DIGITAL TWIN AI and Machine Learning: Numerical Programming and Reproducible Science

Prof. Andrew D. Bagdanov
andrew.bagdanov AT unifi.it

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze

29 October 2020

Outline

Introduction

My solutions

- If you are curious, you will find my solutions to the previous lab exercises here (this is the same notebook, now with solutions!):

`http://bit.ly/DTwin-ML2`

Overview

- So far we have seen a bit of the mathematics we will be using to model machine learning problems.
- We also got our hands a little dirty with some basics of numpy and how to work with numerical objects.
- Today we will take a deeper look at the numerical tools we will be using for the rest of the course.

A recap of the tools

Scikit-learn (sklearn)

- A very complete toolkit for classical machine learning.
- We will see some of the key concepts and features of sklearn today: datasets, model fitting, and data preprocessing.

Matplotlib

- A full-featured toolkit for producing high-quality plots and other visualizations.
- We will take a tour of the major types of plots we will use for analyzing data, monitoring progress, and analyzing results.

Pandas

- The Pandas library is for managing, querying, and manipulating large amounts of numerical data.
- Today we will see how to work with Pandas Dataframes and Series, and how to query Dataframes in sophisticated ways.

Tools of the Trade

Jupyter and the Notebook Paradigm

- What ties everything together in a sustainable way is the Jupyter Notebook paradigm.
- Jupyter notebooks are self-documenting: keep your notes and observations and code and plots and experimental results all in one place.
- Jupyter notebooks support rich content: you can write inline LaTeX formulas in markdown cells, embed interactive widgets.
- In this lecture (and in the whole course) we will barely scratch the surface of what is possible.
- Jupyter has become the *de facto* standard to exploratory data analysis and experimental machine learning.

A very useful Python 3 feature (that I forgot to present!)

Python format strings

- In Python 3 you can use a special syntax for structured, formatted output.
- If you prefix your strings with a `f` (e.g. f'Hello') then anything enclosed in curly braces is evaluated by Python and interpolated into the string.

```
foo = np.random.random()
print(f'This is a random number: {foo}')
print(f'This is a random array:\n {np.random.random((3,3))}')

This is a random number: 0.944148368485749
This is a random array:
[[0.88258024 0.46450465 0.9531795 ]
[0.92952619 0.10499386 0.745663  ]
[0.58531161 0.46065158 0.61286006]]
```

A few more Numpy idioms

Adding dimensions

```
B = np.array([1, 2, 3])
print(B[:, np.newaxis])
print(B[np.newaxis, :])
[[1]
 [2]
 [3]]
[[1 2 3]]
```

Constructing arrays piecemeal

```
arrays = [[1, 1], [2, 2], [3, 3]]
print(np.vstack(arrays))
print(np.hstack(arrays))

[[1 1]
 [2 2]
 [3 3]]
[1 1 2 2 3 3]
```

A few more Numpy idioms

Broadcasting: good news for Matlab `bsxfun()` haters!

```
A = np.array([[1, 1], [2, 2], [3, 3]])
v = np.array([2, 4])
print(A)
print(v * A)
print(v[:,None] * A.T)
```

```
[[1 1]
 [2 2]
 [3 3]]
[[ 2  4]
 [ 4  8]
 [ 6 12]]
[[ 2  4  6]
 [ 4  8 12]]
```

Scikit-learn: basics

- Scikit-learn (usually abbreviated sklearn) is a collection of simple and efficient tools for predictive data analysis.
- It is built on NumPy, SciPy, and matplotlib.
- Its functionality is broken down into macrocategories:
  - Classification: supervised categorical prediction.
  - Regression: supervised estimation of continuous outputs.
  - Clustering: unsupervised latent structure discovery.
  - Dimensionality reduction:
  - Model selection: hyperparameter optimization.
  - Preprocessing: normalization and data munging.
- The Scikit-learn User Guide is a great place to start.

Scikit-learn: the dataset object

A common dataset structure (dictionary)

```
from sklearn.datasets import load_boston
ds = load_boston()
print(ds.keys())
print(ds['data'].shape)
print(ds['target'])

dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])
(506, 13)
[24.  21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 15.  18.9 21.7 20.4
 18.2 19.9 23.1 17.5 20.2 18.2 13.6 19.6 15.2 14.5 15.6 13.9 16.6 14.8
...
 8.1 13.6 20.1 21.8 24.5 23.1 19.7 18.3 21.2 17.5 16.8 22.4 20.6 23.9
 22.  11.9]
```

- Note: the convention is each row corresponds to a data point.

Scikit-learn: some datasets

- Scikit-learn provides a range of common toy datasets that we will work with:
  - Boston Housing Prices: A regression dataset with 13 numeric/categorical predictive variables, one continuous target (`sklearn.datasets.load_boston()`)
  - The Iris Plant Dataset: A classification dataset with 4 numeric predictive variables and one categorical target in three classes (`sklearn.datasets.load_iris()`)
  - Handwritten digits: An image classification dataset consisting of 8×8 pixel images of digits $[0, 1, \ldots, 9]$ \ (`sklearn.datasets.load_digits()`).

Scikit-learn: data preprocessing

Standardization

- Standardization is a common requirement for many machine learning models – they often behave badly individual features are badly scaled.

```
from sklearn import preprocessing
from sklearn.datasets import load_iris

ds = load_iris()
print(f'Original means: {ds["data"].mean(0)}')
print(f'Original sdevs: {ds["data"].std(0)}')

scaled = preprocessing.scale(ds['data'])
print(f'Scaled means: {scaled.mean(0)}')
print(f'Scaled sdevs: {scaled.std(0)}')
```

```
Original means: [5.84333333 3.05733333 3.758      1.19933333]
Original sdevs: [0.82530129 0.43441097 1.75940407 0.75969263]
Scaled means: [-1.6903e-15 -1.8429e-15 -1.6986e-15 -1.4092e-15]
Scaled sdevs: [1. 1. 1. 1.]
```

Scikit-learn: data preprocessing

- Normalization means scaling samples to have unit norm – this is useful for methods using dot products to measure similarity.

```
print(np.sqrt((scaled ** 2.0).sum(1))) # Each row is a vector sample!
normalized = preprocessing.normalize(scaled)
print(np.sqrt((normalized ** 2.0).sum(1)))
```

```
[2.31866282 2.20238668 2.38940142 2.37838853 2.47614211 2.55473374
 2.46767902 2.24585711 2.59157687 2.24883352 2.41964028 2.33563766
...
 1.01457837 1.9812829  2.16357595 2.08082472 1.45321046 2.24340213
 2.35353117 1.96614407 1.81690378 1.55981534 1.94398848 1.1086448 ]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
...
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1.]
```

Scikit-learn: models and model fitting

First an example:

```
from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression

ds = load_boston()
X = ds['data']      # X are our independent variables.
y = ds['target']    # y is our dependent variable.

# Instantiate the model, and fit() it to our data.
model = LinearRegression()
model.fit(X, y)
predictions = model.predict(X)
RMSE = np.sqrt(np.mean((y - predictions)**2))
print(f'Coefficients: {model.coef_}')
print(f'y-intercept:  {model.intercept_}')
print(f'RMSE: {RMSE}')
```

Scikit-learn: models and model fitting (continued)

Which produces an output:

```
Coefficients: [-1.0801e-01  4.6420e-02  2.0554e-02  2.686e+00
 -1.7766e+01  3.8098e+00  6.9222e-04 -1.4755e+00
  3.0604e-01 -1.2334e-02 -9.5274e-01  9.3116e-03
 -5.2475e-01]
y-intercept:  36.459488385089855
RMSE: 4.679191295697281
```

- There's a lot going on here (after loading dataset and extracting variables):
  - First we instantiate a regression model `LinearRegression` – this is an object that holds parameters and provides methods for...
  - Then we fit the model parameters with the `fit()` method.
  - Finally we apply the trained model to data in `X` by calling the `predict()` method.

Scikit-learn: models and model fitting (continued)

- You should almost always use the `predict()` method to apply trained models.
- But, sklearn objects allow us to extract trained parameters if we want them.
- We have a vector of 13 coefficients (corresponding to the 13 independent variables in `X`) and one scalar y-intercept (the bias).
- Our regression model is:

$$y = \mathbf{W}\mathbf{x} + b$$

- So if we want to we can manually compute predictions like this:

```
model.coef_ @ X.T + model.intercept_
```

Scikit-learn: metrics

- How do we know how well our model has fit the data?
- In the `sklearn.metrics` package you will find a vast number of different metric used for evaluating performance of many types of models.
- For regression, measures of error or correlation are usually used:

```
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
print(np.sqrt(mean_squared_error(predictions, y)))
print(mean_absolute_error(predictions, y))
```

```
4.679191295697281
3.2708628109003115
```

Matplotlib: the visualization workhorse

- The basic (and not so basic!) plotting library for numerical programming in Python is Matplotlib.
- Matplotlib is a Python 2D plotting library which produces publication quality figures.
- It can be used in Python scripts, the Python and IPython shells, and Jupyter notebooks.
- There is a HUGE number of addons and extensions to Matplotlib.

The short version

```
import matplotlib.pyplot as plt # Standard import alias.
plt.plot([1, 2, 3, 4])
plt.plot([1, 2, 3, 4] ** 2)
plt.ylabel('some numbers, and some numbers squared')
plt.show() # Not always needed (e.g in a notebook
```

Matplotlib: plotting nultiple sets of data

- Matplotlib is a state-based plotting library designed to be familiar to Matlab users.
- If you issue multiple plot commands, the default is to plot them on the same axes.
- Let's look at how we can use this to analyze the behavior of my gradient descent implementation.

Matplotlib: plotting multiple sets of data (continued)

My implementation

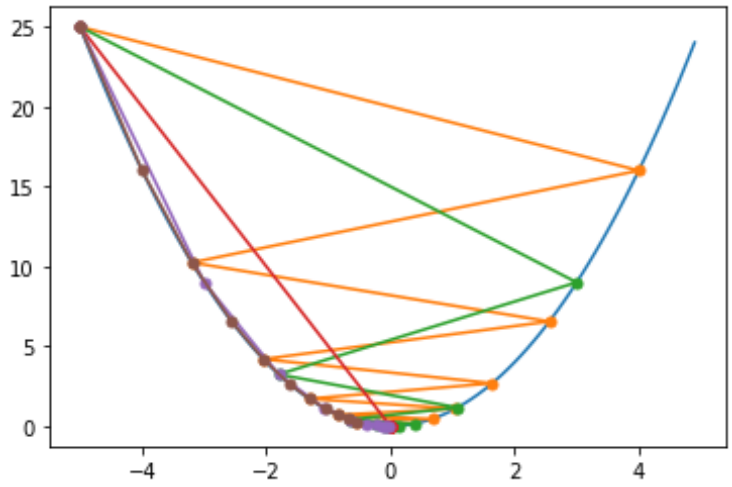
- Here is my implementation of steepest descent (as a function).
- Note how I have instrumented this code to collect the sequence of intermediate solutions.

```
# First refactor code into a function.
def gd(f, f_prime, x0=0.0, eta=0.1, maxiter=10):
  x_star = x0           # Initialize initial "guess".
  solutions = [x_star] # This list will track our solution.

  # Standard descent loop.
  for it in range(maxiter):
    x_star -= eta * f_prime(x_star)
    solutions.append(x_star)

  # Return current solution and list of intermediate solutions.
  return (x_star, np.array(solutions))
```

Matplotlib: plotting multiple sets of data (continued)

A visual analysis

```
rng = np.arange(-5, 5, 0.1)
plt.plot(rng, tpara(np.arange(-5, 5, 0.1)))
for eta in [0.9, 0.8, 0.5, 0.2, 0.1]:
  (solution, intermediates) = gd(tpara, tpara_prime, eta=eta, x0=-5)
  plt.plot(intermediates, tpara(intermediates), '.-', ms=10)
```



- I often use these types of quick and dirty plots to gain insight.

Matplotlib: reading the FM

- I should comment at this point that the Matplotlib documentation is extensive.
- For even simple plotting functionality, the functions provided by Matplotlib provide a vast number of features.

matplotlib.pyplot.plot

matplotlib.pyplot.plot(*args, scalex=True, scaley=True, data=None, **kwargs) [source]

Plot y versus x as lines and/or markers.

Call signatures:

```
plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

The coordinates of the points or line nodes are given by *x*, *y*.

The optional parameter *fmt* is a convenient way for defining basic formatting like color, marker and linestyle. It's a shortcut string notation described in the *Notes* section below.

```
>>> plot(x, y)        # plot x and y using default line style and color
>>> plot(x, y, 'bo')  # plot x and y using blue circle markers
>>> plot(y)           # plot y using x as index array 0..N-1
>>> plot(y, 'r+')     # ditto, but with red plusses
```

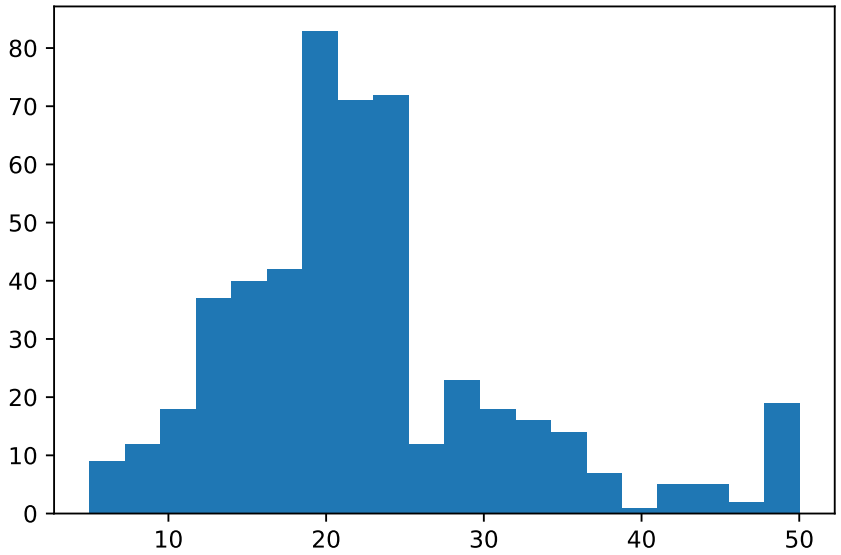
You can use Line2D properties as keyword arguments for more control on the appearance. Line properties and *fmt* can be mixed. The following two calls yield identical results:

```
>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o', linestyle='dashed',
...      linewidth=2, markersize=12)
```

Matplotlib: histogramming

- Of course, using `plt.plot()` presumes that we have naturally ordered data.
- Usually this is not the case – in fact, a large part of ML and data science is about finding structure in our data.
- A useful tool for gaining insight about the distribution is the histogram.
- A histogram quantizes univariate data into fixed-width bins and counts the frequency of each discretized value.
- In Matplotlib the function we want is `plt.hist()`:

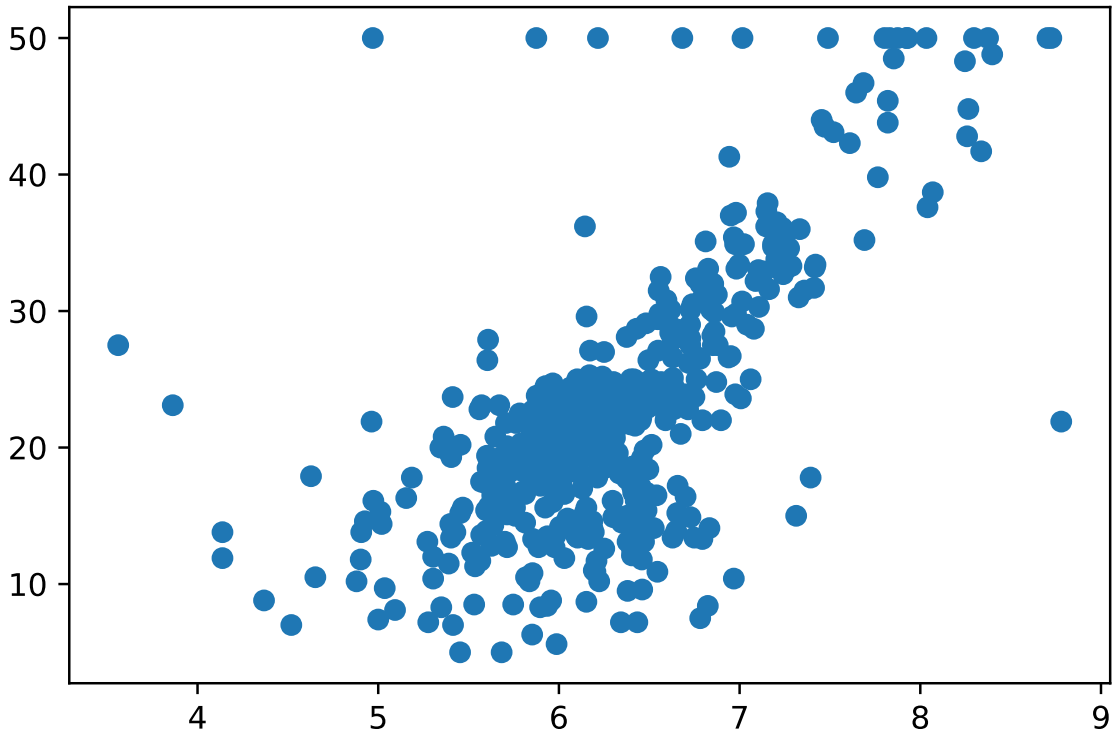
```
plt.hist(y, bins=20)
```



Matplotlib: scatterplotting

- Of course, if we have independent variables in our data (like for our regression problem) we can use an independent variable to induce an order.
- What we want here is called a scatter plot (`plt.scatter()` in Matplotlib).
- This puts the independent variable on the x-axis and the target on the y-axis.

```
plt.scatter(X[:,5], y)
```

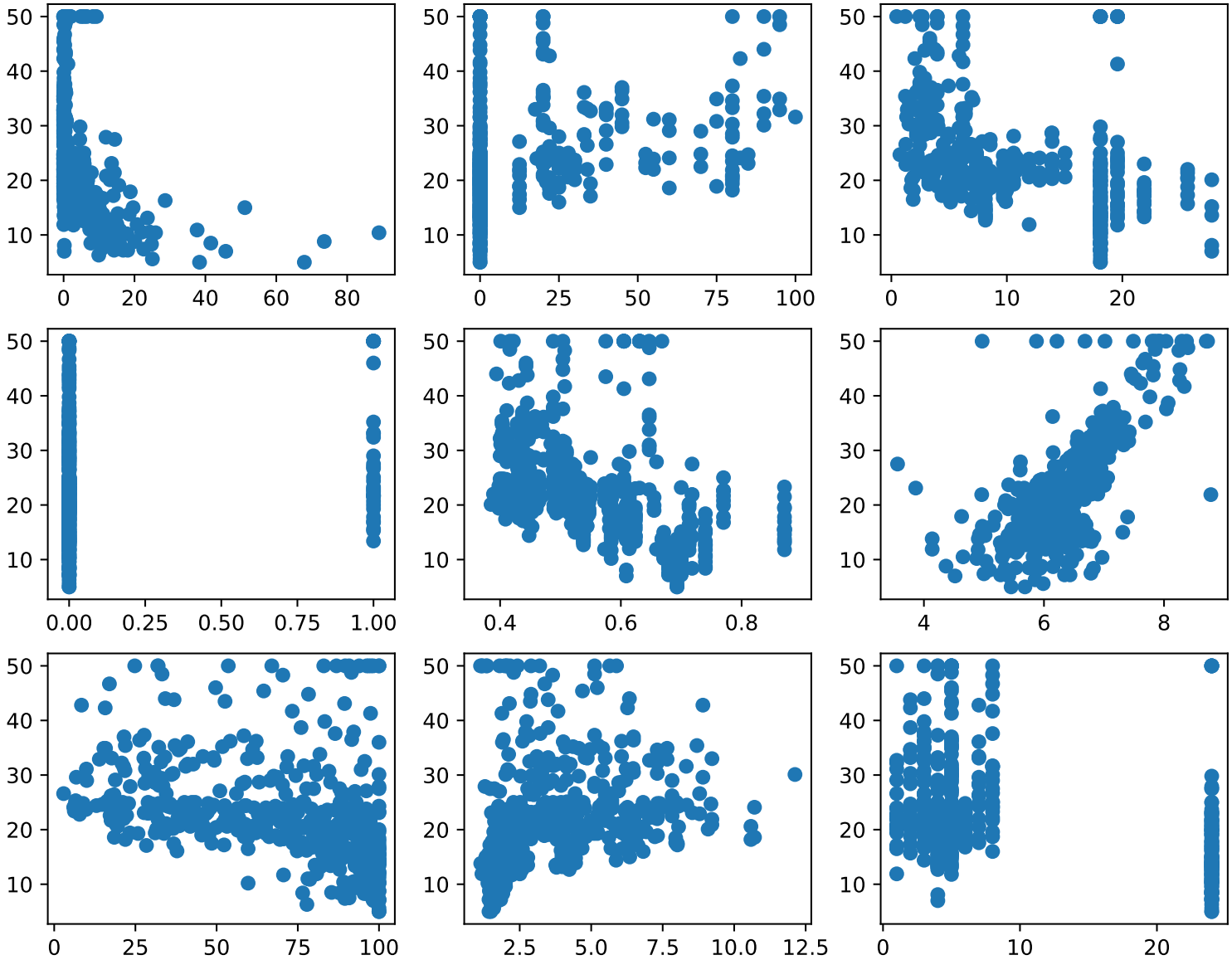


Matplotlib: subplots

- But we have 13 dependent variables in our Boston dataset...
- Matplotlib allows us to create figure containing subplots using the `plt.subplot()` function.

```
plt.figure(figsize=(10, 8))
for p in range(0, 9):
  plt.subplot(3, 3, p+1)
  plt.scatter(X[:,p], y)
```

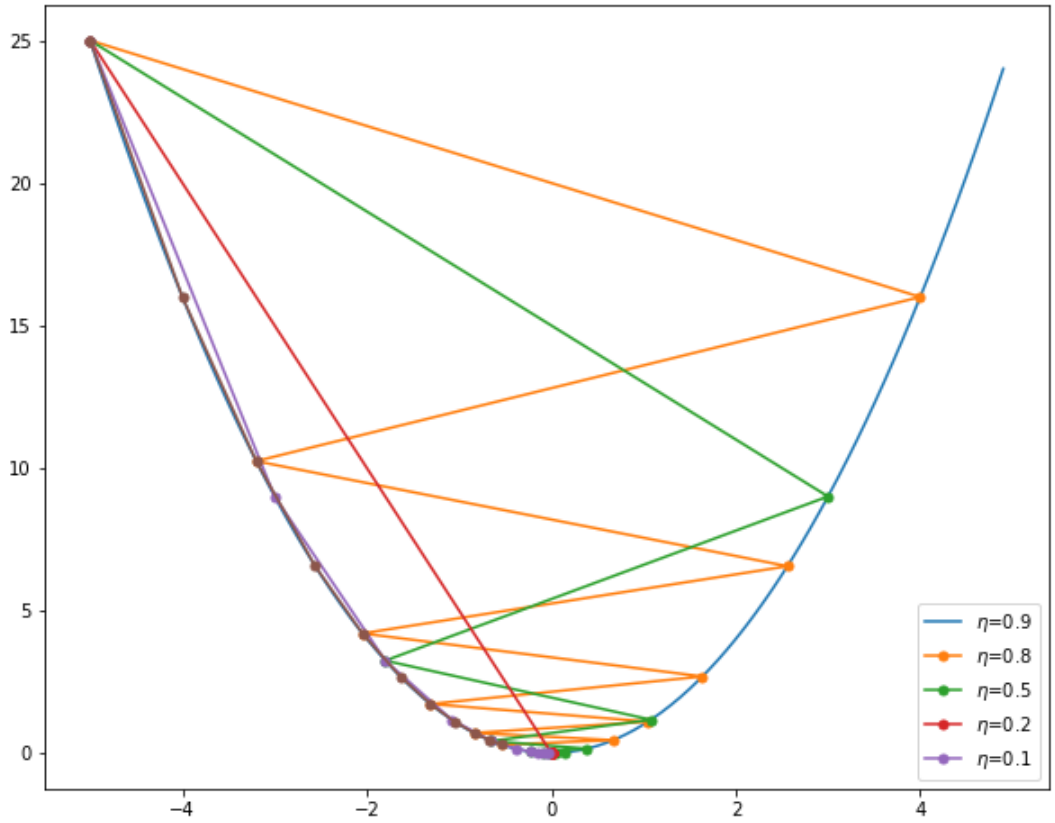
Matplotlib: subplots (continued)



Matplotlib: making things pretty

- Our plots so far are useful, but they have a big defect: they look like crap.
- Let's prettify them a bit with useful annotations.

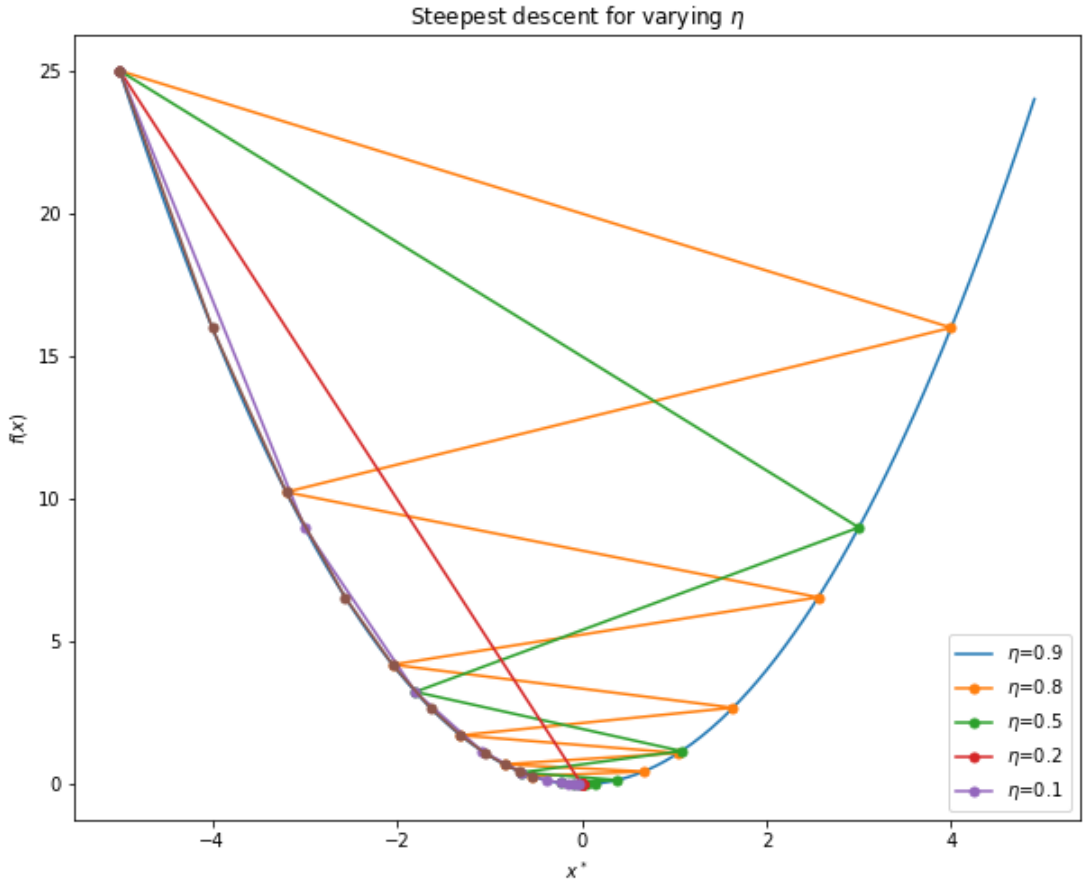
```
plt.legend(labels=[f'$\eta$={eta}' for eta in [0.9, 0.8, 0.5, 0.2, 0.1]],
           loc='lower right')
```



Matplotlib: making things pretty (continued)

- And we can add labels and titles (a la Matlab):

```
plt.xlabel('$x^*$')
plt.ylabel('$f(x)$')
plt.title('Steepest descent for varying $\eta')
```



Managing Data

Pandas: overview

- The Python Pandas library is the most preferred tools for data scientists for data manipulation and analysis.
- Along with Matplotlib and Numpy it is a fundamental library for scientific computing in Python.
- It provides fast, flexible, and expressive data structures to facilitate easier data analysis.
- It provides so much functionality, in fact, that is can be overwhelming to new users.
- Here I will give a high-entropy overview of the important features and concepts.

Pandas: the three important concepts

DataFrame

- A Pandas `DataFrame` is a two-dimensional, size-mutable, tabular data structure with labeled axes (rows and columns).
- Arithmetic operations align on both row and column labels.
- It is a dictionary container for `Series` objects – you can think of it as a wrapped array.

Series

- A Pandas `Series` is a one-dimensional with axis labels (an index).
- Series support both integer- and label-based indexing, and wrap most of the `np.array` functionality.
- Statistical methods from ndarray have been overridden to automatically exclude missing data.

Pandas: the three important concepts (continued)

Index

- A Pandas `Index` is an immutable array implementing an ordered, sliceable set.
- It is the basic object storing axis labels for Pandas.
- Note: we will almost always use integer indexes (i.e. row indices).

Pandas: basic DataFrame usage

- Let's create and DataFrame from a random array.

```
import numpy as np
import pandas as pd

data = np.random.random((1000, 4))
df = pd.DataFrame(data, columns=['A', 'B', 'C', 'D'])
df.describe()
```

	A	B	C	D
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.510549	0.503621	0.495730	0.471495
std	0.283081	0.290694	0.293699	0.287029
min	0.001521	0.000076	0.001319	0.000416
25%	0.273882	0.258171	0.232633	0.227963
50%	0.517267	0.499417	0.492677	0.466878
75%	0.752321	0.745992	0.754617	0.714475
max	0.999547	0.998621	0.999480	0.997759

Pandas: basic `DataFrame` usage (continued)

- A nice feature of Pandas is that it keeps track of column labels (and indexes) when performing computations:

```
print(df.min(0),'\n'); print(df.max(1)); print(type(df.min(1)))
```

```
A    0.001521
B    0.000076
C    0.001319
D    0.000416
dtype: float64

0      0.937014
1      0.570628
         ...
998    0.842050
999    0.735416
Length: 1000, dtype: float64
<class 'pandas.core.series.Series'>
```

Pandas: basic usage (continued)

- Perhaps most importantly, we can use column names to index:

```
# Indexing is *semantic* and flexible.
df['A']                    # Returns a Series
df.A                       # Returns the *same* series.
newdf = df[['A', 'B']] # Returns a DataFrame

# ADDs a new column to newdf derived from A and B.
newdf['A/B'] = newdf.A / newdf.B

# We can filter rows using boolean queries.
newdf[newdf['A/B'] > 100.0]
```

Out[83]:

	A	B	A/B
66	0.825944	0.003580	230.719818
334	0.713525	0.000453	1576.654724
537	0.374274	0.001938	193.141976
812	0.892326	0.008207	108.728546
816	0.309119	0.000076	4078.517572
990	0.741138	0.002853	259.750430

Pandas: datasets revisited

- Especially if you are working with a lot of data, you will probably want to convert your data into a Pandas `DataFrame`.
- Most of the tools (Matplotlib, sklearn) are able to transparently work with Pandas.
- Pandas has a ton of import/export functions for reading/writing to/from CSV, Excel, and other formats.
- Note: when importing – especially from CSV – make sure you get what you really want for column names.
- ALSO Note: if you import data from external formats often the targets will be included in the resulting `DataFrame` – this is usually a Very Bad Idea.

```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris
df = pd.read_csv(url)
```

The rules of the Game

Overview

- In this last part of the lecture today I want to give a simple recipe that we will try to follow for most ML problems we will look at.
- These are a loose sort of best practices you can follow when working with data.
- There is, of course, no strict set of rules that you can (or should) blindly follow.
- In fact, one of the overarching objectives of this course is to give you some exposure and experience to a set of tools and techniques.
- Enough so that you can develop your own practices and draw well-founded conclusions about your own learning and data analysis problems.

Step 1a: Data design

- The first step, of course, is to get your hands on some data.
- Depending on the data, you should design a Pandas `Dataframe` to encapsulate it.
- Think about data types – are features continuous, categorical, or discrete.
- Pick good names for the columns in your dataset – you will be using them a lot, `['A', 'B', ..., 'Z']` is probably a bad idea.

Step 2b: Getting a feel for the data

- When you have data in a `DataFrame`, you can start to get a feel for it.
- Look at the descriptive statistics that `describe()` gives you.
- What you are looking for are surprises and insights.
- Are there any undefined values in your data? How will you deal with them?

Step 2c: Visualization

- An important tool for data exploration is visualization.
- This is especially true if you have very many features (high dimensionality).
- Use simple plots, scatter plots, and histograms to get a better picture of the nature and behavior of your data.
- More advanced plotting capabilities can be employed as needed:
  - Seaborn: prettier plots, better Pandas integration
  - Bokeh: interactive plotting widgets, great for exploration.

Step 2d: Normalization and standardization

- Are some (or all) features badly scaled with respect to others?
- Do you have categorical variables that might need to be embedded or mapped to other spaces?
- You might think about standardization or normalization at this point.
- However, the decision about how to preprocess data is often intimately tied to downstream modeling decisions.

Step 2e: Iterate

- Finally, repeat.
- What you discover via visualization and data exploration can often change how you decide to model your data.
- It is most important to ensure you understand your data and have a good data model going forward.
- Take your time.

Step 2a: Decide how to model your problem

- What type of learning problem are you faced with?
- Is it supervised (do you have target values?):
  - Is it a regression (continuous target) problem?
  - Is it a classification (categorical outputs) problem?
  - During exploratory data analysis (step 1) you should have acquired an idea of which features are correlated with targets.
- Is it an unsupervised learning problem (do you only have blobs of data?):
  - In this case during exploratory data analysis you should have acquired an idea if there is latent structure to learn.

Step 2b: Pick a model parameterization

- Depending on which type of learning problem (supervised or unsupervised), you can now think about selecting a model to try.
- Do there appear to be simple and linear correlations between features and targets?
- Or, is the correlation structure not immediately evident (which might indicate that linear models won't work)?
- Whichever model you start with, you should have a good idea of what the model parameters are that will be estimated.
- General advice: start with a simple model and gradually increase complexity.

Step 2c: Understand hyperparameters

- Most models, in addition to the learnable parameters, will have one of more hyperparameters.
- Some of these are architectural choices (e.g. whether to fit both slope AND y-intercept in a regression).
- Some will be continuous parameters that cannot be fit by gradient-based optimization (e.g. regularization wights).
- The important thing here is to be aware of what hyperparameters exist and to pick reasonable defaults.

Step 2d: Understand how to evaluate

- Finally, we need to know how to evaluate the performance of our models.
- For regression, this might be a simple RMS error.
- For classification, you might be interested in accuracy.
- This can be a delicate decision, however.
- Question: let's say you have an unbalanced binary classification problem (one class has 1000x more example than the other). Why might accuracy not be a good choice?

Step 3a: The very least: training/testing

- This might be the easiest, but MOST IMPORTANT step.
- Whenever you are working with machine learning you MUST be sure to work with *independent training and testing* sets.
- These are usually referred to as splits:
  - Training split: a randomly chosen portion (say, 75%) of the data you set aside ONLY for estimating model parameters.
  - Testing split: a portion (the remaining 25%) of the data you use ONLY FOR EVALUATING PERFORMANCE.
- Very important: using independent training/testing splits like this is the *only way to guarantee generalization*.

Step 3a: Even better: training/validation/testing

- If you have enough data, an even better way is to have three splits:
  - Training split: a randomly chosen portion (say, 60%) of the data you set aside ONLY for estimating model parameters.
  - Validation split: a randomly chosen portion (50% of the remaining data) used to monitor learning and to select hyperparameters.
  - Testing split: a portion (the remaining part) of the data you use ONLY FOR EVALUATING PERFORMANCE.
- Later we will see how cross-validation techniques can be used to make the most of available data without violating the independence of train/validation/test splits.

Step 3a: ALWAYS obey this rule

- If you want to draw conclusions about the performance of your models, you must use independent splits.
- *I cannot emphasize this enough*.

Step 4: Fit your model, evaluate, repeat.

- Now we can actually start doing some machine learning.
- Frameworks like sklearn provide tools with a consistent API (e.g. `model.fit()` to estimate parameters).
- Frameworks like sklearn also usually provide most of the evaluation (and splitting) functions you need.
- We usually talk about building a pipeline that, given data and values for hyperparameters:
  1. Fits the model to the training data.
  2. Evaluates the model on the test (or validation) data.
  3. Visualizes model output and/or performance as appropriate.
- Having a pipeline allows us to repeatably perform experiments with different hyperparameters, with different data, etc.

Reflections

An art and a science

- Data science is really part science and part art.
- It is an experimental science in this respect:
  - We formulate hypotheses about the nature and behavior of our data.
  - This can exploit prior knowledge about the source of the data.
  - And it can involve insight gained through exploratory data analysis.
  - We then design experiments to validate our hypotheses.
  - We perform (often many) experiments to confirm or refute our hypotheses.
  - To do this, we need to bring a wide array of tools and techniques to bear.

Laboratory

- The laboratory notebook for today:

`http://bit.ly/DTwin-ML3`