (rmse, lower the better)
##
##
## NOTE: all required modules are imported. DO NOT import new modules.
## NOTE: references are given intline
## tested on Ubuntu14.04, 22Oct2017, Abhilash Srikantha
################################################################################
########

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

```
import time

def load_data(fname):
points = np.loadtxt(fname, delimiter=',')
y_ = points[:,1]
# append '1' to account for the intercept
x_ = np.ones([len(y_),2])
x_[:,0] = points[:,0]
# display plot
#plt.plot(x_[:,0], y_, 'ro')
#plt.xlabel('x-axis')
#plt.ylabel('y-axis')
#plt.show()
print('data loaded. x:{} y:{}'.format(x_.shape, y_.shape))
return x_, y_

def evaluate_cost(x_,y_,params):
tempcost = 0
for i in range(len(y_)):
tempcost += (y_[i] - ((params[0] * x_[i,0]) + params[1])) ** 2
return tempcost / float(10000)

def evaluate_gradient(x_,y_,params):
m_gradient = 0
b_gradient = 0
N = float(len(y_))
for i in range(len(y_)):
m_gradient += -(2/N) * (x_[i,0] * (y_[i] - ((params[0] * x_[i,0]) +
params[1])))
b_gradient += -(2/N) * (y_[i] - ((params[0] * x_[i,0]) + params[1]))
return [m_gradient,b_gradient]

def update_params(old_params, grad, alpha):
new_m = old_params[0] - (alpha * grad[0])
new_b = old_params[1] - (alpha * grad[1])
return [new_m,new_b]

# initialize the optimizer
optimizer = {'init_params':np.array([4.5,2.0]) ,
'max_iterations':10000,
'alpha':0.69908,
```

```
'eps':0.0000001,
'inf':1e10}

# load data
x_, y_ = load_data("./data/1_points.csv")

# time stamp
start = time.time()

try:
# gradient descent
params = optimizer['init_params']
old_cost = 1e10
for iter_ in range(optimizer['max_iterations']):
# evaluate cost and gradient
cost = evaluate_cost(x_,y_,params)
grad = evaluate_gradient(x_,y_,params)
# display
if(iter_ % 10 == 0):
print('iter: {} cost: {} params: {}'.format(iter_, cost, params))
# check convergence
if(abs(old_cost - cost) < optimizer['eps']):
break
# udpate parameters
params = update_params(params,grad,optimizer['alpha'])
old_cost = cost
except:
cost = optimizer['inf']

# final output
print('time elapsed: {}'.format(time.time() - start))
print('cost at convergence: {} (lower the better)'.format(cost))
```

Static Code Metrics:

Metric	Value
Lines of Code (LOC)	92
Comment Density	0.84
Cyclomatic Complexity	6

Maintainability Index	70.81439155772465
Halstead Metrics	
Unique Operators	9
Unique Operands	54
Total Operators	30
Total Operands	58
Volume	526.0006332679927
Effort	2542.336394128631
Depth of Inheritance Tree (DIT)	0
Coupling Between Object classes (CBO)	0
Lack of Cohesion of Methods (LCOM)	0
Fan-in	9
Fan-out	9
Number of Methods (NOM)	0

McCall's Intermediate Quality Metrics:

Metric	Value
Modifiability	0.82
Testability	0.79
Reliability	0.65
Understandability	0.84
Self-Descriptiveness	0.47
Reusability	1.0
Portability	0.9
Efficiency	0

GPT Analysis Report:

Code Analysis and Improvement Suggestions

1. **Syntax Error in `evaluate_cost` Function**:

- **Issue**: There's a missing parenthesis `)` on the line calculating `tempcost`.

- **Improvement**: Correct the syntax by adding the missing parenthesis.

```
python
tempcost += (y_[i] - ((params[0] * x_[i,0]) + params[1])) ** 2
```

2. **Vectorization**:

- **Issue**: The cost and gradient computations are performed using Python loops.

- **Improvement**: Use NumPy's vectorized operations for better performance and readability. This can reduce complexity and improve speed for larger datasets.

```
python
def evaluate_cost(x_, y_, params):
predictions = x_ @ params # Dot product for predictions
errors = y_ - predictions
return np.mean(errors ** 2)

def evaluate_gradient(x_, y_, params):
predictions = x_ @ params
errors = y_ - predictions
gradient = -2/len(y_) * (x_.T @ errors) # Vectorized gradient calculation
return gradient
```

3. **Function Naming**:

- **Issue**: The function name `udpate_params` contains a typo.

- **Improvement**: Rename the function to `update_params` for correctness.

4. **Constants and Hardcoded Values**:

- **Issue**: The constant values for both cost averaging and the initial parameters are hardcoded, which may lead to difficulties if further adjustments are needed.

- **Improvement**: Define these constants at the top of the file to enhance clarity and ease of updates.

```
python
COST_AVERAGE_DIVISOR = 10000 # Define clear constant
```

5. **Parameter Handling**:

- **Issue**: The optimizer parameters are stored in a dictionary, which affects readability and makes accessing them a bit cumbersome.

- **Improvement**: Consider using a class to encapsulate the optimizer logic and parameters, enhancing organization and maintainability.

6. **Error Handling**:

- **Issue**: The current error handling is broad and is likely to catch all exceptions, which obscures the underlying issue.

- **Improvement**: Use specific exception handling to make debugging easier. For instance, handle file reading errors and ensure that exceptions provide more informative messages.

7. **Logging Instead of Print**:

- **Issue**: The use of print statements for tracking iteration progress and errors may clutter the output.

- **Improvement**: Use Python's `logging` module for more flexible logging and easier toggling between levels of verbosity.

```
python
import logging

logging.basicConfig(level=logging.INFO)
# Replace print with logging
logging.info('data loaded. x:{} y:{}'.format(x_.shape, y_.shape))
```

8. **Docstrings and Comments**:

- **Issue**: While comments are present, they could be more systematically applied, especially with function-level docstrings.

- **Improvement**: Add docstrings for each function detailing parameters, return values, and purpose. This enhances self-descriptiveness and understandability.

```
python
def evaluate_cost(x_, y_, params):
"""
Calculate the cost (mean squared error) for the given parameters.

Parameters:
x_ (np.ndarray): Input features.
y_ (np.ndarray): Target values.
params (np.ndarray): Model parameters.

Returns:
float: Computed cost.
"""
```

9. **Parameter Validation**:

- **Issue**: There's no validation of input parameters before calculations.

- **Improvement**: Before calculations, you can include basic validation checks to ensure parameters and data are of the expected types and shape.

10. **Performance Metrics Tracking**:

- **Issue**: While the cost is being printed during training, tracking other metrics indicating performance, such as gradient norm, could provide more insight.

- **Improvement**: Add performance indicators where necessary, especially if adaptations are made to the gradient descent logic.

11. **Separation of Concerns**:

- **Issue**: The code intermingles data loading, model training, and printed outputs.

- **Improvement**: Consider separating the data loading, model training, and result reporting into different functions or classes to clarify the workflow.

Summary of Changes

- Fix syntax errors and function typos.
- Utilize NumPy's capabilities to vectorize calculations.
- Define constants clearly at the top of the script.
- Adopt a class to encapsulate optimizer parameters.
- Improve error handling for clarity and debugging.
- Use logging instead of print statements for better control over output.
- Enhance code readability through docstrings and better parameter validation.
- Track additional performance metrics if applicable.
- Organize code into distinct functions to delineate responsibilities.

By implementing these changes, the code will not only become cleaner and more efficient but also easier to maintain and extend in the future.