Overview of Machine Learning (II) Regression and Gradient Descent

Concepts, Supervised Learning, Unsupervised Learning, Regression, Classification

The Essential Elements of (most) ML

To use most ML methods, we will need to conceptualize our problem into the following:

Features (inputs, descriptors)

$$\{\boldsymbol{x}_i\} \leftrightarrow \boldsymbol{X}$$

$\boldsymbol{x}_i$ *feature vector of sample i*

- a numerical description of (ideally) characteristics that distinguish one sample from another
- may (or may not) have direct implications on the modeling outputs

Labels (outputs)

$$\{y_i\} \leftrightarrow \boldsymbol{y}; \{\boldsymbol{y}_i\} \leftrightarrow \boldsymbol{Y}$$

y_i or $\boldsymbol{y}_i$ *scalar or vector label of sample i*

- also a numerical (integer or real) description of sample i
- usually reserved for some special quantity or property of interest

Labeled Data

$$\{(\boldsymbol{x}, y)_i\}$$

a set of tuples where features and labels are known

Unlabeled Data

$$\{\boldsymbol{x}_i\}$$

labels are not necessarily known or provided with features

Model

a function that operates on features

$$f(\boldsymbol{x}) \text{ or } \boldsymbol{f}(\boldsymbol{x})$$

- *often defines a mapping from feature space to label space*

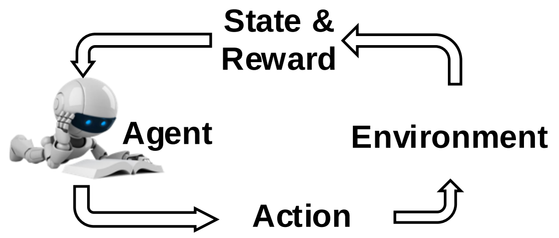
Predictions

the function output or predicted labels

$$\hat{y} = f(\boldsymbol{x}) \text{ or } \hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x})$$

Classes of Machine Learning

Machine learning is deployed in three main modes:



Supervised Learning

- In ***supervised learning***, we aim to create a model that can predict ***y*** as a function of ***x***.
- The optimization/learning of our model is ***supervised*** because the algorithm will exploit knowledge of labels over the dataset

Supervised learning can be used for either

- ***Regression*** –predict a *continuous* label. This is likely to be true for QSPR problems in physical science.
  e.g., conductivity, melting point, band gaps
- ***Classification*** – predict *categorical* labels or class membership. This can be useful for characterizing discrete outcomes
  e.g., (in)soluble, (un)sythesizable, (in)activity, hazardous

Unsupervised Learning

- In ***unsupervised learning***, we aim to create a model that identifies patterns in ***x***.
- The optimization/learning of our model is ***unsupervised*** because the algorithm will not exploit knowledge of labels over the dataset

Unsupervised learning is usually used for

- ***Clustering*** –partition features into a set of different classes/groups, which is the *y*.
  e.g., chemical classes
- ***Signal processing*** – Uncover the underlying signal within a set of features. This is often a part of representation learning.
  e.g., protein folding pathways
- ***Generating*** **–** create a model distribution over **x** such that we can generate new samples

Reinforcement Learning

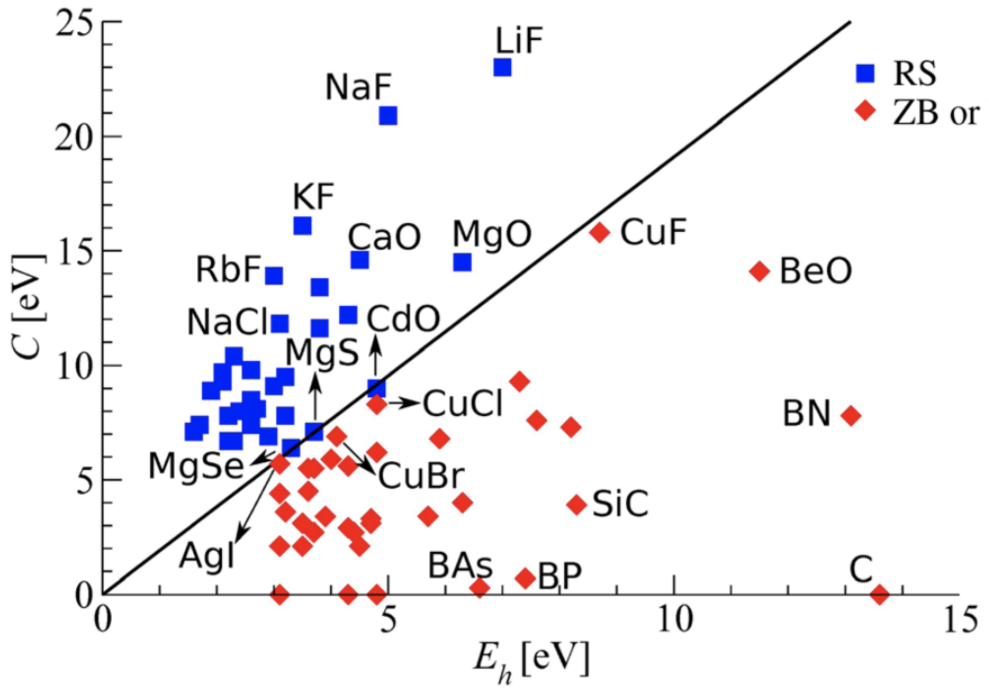
- In reinforcement learning, an "agent" learns how to interact with its environment based on feedback via cumulative rewards/penalties
- Many things that people think are reinforcement learning are probably not reinforcement learning
- Usually about planning and scheduling

e.g., automated process synthesis, process control

- In ***semi-supervised learning***, we want a model that can predict ***y*** as a function of ***x,*** just as in supervised learning
- ***Both labeled and unlabeled data*** are used in modes like *co-training, pseudo-labeling, and label propagation*

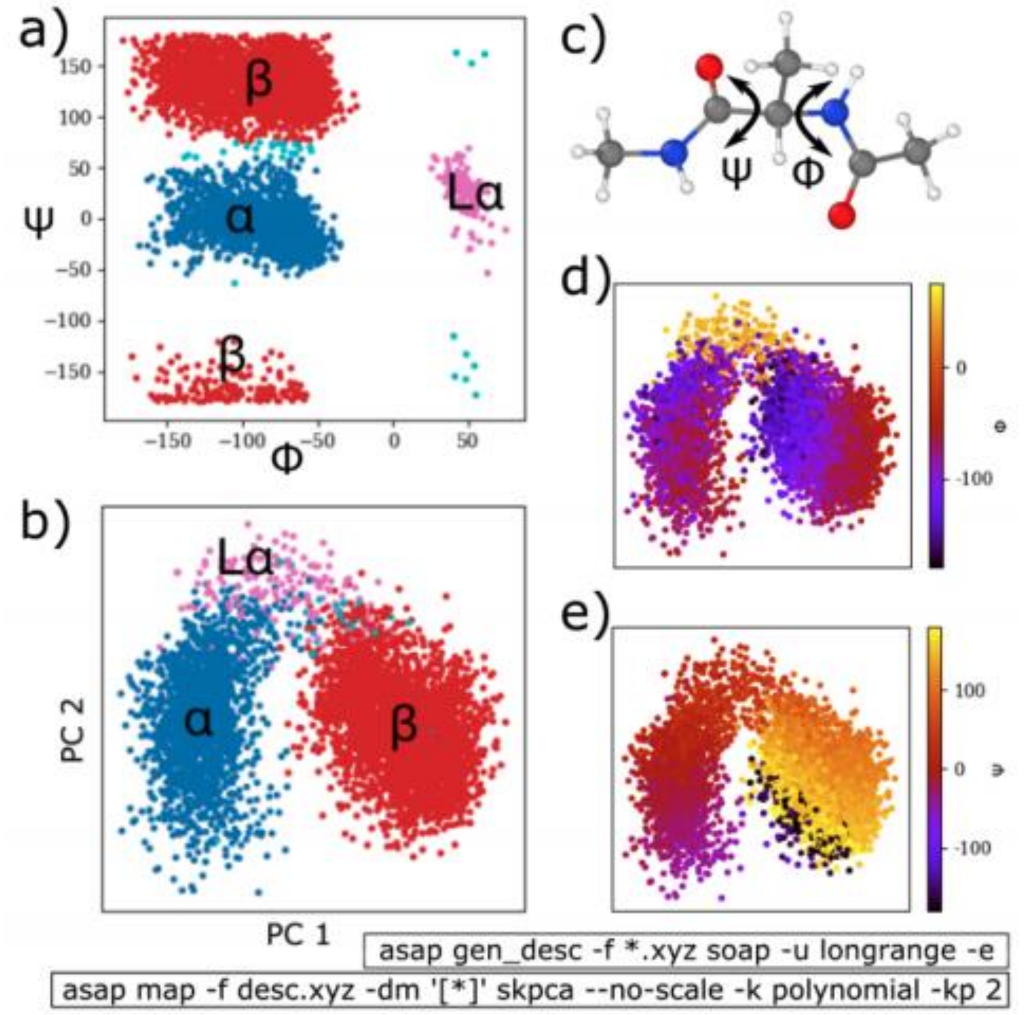
- In ***self-supervised learning***, we eventually want a model that can predict ***y*** as a function of ***x,***
- ***Only unlabeled data*** are used during training; one form is *contrastive learning*

Example: supervised & unsupervised Learning



Ghiringhelli et al. *PRL* 114 (**2015**)

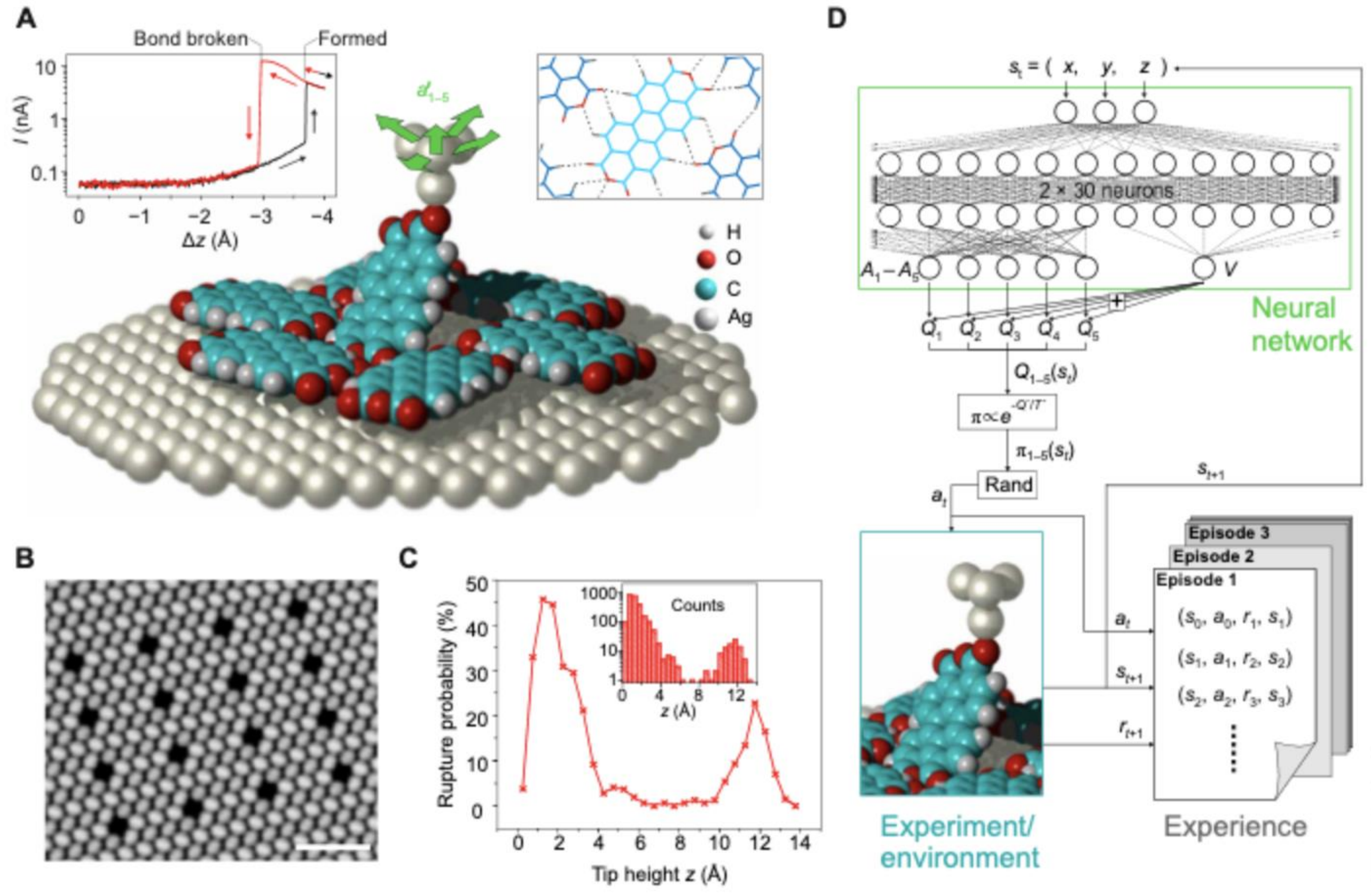
Two simple "descriptors" (related to nearest neighbor distance and dielectric constant) define a function that serves as a decision boundary that distinguishes between rocksalt and zinc blend or wurtzite crystal structures.



```
asap gen_desc -f *.xyz soap -u longrange -e
asap map -f desc.xyz -dm '[*]' skpca --no-scale -k polynomial -kp 2
```

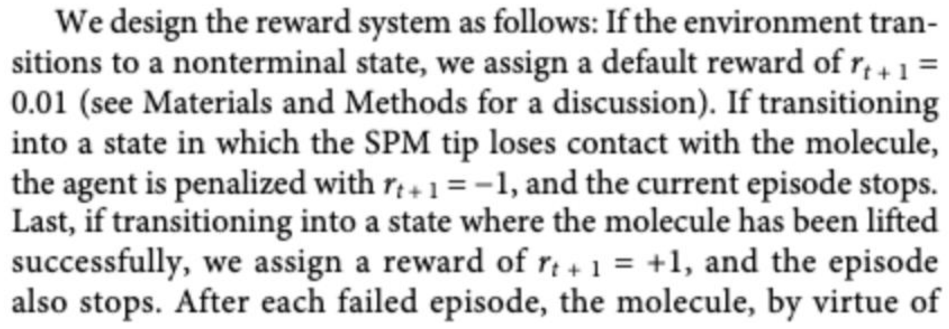
This illustrates a typical Ramachandran plot of alanine dipeptide by comparison to a unsupervised learning over molecular configurations.

Example: Reinforcement Learning



Guiding nanofabrication with single-molecule manipulation

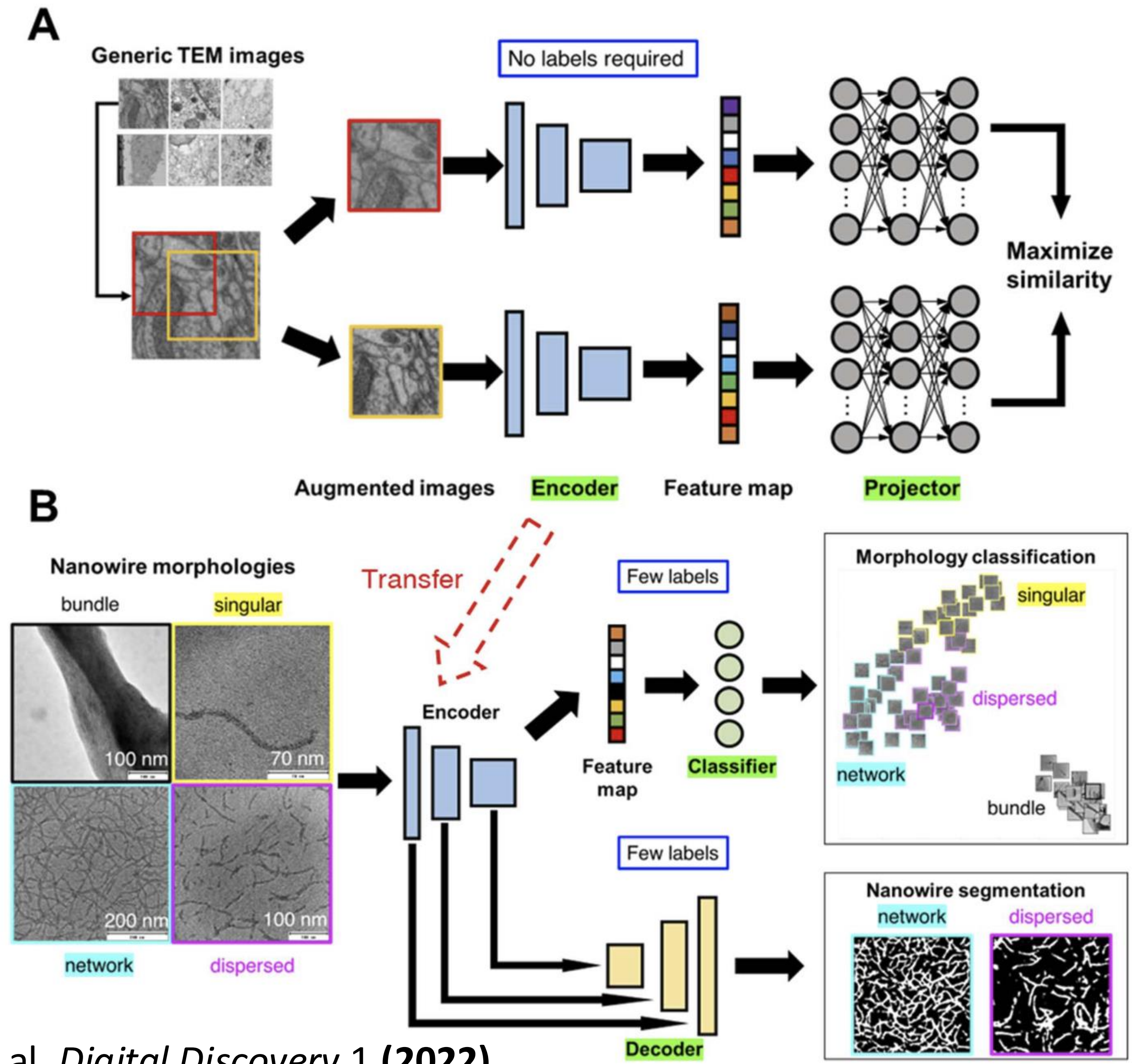
- Scanning probe microscope can remove molecules from supramolecular assembly, but apparently this is non-trivial manual task
- Reinforcement learning is used to develop a protocol to move the tip in a manner that enables effective molecule lifting



We design the reward system as follows: If the environment transitions to a nonterminal state, we assign a default reward of $r_{t+1} = 0.01$ (see Materials and Methods for a discussion). If transitioning into a state in which the SPM tip loses contact with the molecule, the agent is penalized with $r_{t+1} = -1$, and the current episode stops. Last, if transitioning into a state where the molecule has been lifted successfully, we assign a reward of $r_{t+1} = +1$, and the episode also stops. After each failed episode, the molecule, by virtue of

einen et al. *Science Advances 6* **(2020)**

Example: Self- + Semi- Supervised Learning



Lu et al. *Digital Discovery* 1 **(2022)**

“Fancy” ML workflow for microscopy segmentation & classification

- Combines many “advanced” architecture concepts with semi-supervised approach in a “transfer learning” paradigm.
- Self-supervised learning component comes from matching an image to itself! (they must come from the same class… probably?)
- Overall goal is efficient labeling of TEM/data efficiency

Regression
Gradient Descent

The basic problem of curve-fitting

Machine learning Regression is often characterized as “fancy curve-fitting”; to understand the (un)fairness of that statement, we will first describe good ole regular curve-fitting

Linear Least-Squares Regression

$$f(x) = \theta_0 + \theta_1 x$$

Objective:

$$\min_{\boldsymbol{\theta}} \mathcal{E}(f) = \min_{\boldsymbol{\theta}} \sum_{k=1}^{n} |e_k|^2$$

$$= \min_{\boldsymbol{\theta}} \sum_{k=1}^{n} (\theta_0 + \theta_1 x - y_k)^2$$

Given $\{(\boldsymbol{x}_i, y_i)\}$ produce “optimal” f $\hat{y} = f(\boldsymbol{x}, \boldsymbol{\theta})$

that minimizes some error metric (“loss”) $\mathcal{E}(\{y_k, \hat{y}_k\})$

$$e_k = \hat{y}_k - y_k$$

Some possible loss functions

$$\mathcal{E}_\infty(f) = \max_k |e_k|$$

$$\mathcal{E}_2(f) = \sqrt{\frac{1}{n}\sum_{k=1}^{n} |e_k|^2}$$

$$\mathcal{E}_1(f) = \sqrt{\frac{1}{n}\sum_{k=1}^{n} |e_k|}$$

$$\mathcal{E}_p(f) = \sqrt[p]{\frac{1}{n}\sum_{k=1}^{n} |e_k|^p}$$

note that the “optimal” f depends on the loss function

The basic problem of curve-fitting

Machine learning Regression is often characterized as “fancy curve-fitting”; to understand the (un)fairness of that statement, we will first describe good ole regular curve-fitting

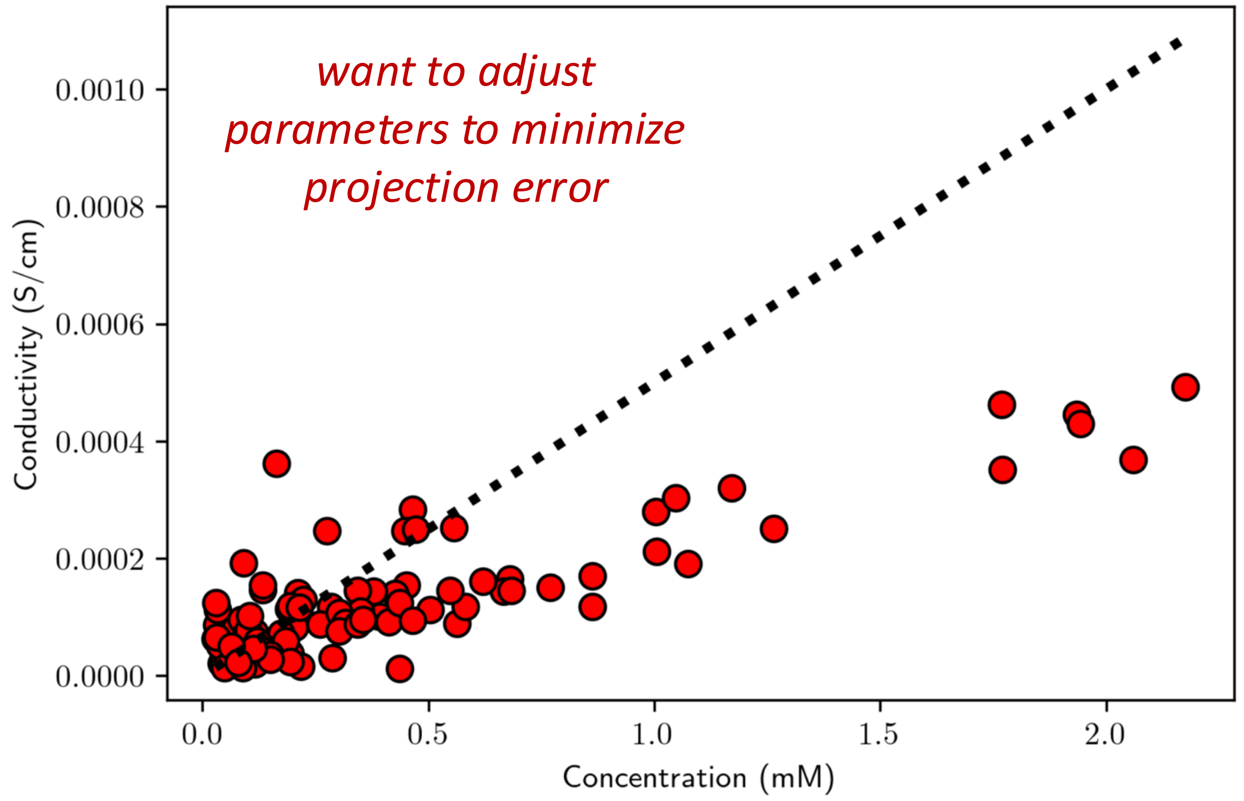
Linear Least-Squares Regression

$$f(x) = \theta_0 + \theta_1 x$$

Objective:

$$\min_{\boldsymbol{\theta}} \mathcal{E}(f) = \min_{\boldsymbol{\theta}} \sum_{k=1}^{n} |e_k|^2$$

$$= \min_{\boldsymbol{\theta}} \sum_{k=1}^{n} (\theta_0 + \theta_1 x_k - y_k)^2$$



We will explore using gradient descent for this problem, but it can be approached easily/exactly. How?

The basic problem of curve-fitting

Machine learning Regression is often characterized as “fancy curve-fitting”; to understand the (un)fairness of that statement, we will first describe good ole regular curve-fitting

Linear Least-Squares Regression

$$f(x) = \theta_0 + \theta_1 x$$

Objective:

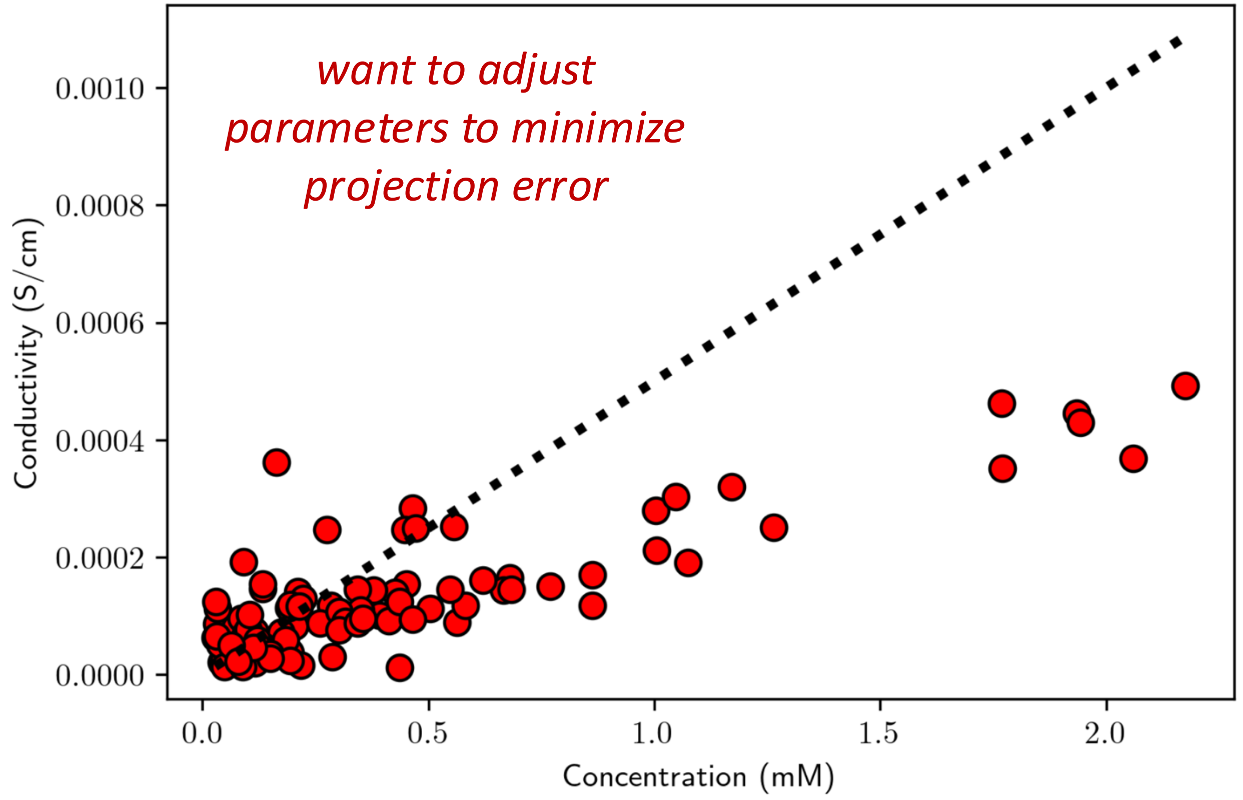
$$\min_{\boldsymbol{\theta}} \mathcal{E}(f) = \min_{\boldsymbol{\theta}} \sum_{k=1}^{n} |e_k|^2$$

set derivatives to zero and solve

$$= \min_{\boldsymbol{\theta}} \sum_{k=1}^{n} (\theta_0 + \theta_1 x_k - y_k)^2$$

$$\begin{pmatrix} n & \sum_{k=1}^{n} x_k \\ \sum_{k=1}^{n} x_k & \sum_{k=1}^{n} x_k^2 \end{pmatrix} \begin{pmatrix} \theta_0 \\ \theta_1 \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{n} y_k \\ \sum_{k=1}^{n} x_k y_k \end{pmatrix}$$

this only possible if our function is linear in all its parameters



Non-linear Regression

Machine learning Regression is often characterized as "fancy curve-fitting"; to understand the (un)fairness of that statement, we will first describe good ole regular curve-fitting

Non-linear regression

$$f(x, \boldsymbol{\theta})$$

now just some general function, which is not necessarily linear in its parameters

e.g.,

$$f(x, \boldsymbol{\theta}) = \theta_0 \cos(\theta_1 x + \theta_2) + \theta_3$$

If we consider a loss related to l2 - norm, then

$$\mathcal{E}(\boldsymbol{\theta}) = \sum_{k=1}^{n} (f(x_k, \boldsymbol{\theta}) - y_k)^2; \quad \frac{\partial \mathcal{E}}{\partial \theta_i} = 0 \;\forall\; i$$

$$\implies \sum_{k=1}^{n} (f(x_k, \boldsymbol{\theta}) - y_k) \frac{\partial f}{\partial \theta_i} = 0 \;\forall\; i$$

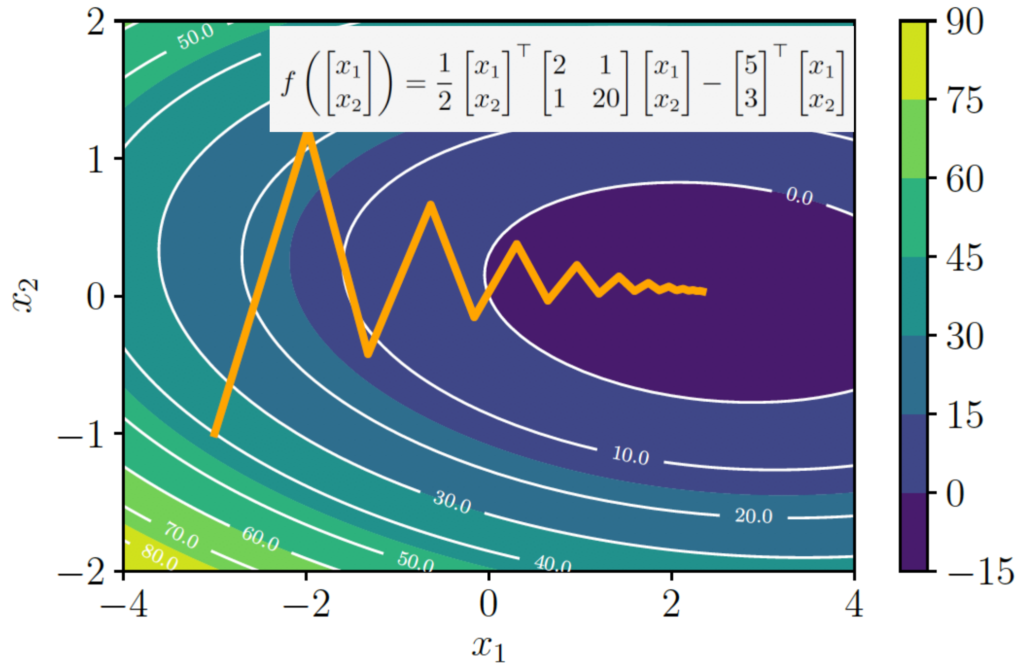
$$\sum_{k=1}^{n} e_k \frac{\partial f}{\partial \theta_i} = 0 \;\forall\; i$$

Gradient Descent

Essential task: $f : \mathbb{R}^n \to \mathbb{R}, \boldsymbol{x} \mapsto f(\boldsymbol{x})$ $\qquad \min_{\boldsymbol{x}} f(\boldsymbol{x})$

Gradient Descent

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \gamma_i \left[\nabla f(\boldsymbol{x}_i)\right]^T$$



- *very simple method*
- *depends on (adaptive) stepsize*
- *slowly convergent to closest minima*

with momentum

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \gamma_i \left[\nabla f(\boldsymbol{x}_i)\right]^T + \alpha \Delta \boldsymbol{x}_i$$

$$\Delta \boldsymbol{x}_i = \alpha \Delta \boldsymbol{x}_{i-1} - \gamma_{i-1} \left[\nabla f(\boldsymbol{x}_{i-1})\right]^T$$

- *uses "memory" to reduce jitter*

stochastic

$$f(\boldsymbol{x}) = \sum_{k=1}^{N} f_k(\boldsymbol{x})$$

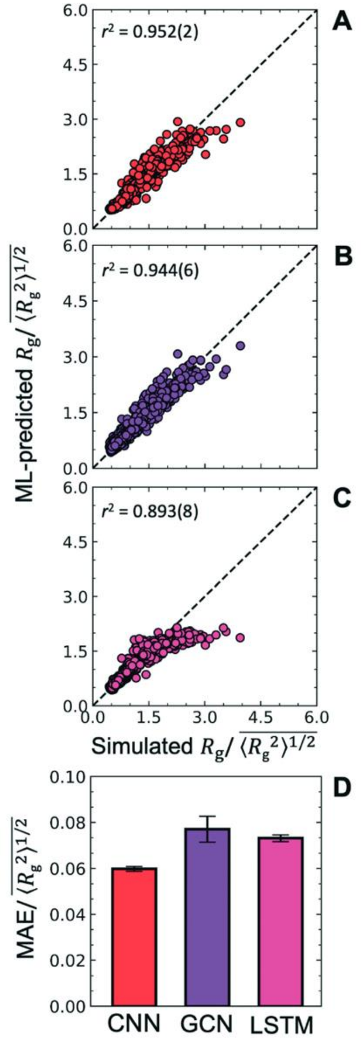
$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \gamma_i \sum_{k \subset \mathbb{N}: k \leq N} \left[\nabla f_k(\boldsymbol{x}_i)\right]^T$$

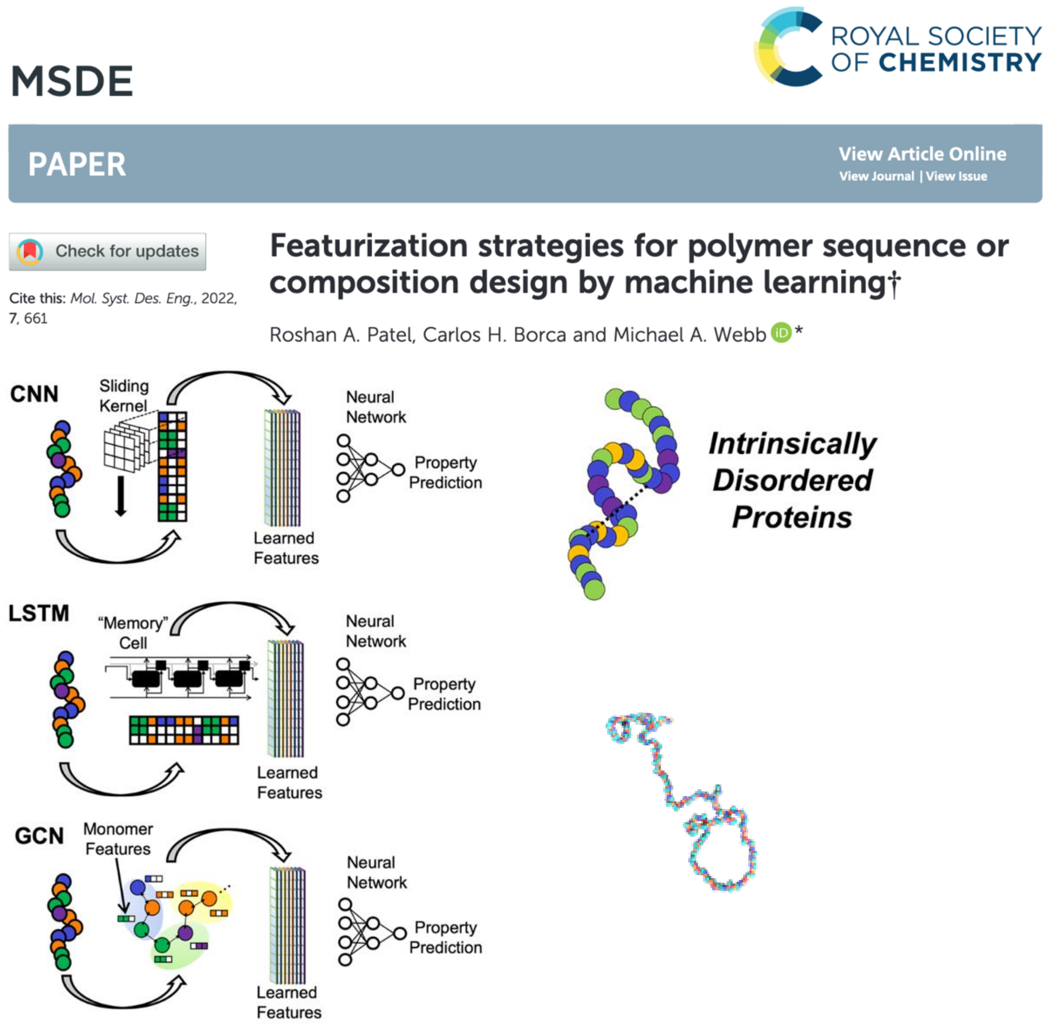
- *useful for large N, which may not be atypical in machine learning applications*

Notebook
Exercise

Activity: Premise and Objective

We will understand basic essence of parameter optimization using the example of linear regression and gradient descent; our problem of study relates to **polymer physics**





ROYAL SOCIETY OF CHEMISTRY

MSDE

PAPER

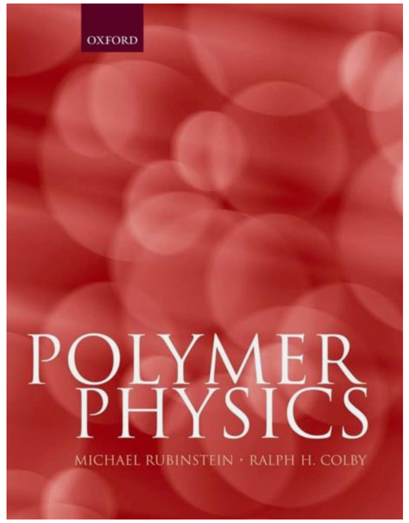
View Article Online
View Journal | View Issue

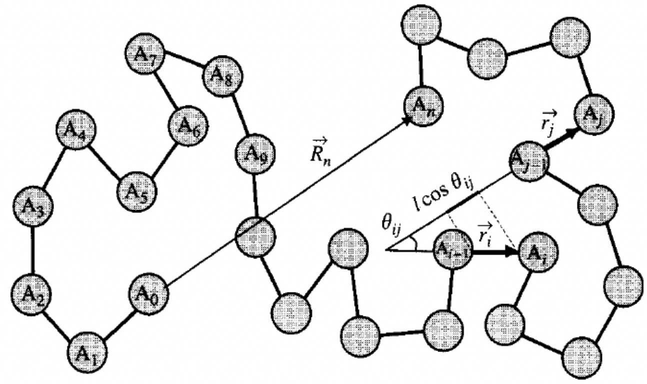
Check for updates

Cite this: *Mol. Syst. Des. Eng.*, 2022, 7, 661

Featurization strategies for polymer sequence or composition design by machine learning†

Roshan A. Patel, Carlos H. Borca and Michael A. Webb*





$$\langle R^2\rangle \equiv \langle \vec{R}_n^{\,2}\rangle = \langle \vec{R}_n\cdot\vec{R}_n\rangle = \left\langle\left(\sum_{i=1}^{n}\vec{r}_i\right)\cdot\left(\sum_{j=1}^{n}\vec{r}_j\right)\right\rangle$$

$$= \sum_{i=1}^{n}\sum_{j=1}^{n}\langle\vec{r}_i\cdot\vec{r}_j\rangle. = l^2\sum_{i=1}^{n}\sum_{i=1}^{n}\langle\cos\theta_{ij}\rangle$$

$$\langle R^2\rangle = nl^2.$$

Initialization and data inspection

Basic Imports

```python
# Modules used by Prof. Webb
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import urllib.request
import random
from sklearn.metrics import r2_score, mean_squared_error,mean_absolute_error
```

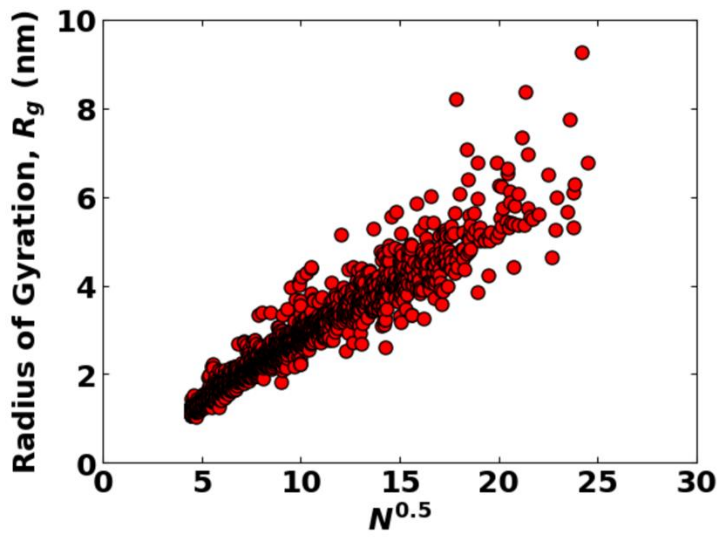
Sourcing the data

```python
url_for_labels    = "https://raw.githubusercontent.com/webbtheosim/featurization/main/Dataset_A/labels.csv"
url_for_sequences = "https://raw.githubusercontent.com/webbtheosim/featurization/main/Dataset_A/sequences.txt"
idpdata = pd.read_csv(
    url_for_labels
)

y = idpdata['ROG (A)'].to_numpy()/10.     # these are now labels
seqs  = [line.strip().split() for line in urllib.request.urlopen(url_for_sequences)]
X     = np.array([len(seq) for seq in seqs])**0.5   # these are features

idpdata.head()
```

	ROG (A)	CV (J/K)	TAUS (fs)
0	11.725914	0.444604	36585.162
1	11.912079	0.370302	40234.011
2	11.375047	0.399939	38123.675
3	11.457038	0.407542	34174.561
4	11.509964	0.449730	34279.740

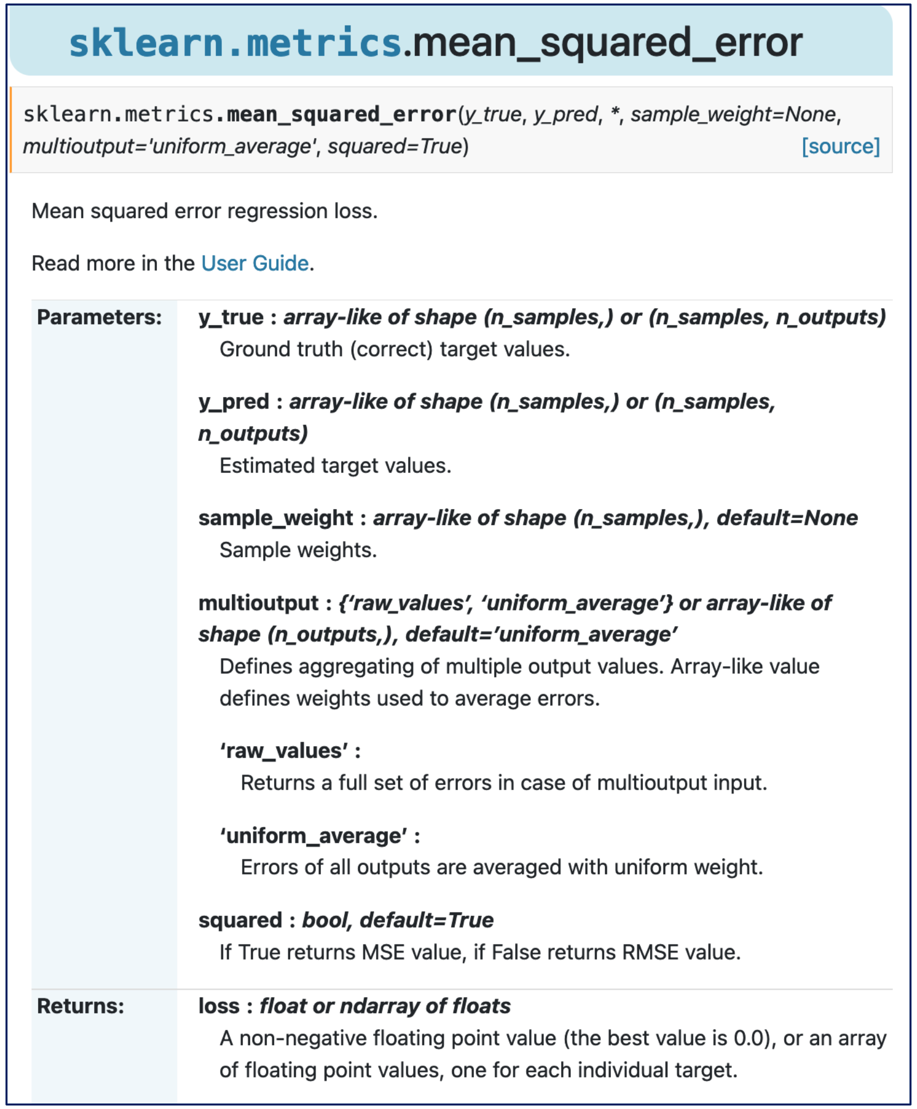
Plotting the data

```python
# global specifications on plots
plt.rcParams.update({'font.size': 18,
                     'font.weight' : 'bold',
                     'axes.labelweight': 'bold'})

def plot_raw_data(x,y):
  plt.plot(x, y,marker='o',linestyle="",markersize=8,\
           color='r',markeredgecolor='k')
  plt.ylabel("Radius of Gyration, $R_g$ (nm)")
  plt.xlabel("$N^{0.5}$")
  plt.xlim(0,30)
  plt.ylim(0,10)
  ax = plt.gca()
  ax.tick_params(direction='in')
  ax.yaxis.set_ticks_position('both')
  ax.xaxis.set_ticks_position('both')
  return ax

ax = plot_raw_data(X,y)
```



Human Hypothesis to the Data

$$R_g = \theta_0 + \theta_1 N^{0.5}$$



sklearn.metrics.mean_squared_error

sklearn.metrics.**mean_squared_error**(*y_true*, *y_pred*, *, *sample_weight=None*, *multioutput='uniform_average'*, *squared=True*) [source]

Mean squared error regression loss.

Read more in the User Guide.

Parameters:

y_true : *array-like of shape (n_samples,) or (n_samples, n_outputs)*
Ground truth (correct) target values.

y_pred : *array-like of shape (n_samples,) or (n_samples, n_outputs)*
Estimated target values.

sample_weight : *array-like of shape (n_samples,), default=None*
Sample weights.

multioutput : *{'raw_values', 'uniform_average'} or array-like of shape (n_outputs,), default='uniform_average'*
Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

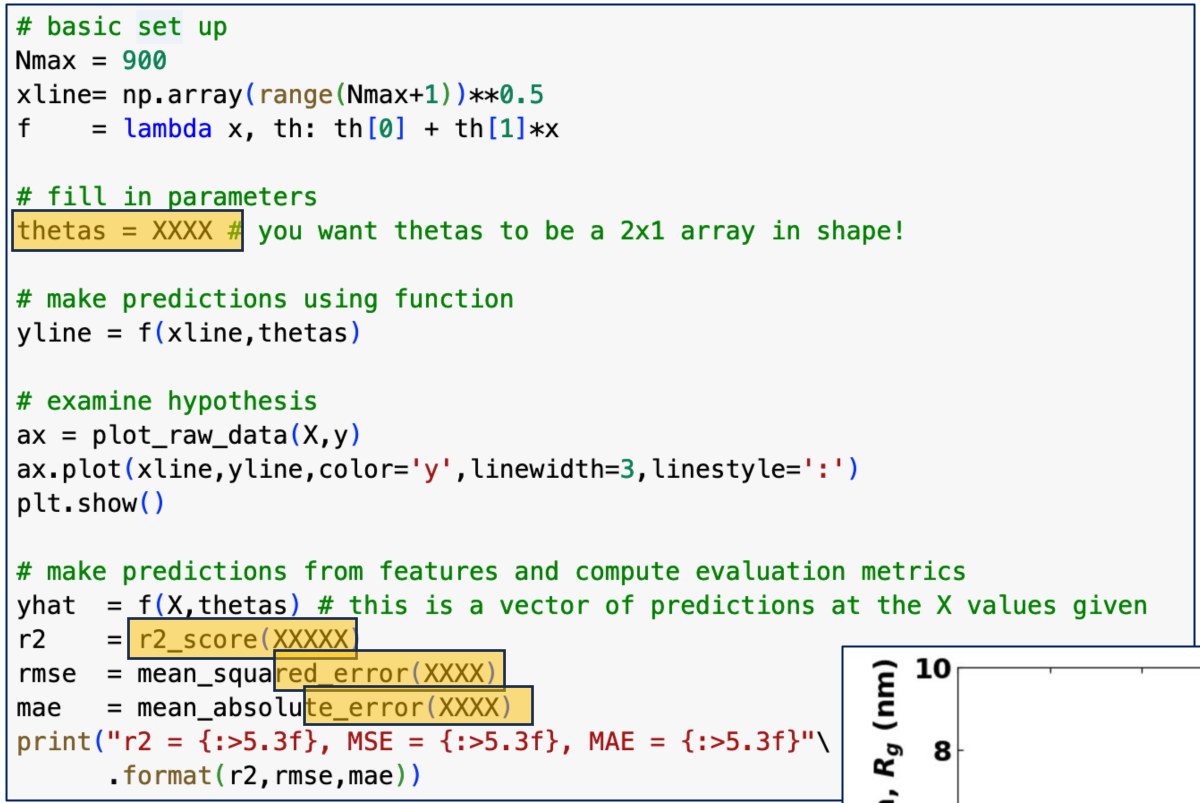
'raw_values' :
Returns a full set of errors in case of multioutput input.

'uniform_average' :
Errors of all outputs are averaged with uniform weight.

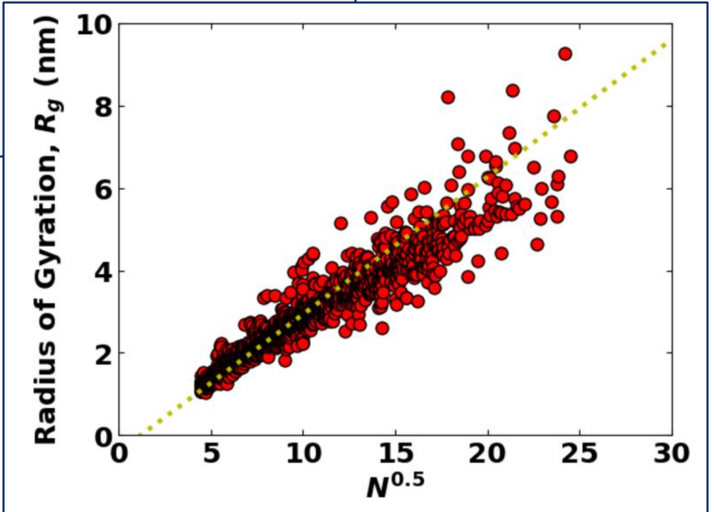
squared : *bool, default=True*
If True returns MSE value, if False returns RMSE value.

Returns:

loss : *float or ndarray of floats*
A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.



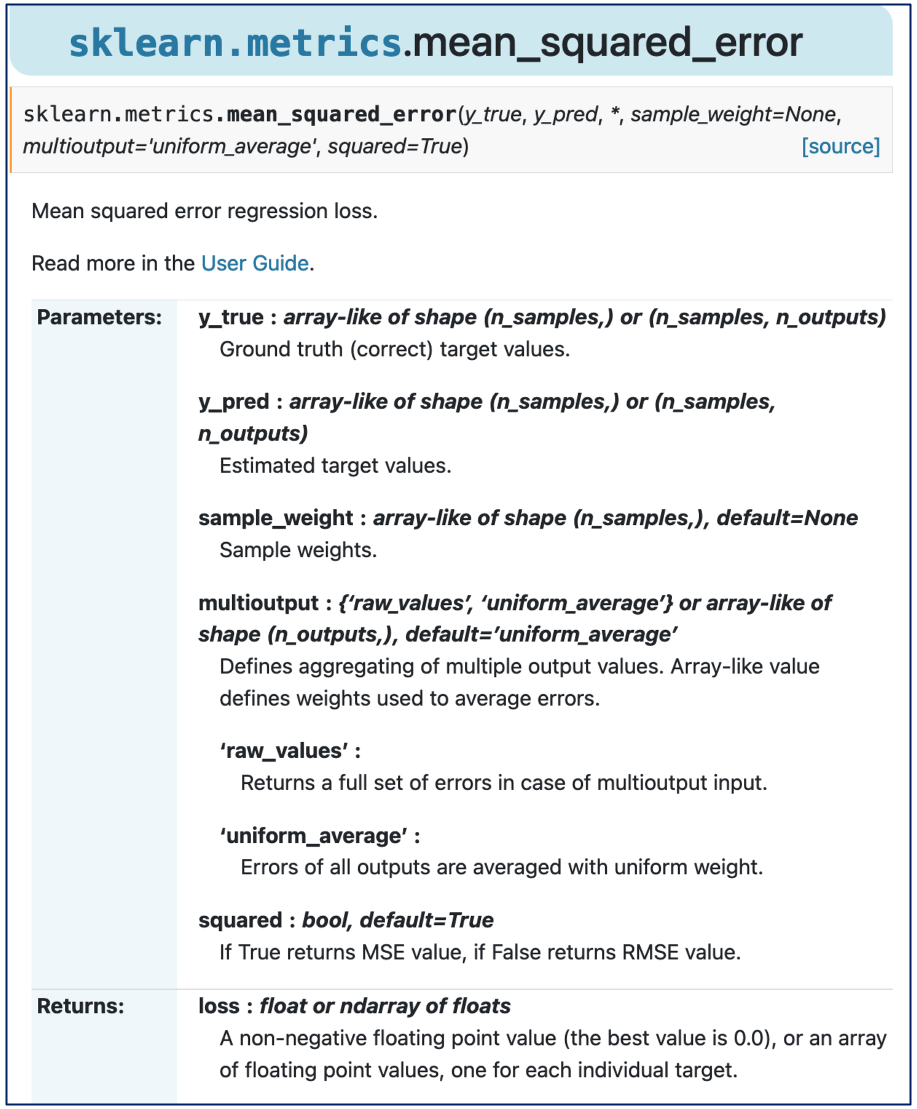
```
# basic set up
Nmax = 900
xline= np.array(range(Nmax+1))**0.5
f    = lambda x, th: th[0] + th[1]*x

# fill in parameters
thetas = XXXX # you want thetas to be a 2x1 array in shape!

# make predictions using function
yline = f(xline,thetas)

# examine hypothesis
ax = plot_raw_data(X,y)
ax.plot(xline,yline,color='y',linewidth=3,linestyle=':')
plt.show()

# make predictions from features and compute evaluation metrics
yhat  = f(X,thetas) # this is a vector of predictions at the X values given
r2    = r2_score(XXXXX)
rmse  = mean_squared_error(XXXX)
mae   = mean_absolute_error(XXXX)
print("r2 = {:>5.3f}, MSE = {:>5.3f}, MAE = {:>5.3f}"\
      .format(r2,rmse,mae))
```



```
r2 = 0.907, MSE = 0.118, MAE = 0.194
```

Human Hypothesis to the Data

$$R_g = \theta_0 + \theta_1 N^{0.5}$$



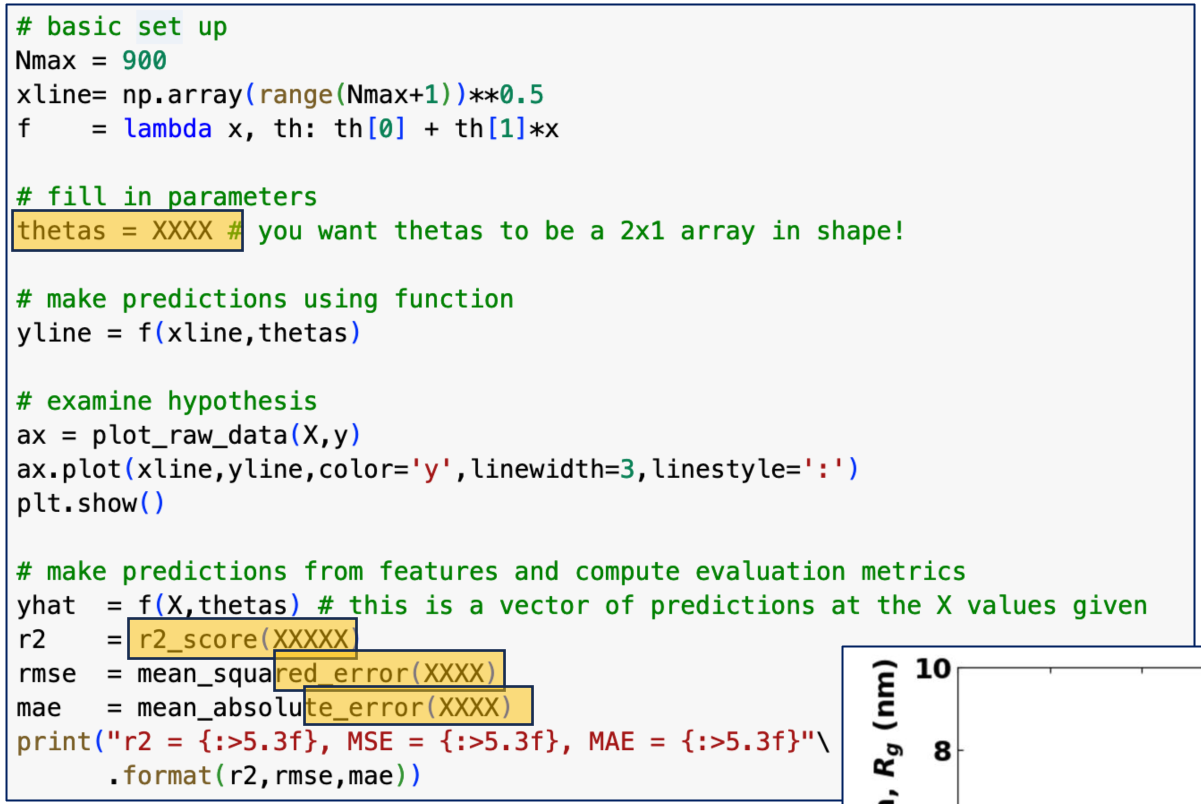
sklearn.metrics.mean_squared_error

sklearn.metrics.**mean_squared_error**(*y_true*, *y_pred*, *, *sample_weight=None*, *multioutput='uniform_average'*, *squared=True*) [source]

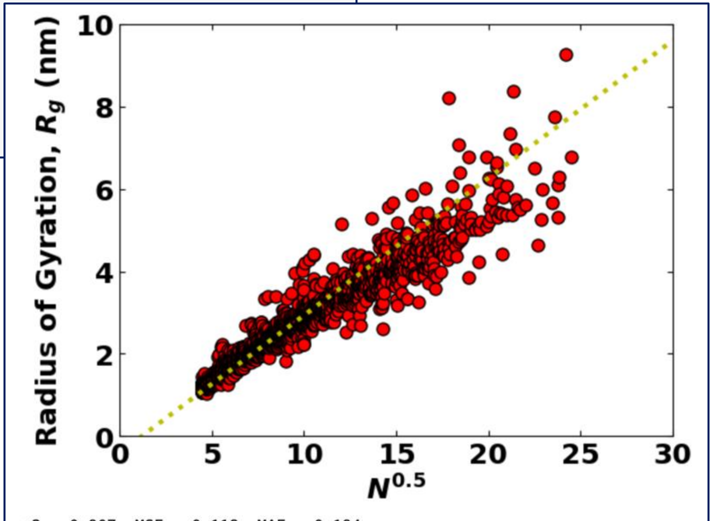
Mean squared error regression loss.

Read more in the User Guide.

Parameters:	**y_true : *array-like of shape (n_samples,) or (n_samples, n_outputs)*** Ground truth (correct) target values. **y_pred : *array-like of shape (n_samples,) or (n_samples, n_outputs)*** Estimated target values. **sample_weight : *array-like of shape (n_samples,), default=None*** Sample weights. **multioutput : *{'raw_values', 'uniform_average'} or array-like of shape (n_outputs,), default='uniform_average'*** Defines aggregating of multiple output values. Array-like value defines weights used to average errors. **'raw_values' :** Returns a full set of errors in case of multioutput input. **'uniform_average' :** Errors of all outputs are averaged with uniform weight. **squared : *bool, default=True*** If True returns MSE value, if False returns RMSE value.
Returns:	**loss : *float or ndarray of floats*** A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.



```
# basic set up
Nmax = 900
xline= np.array(range(Nmax+1))**0.5
f    = lambda x, th: th[0] + th[1]*x

# fill in parameters
thetas = XXXX # you want thetas to be a 2x1 array in shape!

# make predictions using function
yline = f(xline,thetas)

# examine hypothesis
ax = plot_raw_data(X,y)
ax.plot(xline,yline,color='y',linewidth=3,linestyle=':')
plt.show()

# make predictions from features and compute evaluation metrics
yhat  = f(X,thetas) # this is a vector of predictions at the X values given
r2    = r2_score(XXXXX)
rmse  = mean_squared_error(XXXX)
mae   = mean_absolute_error(XXXX)
print("r2 = {:>5.3f}, MSE = {:>5.3f}, MAE = {:>5.3f}"\
      .format(r2,rmse,mae))
```



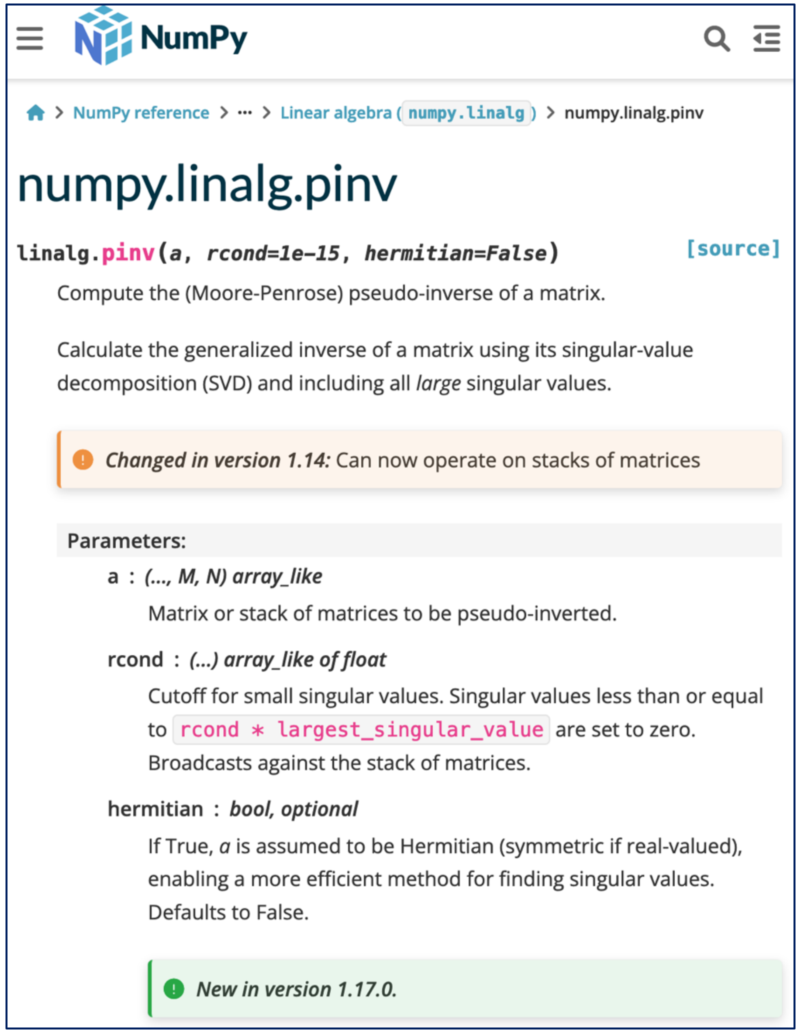
```
r2 = 0.907, MSE = 0.118, MAE = 0.194
```

Linear Algebraic Solution

Because our model is linear in all its parameters,
we can find an exact solution using linear algebra

$$R_g = \theta_0 + \theta_1 N^{0.5}$$

$$\boldsymbol{Ax} = \boldsymbol{b} \Leftrightarrow \boldsymbol{A}^T\boldsymbol{Ax} = \boldsymbol{A}^T\boldsymbol{b} \Leftrightarrow \boldsymbol{x} = \underbrace{(\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T}_{\text{Moore-Penrose pseudo-inverse}}\boldsymbol{b}$$



NumPy

NumPy reference > ··· > Linear algebra (`numpy.linalg`) > numpy.linalg.pinv

numpy.linalg.pinv

`linalg.pinv(a, rcond=1e-15, hermitian=False)` [source]

Compute the (Moore-Penrose) pseudo-inverse of a matrix.

Calculate the generalized inverse of a matrix using its singular-value decomposition (SVD) and including all *large* singular values.

Changed in version 1.14: Can now operate on stacks of matrices

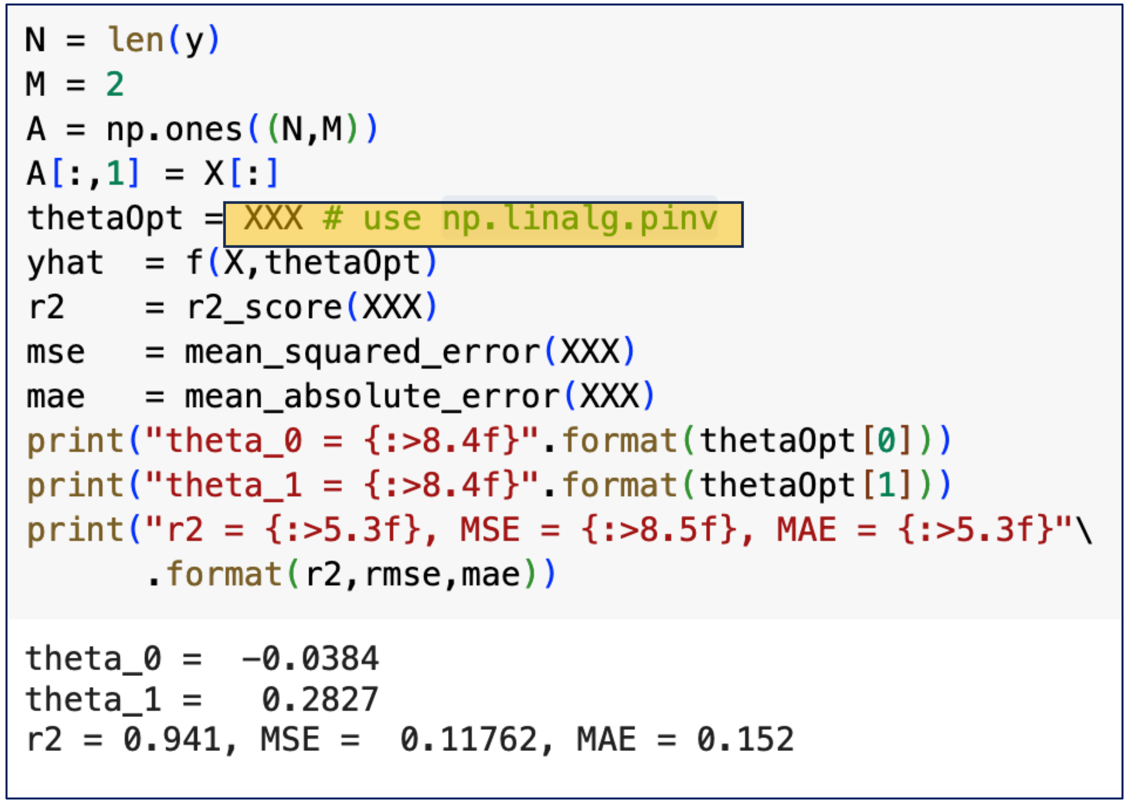
Parameters:

a : *(..., M, N) array_like*
Matrix or stack of matrices to be pseudo-inverted.

rcond : *(...) array_like of float*
Cutoff for small singular values. Singular values less than or equal to `rcond * largest_singular_value` are set to zero. Broadcasts against the stack of matrices.

hermitian : *bool, optional*
If True, *a* is assumed to be Hermitian (symmetric if real-valued), enabling a more efficient method for finding singular values. Defaults to False.

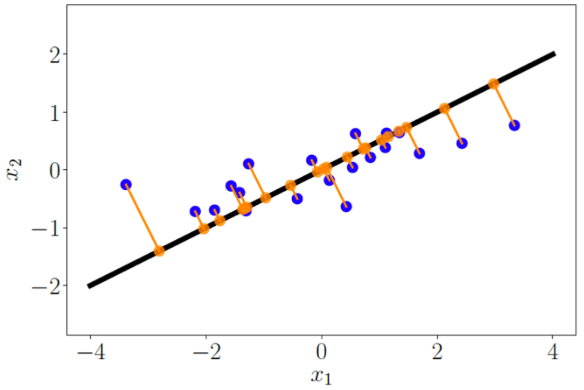
New in version 1.17.0.

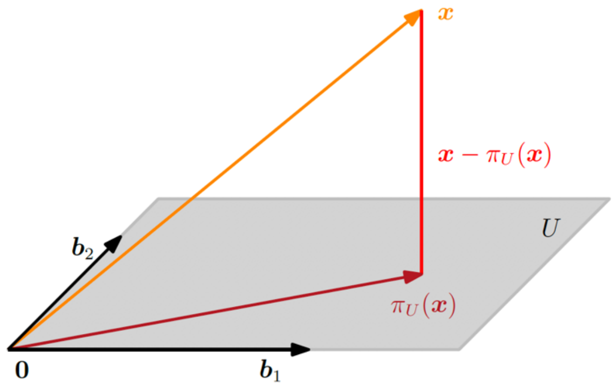


```
N = len(y)
M = 2
A = np.ones((N,M))
A[:,1] = X[:]
thetaOpt = XXX # use np.linalg.pinv
yhat   = f(X,thetaOpt)
r2     = r2_score(XXX)
mse    = mean_squared_error(XXX)
mae    = mean_absolute_error(XXX)
print("theta_0 = {:>8.4f}".format(thetaOpt[0]))
print("theta_1 = {:>8.4f}".format(thetaOpt[1]))
print("r2 = {:>5.3f}, MSE = {:>8.5f}, MAE = {:>5.3f}"\
      .format(r2,rmse,mae))
```

```
theta_0 =  -0.0384
theta_1 =   0.2827
r2 = 0.941, MSE =  0.11762, MAE = 0.152
```

Reminder: least-squares is minimizing projection error





We want $\pi_U(\boldsymbol{x}) = \sum_{i=1}^{M} \lambda_i \boldsymbol{b}_i = \boldsymbol{B\lambda}$ given $\mathcal{B}_U = (\boldsymbol{b}_1, \dots, \boldsymbol{b}_M)$

such that $\pi_U(\boldsymbol{x}) - \boldsymbol{x}$ is ***orthogonal*** to *U* and its ***distance*** in ***minimized***

Assuming the dot product as the inner product…

$$\Longrightarrow \boldsymbol{b}_i^T(\boldsymbol{x} - \boldsymbol{B\lambda}) = 0, \; i = 1, \dots, M$$

$$\Updownarrow$$

$$\boldsymbol{B}^T(\boldsymbol{x} - \boldsymbol{B\lambda}) = \boldsymbol{0} \iff \boldsymbol{B}^T\boldsymbol{B\lambda} = \boldsymbol{B}^T\boldsymbol{x}$$

normal equation

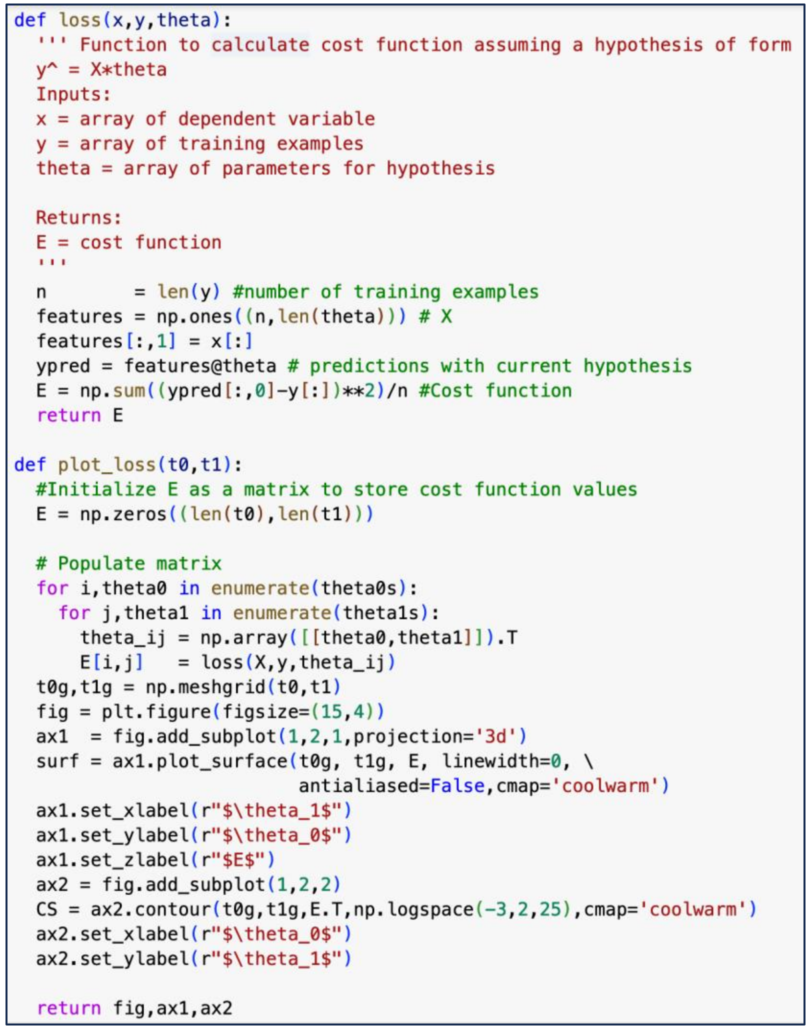
$$\boldsymbol{\lambda} = (\boldsymbol{B}^T\boldsymbol{B})^{-1}\boldsymbol{B}^T\boldsymbol{x}$$

if ***B*** describes an orthonormal basis??

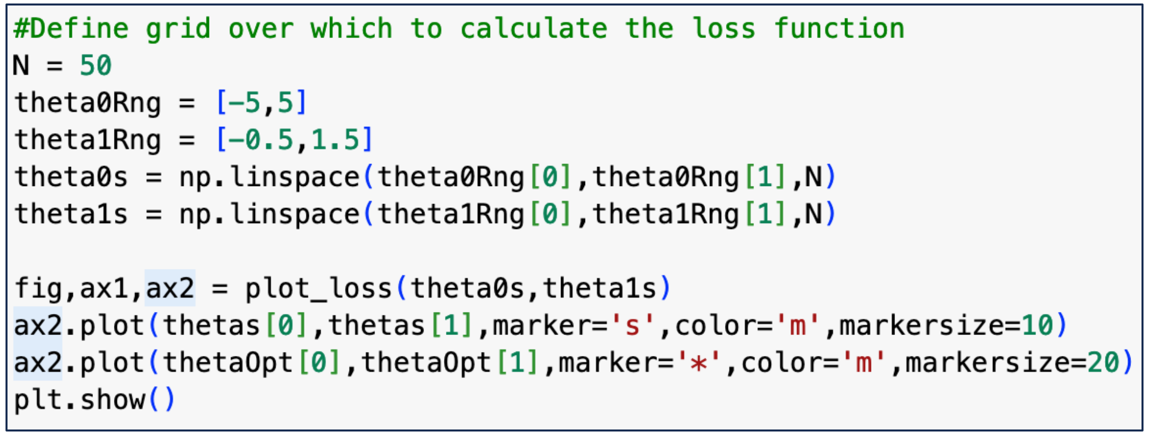
$$\Longrightarrow \boldsymbol{P}_\pi = \boldsymbol{B}(\boldsymbol{B}^T\boldsymbol{B})^{-1}\boldsymbol{B}^T$$

Optimization with a Loss Function

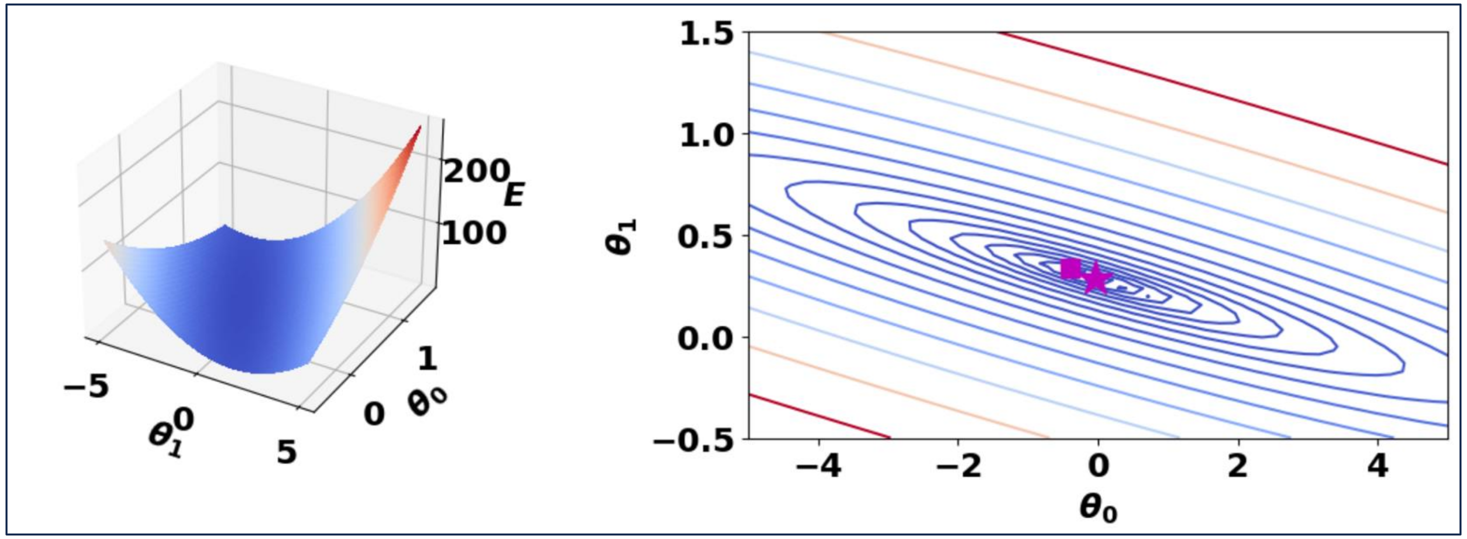
Training (optimizing parameters for) supervised ML models requires specification of a loss function and means to navigate it; let’s take a look at a simple loss function



```python
def loss(x,y,theta):
  ''' Function to calculate cost function assuming a hypothesis of form
  y^ = X*theta
  Inputs:
  x = array of dependent variable
  y = array of training examples
  theta = array of parameters for hypothesis

  Returns:
  E = cost function
  '''
  n        = len(y) #number of training examples
  features = np.ones((n,len(theta))) # X
  features[:,1] = x[:]
  ypred = features@theta # predictions with current hypothesis
  E = np.sum((ypred[:,0]-y[:])**2)/n #Cost function
  return E

def plot_loss(t0,t1):
  #Initialize E as a matrix to store cost function values
  E = np.zeros((len(t0),len(t1)))

  # Populate matrix
  for i,theta0 in enumerate(theta0s):
    for j,theta1 in enumerate(theta1s):
      theta_ij = np.array([[theta0,theta1]]).T
      E[i,j]   = loss(X,y,theta_ij)
  t0g,t1g = np.meshgrid(t0,t1)
  fig = plt.figure(figsize=(15,4))
  ax1  = fig.add_subplot(1,2,1,projection='3d')
  surf = ax1.plot_surface(t0g, t1g, E, linewidth=0, \
                          antialiased=False,cmap='coolwarm')
  ax1.set_xlabel(r"$\theta_1$")
  ax1.set_ylabel(r"$\theta_0$")
  ax1.set_zlabel(r"$E$")
  ax2 = fig.add_subplot(1,2,2)
  CS = ax2.contour(t0g,t1g,E.T,np.logspace(-3,2,25),cmap='coolwarm')
  ax2.set_xlabel(r"$\theta_0$")
  ax2.set_ylabel(r"$\theta_1$")

  return fig,ax1,ax2
```

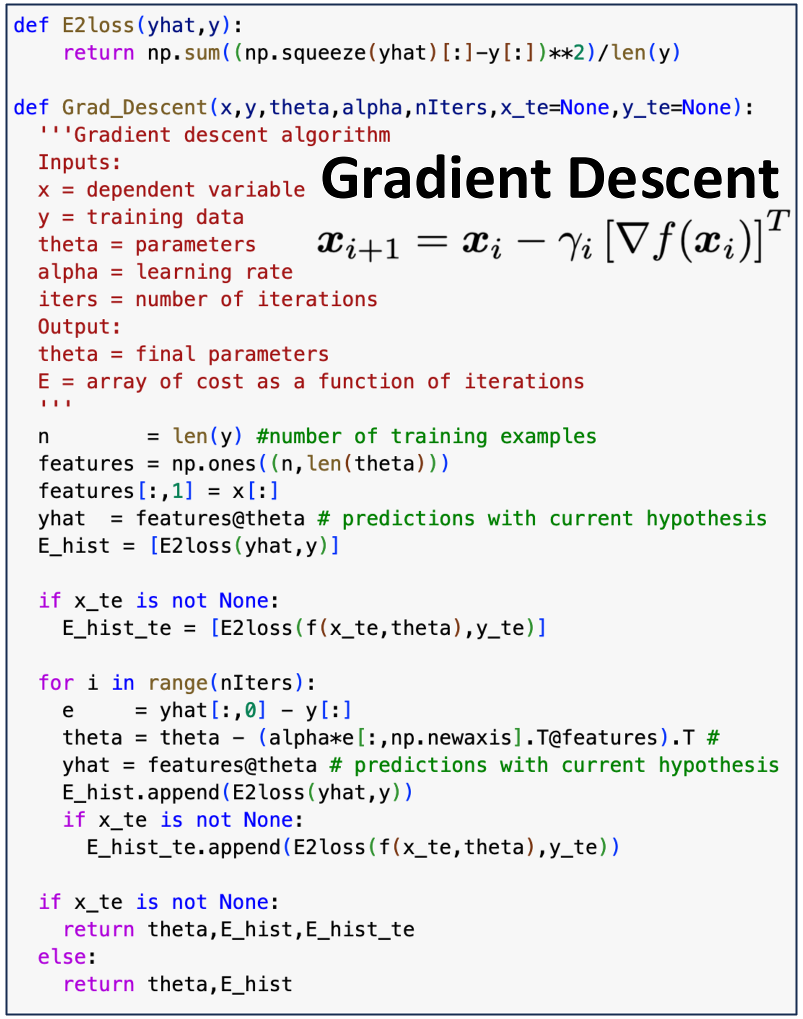


```python
#Define grid over which to calculate the loss function
N = 50
theta0Rng = [-5,5]
theta1Rng = [-0.5,1.5]
theta0s = np.linspace(theta0Rng[0],theta0Rng[1],N)
theta1s = np.linspace(theta1Rng[0],theta1Rng[1],N)

fig,ax1,ax2 = plot_loss(theta0s,theta1s)
ax2.plot(thetas[0],thetas[1],marker='s',color='m',markersize=10)
ax2.plot(thetaOpt[0],thetaOpt[1],marker='*',color='m',markersize=20)
plt.show()
```



Results with Gradient Descent Optimization

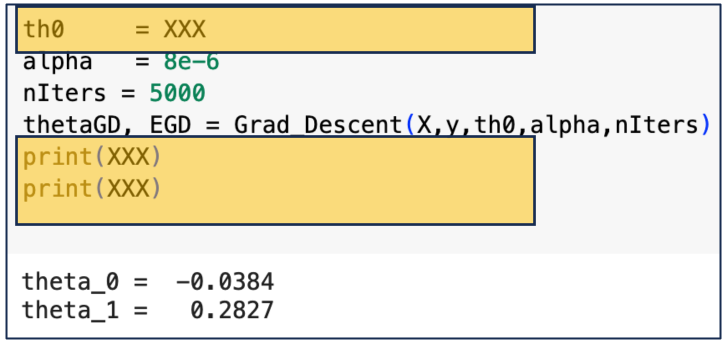
Most optimization methods make use of information regarding the gradients of the loss function with respect to the parameters; these guide selection of the next parameters



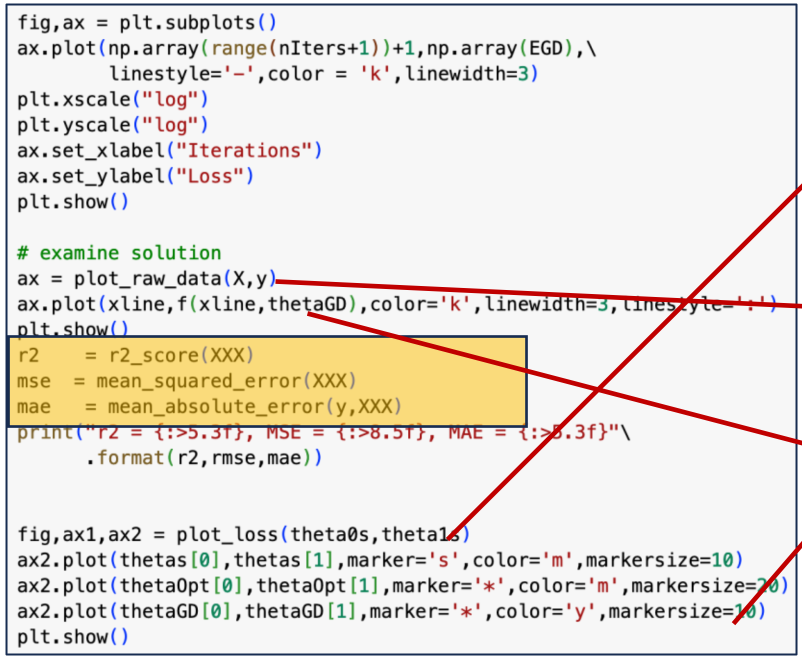
```
def E2loss(yhat,y):
    return np.sum((np.squeeze(yhat)[:]-y[:])**2)/len(y)

def Grad_Descent(x,y,theta,alpha,nIters,x_te=None,y_te=None):
  '''Gradient descent algorithm
  Inputs:
  x = dependent variable
  y = training data
  theta = parameters
  alpha = learning rate
  iters = number of iterations
  Output:
  theta = final parameters
  E = array of cost as a function of iterations
  '''
  n        = len(y) #number of training examples
  features = np.ones((n,len(theta)))
  features[:,1] = x[:]
  yhat  = features@theta # predictions with current hypothesis
  E_hist = [E2loss(yhat,y)]

  if x_te is not None:
    E_hist_te = [E2loss(f(x_te,theta),y_te)]

  for i in range(nIters):
    e     = yhat[:,0] - y[:]
    theta = theta - (alpha*e[:,np.newaxis].T@features).T #
    yhat = features@theta # predictions with current hypothesis
    E_hist.append(E2loss(yhat,y))
    if x_te is not None:
      E_hist_te.append(E2loss(f(x_te,theta),y_te))

  if x_te is not None:
    return theta,E_hist,E_hist_te
  else:
    return theta,E_hist
```

Gradient Descent

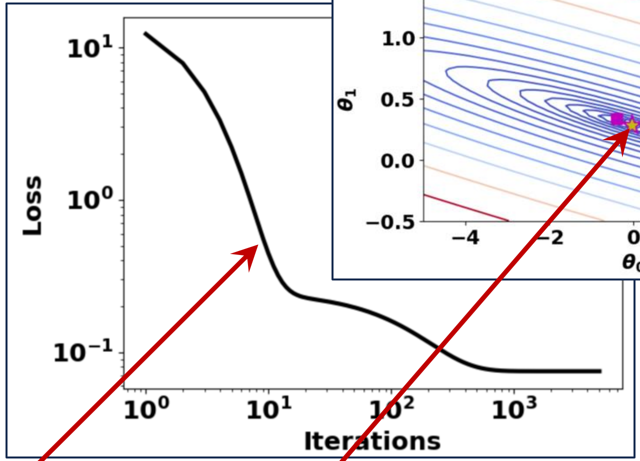
$$x_{i+1} = x_i - \gamma_i \left[\nabla f(x_i)\right]^T$$

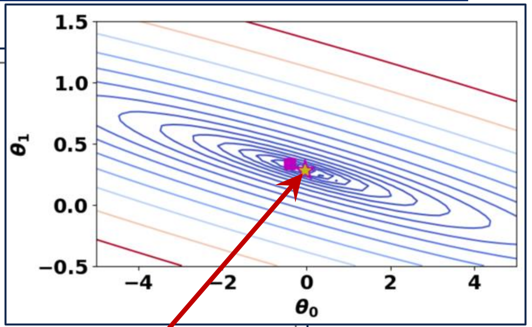


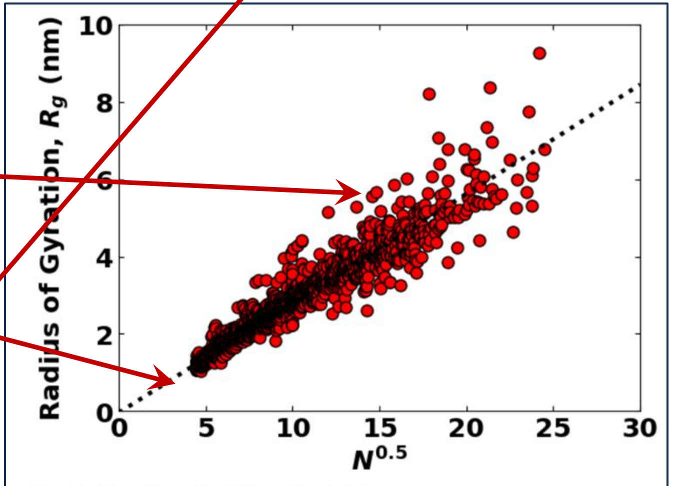
```
th0     = XXX
alpha   = 8e-6
nIters = 5000
thetaGD, EGD = Grad_Descent(X,y,th0,alpha,nIters)
print(XXX)
print(XXX)

theta_0 =  -0.0384
theta_1 =   0.2827
```



```
fig,ax = plt.subplots()
ax.plot(np.array(range(nIters+1))+1,np.array(EGD),\
        linestyle='-',color = 'k',linewidth=3)
plt.xscale("log")
plt.yscale("log")
ax.set_xlabel("Iterations")
ax.set_ylabel("Loss")
plt.show()

# examine solution
ax = plot_raw_data(X,y)
ax.plot(xline,f(xline,thetaGD),color='k',linewidth=3,linestyle=':')
plt.show()
r2    = r2_score(XXX)
mse  = mean_squared_error(XXX)
mae   = mean_absolute_error(y,XXX)
print("r2 = {:>5.3f}, MSE = {:>8.5f}, MAE = {:>5.3f}"\
      .format(r2,rmse,mae))


fig,ax1,ax2 = plot_loss(theta0s,theta1s)
ax2.plot(thetas[0],thetas[1],marker='s',color='m',markersize=10)
ax2.plot(thetaOpt[0],thetaOpt[1],marker='*',color='m',markersize=20)
ax2.plot(thetaGD[0],thetaGD[1],marker='*',color='y',markersize=10)
plt.show()
```







```
r2 = 0.941, MSE =  0.11762, MAE = 0.152
```