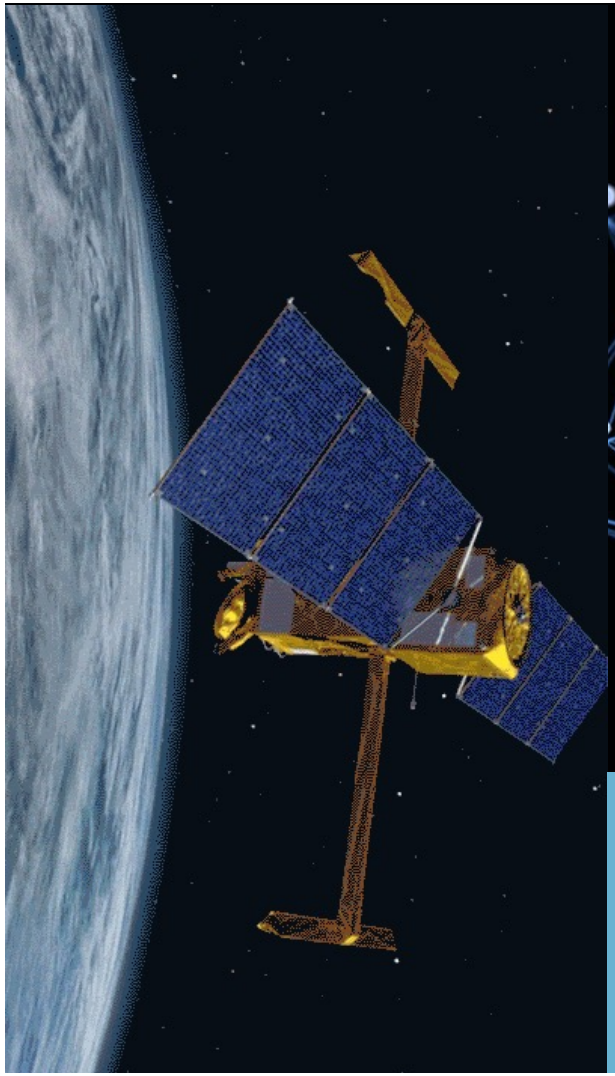


Lecture 4
Multivariate &
Non-Linear Numerical Regression

First, Multivariate Linear Regression

Problem Statement

- ***n* features:** $x_j^{(i)}$ (j=1,…,m for m measured values)
- 1 **target**: y_j
- Supervised learning: m known labels $y_{j,\text{true}}$
- **Hypothesis**: $\boldsymbol{y}_{\text{pred}} = \theta_0 + \theta_1 \boldsymbol{x}_1 + \theta_2 \boldsymbol{x}_2 , \ldots , + \theta_n \boldsymbol{x}_n ,$
- Or including bias $\boldsymbol{x}_0$=1, in matrix notation: $\boldsymbol{y}_{\text{pred}} = h(\boldsymbol{\theta} ; \boldsymbol{X}) = \boldsymbol{X} . \boldsymbol{\theta}$

- Example with just 2 features (plus bias): $\theta_0 = 100$, $\theta_1 = 50$, $\theta_2 = 12$
- Mean-squared-error (MSE) is still:

$$\frac{1}{2\,m}\sum_{i=1}^{m}(y_{i,\text{pred}} - y_{i,\text{true}})^2 = \frac{1}{2\,m}\sum_{i=1}^{m}(\boldsymbol{X}.\boldsymbol{\theta} - y_{i,\text{true}})^2$$

```python
nr_measurements=100

abs_noise = True

x1_range = np.linspace(0,10,nr_measurements)
x2_range = np.linspace(-5,5,nr_measurements)

x1, x2 = np.meshgrid(x1_range, x2_range)

m = np.size(x1)

############################################################
y_withoutnoise = 100 + 50*x1 + 12*x2

print("Number of x1 and x2 features and targets y:", np.size(x1), np.size(x2), np.size(y_withoutnoise))
############################################################
if abs_noise:
  y = y_withoutnoise + np.random.uniform(-50,50,x1.shape)
else:
  y = y_withoutnoise*(1+ np.random.uniform(-0.1,0.1,x1.shape))

# Save arrays for plotting but also define as 'unrolled' vectors:
x1_m = x1; x2_m = x2 ; y_m = y

x1 = np.reshape(x1,m)
x2 = np.reshape(x2,m)
y = np.reshape(y,m)

fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111, projection='3d')
ax.view_init(45, 45)
ax.scatter(x1, x2, y, c='r', marker='o')

ax.set_title('Noisy synthetic bi-linear "measurement" data.')
ax.set_xlabel(r'$x^{(1)}$')
ax.set_ylabel(r'$x^{(2)}$')
ax.set_zlabel(r'$y$')

measurement_error = mean_squared_error(y_m,y_withoutnoise)/2
print("(Artificial) measurement error is:", round(measurement_error,2))
```

100 values of *each* feature

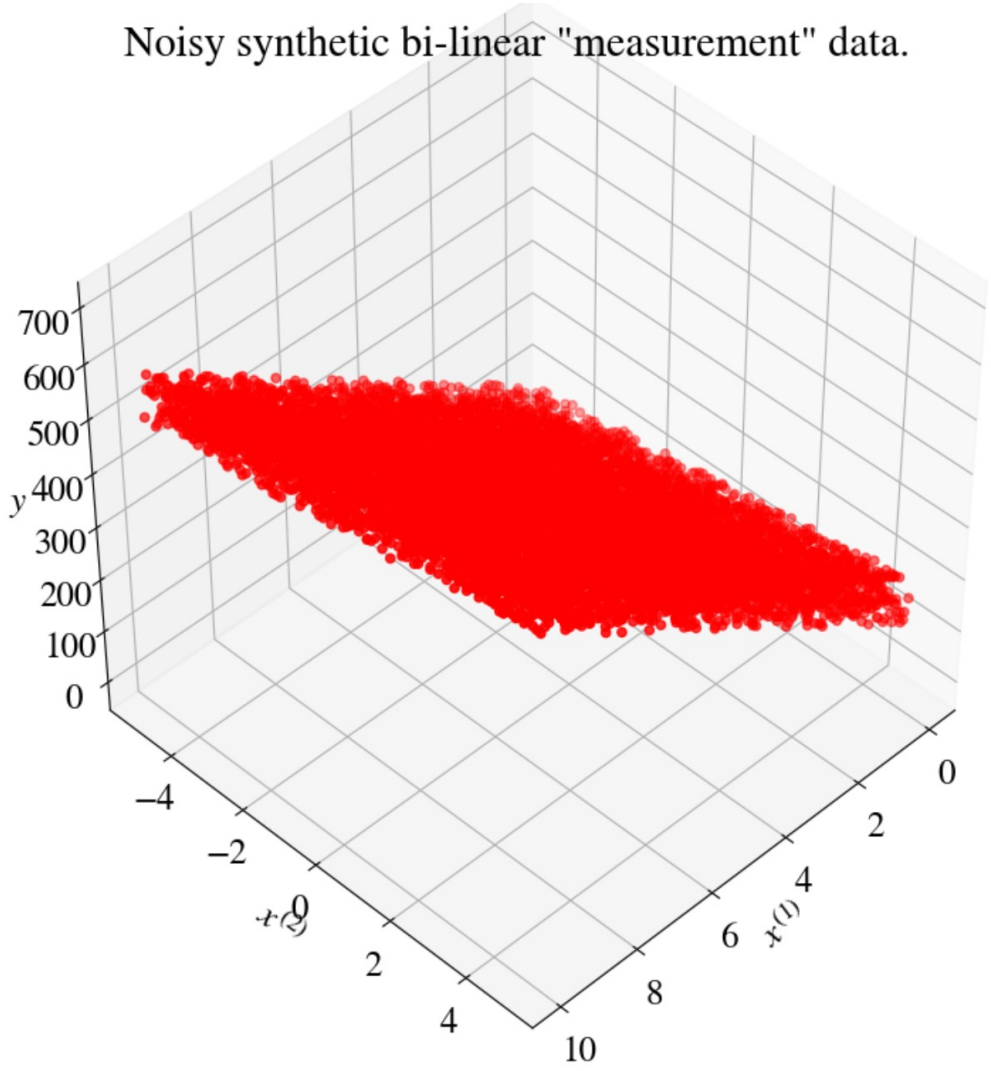
100 × 100 = 10,000 combinations

Example linear function in 2 features

10,000-length feature vectors

Plot of data

- Experimental error:
- MSE = 420.6

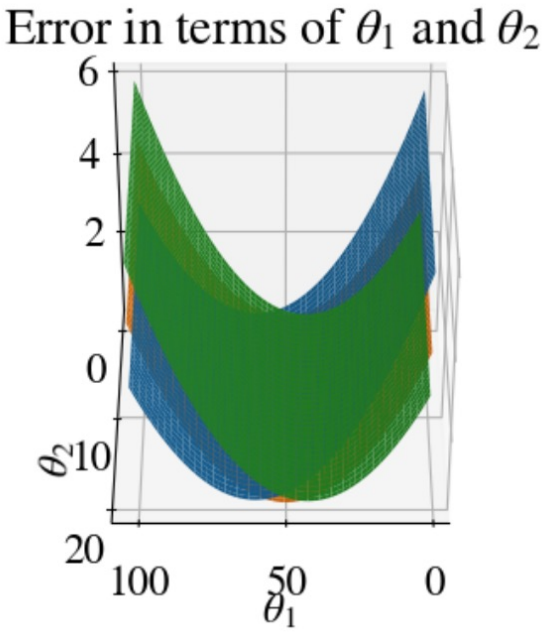


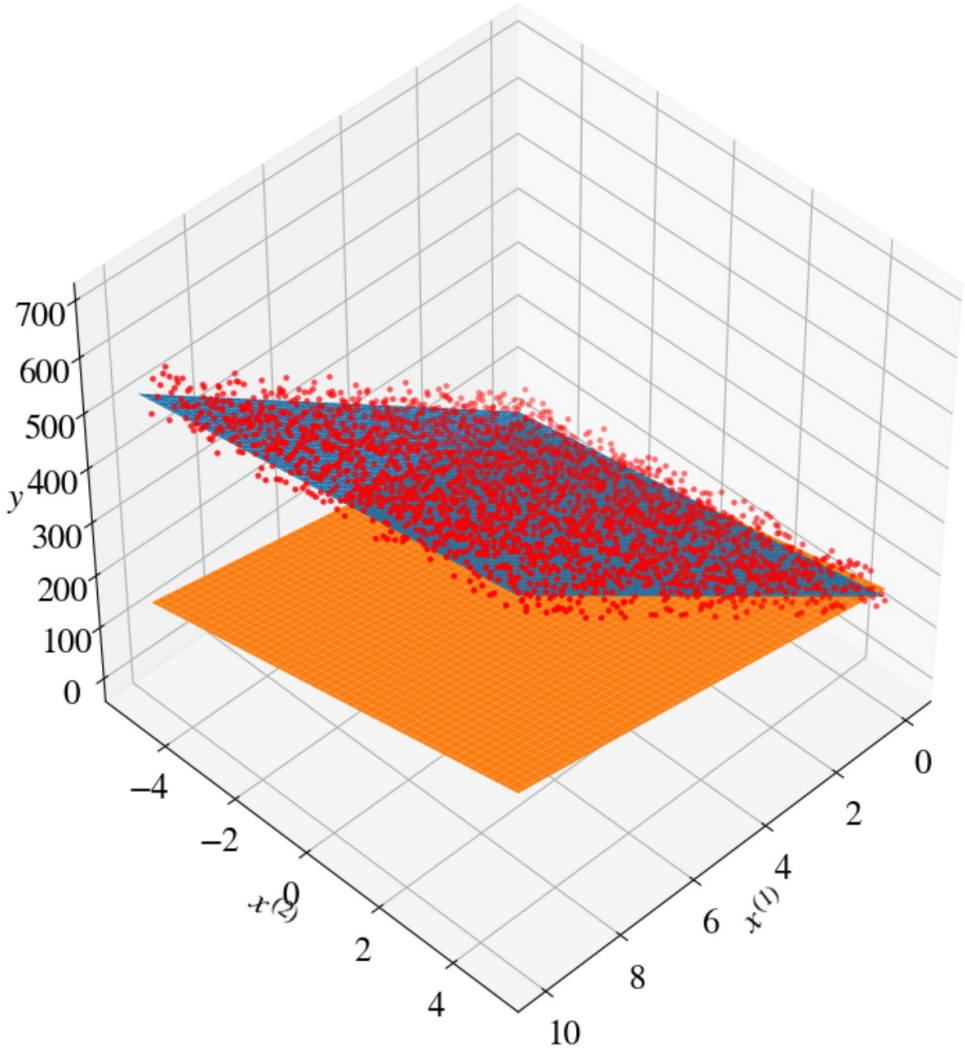
Brute force fitting

- 3 nested loops,
- 100 values each of intercept and 2 slopes = 1,000,000 possible combinations!
- Instead, for visualization, only try 3 intercepts (30,100,150)
- Results:
  - θ_0 = 100, θ_1 = 49.6, θ_2 = 12
  - MSE = 414.3
  - CPU time = **18 s** ☹

```
[5]  1  %%time
     2  tries = 0
     3  for k in range(3):
     4      for j in range(nrfits1):
     5          for i in range(nrfits2):
     6              y_pred = theta0[k] + theta1[j]*x1 + theta2[i]*x2
     7              errors[i,j,k] = mean_squared_error(y_pred , y)/2
     8              tries = tries+1
```

Visualization of fits





Gradient descent

- $\theta_0^{\text{new}} = \theta_0 - \alpha \frac{\partial J}{\partial \theta_0} = \theta_0 - \frac{\alpha}{m} \sum_{i=1}^{m} (\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \ldots + \theta_n x_n^{(i)} - y^{(i)}) \times x_0^{(i)}$,
- $\theta_1^{\text{new}} = \theta_1 - \alpha \frac{\partial J}{\partial \theta_1} = \theta_1 - \frac{\alpha}{m} \sum_{i=1}^{m} (\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \ldots + \theta_n x_n^{(i)} - y^{(i)}) \times x_1^{(1)}$.
- $\ddots$
- $\theta_n^{\text{new}} = \theta_n - \alpha \frac{\partial J}{\partial \theta_n} = \theta_n - \frac{\alpha}{m} \sum_{i=1}^{m} (\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \ldots + \theta_n x_n^{(i)} - y^{(i)}) \times x_n^{(1)}$.

- Or vectorized, in terms of X with bias (x_0=1) column:

$$\vec{\theta}^{\text{new}} = \vec{\theta} - \frac{\alpha}{m}((\vec{\tilde{y}} - \vec{y}) \cdot X)$$

- Note: the definition of $J(\theta)$ = MSE/2 is such that factor 2 cancels out in error gradient.

Python code for multivariate gradient descent

```
%%time
nrsteps = 10000

thetas = np.zeros([nrsteps,3])
errorsGD = np.zeros(nrsteps)

alpha = 0.05
alpha_m = alpha/m

x_aug = np.zeros([m,3])
x_aug[:,0] = 1
x_aug[:,1] = x1
x_aug[:,2] = x2

# Initial guess:
thetas[0,:] = [25,10,2]

i=0
converged = False
tolerance = 1e-4
while not converged:

    hypothesis = x_aug.dot(thetas[i,:])

    errorsGD[i] = mean_squared_error(hypothesis,y) /2

    thetas[i+1,:] = thetas[i,:] -  alpha_m * (hypothesis - y).dot(x_aug)

    if np.abs(errorsGD[i]-errorsGD[i-1]) < tolerance:
      converged = True
      print("Converged in", i, "iterations with thetas: ", round(thetas[i+1,0],2),round(thetas[i+1,1],2),round(thetas[i+1,2],2), "and error "
    i+=1
```

Bias and features

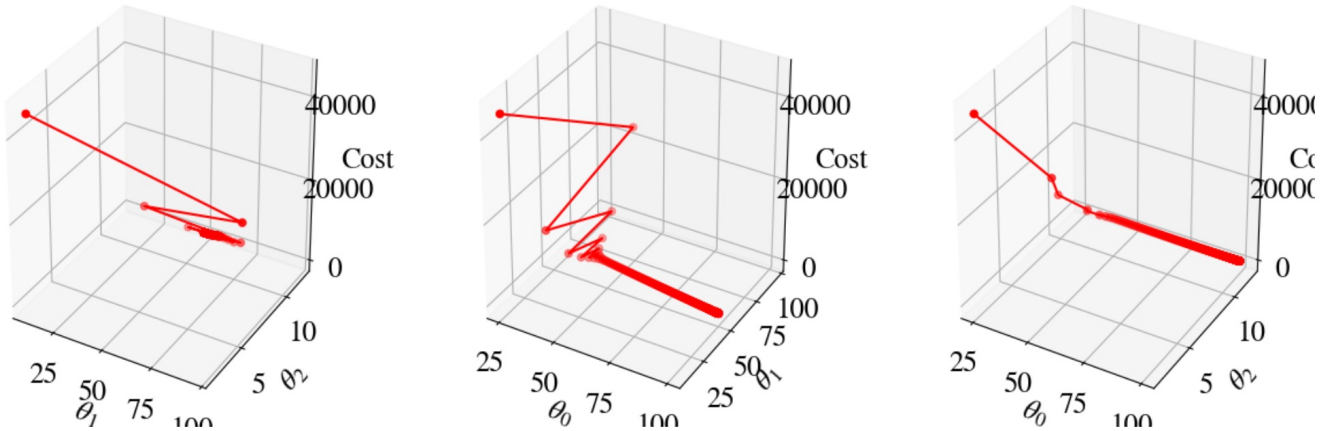
While-loop until tolerance

vectorized

Identical to univariate gradient descent!

How did we do?

- 477 iterations (trials for combinations of θ_0, θ_1, θ_2),
- θ_0 = 99.9, θ_1 = 50.02, θ_2 = 11.95,
- MSE = 411.8
- CPU time: **206 ms** ☺ (100x faster)



Accelerated gradient descent

- **3** iterations (trials for combinations of θ_0, θ_1, θ_2),
- θ_0 = 100.1, θ_1 = 49.99, θ_2 = 11.95,
- MSE = 411.8
- CPU time: **36.4 ms** ☺, is 360× faster!

- Unlike brute-force nested loops, nr of iterations of GD does not increase much for additional features.

Non-Linear Multivariate Regressions

Problem statement

- First, consider non-linear regression in 1 variable (univariate):
- **Hypothesis**: $y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \ldots + \theta_n x^n$

- How do we use ML numerical regression to fit this?

- Embarrassingly simple: just define each term as separate feature, and then do ***same*** multivariate linear regression!
- Features: $x_0 = 1$, $x_1 = x$, $x_2 = x^2$, …, $x_n = x^n$.
- Think of spreadsheet with *n* feature columns. One target vector.

How far can we push this?

We can also try models like:

$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 \sin(x) + \theta_4 \log(x) + \theta_3 \exp(x/2) \ldots + \theta_n x^n$

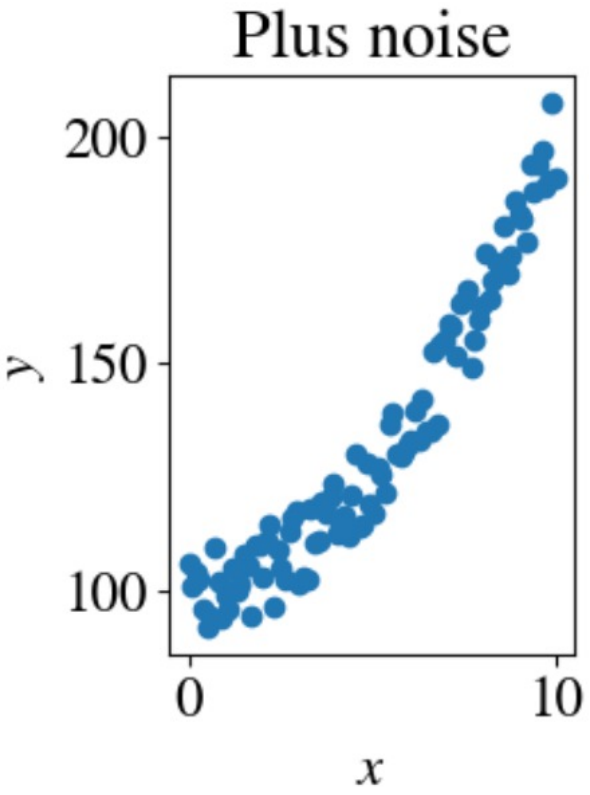
as long as...: the relation is *linear in fitting parameters* θ, such that our loss/cost function $J(\theta)$ is quadratic in θ and loss gradient is linear in θ.

So, what's an example of a model that does *not* work with this approach?

Answer: models like $y = \cos(\theta_1 x)$, $y = \exp(x/\theta_1)$, $y = (\theta_1 x)^2$ etc.

Simple example: quadratic function in one variable/feature

```python
nr_measurements=100

x = np.linspace(0,10,nr_measurements)
y = x**2

plt.subplot(1,2,1)
plt.title('Quadratic function')
plt.scatter(x,y)
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')

# Now add some noise:
y = 100 + y + np.random.uniform(-10,10,x.shape)

plt.subplot(1,2,2)
plt.title('Plus noise')
plt.scatter(x,y)
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')

measurement_error = mean_squared_error(100+x**2 , y)/2
print("Measurement error is", round(measurement_error,2))
plt.tight_layout()
```




```
Measurement error is 17.41
```

Fit with *linear* regression from scikit-learn

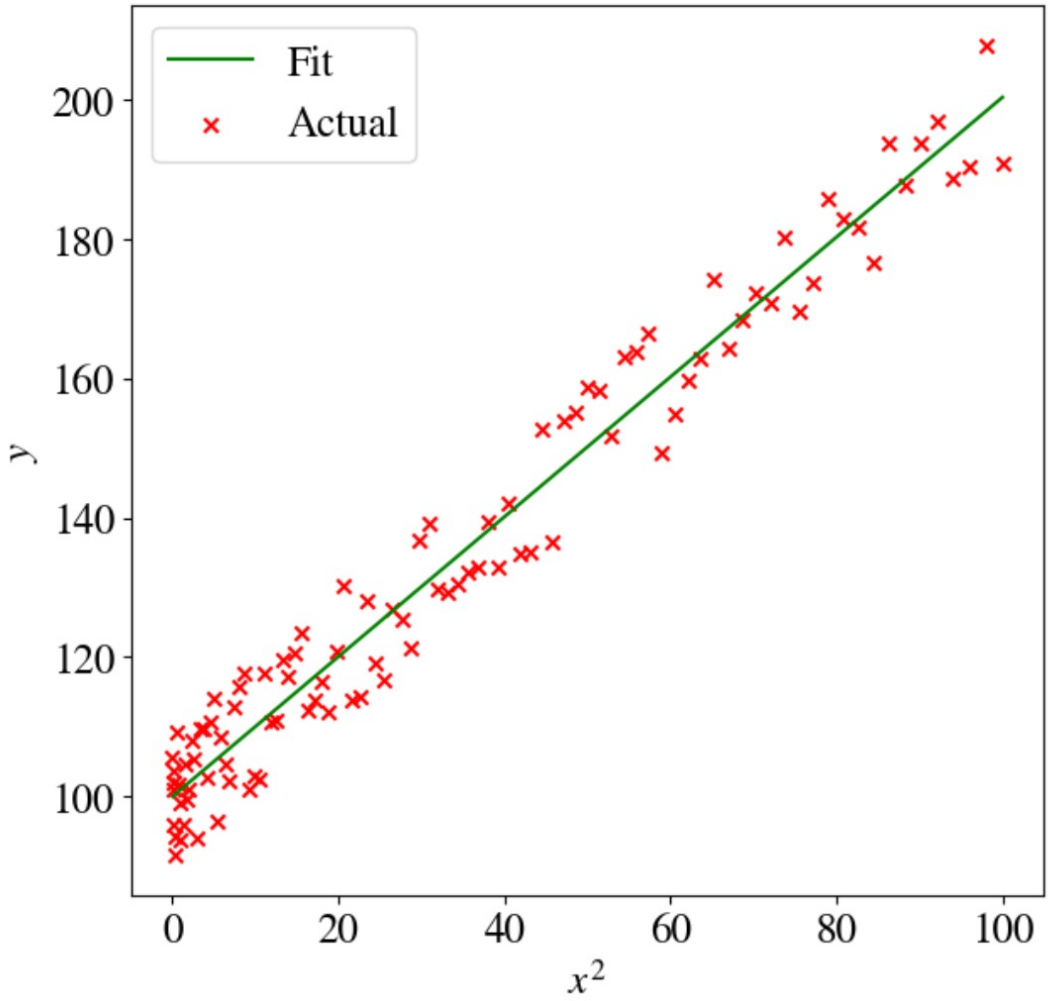
- Define feature $x_1 = x^2$ and perform *linear* fit

```
1  x1 = x**2
2
3  sq_linear_regression_model = LinearRegression().fit(x1, y)
4  y_pred = sq_linear_regression_model.predict(x1)
5
```

```
Regression error over entire data set is 17.39
```

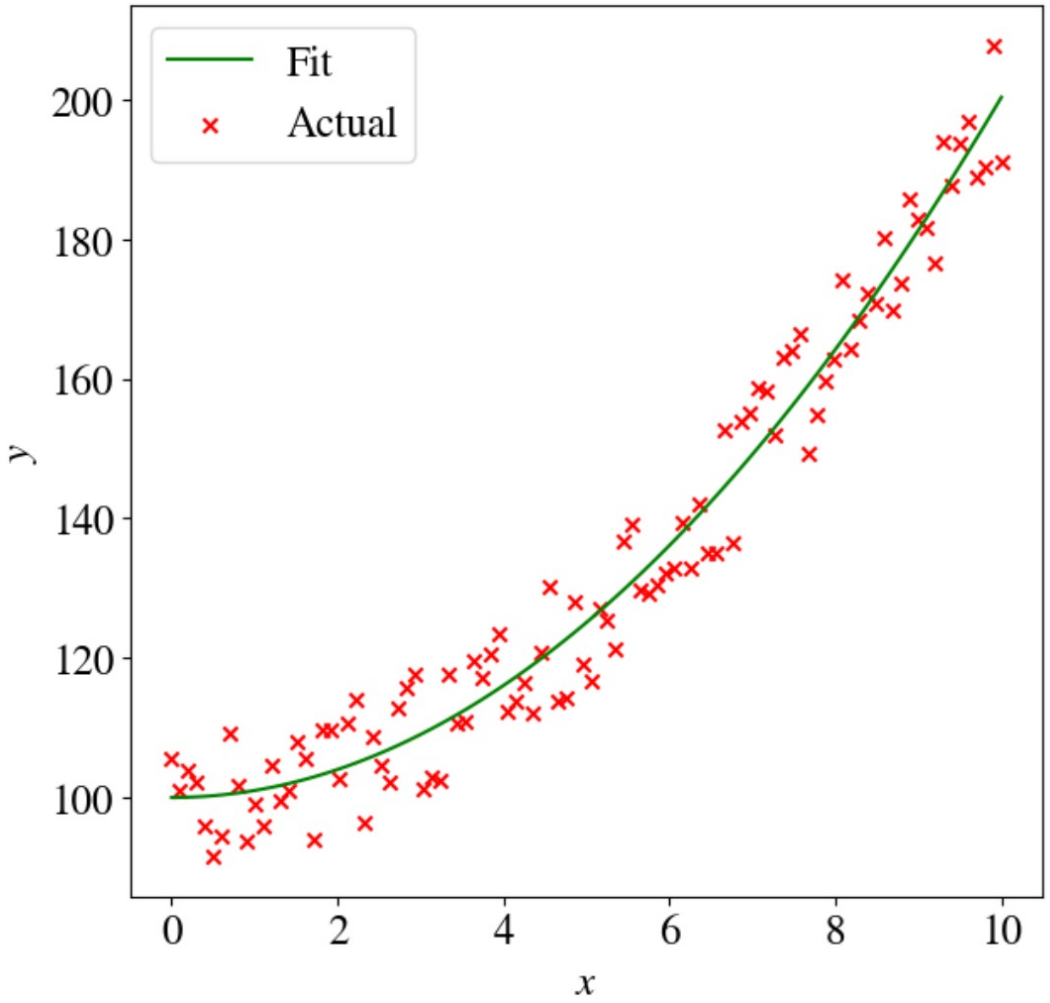
Result

- Plotted as y vs feature x^2:



Result

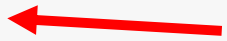
- Plotted as *y* vs feature *x*
- Optimized/best fit finds:
- $\theta_0 = 100$, $\theta_1 = 1$
- MSE = 17.4



Higher-order polynomial

- Often we don't know what order to fit, so let's try full 3rd order polynomial: $y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$

```
xx = np.zeros([len(x),3])
xx[:,0] = x[:,0]
xx[:,1] = x[:,0]**2
xx[:,2] = x[:,0]**3

# Note that 'poly_regression_model' etc are names I give my models
poly_regression_model = LinearRegression().fit(xx, y)
y_pred = poly_regression_model.predict(xx)


theta0 = poly_regression_model.intercept_[0] # the result is given as an array; [0] is the index of the
theta1 = poly_regression_model.coef_[0][0] # similar to above, the [0][0] only takes the first (and only
theta2 = poly_regression_model.coef_[0][1]
theta3 = poly_regression_model.coef_[0][2]

y_poly = theta0 + theta1*x + theta2*x**2 + + theta3*x**3
```

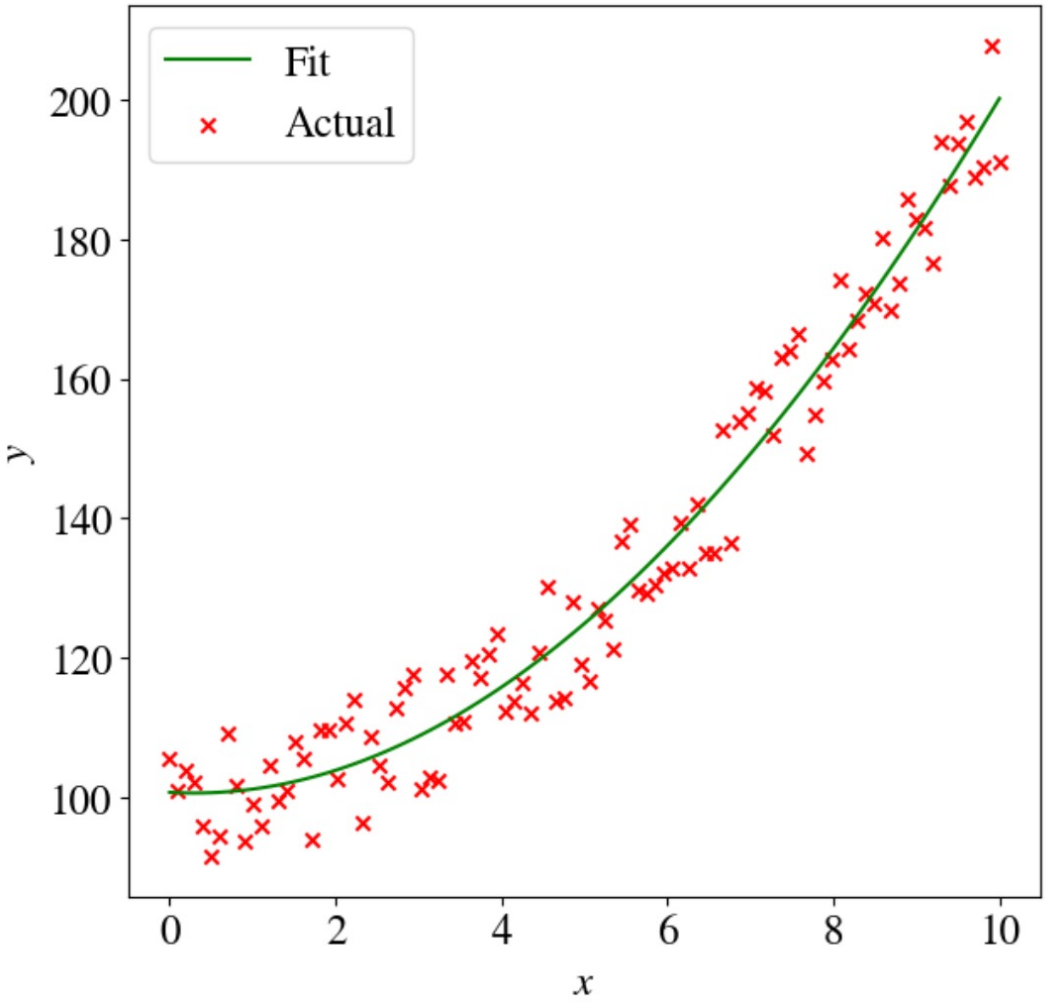


Note, LinearRegression does not need bias

Results

- $\boldsymbol{\theta_0 = 100.7}$, θ_1 = -0.68,
- $\boldsymbol{\theta_2 = 1.15}$, θ_2 = -0.01,
- MSE = 17.4
- CPU = 0.6 s

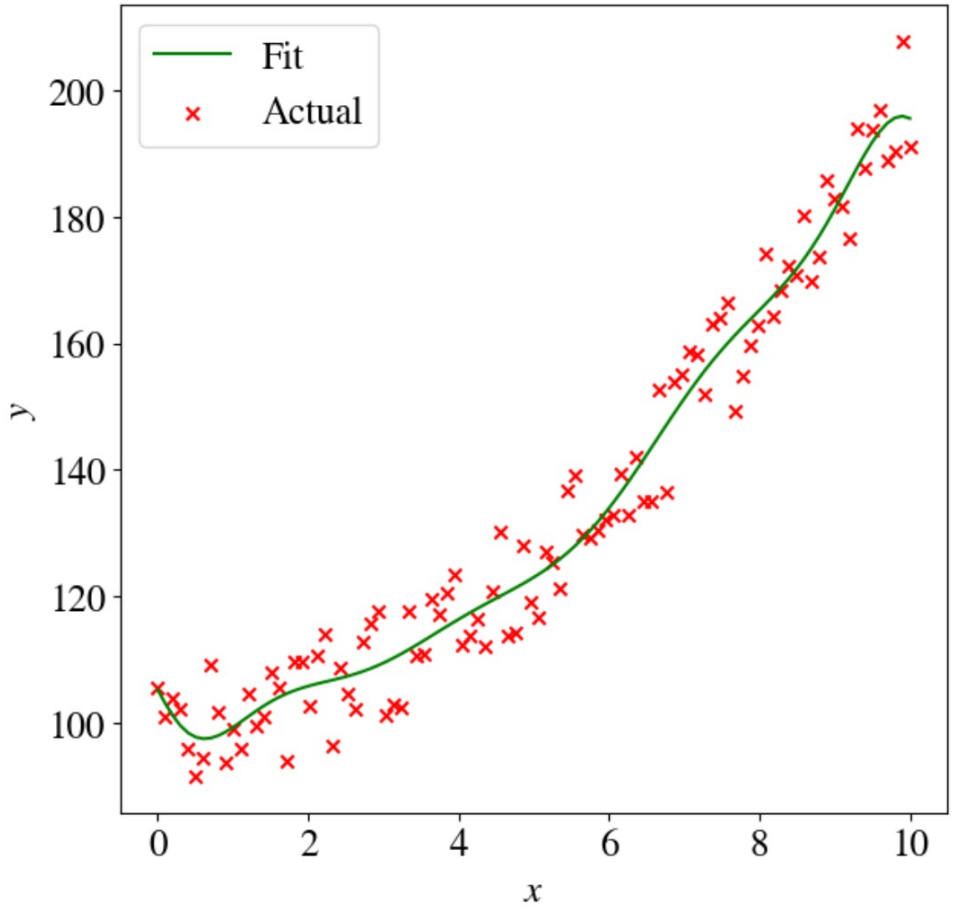
- Basically same results, but without knowing *a priori*



What about 12^{th} order polynomial!?

```
poly_order = 12
features = np.zeros([len(x),poly_order])

for i in range(poly_order):
    features[:,i] = x[:,0]**(i+1)

poly_regression_model = LinearRegression().fit(features, y)
y_pred = poly_regression_model.predict(features)

thetas = np.zeros(poly_order+1)
thetas[0] = poly_regression_model.intercept_[0]
thetas[1:] = poly_regression_model.coef_[0]
thetas_sq = thetas
print("Polynomial fitting parameters are:", thetas)

y_poly = thetas[0] + features.dot(thetas[1:])

```

```
Polynomial fitting parameters are: [ 1.05226198e+02 -2.18009092e+01 -1.73897461e+00  5.50421046e+01
 -6.23101537e+01  3.36953612e+01 -1.06709673e+01  2.12089282e+00
 -2.69848487e-01  2.16561280e-02 -1.03558550e-03  2.57185467e-05
 -2.26234320e-07]
Regression error for 12 order polynomial is 15.92
```



'Better' fit with MSE = 15.9, but risk of overfitting! (more on that later)

Power of polynomial fitting

- *Any* function can be approximated by polynomial (Taylor expansion)

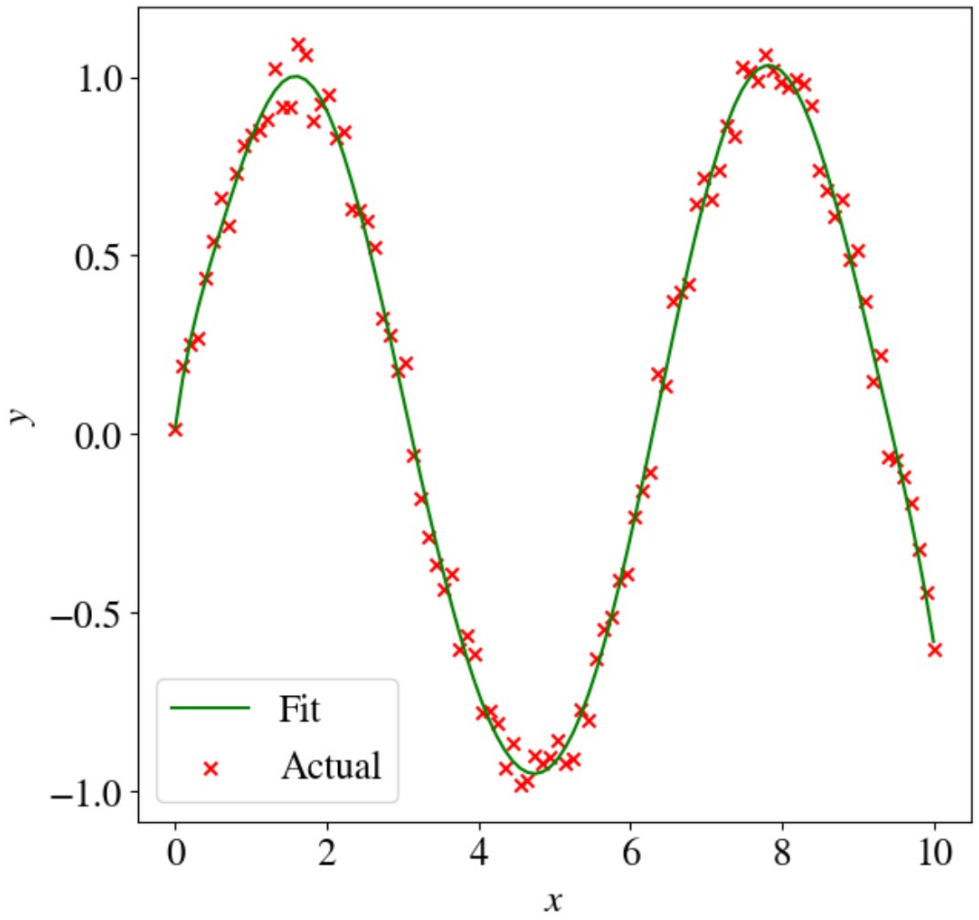
- Demonstration: fit $y = \sin(x)$ with 12^{th}-order polynomial

Power of polynomial fitting

```
y_sin = np.sin(x) + np.random.uniform(-0.1,0.1,x.shape)

poly_order = 12
features = np.zeros([len(x),poly_order])

for i in range(poly_order):
    features[:,i] = x[:,0]**(i+1)

poly_regression_model = LinearRegression().fit(features, y_sin)
y_pred = poly_regression_model.predict(features)

thetas = np.zeros(poly_order+1)
thetas[0] = poly_regression_model.intercept_[0]
thetas[1:] = poly_regression_model.coef_[0]
```

Results

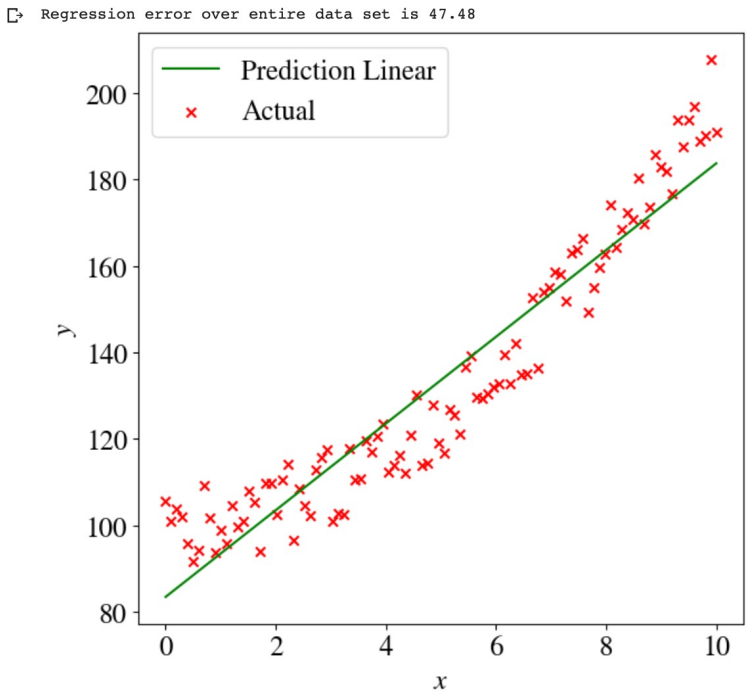
```
Polynomial fitting parameters are: [ 1.65368246e-02  1.58637449e+00 -2.35995737e+00  3.36511323e+00
 -2.75373108e+00  1.28140061e+00 -3.73374338e-01  7.18367784e-02
 -9.27062462e-03  7.92962791e-04 -4.30853691e-05  1.34699793e-06
 -1.84882074e-08]
Regression error for 12  order polynomial is 0.003
```



Underfitting and Overfitting

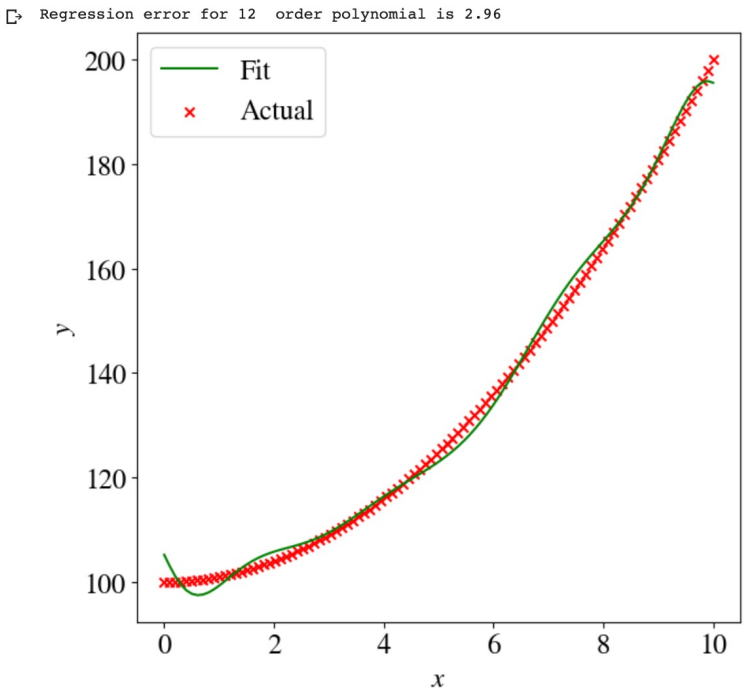
Underfitting

- Model (hypothesis) not complex enough for data
- E.g., linear model to fit quadratic data



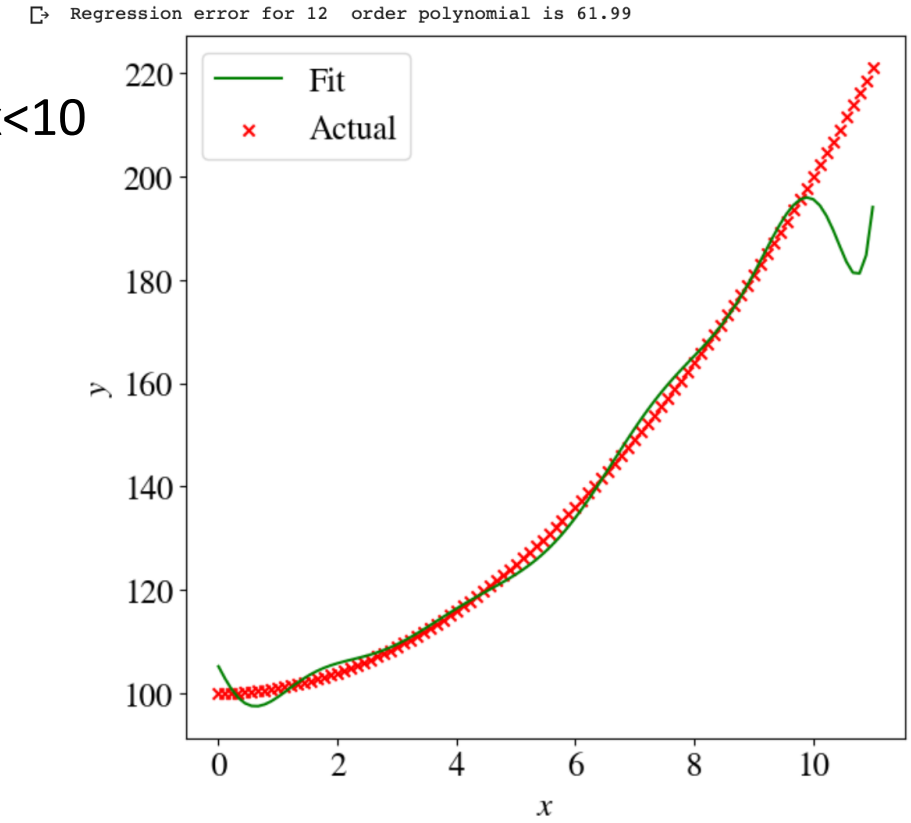
Overfitting

- *Too* complex for (amount of) training data
- How to know?
- Compare MSE for training data vs. MSE for independent validation data



Example

- 12^{th} order polynomial fitted to 0<x<10
- Make predictions of 0<*x*<11
- MSE increases from 3 to 62!

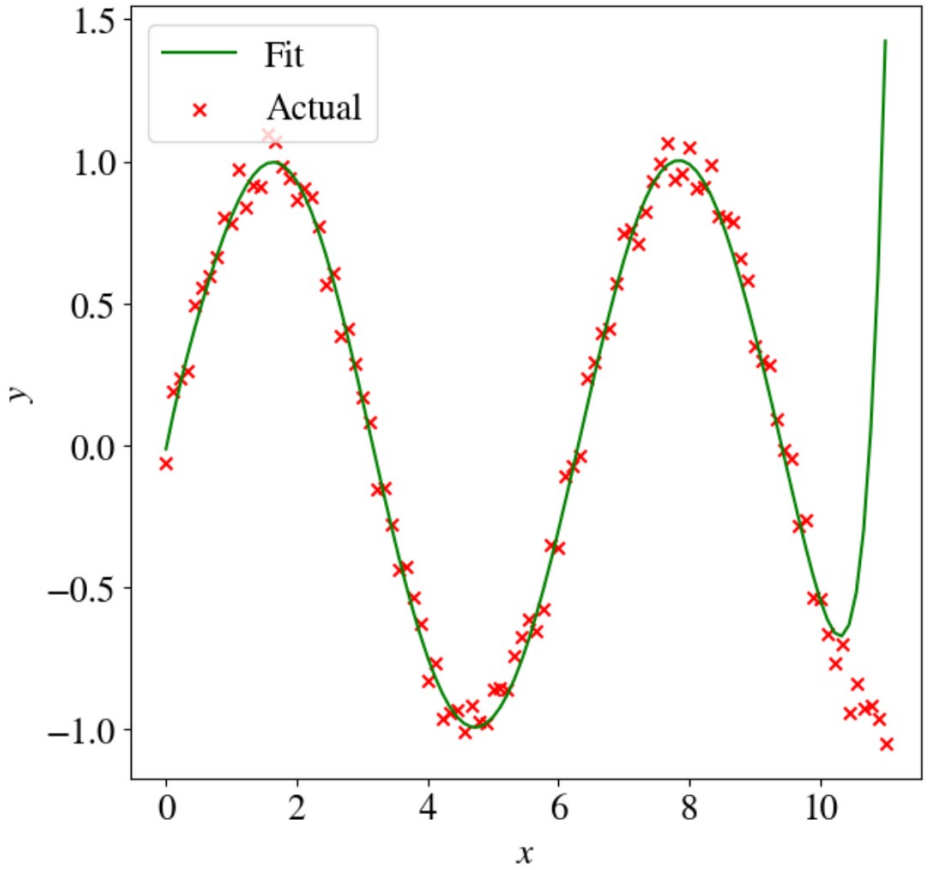


Another example

- 12^{th} order polynomial to fit sine
- Fit between $0<x<10$,
- Predict for $0<x<11$

- Overfitting can be reduced by *regularization* (other lecture)

- But, ***extrapolation*** outside of range of training data is often problematic



Multivariate non-linear regression

Multivariate non-linear regression

- Simple example: 4^{th} order polynomial in 2 (initial) features
- Say temperature T and pressure p
- $y = \theta_0 + \theta_1 T + \theta_2 p + \theta_3 T^2 + \theta_4 p^2 + \theta_5 T p + \theta_6 T^3 + \theta_7 p^3 + \theta_8 T p^2 + \theta_9 T^2 p + \theta_{10} p^4 + \theta_{11} T^4 + \theta_{12} T^2 p^2 + \theta_{13} T^3 p + \theta_{14} T p^3$
- Need to define 14 features
- Cross-product terms add up!

- Gradient descent can handle many features, but
- Mini-batch GD quickly becomes attractive for higher-order polynomial in multiple (physical) features

Summary

- Non-linear multivariate numerical regression ML algorithms are ***identical*** to univariate linear regression,
- Just need to define (many) new features,
- Polynomial regression can approximate *any* function,
- Beware of under- and overfitting.

Lecture Notes & Lab(s)

- Lecture notes:
  - ✓Week_3_Lecture_Part1_Multivariate_Linear.ipynb
  - ✓Week_3_Lecture_Part2_Multivariate_Nonlinear.ipynb

- Labs:
  - ✓Week_3_Lab_Part1_Multivariate_Linear.ipynb
  - ✓Week_3_Lab_Part2_Multivariate_Non_Linear.ipynb

- Maybe start with lab 2 in class (bit harder) and do lab 1 (easier) at home, or lab 1 today and lab 2 on Thursday in class.

Next Week

- **Logistic regression** (univariate and multivariate, linear and non-linear)

- These are **classification** schemes of a target variable that can only have discrete categories (e.g., satellite image pixel is land/water/grass).