Start menu -> Jupyter (Desktop App)

Locate files on directory

Python is a high-level language.

Web programming

Data science

Oriented in terms of cells

# - increase size

myText = “Hello World”

print(myText)

“Type” variable

Will say what variable it is

Print(type(“bob”))

<class ‘str’>

Print(type(1))

<class ‘int’>

In Jupyter notebook

Create a new cell -> Type a or b. x to remove. Z to undo.

**List**

our\_list =[“steve”, “bob”, “emily”]

our\_list

output:

[‘steve’ ‘bob’, ‘emily’]

mynewlist = ["hello", 0, [3.14, 2.17]]

# pop from right side (default) [‘steve’, ‘bob’, “DOUG WAS HERE”]

our\_list.pop()

print(our\_list)

our\_list.pop()

print(our\_list)

output:

['steve', 'bob']

['steve']

# pop from beginning via index

our\_list.pop(0)

our\_list

output:

['bob', 'emily']

**Exercise 1:** Let's do 'hello world' but with lists. (solutions to all exercises at the bottom)

Create a variable called hello\_world\_list that has two elements: the strings 'hello' and 'world'. Then print out that variable.

#doesn’t matter if single or double quotes for this exercise

hello\_world\_list = ["hello", "world"]

hello\_world\_list #print: print(hello\_world\_list)

output:

['hello', 'world']

hello\_world\_list = ["hello", "world"]

hello\_world\_list[0]

output:

'hello'

hello\_world\_list[1]

myIndex = 1

hello\_world\_list[myIndex]

output:

'world'

print(hello\_world\_list[0], hello\_world\_list[1])

hello world #auto comma

print(hello\_world\_list[0], hello\_world\_list[1], sep ='')

helloworld #remove comma

Heterogenous list = means it has different kinds

print(type(mixed\_list[0]))

print(type(mixed\_list[1]))

print(type(mixed\_list[2]))

<class 'str'>

<class 'int'>

<class 'float'>

for i in [0,1,2]:

# Do this part once for each value of i

print(type(mixed\_list[i]))

<class 'str'>

<class 'int'>

<class 'float'>

**FOR LOOPS**

squares = []

for i in [0,1,2,3,4,5]:

squares.append(i\*i)

print(squares)

[0, 1, 4, 9, 16, 25]

squares = []

for i in [0,1,2,3,4,5]:

squares.append(i\*i)

print(squares) #indention

[0]

[0, 1]

[0, 1, 4]

[0, 1, 4, 9]

[0, 1, 4, 9, 16]

[0, 1, 4, 9, 16, 25]

[i\*i for i in [0,1,2,3,4,5]]

[0, 1, 4, 9, 16, 25]

[i\*i for i in range(6)]

[0, 1, 4, 9, 16, 25]

**Exercise 2:**

Okay, so let's create a list with the values 1, 2, 3, 4, and 5 in it. Then let's use indexing to calculate the sum.

list = [1, 2, 3, 4, 5]

list[0]+list[1]+list[2]+list[3]+list[4]

15

# put a range in the list

list = [1, 2, 3, 4, 5]

sum = 0

for i in range(5):

sum = sum + list[i]

print(sum)

15

# all values in the list

list = [1, 2, 3, 4, 5]

sum = 0

for v in list:

sum = sum + v

print(sum)

15

# using +=

list = [1, 2, 3, 4, 5]

sum = 0

for v in list:

sum += v

print(sum)

15

# sum list

mylist = [1, 2, 3, 4, 5]

sum(mylist)

15

**Nesting Lists**

list\_of\_lists = [[1,0],[2,3],[5,7],[1,2],[5,3]]

print(list\_of\_lists)

[[1, 0], [2, 3], [5, 7], [1, 2], [5, 3]]

# access second element in sublist

for sublist in list\_of\_lists:

print("Sublist:", sublist)

Sublist: [1, 0]

Sublist: [2, 3]

Sublist: [5, 7]

Sublist: [1, 2]

Sublist: [5, 3]

# access 2nd element of 3rd list

print(list\_of\_lists[2])

print(list\_of\_lists[2][1])

[5, 7]

7

# printing sublist

# access second element in sublist

for sublist in list\_of\_lists:

print("Sublist:", sublist[1])

Sublist: [1, 0]

Sublist: [2, 3]

Sublist: [5, 7]

Sublist: [1, 2]

Sublist: [5, 3]

# prints index 1 of all list

# Or, using a list comprehension:

[sublist[1] for sublist in list\_of\_lists]

Out[66]:

[0, 3, 7, 2, 3]

for i in range(5):

print("i = ",i)

for j in range(3):

print(" j = ",j)

i = 0

j = 0

j = 1

j = 2

i = 1

j = 0

j = 1

j = 2

i = 2

j = 0

j = 1

j = 2

i = 3

j = 0

j = 1

j = 2

i = 4

j = 0

j = 1

j = 2

**Slicing Lists** a\_list[start:stop:skip]

a\_list = ['a', 'b', 'c', 'd', 'e']

a\_list

['a', 'b', 'c', 'd', 'e']

# First two elements

a\_list[0:2]

['a', 'b']

# Last two elements

a\_list[-2:] #negative numbers wrap around, starts at the end of the list

['d', 'e'] # -1 is the end index

# Last two elements

a\_list[:-2] # colon before the number

['a', 'b', 'c']

# print odd numbers

nums = [0,1,2,3,4,5,6]

odds = nums[1::2]

odds

[1, 3, 5]

# print even numbers

nums = [0,1,2,3,4,5,6]

odds = nums[1::2]

evens = nums [::2]

evens

[0, 2, 4, 6]

# backwards evens

nums = [0,1,2,3,4,5,6]

odds = nums[1::2]

evens = nums [::2]

backwards = nums[::-1]

backwards

[6, 5, 4, 3, 2, 1, 0]

**List Comprehensions**

a\_list = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

a\_list

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

# We can also use extra conditions like:

c\_list = [x for x in a\_list if 5 < x < 9]

c\_list

[8, 7, 6]

# for each elements put it on x

# We can also use extra conditions like:

c\_list = [x for x in a\_list if 5 <= x <= 9]

c\_list

[9, 8, 7, 6, 5]

Using “.” plus “tab” shows list of functions

**Some other data structures: Sets and Tuples**

set() # gives ordering to elements

list\_of\_numbers = [1,2,2,2,4,3,5]

set(list\_of\_numbers)

{1, 2, 3, 4, 5}

list\_of\_numbers = [5,3,1]

set(list\_of\_numbers)

{1, 3, 5}

mySet = {3, 2, 1, 4, 5}

mySet

{1, 2, 3, 4, 5}

for i in list\_of\_numbers:

print(i)

1

2

2

2

4

3

5

for i in set(list\_of\_numbers):

print(i)

1

2

3

4

5

When you type for a variable and turns “green” means variable name used already

Else, reinitialize with:

del list # delete list variable

del list

list\_of\_numbers = [5,3,4,1]

set(list\_of\_numbers)

mySet = {3, 2, 1, 4, 5}

list(mySet)

[1, 2, 3, 4, 5]

# changing element inside an array

li = [1,2,3]

print(type(li))

li[0] = 7

print(li)

<class 'list'>

[7, 2, 3]

s = set([1,2,3])

print(type(s))

s[0] = 7 # Can't do this - sets are immutable!

print(s) # it isn’t in ordered

# Intro to Linear Regression

import numpy as np # can use as np

import matplotlib.pyplot as plt # can use as plt

%matplotlib inline #jupyter notebook specific, display after the cell

raw\_inputs = np.linspace(0,10,21)

x = []

y = []

for val in raw\_inputs:

x.append(val)

y.append(val + np.random.normal(0,0.4))

x = np.array(x).reshape(-1,1)

y = np.array(y).reshape(-1,1)

plt.scatter(x,y)

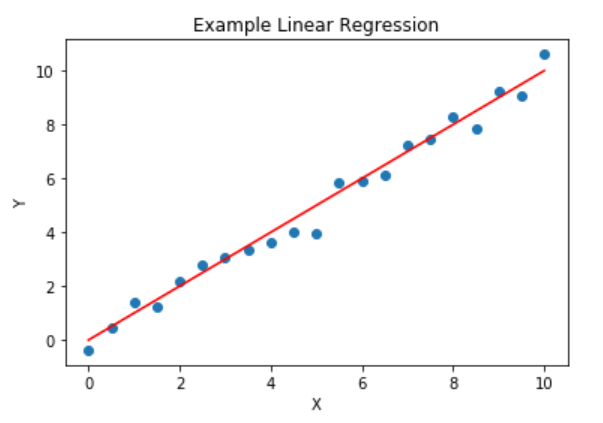
plt.plot(raw\_inputs,raw\_inputs,'r')

plt.xlabel("X")

plt.ylabel("Y")

plt.title("Example Linear Regression");

**output:**



numpy arrays are tailored to numeric data

should be homogeneous

NOTE: ctrl + shift + (-) breaks cells in jupyter notebook

NOTE: install i.e., “conda install matplotlib”

import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline

raw\_inputs = np.linspace(0,10,21)

raw\_inputs

array([ 0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5, 10. ])

np.random.normal(0,0.4) #0.4 is the standard deviation

numpy – module/library

random – another collection of code

normal –

go inside numpy -> then go inside random -> then go inside normal

x = []

y = []

for val in raw\_inputs:

x.append(val)

y.append(val + np.random.normal(0,0.4))

x

[0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0]

x = []

y = []

for val in raw\_inputs:

x.append(val)

y.append(val + np.random.normal(0,0.4))

y #random numbers

[-0.31086159717291612, 0.30241083692378457, 0.93531796735786032, 1.7492674244372581, 2.1425906372840982, 2.5652269618715291, 2.79750648920374, 2.7122019228267922, 4.4336719593345757, 3.9067833955381746, 4.8343109723504378, 5.922934337376522, 5.8460649015816513, 6.0527689734654686, 7.1965010716636462, 6.4919149774490679, 7.9058797361333255, 8.2888536604773702, 8.089420311578424, 8.7616020856254853, 10.526363375449357]

x = np.array(x).reshape(-1,1)

y = np.array(y).reshape(-1,1)

x

array([[ 0. ], [ 0.5], [ 1. ], [ 1.5], [ 2. ], [ 2.5], [ 3. ], [ 3.5], [ 4. ], [ 4.5], [ 5. ], [ 5.5], [ 6. ], [ 6.5], [ 7. ], [ 7.5], [ 8. ], [ 8.5], [ 9. ], [ 9.5], [ 10. ]])

reshape(-1,1) # -1 -do anything you want, 1 = no. of column

represent as a column matrix

else, array is represented as a one-dimensional matrix

more one SciPy.org

NOTE: Shift + tab x2 shows description/documentation

plt.scatter(x,y)

plt.plot(raw\_inputs,raw\_inputs,'r') # raw inputs vs raw inputs because y is just small noise

plt.title("Example Linear Regression") # without ; shows the value

Text(0.5,1,'Example Linear Regression')

*<prints the linear regression>*

plt.title("Example Linear Regression"); # ; is evaluate the value

*<prints the linear regression>*

Last line in the cell, assumed you want the info, if not put ;

plt.plot(raw\_inputs,raw\_inputs,'r') # r = red

**Ordinary Least Squares**

xpoint = x[3] # x coordinate

ypoint = y[3] # y coordinate

predictions = [5 for i in raw\_inputs]

ypred = predictions[3]

plt.scatter(x,y)

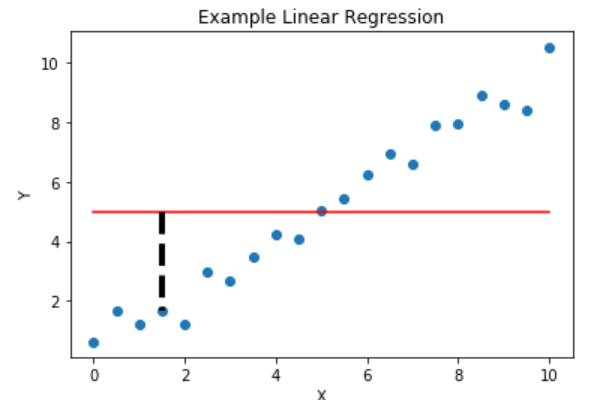
plt.plot(raw\_inputs,predictions,'r')

plt.plot([xpoint,xpoint],[ypred,ypoint],'k--',lw=4)

plt.xlabel("X")

plt.ylabel("Y")

plt.title("Example Linear Regression");



plt.plot([xpoint,xpoint],[ypred,ypoint],'k--',lw=4) ## color black, (--) trace line, lw line width

**FUNCTIONS**

def f(x):

return x\*x-3

f(4)

13

def f(x,y):

return x\*y-3

f(4,5)

17

* Ordinary: We're not doing anything crazy, we're just using the regular errors (how much our model missed).
* Least: We're trying to find a result that minimizes our errors.
* Squares: We're using the square of the errors to optimize... really punishing us if we miss any values by a lot and rewarding us for errors < 1.

def get\_error(true\_y, predictions):

error = 0

for y, pred in zip(true\_y,predictions):

error += (y-pred)\*(y-pred)

return error

get\_error(y,predictions)

array([ 187.4861156])

for y, pred in zip(true\_y,predictions): #zip pairs up values, like a zipper

NOTE: like pairing of socks you put them as a pair, not pair them and put them in the drawer

a = [1,2,3,4]

b = ['a','b','c','d']

z = zip(a,b)

list(z)

[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# stops when a list runs out

a = [1,2,3,4,5]

b = ['a','b','c','d']

z = zip(a,b) # zip is an iterator

list(z)

[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

xpoint = x[3]

ypoint = y[3]

m = 0.5

b = 0.

predictions = [b+m\*i for i in raw\_inputs] # THIS LINE CHANGES

ypred = predictions[3]

plt.scatter(x,y)

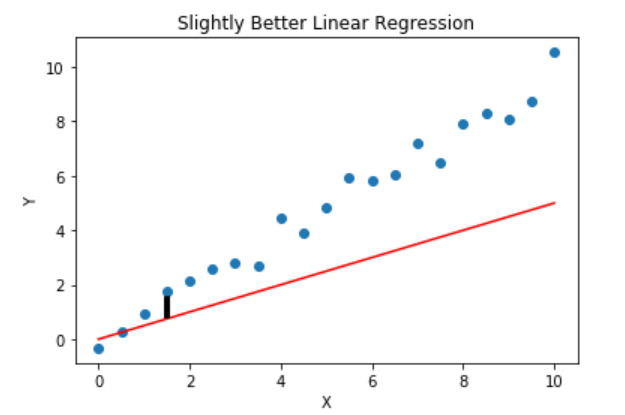
plt.plot(raw\_inputs,predictions,'r')

plt.plot([xpoint,xpoint],[ypred,ypoint],'k--',lw=4)

plt.xlabel("X")

plt.ylabel("Y")

plt.title("Slightly Better Linear Regression");



Change b and m pair to make a good fit.

get\_error(y,predictions)

array([ 159.33995464])

# Initialize the program so that we can start it!

prev\_error = 10000

slope = 0.0

# Start with a line that's all 0's and get the error there.

predictions = [0+slope\*i for i in raw\_inputs]

current\_error = get\_error(y,predictions)

# Now we loop through and see if we're getting better or worse errors!

# If we get worse errors, we'll stop trying to go higher. This is an over-simplification

# but we're going to use it for demo purposes.

# ramps up lines until we get an error

while current\_error < prev\_error:

prev\_error = current\_error

slope += 0.1

predictions = [0+slope\*i for i in raw\_inputs]

current\_error = get\_error(y,predictions)

# Uncomment below to see all the lines being tried before we stop!

# plt.plot(raw\_inputs,predictions,'b',alpha=slope-.1)

# Remove the last step since it made it worse.

slope -=0.1

predictions = [0+slope\*i for i in raw\_inputs]

current\_error = get\_error(y,predictions)

ypred = predictions[3]

# Now print our results

print("The slope we found is:", slope)

plt.scatter(x,y)

plt.plot(raw\_inputs,predictions,'r')

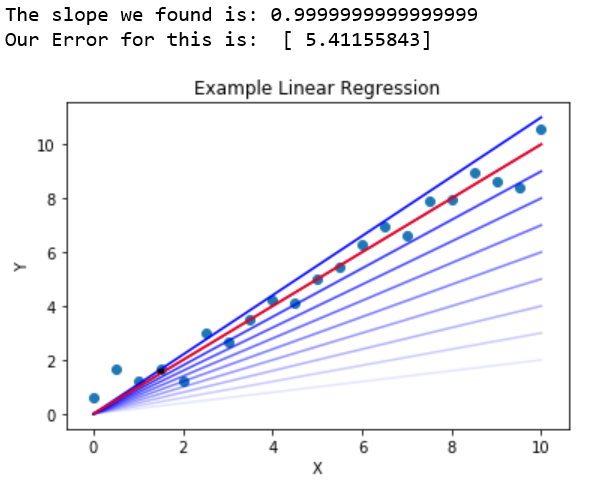
plt.plot([xpoint,xpoint],[ypred,ypoint],'k--',lw=4)

plt.xlabel("X")

plt.ylabel("Y")

plt.title("Example Linear Regression");

print("Our Error for this is: ", get\_error(y,predictions))



# Initialize the program so that we can start it!

prev\_error = 10000

slope = 0.0

# Start with a line that's all 0's and get the error there.

predictions = [0+slope\*i for i in raw\_inputs]

current\_error = get\_error(y,predictions)

current\_error

array([ 672.81194882])

# Now we loop through and see if we're getting better or worse errors!

# If we get worse errors, we'll stop trying to go higher. This is an over-simplification

# but we're going to use it for demo purposes.

while current\_error < prev\_error:

prev\_error = current\_error

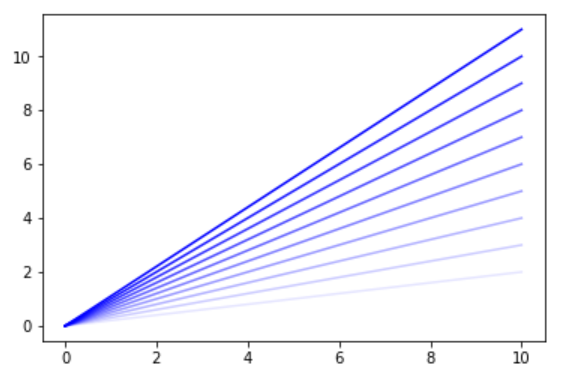
slope += 0.1

predictions = [0+slope\*i for i in raw\_inputs]

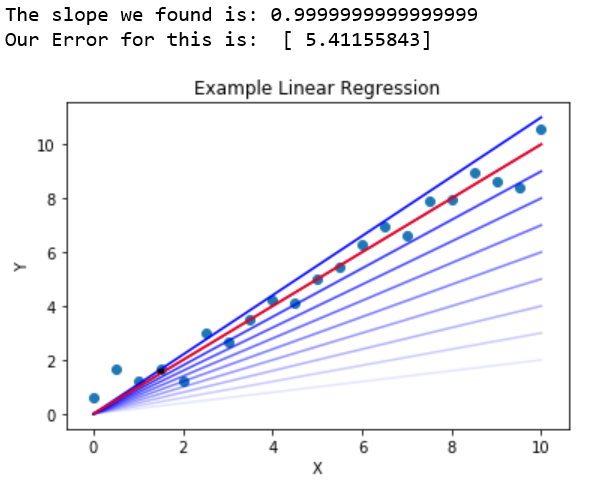
current\_error = get\_error(y,predictions)

# Uncomment below to see all the lines being tried before we stop!

plt.plot(raw\_inputs,predictions,'b',alpha=slope-.1)



**Compare with this**



current\_error # sum of the squared residuals

array([ 16.72356163])

# Remove the last step since it made it worse.

slope -=0.1

predictions = [0+slope\*i for i in raw\_inputs]

current\_error = get\_error(y,predictions)

ypred = predictions[3]

# Now print our results

print("The slope we found is:", slope)

plt.scatter(x,y)

plt.plot(raw\_inputs,predictions,'r')

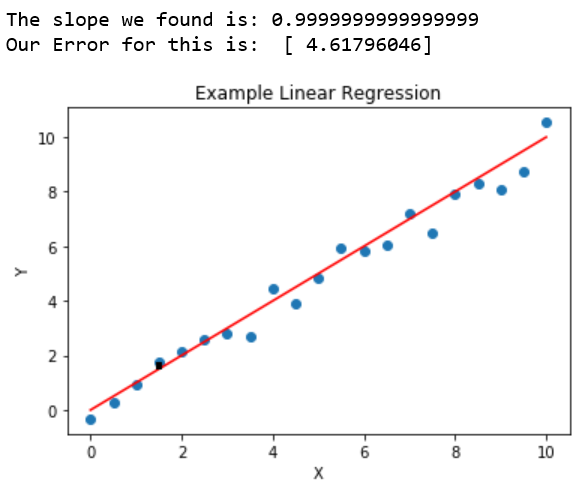
plt.plot([xpoint,xpoint],[ypred,ypoint],'k--',lw=4)

plt.xlabel("X")

plt.ylabel("Y")

plt.title("Example Linear Regression");

print("Our Error for this is: ", get\_error(y,predictions))



In real world machine learning,

Should you increase? Or decrease in the slope? Or direction.

Regularization

Overfitting

i.e., every Tuesday, its raining. So I bring umbrella in the whole year

-> false conclusion on few cases scene! Overfitting

-> be more cautious

# **R (programming language)**

**For data statistics**

**Udemy**

**Machine Learning A-Z**

**Hands-On Python & R**

INTRO to SKLEARN

import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline

raw\_inputs = np.linspace(0,10,21)

x = []

y = []

for val in raw\_inputs:

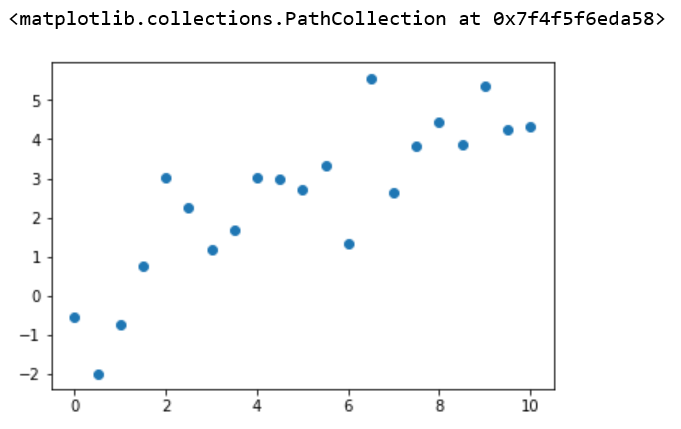
x.append(val)

y.append(0.5\*val + np.random.randn())

x = np.array(x).reshape(-1,1)

y = np.array(y).reshape(-1,1)

plt.scatter(x,y)



from sklearn.linear\_model import LinearRegression #import the regression called "LinearRegression"

lr = LinearRegression() # Initialize the model

lr.fit(x,y) # "Fit" the model, which means "I want you to learn from my data!"

LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=1, normalize=False)

new\_x = np.array([7,8,9]).reshape(-1,1)

predicted\_y = lr.predict(new\_x) # Predict for new data!

print(predicted\_y)

[[ 3.6139032 ] [ 4.15020714] [ 4.68651107]]

from sklearn.linear\_model import Lasso

lr = Lasso() # Initialize the model

lr.fit(x,y) # "Fit" the model, which means "I want you to learn from my data!"

Lasso(alpha=1.0, copy\_X=True, fit\_intercept=True, max\_iter=1000, normalize=False, positive=False, precompute=False, random\_state=None, selection='cyclic', tol=0.0001, warm\_start=False)

new\_x = np.array([7,8,9]).reshape(-1,1)

predicted\_y = lr.predict(new\_x) # Predict for new data!

print(predicted\_y)

[ 3.39572139 3.82293441 4.25014743]

def do\_regression(model, x, y, new\_x):

model.fit(x,y)

ypred = model.predict(new\_x)

return ypred

do\_regression(Lasso(alpha=0.5, fit\_intercept=False), x, y, new\_x)

array([[ 3.50804407],

[ 4.00919323],

[ 4.51034238]])

do\_regression(LinearRegression(), x, y, new\_x)

array([[ 3.6139032 ], [ 4.15020714], [ 4.68651107]])

Note: it does smoothing a model, not overfitting.

Check the weather do not need to bring umbrella every Tuesday.

Note: alpha smooths things out (how much do you want to smooth things)

Linear regression goal is to minimize sum of squared errors

This section describes to us some of the attributes associated with the model. So for instance, we know that linear regression is trying to find a line... so it should have a formula like y = coefficient \* x + intercept. Once we've trained our model, we can find these by looking at the attributes like so:

print (lr.coef\_,lr.intercept\_)

[ 0.42721302] [ 0.40523023]

**Score**

raw\_inputs = np.linspace(0,10,11)

test\_x = []

test\_y = []

for val in raw\_inputs:

test\_x.append(val)

test\_y.append(0.5\*val + np.random.randn())

test\_x = np.array(test\_x).reshape(-1,1)

test\_y = np.array(test\_y).reshape(-1,1)

lr.score(test\_x,test\_y) # This model isn't very good yet...

0.78881512384442787

**Transform (to check in other viewpoint i.e., rotate it)**

This is another method that comes up a lot, though not in linear regression. We'll look at it in terms of a thing called PCA ( <http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html#sklearn.decomposition.PCA> ).

PCA is a **dimensionality reduction**

from sklearn.decomposition import PCA

X = np.array([[-1, -1, 1], [-2, -1, 2], [-3, -2, 3], [1, 1, 4], [2, 1, 5], [3, 2, 6]])

pca = PCA(n\_components=2)

pca.fit(X)

PCA(copy=True, iterated\_power='auto', n\_components=2, random\_state=None, svd\_solver='auto', tol=0.0, whiten=False)

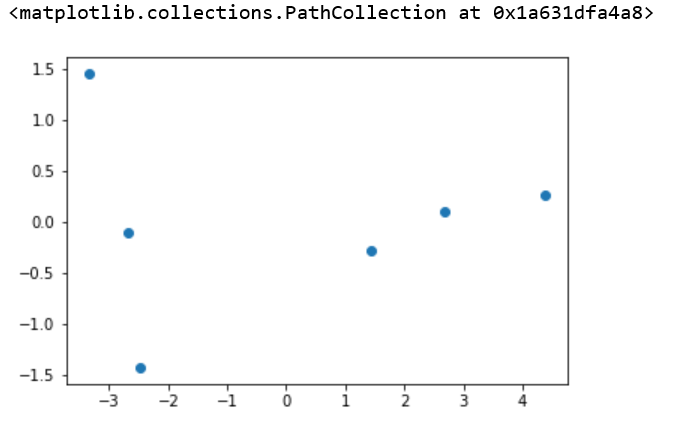
Note: 6 features with 3 data points

pca\_x = pca.transform(X)

print(pca\_x)

[[-2.48455163 -1.4257134 ] [-2.67742007 -0.10763794] [-3.33740564 1.44882813] [ 1.44194245 -0.27838915] [ 2.67742007 0.10763794] [ 4.38001481 0.25527443]]

plt.scatter(pca\_x[:,0], pca\_x[:,1])



Note: deep learning, tuning the model is difficult.

**Let's see some examples (stolen shamelessly from SkLearn's tutorials)**

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

from sklearn import neighbors, datasets

n\_neighbors = 15

# import some data to play with

iris = datasets.load\_iris()

X = iris.data[:, :2] # we only take the first two features. We could

# avoid this ugly slicing by using a two-dim dataset

y = iris.target # what species

h = .02 # step size in the mesh

# Create color maps

cmap\_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])

cmap\_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

#### Spawn the model with some parameter set!

clf = neighbors.KNeighborsClassifier(n\_neighbors, weights='distance')

#### Fit the model!

clf.fit(X, y)

# Plot the decision boundary. For that, we will assign a color to each

# point in the mesh [x\_min, x\_max]x[y\_min, y\_max].

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h),

np.arange(y\_min, y\_max, h))

### Use predict on literally every point in the plot to see what class we think it is!

Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])

##### Just making the plots after this point

# Put the result into a color plot

Z = Z.reshape(xx.shape)

plt.figure()

plt.pcolormesh(xx, yy, Z, cmap=cmap\_light)

# Plot also the training points

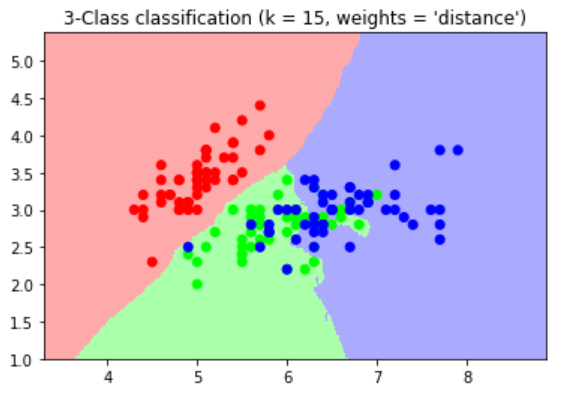
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap\_bold)

plt.xlim(xx.min(), xx.max())

plt.ylim(yy.min(), yy.max())

plt.title("3-Class classification (k = %i, weights = '%s')"

% (n\_neighbors, clf.weights));



The dots are the training data.

Region is prediction.

For example,

Hair color, look at the average

How many would you like to look at

**What about something more complicated like handwriting recognition? Same pipeline, just more data wrangling to get everything ready to go**

# Author: Gael Varoquaux <gael dot varoquaux at normalesup dot org>

# License: BSD 3 clause

# Standard scientific Python imports

import matplotlib.pyplot as plt

# Import datasets, classifiers and performance metrics

from sklearn import datasets, svm, metrics

# The digits dataset

digits = datasets.load\_digits()

# The data that we are interested in is made of 8x8 images of digits, let's

# have a look at the first 4 images, stored in the `images` attribute of the

# dataset. If we were working from image files, we could load them using

# matplotlib.pyplot.imread. Note that each image must have the same size. For these

# images, we know which digit they represent: it is given in the 'target' of

# the dataset.

images\_and\_labels = list(zip(digits.images, digits.target))

for index, (image, label) in enumerate(images\_and\_labels[:4]):

plt.subplot(2, 4, index + 1)

plt.axis('off')

plt.imshow(image, cmap=plt.cm.gray\_r, interpolation='nearest')

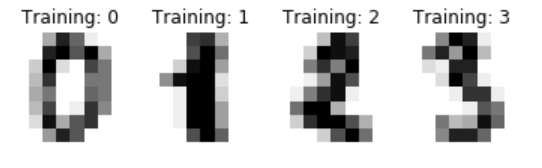
plt.title('Training: %i' % label)

# To apply a classifier on this data, we need to flatten the image, to

# turn the data in a (samples, feature) matrix:

n\_samples = len(digits.images)

data = digits.images.reshape((n\_samples, -1))



All of that was just to get the data into a format we can work with. So now it's in a format like x = [pixel\_0's strength, pixel\_1's strength, pixel\_2' strength,...] and y = [this is a 3].

Now we spawn our model (called SVC this time, again don't worry much about what an SVC is now, but the documentation is here:

<http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC>

Then we train it on the first half of our data using fit().

# Create a classifier: a support vector classifier

classifier = svm.SVC(gamma=0.001)

# We learn the digits on the first half of the digits

classifier.fit(data[:n\_samples // 2], digits.target[:n\_samples // 2])

SVC(C=1.0, cache\_size=200, class\_weight=None, coef0=0.0, decision\_function\_shape='ovr', degree=3, gamma=0.001, kernel='rbf', max\_iter=-1, probability=False, random\_state=None, shrinking=True, tol=0.001, verbose=False)

This whole section is just looking at how good our model is. At this point, our model is built already and we can just see how we did!

# Now predict the value of the digit on the second half:

expected = digits.target[n\_samples // 2:]

predicted = classifier.predict(data[n\_samples // 2:])

images\_and\_predictions = list(zip(digits.images[n\_samples // 2:], predicted))

plt.figure(figsize=(10,6))

for index, (image, prediction) in enumerate(images\_and\_predictions[:10]):

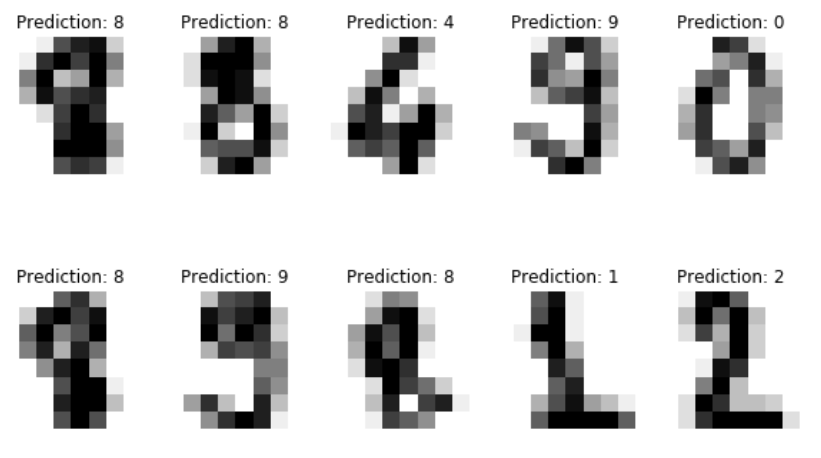
plt.subplot(2, 5, index+1)

plt.axis('off')

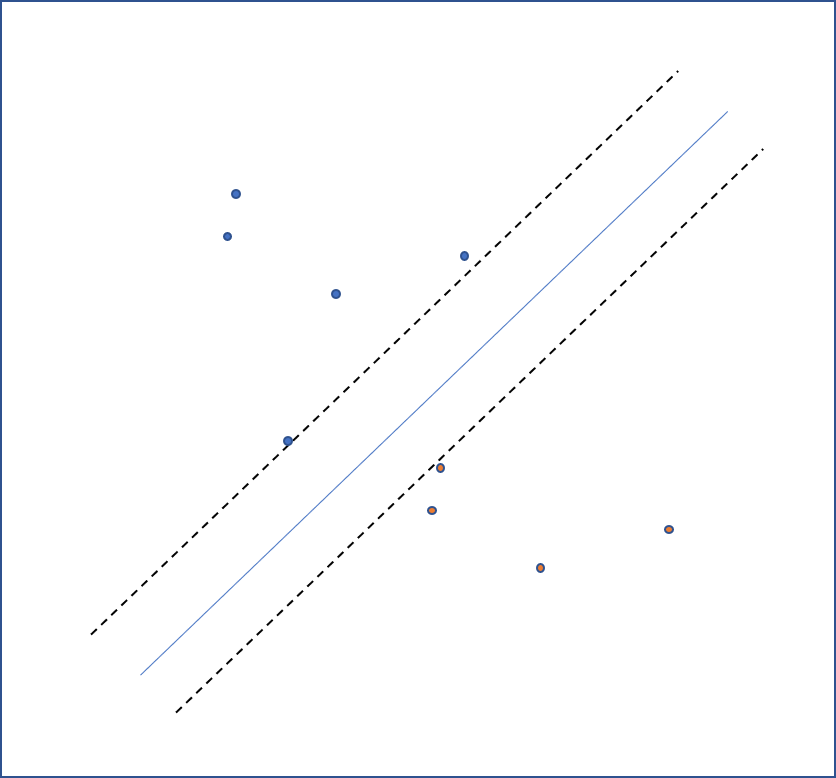
plt.imshow(image, cmap=plt.cm.gray\_r, interpolation='nearest')

plt.title('Prediction: %i' % prediction)

plt.show()



Note:



Separation of dimensions

Points that hit, are support vectors

NOTE: predicts this or that, no in-betweens

5 / 2

Output: 2.5

5 // 2

Output: 2