Machine Learning

Prepping Data

Numpy Intro

Slides adapted from material created by E. Alpaydin
Prof. Mordohai, Prof. Greenstadt, Pattern Classification (2nd Ed.),
Pattern Recognition and Machine Learning

Objectives

- Preparing Data
- Numpy Intro

The Data

- The first thing we'll need to do in this class is get data!
- Data can come in many different forms.
- However, most algorithms require an *observed data matrix*, X
- Each *row* of X pertains to an *observation.*
- Each *column* pertains to information for the observations.
  - Knows as a *feature*

$$X = \begin{array}{cc} \text{Feature \#1 (car)} & \text{Feature \#2 (age)} \\ \left[\begin{matrix} Honda & 2 \\ Ford & 20 \\ Toyota & 3 \\ Toyota & 1 \end{matrix}\right] & \end{array} \left.\vphantom{\begin{matrix} a \\ a \\ a \\ a \end{matrix}}\right\} \text{Four observations}$$

Feature Types

- As mentioned in the introduction, we categorize each feature as one of three types:
  1. Continuous valued
  2. Categorical-Nominal
  3. Categorical-Ordinal
- Depending on the algorithm being used we may need to
  - Convert categorical features into a set of binary features
  - Convert continuous features to categorical-ordinal
  - “Standardize” features so that they have the same range

Prepping Data

- Categorical→ Binary
  - If our algorithm requires (or works best with) binary features, we need to *one-hot-encode* our categorical features to be binary features.
  - *One-hot-encoding* means to create a set of binary features, one per category, and for any given observation only one of those features has a value of one (all the rest have a value of zero).
  - Example:

$$X = \begin{bmatrix} Honda \\ Ford \\ Toyota \\ Toyota \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \\ 1 \\ 2 \\ 2 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Prepping Data

- Continuous → Categorical-Ordinal
  - Some algorithms work better/easier on categorical-ordinal data.
  - If we just have a single threshold, then we are converting our continuous features into binary features.
  - If we have multiple thresholds, then we can turn them into categorical-ordinal features.

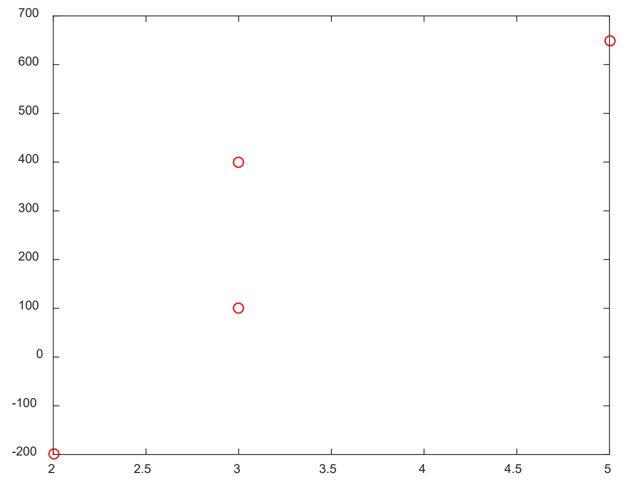
Standardizing Data

- Standardizing Features
  - Some algorithms work best when all features to have mean and standard deviation across the observations.
  - Otherwise, a feature might dominate just due to its scale.
- Example: Imagine two features for a person
  - Height
  - Weight
- These are both on very different scales
  - Height (inches): Maybe 12 → 80
  - Weight (lbs): Maybe 5→400
- If we used the data as-is, then one feature may have more influence than the other

Standardizing Data

- Standardized data has
  - Zero mean
  - Unit deviation
- We treat each feature independently and
  1. Center it (subtract the mean from all samples)
  2. Make them all have the same span (divide by standard deviation).
- This is often referred to as **zscoring** our data
- Example

```
X = [3, 400;
     2, -200;
     3, 100;
     5, 650];
figure(1);
plot(X(:,1),X(:,2),'or');
```

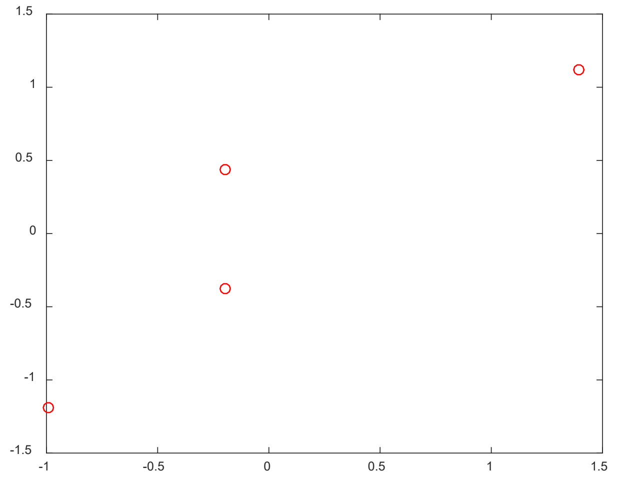


Standardizing Data

- We can compute the mean and standard deviation of each feature easily and then subtract the means from each observation and divide each (centered) observation by the standard deviation of each feature.
- Would it make sense to do this with categorical (nominal and/or ordinal) data?
- How about if it was made into a binary feature set?

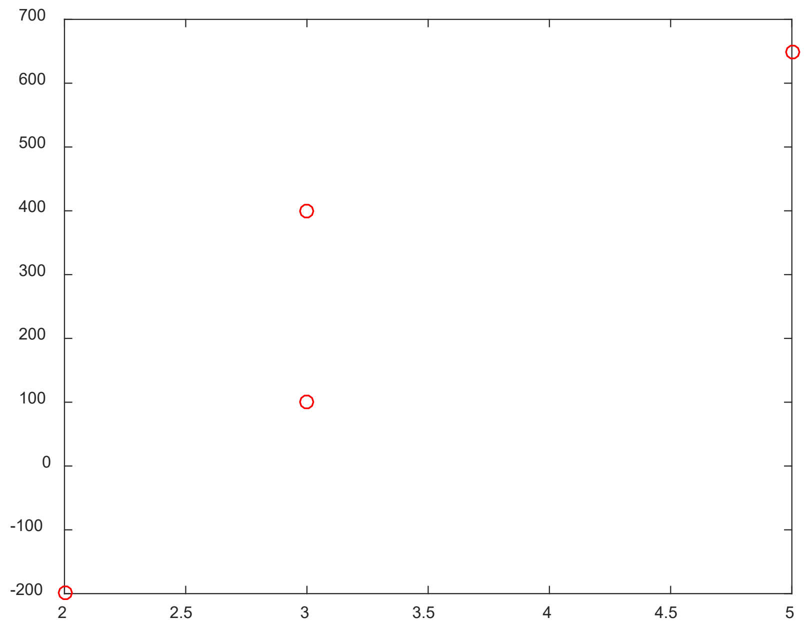
```
m = mean(X);  %get mean of each feature
s = std(X);   %get std of each feature
%m = 3.2500  237.5000
%s = 1.2583  368.2730

X = X - repmat(m,size(X,1),1);
X = X./repmat(s,size(X,1),1);
figure(2);
plot(X(:,1),X(:,2),'or');
```

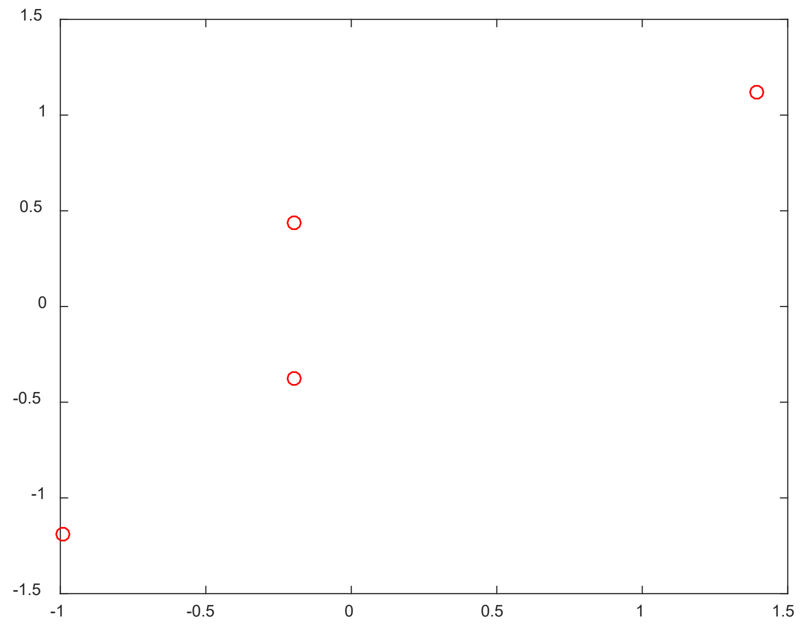


Standardizing Data

Original



Standardized



Note on Standard Deviation

- There are two commonly used, slightly different computations for standard deviation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$$

- Typically, the first version is to be used if we got the mean, μ, somehow other than computing it from the data itself.
- The second version adjusts for this lack of independence of σ and , u if in fact μ was computed as $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$
- For this course, we will typically want to use the version that divides by $N - 1$

Note: Python's std defaults to using N instead of $N - 1$, use it with `ddof=1`!

Target Values

- In addition to the observed data, X, for *supervised* learning problems we also know the *target values.*
- We typically store the target values in a matrix Y, where again each row pertains to an observation.
    - If we have several target values for our observations, then each column pertains to a target.

Numpy

- In the course we'll be using the *numpy* library for storing and manipulating our data and doing many numerical computations.
  - Note that we **will not** be using ML libraries like Keras or Tensorflow.
  - Usage of these on assignments will result in zeros.
- First thing you'll need to do is install numpy
  - `pip install numpy`
- It's important that throughout we keep our data as *matrices.*
  - Otherwise numpy will perform other (vector) types of operations.

Numpy Matrices

- Now we can create our data

```
import numpy as np
X = np.array([[3, 400],[2, -200],[3, 100],[ 5, 650]]);
print(X.shape)  #(4,2)
```

- Let's try to multiply the first row of this with the transpose of the rest?

$$[3 \quad 400]\begin{bmatrix} 2 & 3 & 5 \\ -200 & 100 & 650 \end{bmatrix} = [-79994 \quad 400009 \quad 260015]$$

- How can we do this in numpy?

```
np.atleast_2d(X[0,:])@X[1:4,:].T
```

Numpy Matrices

```
np.atleast_2d(X[0,:])@X[1:4,:].T
```

- A few syntax things:
  - @ is an overloaded operator for the `matmult` method.
  - `.T` is an alias for the `transpose` method.
  - `np.atleast_2d` forces `X[0,:]` to be a matrix
    - Actually, not an issue in this example, but without it the shape of $X[0,:]$ is $(2,)$ which is odd, and can cause issues with later computations.

Numpy Axes and Statistics

$$X = \begin{bmatrix} 3 & 400 \\ 2 & -200 \\ 3 & 100 \\ 5 & 650 \end{bmatrix}$$

- Of we were to do `X.shape` we'd get the *dimensions* of each *axis*
  - The first axis, axis=0, is the rows.
  - The second axis, axis=1, is the columns.
- If we want to get the mean *over* each row (axis=0) we can do:

```
np.mean(X,axis=0,keepdims=True)  #[3.25, 237.5]
```

- And likewise, the standard deviation over each row, but with a loss of degree of freedom since we're using the sample mean

```
np.std(X,axis=0,ddof=1, keepdims=True) #[1.26, 368.27]
```

- And finally, our covariance matrix should be 2×2, so to compute the covariance using the columns as variables:

```
np.cov(X,rowvar=False,ddof=1)
```

Numpy Broadcasting

$$X = \begin{bmatrix} 3 & 400 \\ 2 & -200 \\ 3 & 100 \\ 5 & 650 \end{bmatrix}$$

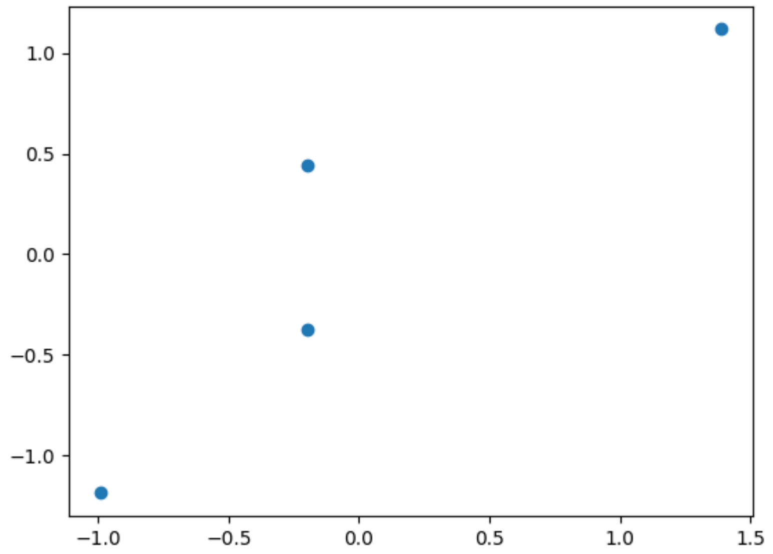
- Now maybe we want to use our means and standard deviations to z-score our data.
- Formally, since $X \in \mathcal{R}^{4\times 2}$ to do $(X-M)/S$ both M and S should be 4×2 matrices.
  - But they're $\in \mathcal{R}^{1\times 2}$
- However, if there are dimensionality mismatches, numpy attempts to *broadcast* (i.e copy) data to make things work.

```
m = np.mean(X,axis=0,keepdims=True)  #1x2
s = np.std(X,axis=0,ddof=1,keepdims=True)  #1x2
Xp = (X-m)/s  #4x2 via broadcasting
```

Matplotlib

- To do our visualization, we can use matplotlib

```
    import matplotlib.pyplot as plt
    plt.scatter(Xp[:,0],Xp[:,1])
    plt.show()
```



1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.5