lab_nlls_partial (2)

October 31, 2023

Name : Shadeeb Hossain ID: sh7492 # Lab: Nonlinear Least Squares for Modeling Materials

Nonlinear least squares (NLLS) is a widely-used method for modeling data. In NLLS, we wish to fit a model of the form,

```
yhat = g(x,w)
```

where `w` is a vector of paramters and `x` is the vector of predictors. We find `w` by minimizing a least-squares function

```
f(w) = \sum_i (y_i - g(x_i,w))^2
```

where the summation is over training samples `(x_i,y_i)`. This is similar to linear least-squares, but the function `g(x,w)` may not be linear in `w`. In general, this optimization has no closed-form expression. So numerical optimization must be used.

In this lab, we will implement gradient descent on NLLS in a problem of physical modeling of materials. Specifically, we will estimate parameters for expansion of copper as a function of temperature using a real dataset. In doing this lab, you will learn to: * Set up a nonlinear least squares as an unconstrained optimization function * Compute initial parameter estimates for a simple rational model * Compute the gradients of the least squares objective * Implement gradient descent for minimizing the objective * Implement momentum gradient descent * Visualize the convergence of the algorithm

We first import some key packages.

[20]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge, LinearRegression
```

0.1 Load the Data

The NIST agency has an excellent nonlinear regression website that has several datasets for nonlinear regression problems. In this lab, we will use the data from a NIST study involving the thermal expansion of copper. The response variable is the coefficient of thermal expansion, and the predictor variable is temperature in degrees kelvin.

Hahn, T., NIST (1979), Copper Thermal Expansion Study. (unpublished}

You can download the data as follows.

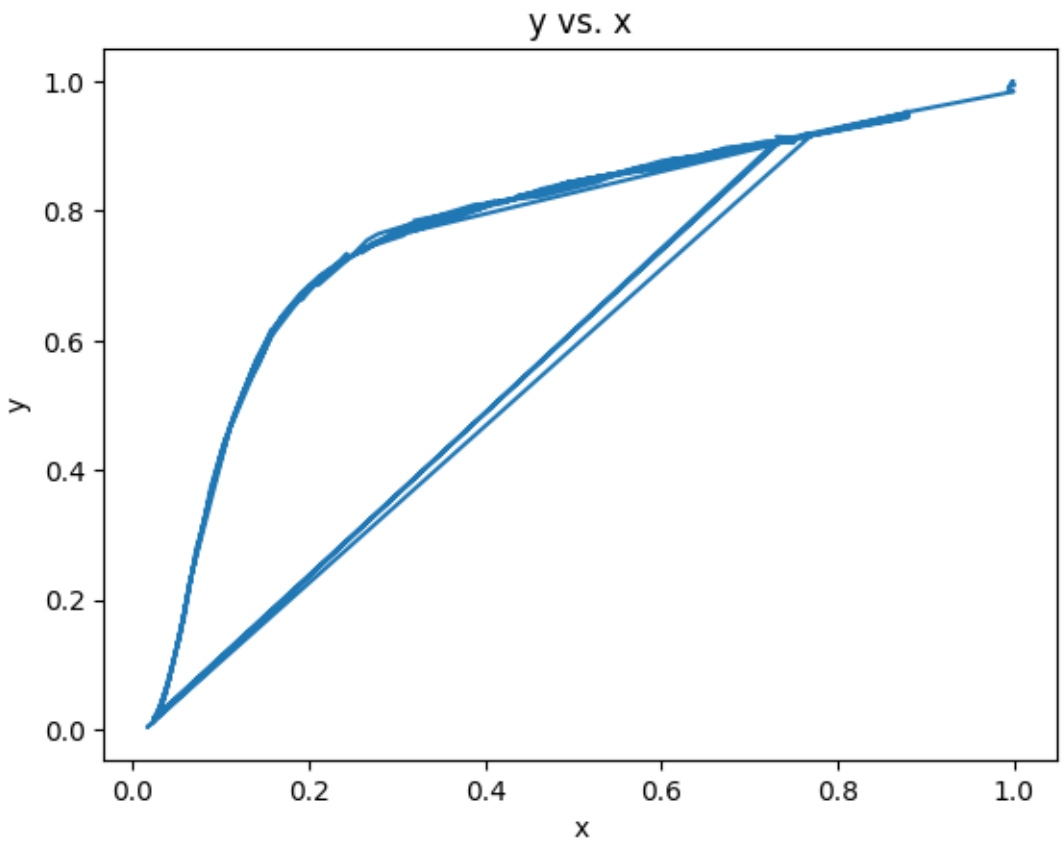
[21]:
```
url = 'https://itl.nist.gov/div898/strd/nls/data/LINKS/DATA/Hahn1.dat'
df = pd.read_csv(url, skiprows=60, sep=' ',skipinitialspace=True,␣
 ↪names=['y0','x0','dummy'])
df.head()
```

[21]:

	y0	x0	dummy
0	0.591	24.41	NaN
1	1.547	34.82	NaN
2	2.902	44.09	NaN
3	2.894	45.07	NaN
4	4.703	54.98	NaN

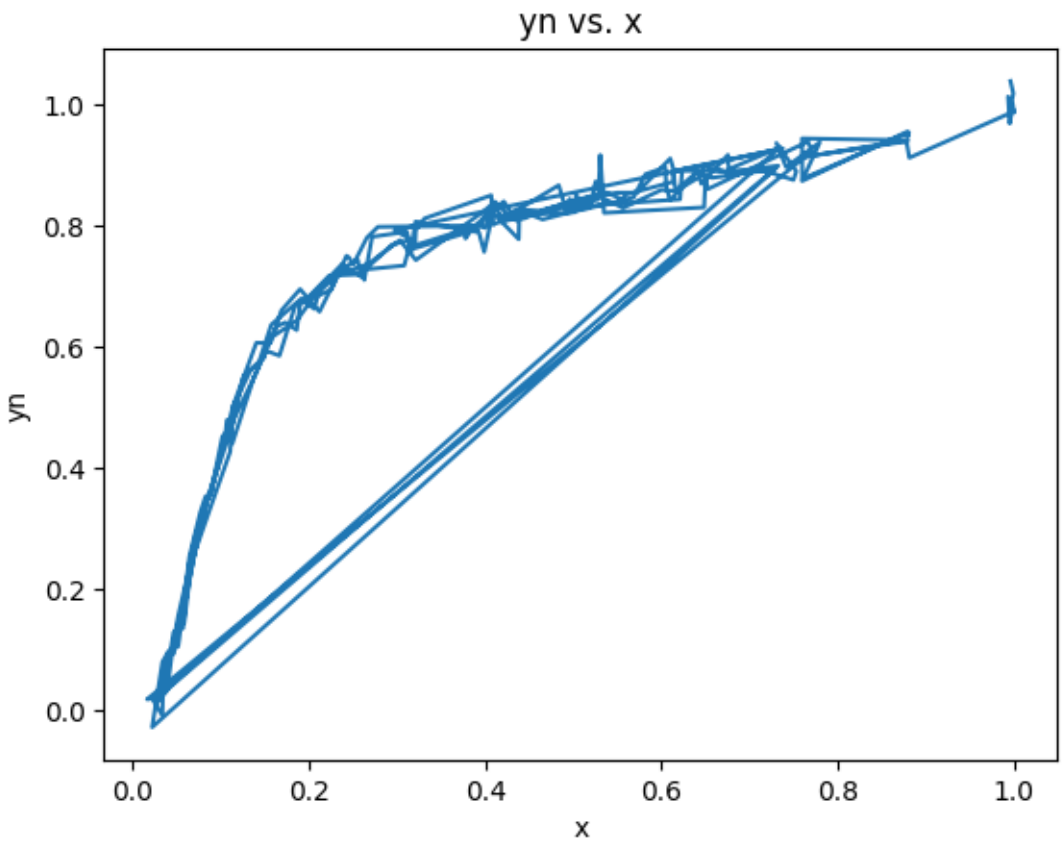
Extract the `x0` and `y0` into arrays. Rescale, `x0` and `y0` to values between `0` and `1` by dividing `x0` and `y0` by the maximum value. Store the scaled values in vectors `x` and `y`. The rescaling will help with the conditioning of the fitting. Plot, `y` vs. `x`.

[22]:
```
# TODO 1
x0 = np.array('x0')
y0 = np.array('y0')
x0 = df['x0'].values   # use this in order to use the function np.max(x0) . np.
 ↪array does not work with np.max
y0 = df['y0'].values
x = x0 / np.max(x0)
y = y0 / np.max(y0)
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.title('y vs. x')
plt.show()
```



To make the problem a little more challenging, we will add some noise. Add random Gaussian noise with mean 0 and std. dev = 0.02 to `y`. Store the noisy results in `yn`. You can use the `np.random.normal()` function to add Gaussian noise. Plot `yn` vs. `x`.

[23]:
```
# TODO 2
std_deviation = 0.02
yn = y + np.random.normal(0, std_deviation, len(y))
plt.plot(x, yn)
plt.xlabel('x')
plt.ylabel('yn')
plt.title('yn vs. x')
plt.show()
```



Split the data `(x,yn)` into training and test. Let `xtr,ytr` be training data and `xts,yts` be the test data. You can use the `train_test_split` function. Set `test_size=0.33` so that 1/3 of the samples are held out for test.

[24]:
```
from sklearn.model_selection import train_test_split

# TODO 3
xtr, xts, ytr, yts = train_test_split(x, yn, test_size=0.33)
```

0.2 Initial Fit for a Rational Model

The NIST website suggests using a *rational* model of the form,

```
yhat = (a[0] + a[1]*x + ... + a[d]*x^d)/(1 + b[0]*x + ... + b[d-1]*x^d)
```

with `d=3`. The model parameters are `w = [a[0],...,a[d],b[0],...,b[d-1]]` so there are `2d+1` parameters total. Complete the function below that takes vectors `w` and `x` and predicts a set of values `yhat` using the above model.

[25]:
```
def predict(w,x):

    # Get the length
    d = (len(w)-1)//2

    # TODO 4.  Extract a and b from w
    a = w[:d + 1]
    b = w[d + 1:]

    # TODO 5.  Compute yhat.  You may use the np.polyval function
    # But, remember you must flip the order the a and b
    yhat = np.polyval(a[::-1], x) / (1 + np.polyval(b[::-1], x))

    return yhat
```

When we fit with a nonlinear model, most methods only get convergence to a local minima. So, you need a good initial condition. For a rational model, one way to get is to realize that if:

`y ~= (a[0] + a[1]*x + ... + a[d]*x^d)/(1 + b[0]*x + ... + b[d-1]*x^d)`

Then:

`y ~= a[0] + a[1]*x + ... + a[d]*x^d - b[0]*x*y + ... - b[d-1]*x^d*y.`

So, we can solve for the the parameters `w = [a,b]` from linear regression of the predictors,

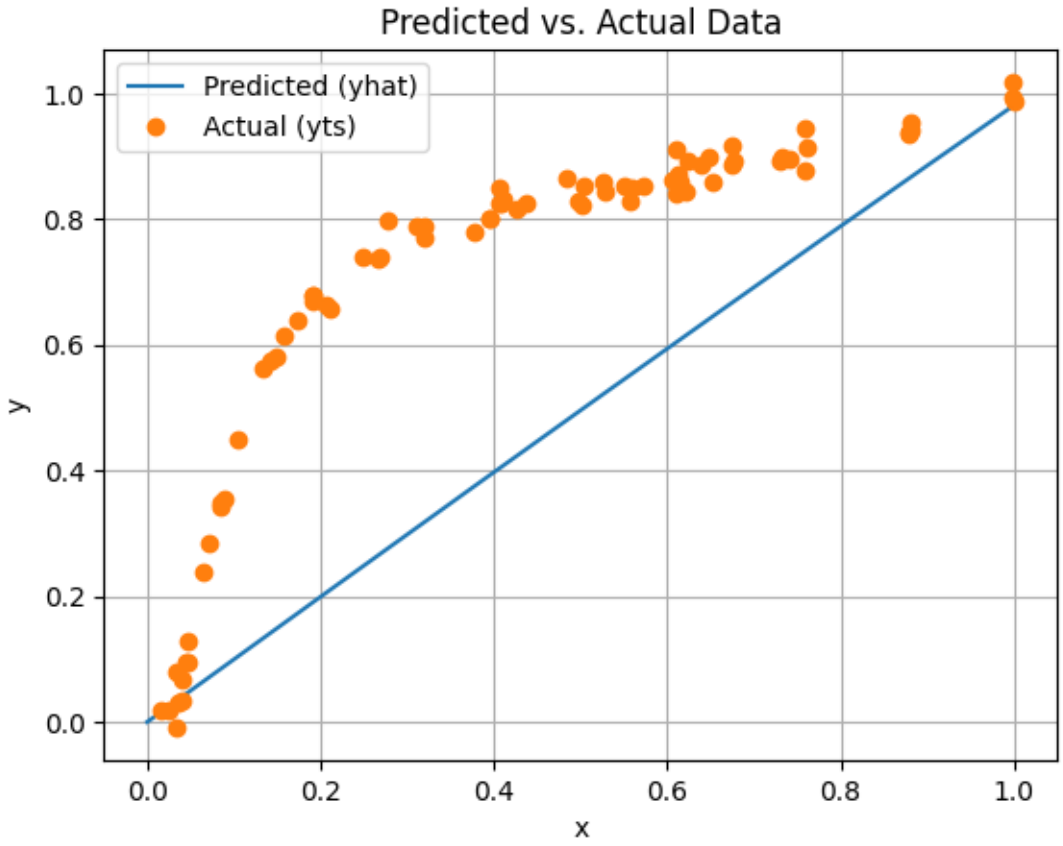
`Z[i,:] = [ x[i], ... , x[i]**d, y[i]*x[i], ... , y[i}*x[i]**d ]`

[37]:
```
d = 3

# TODO 6.  Create the transformed feature matrix
Z = np.column_stack([x**i for i in range(1, d + 1)] + [x**i * y for i in␣
 ↪range(d)])


# TODO 7.  Fit with parameters with linear regression
regr = LinearRegression()
regr.fit(Z, y)
# TODO 8
# Extract the parameters from regr.coef_ and regr.intercept_ and store the␣
 ↪parameter vector in winit
#winit = np.concatenate((regr.intercept_, regr.coef_[-d:]))   - This was␣
 ↪generating an error that zero dimensional array cannot be concatenated.
winit = np.concatenate((regr.intercept_.reshape(1), regr.coef_[-d:]))
```

Now plot the predicted values of the `yhat` vs. `x` using your estimated parameter `winit` for 1000 values `x` in `[0,1]`. On the same plot, plot `yts` vs. `xts`. You will see that you get a horrible fit.

[56]:
```
# TODO 9
xp = np.linspace(0, 1, 1000)
```

```
yhat = np.polyval(winit[:d+1][::-1], xp) / (1 + np.polyval(winit[d+1:][::-1],␣
 ↪xp))

plt.plot(xp, yhat, label='Predicted (yhat)')
plt.plot(xts, yts, 'o', label='Actual (yts)')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Predicted vs. Actual Data')
plt.legend()
plt.grid(True)
plt.show()
# xp = ...
# yhat = ...
# plot(...)
```



The reason the previous fit is poor is that the denominator in `yhat` goes close to zero. To avoid this problem, we can use Ridge regression, to try to keep the parameters close to zero. Re-run the fit above with `Ridge` with `alpha = 1e-3`. You should see you get a reasonable, but not perfect fit.

[58]:
```
# TODO 10.  Fit with parameters with linear regression
X_powers = np.column_stack([xp**i for i in range(1, d + 1)])
XY_powers = np.column_stack([xp**i * np.interp(xp, x, y) for i in range(1, d +␣
 ↪1)])
Z = np.column_stack((X_powers, XY_powers))
regr = Ridge(alpha=1e-3)
regr.fit(Z, y)

# TODO 11
# Extract the parameters from regr.coef_ and regr.intercept_
winit = np.concatenate((regr.intercept_.reshape(1), regr.coef_[-d:]))  # make␣
 ↪sure to add.reshape(1) or you would get the same error as above


# TODO 12
X_powers = np.column_stack([xp**i for i in range(1, d + 1)])
XY_powers = np.column_stack([xp**i * yhat for i in range(1, d + 1)])
Z = np.column_stack((X_powers, XY_powers))
yhat = np.polyval(winit[:d+1][::-1], xp) / (1 + np.polyval(winit[d+1:][::-1],␣
 ↪xp))

# Plot the results as above.
xp = np.linspace(0, 1, 1000)
plt.plot(xp, yhat, label='Predicted (yhat)')
plt.plot(xts, yts, 'ro', label='Actual (yts)')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Predicted vs. Actual Data (Ridge Regression)')
plt.legend()
plt.grid(True)
plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-58-d6e0292414c7> in <cell line: 6>()
      4 Z = np.column_stack((X_powers, XY_powers))
      5 regr = Ridge(alpha=1e-3)
----> 6 regr.fit(Z, y)
      7
      8 # TODO 11

/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_ridge.py in␣
 ↪fit(self, X, y, sample_weight)
   1124
   1125         _accept_sparse = _get_valid_accept_sparse(sparse.issparse(X),␣
 ↪self.solver)
```

```
-> 1126         X, y = self._validate_data(
   1127             X,
   1128             y,

/usr/local/lib/python3.10/dist-packages/sklearn/base.py in _validate_data(self,
 ↪X, y, reset, validate_separately, **check_params)
    582                 y = check_array(y, input_name="y", **check_y_params)
    583             else:
--> 584                 X, y = check_X_y(X, y, **check_params)
    585             out = X, y
    586

/usr/local/lib/python3.10/dist-packages/sklearn/utils/validation.py in␣
 ↪check_X_y(X, y, accept_sparse, accept_large_sparse, dtype, order, copy,␣
 ↪force_all_finite, ensure_2d, allow_nd, multi_output, ensure_min_samples,␣
 ↪ensure_min_features, y_numeric, estimator)
   1122     y = _check_y(y, multi_output=multi_output, y_numeric=y_numeric,␣
 ↪estimator=estimator)
   1123
-> 1124     check_consistent_length(X, y)
   1125
   1126     return X, y

/usr/local/lib/python3.10/dist-packages/sklearn/utils/validation.py in␣
 ↪check_consistent_length(*arrays)
    395     uniques = np.unique(lengths)
    396     if len(uniques) > 1:
--> 397         raise ValueError(
    398             "Found input variables with inconsistent numbers of samples␣
 ↪%r"
    399             % [int(l) for l in lengths]

ValueError: Found input variables with inconsistent numbers of samples: [1000,␣
 ↪236]
```

0.3 Creating a Loss Function

We can now use gradient descent to improve our initial estimate. Complete the following function to compute

`f(w) = 0.5*\sum_i (y[i] - yhat[i])^2`

and `fgrad`, the gradient of `f(w)`.

[70]:
```
def feval(w,x,y):


    # TODO 13.  Parse w
```

```
    a = w[:d+1]
    b = w[d+1:]
    # a = ...
    # b = ...

    # TODO 14.  Znum[i,j] = x[i]**j
    Znum = np.column_stack([x**j for j in range(d+1)])
    # TODO 15.  Zden[i,j] = x[i]**(j+1)
    Zden = np.column_stack([x**(j+1) for j in range(d)])
    # TODO 16.  Compute yhat
    # Compute the numerator and denominator
    yhat_num = np.dot(Znum, a)  # numerator
    yhat_den = np.dot(Zden, b)   # Denominator

    yhat = yhat_num / (1 + yhat_den)
    # TODO 17.  Compute loss
    f = 0.5 * np.sum((y - yhat) ** 2)

    # TODO 18.  Compute gradients
    # fgrad = ...
    ydiff = y - yhat
    fgrad_a = -np.dot(Znum.T, ydiff / (1 + yhat_den))
    fgrad_b = np.dot(Zden.T, ydiff * yhat_num / (1 + yhat_den)**2)
    fgrad = np.concatenate((fgrad_a, fgrad_b))
    return f, fgrad
```

Test the gradient function: * Take `w0=winit` and compute `f0,fgrad0 = feval(w0,xtr,ytr)` * Take `w1` very close to `w0` and compute `f1,fgrad1 = feval(w1,xtr,ytr)` * Verify that `f1-f0` is close to the predicted value based on the gradient.

[76]:
```
# TODO 19
w0=winit
f0,fgrad0=feval(w0,xtr,ytr)

w1=w0+0.001
f1,fgrad1=feval(w1,xtr,ytr)


actual=f1-f0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-76-b4c4fd83c89b> in <cell line: 2>()
      1 # TODO 19
----> 2 xtr.shape()
      3 w0=winit
      4 f0,fgrad0=feval(w0,xtr,ytr)
      5
```

```
TypeError: 'tuple' object is not callable
```

0.4 Implement gradient descent

We will now try to minimize the loss function with gradient descent. Using the function `feval` defined above, implement gradient descent. Run gradient descent with a step size of `alpha=1e-6` starting at `w=winit`. Run it for `nit=10000` iterations. Compute `fgd[it]`= the objective function on iteration `it`. Plot `fgd[it]` vs. `it`.

You should see that the training loss decreases, but it still hasn't converged after 10000 iterations.

[77]:
```
# TODO 20
nit = 10000
step = 1e-6
fgd = np.zeros(nit)

w = winit
for it in range(nit):
    f, fgrad = feval(w, xtr, ytr)
    w -= step * fgrad
    fgd[it] = f
plt.figure(figsize=(10, 6))
plt.plot(range(nit), fgd)
plt.xlabel('Iteration')
plt.ylabel('Objective Function Value')
plt.title('Gradient Descent Training Loss')
plt.grid(True)
plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-77-566a803a74a7> in <cell line: 7>()
      6 w = winit
      7 for it in range(nit):
----> 8     f, fgrad = feval(w, xtr, ytr)
      9     w -= step * fgrad
     10     fgd[it] = f

<ipython-input-70-65f41e0f3c58> in feval(w, x, y)
     15     # Compute the numerator and denominator
     16     yhat_num = np.dot(Znum, a)  # numerator
---> 17     yhat_den = np.dot(Zden, b)   # Denominator
     18
     19     yhat = yhat_num / (1 + yhat_den)
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/overrides.py in dot(*args,␣
↪**kwargs)

ValueError: shapes (158,3) and (0,) not aligned: 3 (dim 1) != 0 (dim 0)
```

Now, try to get a faster convergence with adaptive step-size using the Armijo rule. Implement the gradient descent with adaptive step size. Let `fadapt[it]` be the loss function on iteration `it`. Plot `fadapt[it]` and `fgd[it]` vs. `it` on the same graph. You should see a slight improvement, but not much.

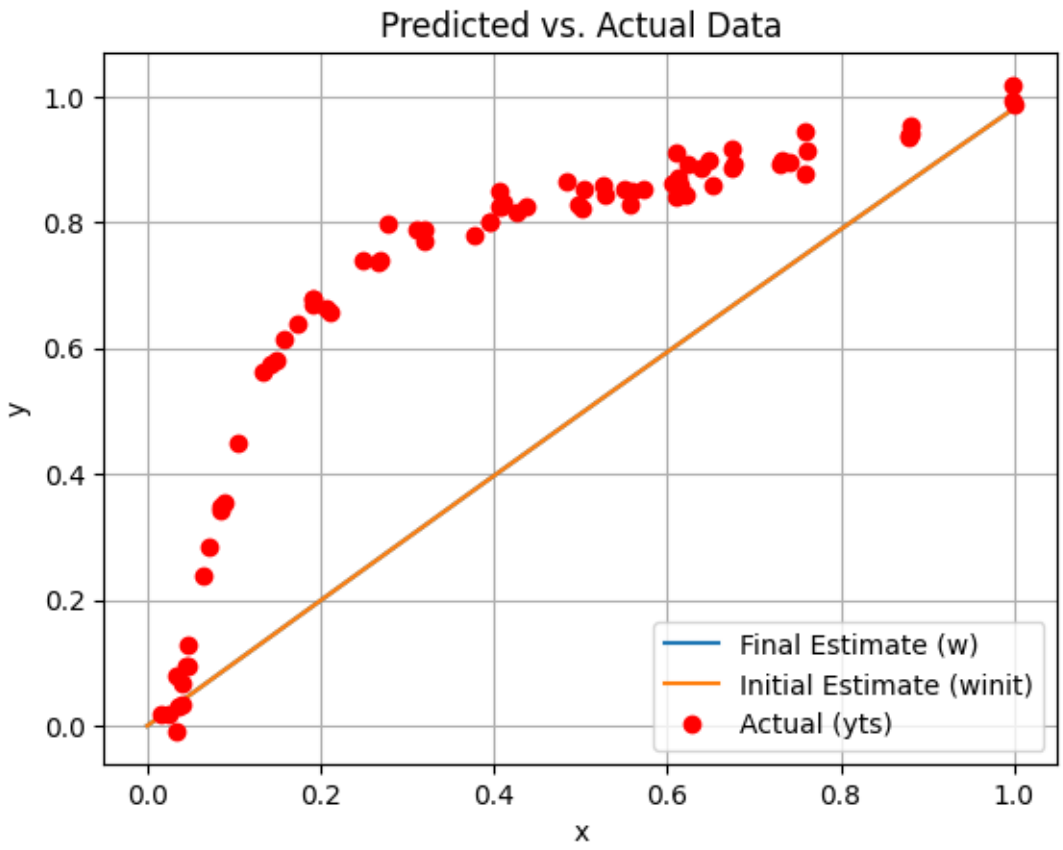
[80]:
```
# TODO 21
# fadapt
nit = 10000
step = 1e-6  # Initial step
fadapt = np.zeros(nit)

w = winit
for it in range(nit):

    f, fgrad = feval(w, xtr, ytr)
while True:
        w_new = w - step * fgrad
        f_new, _ = feval(w_new, xtr, ytr)
        if f_new <= f + step * 1e-4 * np.dot(fgrad, fgrad):
            break
w = w_new
fadapt[it] = f
plt.plot(range(nit), fgd, label='Gradient Descent (Fixed Step)')
plt.plot(range(nit), fadapt, label='Gradient Descent (Adaptive Step)')
plt.xlabel('Iteration')
plt.ylabel('Objective Function Value')
plt.title('Gradient Descent Training Loss')
plt.legend()
plt.grid(True)
plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-80-75e258684533> in <cell line: 8>()
      8 for it in range(nit):
      9
---> 10     f, fgrad = feval(w, xtr, ytr)
     11 while True:
     12         w_new = w - step * fgrad

<ipython-input-70-65f41e0f3c58> in feval(w, x, y)
     15     # Compute the numerator and denominator
```

```
     16     yhat_num = np.dot(Znum, a)  # numerator
---> 17     yhat_den = np.dot(Zden, b)   # Denominator
     18
     19     yhat = yhat_num / (1 + yhat_den)

/usr/local/lib/python3.10/dist-packages/numpy/core/overrides.py in dot(*args,␣
 ↪**kwargs)

ValueError: shapes (158,3) and (0,) not aligned: 3 (dim 1) != 0 (dim 0)
```

Using he final estimate for `w` from the adaptive step-size plot the predicted values of the `yhat` vs. `x` usfor 1000 values `x` in `[0,1]`. On the same plot, plot `yhat` vs. `x` for the initial parameter `w=winit`. Also, plot `yts` vs. `xts`. You should see that gradient descent was able to improve the estimat slightly, although the initial estimate was not too bad.

[82]:
```
# TODO 22
xp = np.linspace(0, 1, 1000)
yhat_final = np.polyval(w[:d+1][::-1], xp) / (1 + np.polyval(w[d+1:][::-1], xp))
yhat_init = np.polyval(winit[:d+1][::-1], xp) / (1 + np.polyval(winit[d+1:][::
 ↪-1], xp))
plt.plot(xp, yhat_final, label='Final Estimate (w)')
plt.plot(xp, yhat_init, label='Initial Estimate (winit)')
plt.plot(xts, yts, 'ro', label='Actual (yts)')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Predicted vs. Actual Data')
plt.legend()
plt.grid(True)
plt.show()
```



0.5 Momentum Gradient Descent

This section is bonus.

One way to improve gradient descent is to use *momentum*. In momentum gradient descent, the update rule is:

```
f, fgrad = feval(w,...)
z = beta*z + fgrad
w = w - step*z
```

This is similar to gradient descent, except that there is a second order term on the gradient. Implement this algorithm with `beta = 0.99` and `step=1e-3`. Compare the convergence of the loss function with gradient descent.

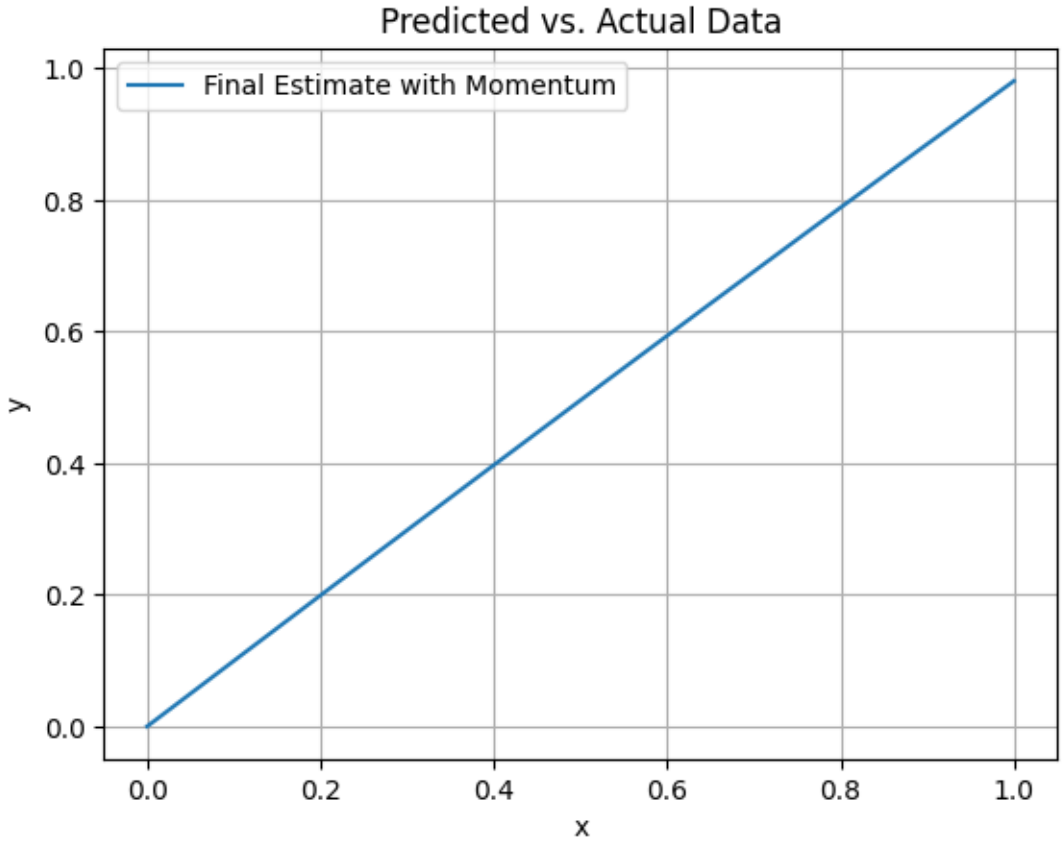
[83]:
```
# TODO 23
nit = 1000
step = 1e-3
beta = 0.99
fgd_momentum = np.zeros(nit)
w = winit
```

```
z = np.zeros_like(w)
for it in range(nit):
    f, fgrad = feval(w, xtr, ytr)
    z = beta * z + fgrad
    w -= step * z
    fgd_momentum[it] = f
plt.plot(range(nit), fgd, label='Gradient Descent (Fixed Step)')
plt.plot(range(nit), fgd_momentum, label='Gradient Descent with Momentum')
plt.xlabel('Iteration')
plt.ylabel('Objective Function Value')
plt.title('Gradient Descent Training Loss')
plt.legend()
plt.grid(True)
plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-83-b4f3fc1a2956> in <cell line: 8>()
      7 z = np.zeros_like(w)
      8 for it in range(nit):
----> 9     f, fgrad = feval(w, xtr, ytr)
     10     z = beta * z + fgrad
     11     w -= step * z

<ipython-input-70-65f41e0f3c58> in feval(w, x, y)
     15     # Compute the numerator and denominator
     16     yhat_num = np.dot(Znum, a)  # numerator
---> 17     yhat_den = np.dot(Zden, b)   # Denominator
     18
     19     yhat = yhat_num / (1 + yhat_den)

/usr/local/lib/python3.10/dist-packages/numpy/core/overrides.py in dot(*args,␣
 ↪**kwargs)

ValueError: shapes (158,3) and (0,) not aligned: 3 (dim 1) != 0 (dim 0)
```

[84]:
```
# TODO 24
# plot yhat vs. x

xp = np.linspace(0, 1, 1000)
yhat_momentum = np.polyval(w[:d+1][::-1], xp) / (1 + np.polyval(w[d+1:][::-1],␣
 ↪xp))
plt.plot(xp, yhat_momentum, label='Final Estimate with Momentum')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Predicted vs. Actual Data')
```

```
plt.legend()
plt.grid(True)
plt.show()
```



0.6 Beyond This Lab

In this lab, we have just touched at some of the ideas in optimization. There are several other important algorithms that you can explore: * Levenberg-Marquardt method for non-linear least squares * Newton's method * More difficult non-linear least squares problems.

[ ]: