

CSC380: Principles of Data Science

Nonlinear Models 1

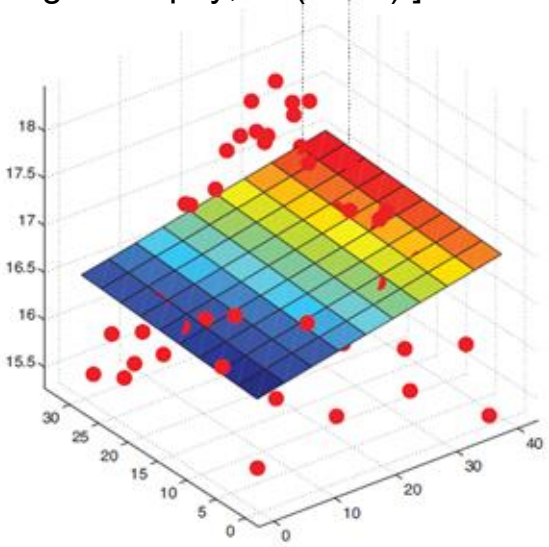
Naren Doraiswamy

Outline

- Support Vector Machine
- Basis Functions

Linear Models

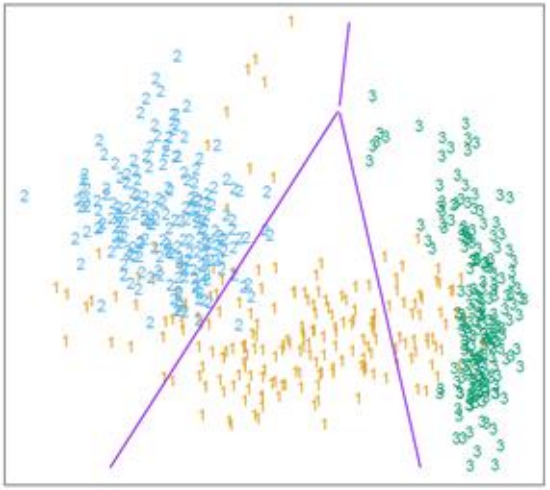
[Image: Murphy, K. (2012)]



Linear Regression Fit a *linear function* to the data,

$$y = w^T x$$

[Image: Hastie et al. (2001)]

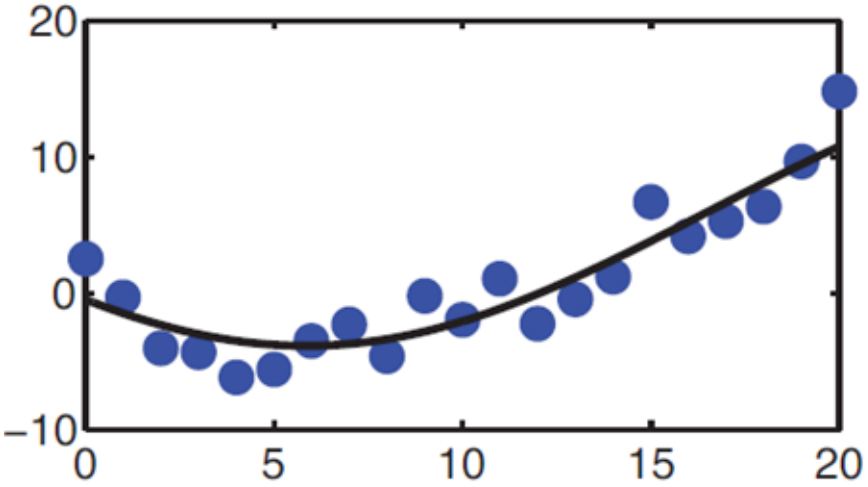


Logistic Regression Learn a decision boundary that is *linear in the data*,

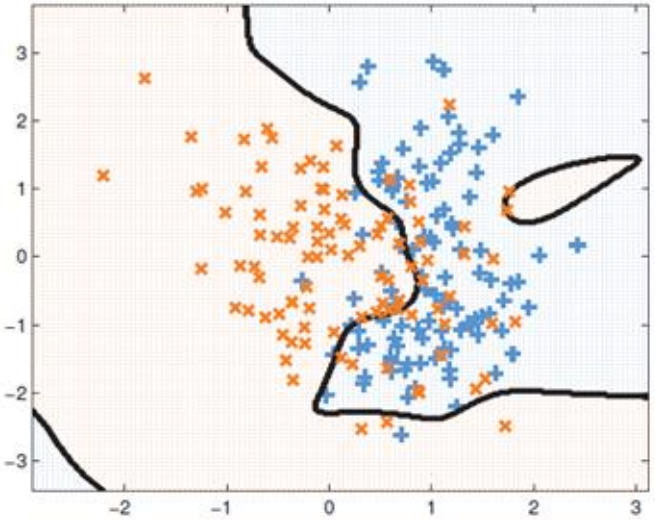
For two classes: $p = \frac{1}{1+e^{w^T x+b}} = 0.5$

$$y = \mathbf{I}\{w^\top x \geq 0\}$$

Nonlinear Data



What if our data are *not* well-described by a linear function?



What if classes cannot be well-distinguished by a linear function?

Example: Earthquake Prediction

Suppose that we want to predict the number of earthquakes that occur of a certain magnitude.

FIGURE 5-3A: WORLDWIDE EARTHQUAKE FREQUENCIES, JANUARY 1964–MARCH 2012

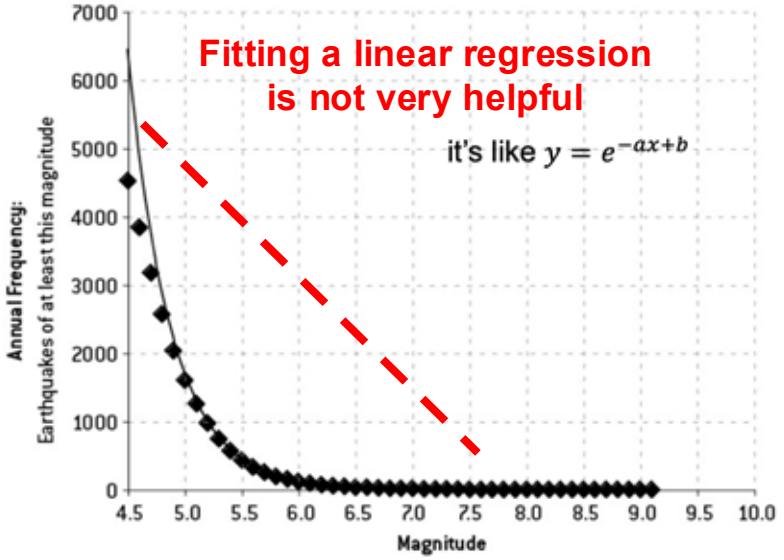
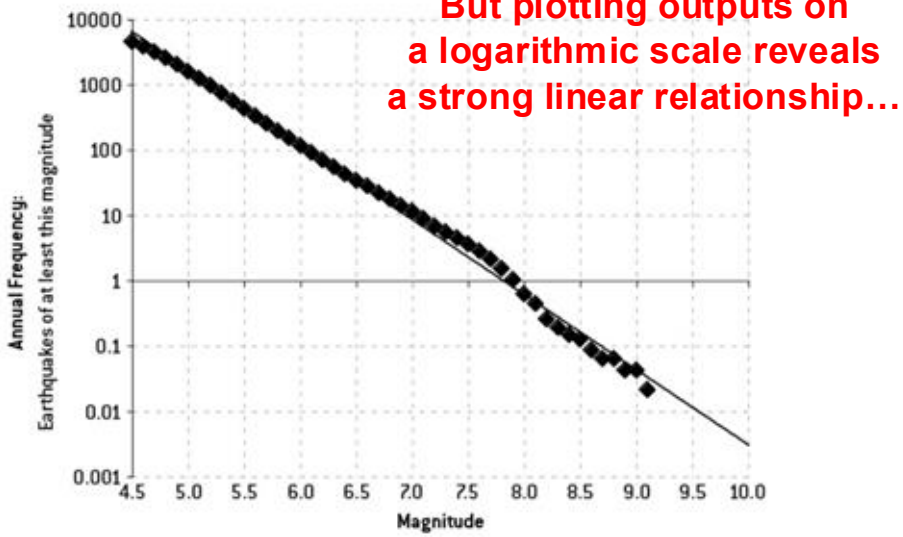


FIGURE 5-3B: WORLDWIDE EARTHQUAKE FREQUENCIES, JANUARY 1964–MARCH 2012, LOGARITHMIC SCALE



Beyond linearity: Transformation in x

- Recall: for 1d problem, we embedded the feature: $x' = (x, 1) \in \mathbb{R}^2$ so we can encode the intercept term.

$$\phi_0(x) = 1 \qquad \phi_1(x) = x \qquad y = \boldsymbol{w}^{\mathsf{T}}\Phi_{\text{lin}}(x) = \phi_0(x)w_0 + \phi_1(x)w_1 = w_0 + w_1 x$$

Beyond linearity: Transformation in x

- Recall: for 1d problem, we embedded the feature: $x' = (x, 1) \in \mathbb{R}^2$ so we can encode the intercept term.

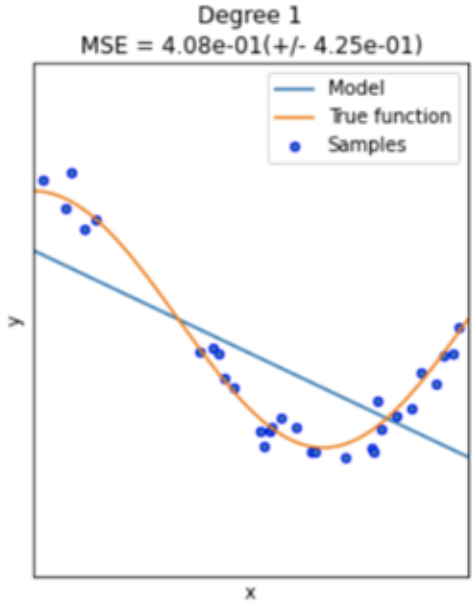
$$\phi_0(x) = 1 \qquad \phi_1(x) = x \qquad y = \boldsymbol{w}^{\mathsf{T}}\Phi_{\text{lin}}(x) = \phi_0(x)w_0 + \phi_1(x)w_1 = w_0 + w_1 x$$

- Actually, the embedding trick is stronger.
  - $(x^2, x, 1)$: 2nd order polynomial with respect to x
  - $(x^d, x^{d-1}, \ldots, 1)$: d-th order polynomial (= degree d)

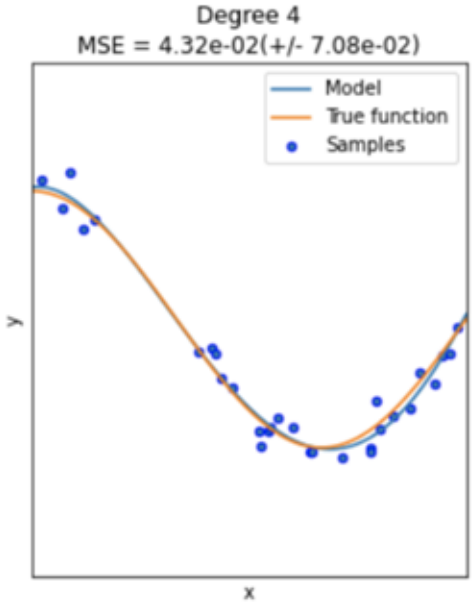
$$\phi_0(x) = 1 \qquad \phi_1(x) = x \qquad \phi_2(x) = x^2$$

$$y = \boldsymbol{w}^{\mathsf{T}}\Phi_{\text{lin}}(x) = \phi_0(x)w_0 + \phi_1(x)w_1 + \phi_2(x)w_2 = w_0 + w_1 x + w_2 x^2$$

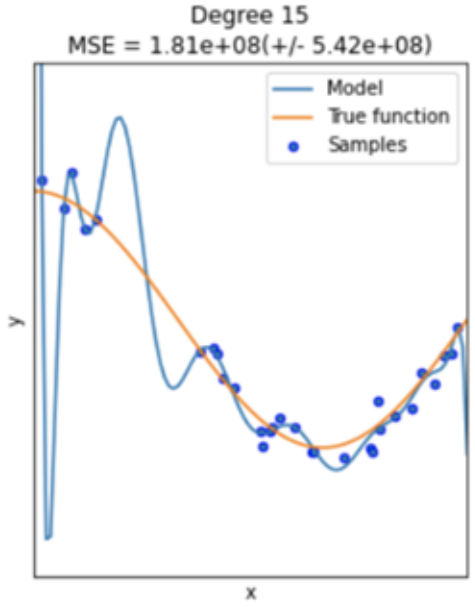
Feature embedding trick



$(x, 1)$



$(x^4, x^3, x^2, x, 1)$



$(x^{15}, \ldots, x^3, x^2, x, 1)$

higher-order polynomial = higher complexity = prone to overfitting!

from https://datascience.foundation/sciencewhitepaper/underfitting-and-overfitting-in-machine-learning

Basis Functions

- A **basis function** can be any function of the input features X
- Define a set of B basis functions $\phi_1(x), \ldots, \phi_B(x)$
- Fit a linear regression model in terms of basis functions,

$$y = \sum_{b=1}^{B} w_b \phi_b(x) = w^T \phi(x)$$

notation:
$\phi(x) := [\phi_1(x), \ldots, \phi_B(x)]$

- The model is *linear in the transformed basis/induced features* $\phi(x)$.
- The model is *nonlinear in the data* X

Example

- Two features: $x = (x_1, x_2)$
- Quadratic features: $\phi_1(x) = 1, \phi_2(x) = x_1, \phi_3(x) = x_2, \phi_4(x) = x_1^2, \phi_5(x) = x_2^2, \phi_6(x) = x_1 \cdot x_2$
- For an input: $x = (2,3)$
- $\phi(x) = (1, 2, 3, 4, 9, 6)$
- $y = w^T \phi(x) = w_1 \times 1 + w_2 \times 2 + w_3 \times 3 + w_4 \times 4 + w_5 \times 9 + w_6 \times 6$

sklearn.preprocessing.PolynomialFeatures

degree : *int or tuple (min_degree, max_degree), default=2*

If a single int is given, it specifies the maximal degree of the polynomial features. If a tuple `(min_degree, max_degree)` is passed, then `min_degree` is the minimum and `max_degree` is the maximum polynomial degree of the generated features. Note that `min_degree=0` and `min_degree=1` are equivalent as outputting the degree zero term is determined by `include_bias`.

interaction_only : *bool, default=False*

If `True`, only interaction features are produced: features that are products of at most `degree` *distinct* input features, i.e. terms with power of 2 or higher of the same input feature are excluded:

- included: `x[0]`, `x[1]`, `x[0] * x[1]`, etc.
- excluded: `x[0] ** 2`, `x[0] ** 2 * x[1]`, etc.

include_bias : *bool, default=True*

If `True` (default), then include a bias column, the feature in which all polynomial powers are zero (i.e. a column of ones - acts as an intercept term in a linear model).

order : *{'C', 'F'}, default='C'*

Order of output array in the dense case. `'F'` order is faster to compute, but may slow down subsequent estimators.

Example: Polynomial Basis Functions

Create three two-dimensional data points [0,1], [2,3], [4,5]:

```
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Compute quadratic features $(1, x_1, x_2, x_1^2, x_1 x_2, x_2^2)$,

```
>>> poly = PolynomialFeatures(degree=2)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

These are now our new data and ready to fit a model…

Example: Polynomial Basis Functions

Create a 3-rd order polynomial (cubic) function,

```
f = lambda x: (x-1)*(x-2)*(x-3)
import numpy.random as ra
ra.seed(20)
train_x = np.arange(5)
train_y = f(train_x) + 1*ra.randn(len(train_x))
train_y
```

✓ 0.3s

```
array([-5.11610689,  0.19586502,  0.35753652, -2.34326191,  4.91516741])
```

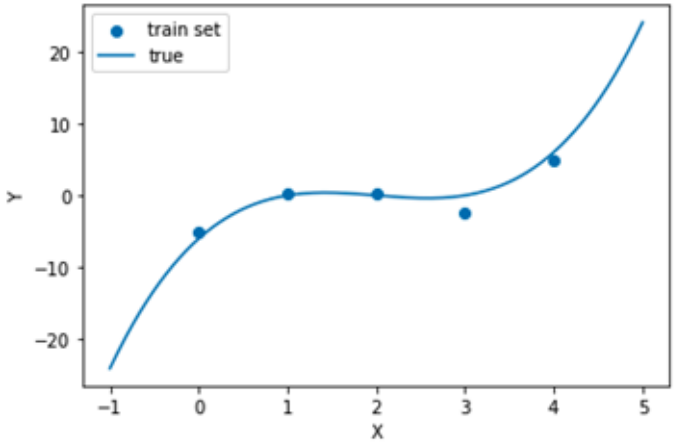
Plot train set and the actual function

```
test_x = np.linspace(-1,5,400)

from matplotlib import pyplot as plt
plt.scatter(train_x,train_y)

plt.plot(test_x, f(test_x))
plt.legend(['train set', 'true'])
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

✓ 0.4s



Example: Polynomial Basis Functions

Create cubic features $(1, x, x^2, x^3)$

```
poly = PolynomialFeatures(degree=3)
train_xx = poly.fit_transform(train_x[:,np.newaxis])
train_xx
```

✓ 0.4s

turns train_x (length 5 array) into a matrix (5 by 1 matrix)

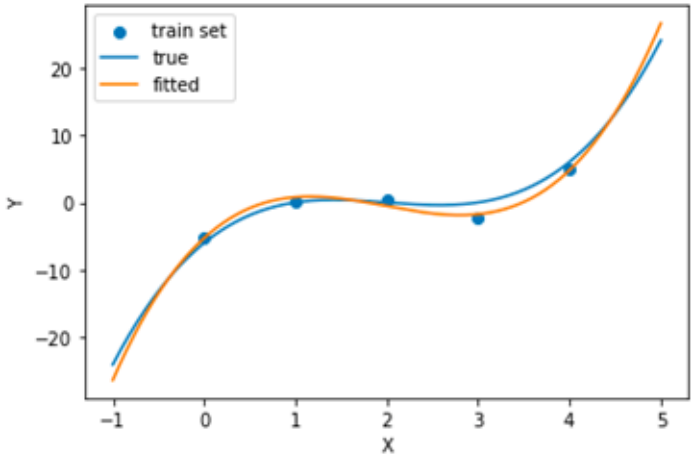
```
array([[ 1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  2.,  4.,  8.],
       [ 1.,  3.,  9., 27.],
       [ 1.,  4., 16., 64.]])
```

Perform linear regression; plot it

```
from matplotlib import pyplot as plt
from sklearn.linear_model import LinearRegression
model = LinearRegression().fit(train_xx, train_y)
test_x = np.linspace(-1,5,400)
test_xx = poly.fit_transform(test_x[:,np.newaxis])
pred_y = model.predict(test_xx)

plt.scatter(train_x,train_y)
plt.plot(test_x, f(test_x))
plt.plot(test_x, pred_y)
plt.legend(['train set', 'true', 'fitted'])
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

✓ 0.2s



Data Preprocessing

- Generally the first step in data science involves *preprocessing* or transforming data in some way
  - Filling in missing values (imputation)
  - Centering / normalizing / standardizing
  - Etc.

- We then fit our models to this preprocessed data

- One way to view preprocessing is simply as computing some basis function $\phi(x)$, nothing more

Basis Functions

PROs

- More flexible modeling that is nonlinear in the original data
- Increases model expressivity

CONs

- Typically requires **more parameters** to be learned
- More sensitive to **overfitting** training data (due to expressivity)
- Requires more **regularization** to avoid overfitting
- Need to find *good* basis functions (feature engineering)