Generalized Linear Models

Overview

- Logarithms and Euler's Number
- Functions
    - Line
    - Quadratic
    - Exponential
    - Log Base e (Natural)
    - Sine
    - Cosine
    - Tangent
    - Hyperbolic
- Distributions
    - Normal
    - Binomial
    - Poisson
    - Gamma
- S-Curve and CDF
- Generalized Linear Models
- Linear and Logistic Regression Formulas
- Assumptions
- Polynomial Regression

- Step Functions
- Basis Functions
- Local Regression
- Regression Splines
- Smoothing Splines
- Generalized Additive Models

Video Reviews

Quant Psych

- Understanding GLM - https://www.youtube.com/watch?v=SqN-qlQOM5A
- GLM Pt 1 - https://www.youtube.com/watch?v=ZWnM-yPUXlA
- GLM Pt 2 - https://www.youtube.com/watch?v=29Droau_6DM
- Mixed Models - https://www.youtube.com/watch?v=5tOifM51ZOk

Stanford

- 1.1 First 4 minutes - https://www.youtube.com/watch?v=LvySJGj-88U&list=PLoROMvodv4rPP6braWoRt5UCXYZ71GZIQ GLM - 4.8
- Splines - 7.2, 7.3,
- GAM - 7.4

- Python: Polynomial Regression... - https://www.youtube.com/watch?v=_OomkfNiU2c&list=PLoROMvodv4rPP6braWoRt5UCXYZ71GZIQ&index=56&pp=iAQB
- Python: Splines - https://www.youtube.com/watch?v=C9h-o6AfNX0&list=PLoROMvodv4rPP6braWoRt5UCXYZ71GZIQ&index=57&pp=iAQB
- Python: GAMs - https://www.youtube.com/watch?v=hQx84r2maaE&list=PLoROMvodv4rPP6braWoRt5UCXYZ71GZIQ&index=58&pp=iAQB
- for linear regression we focus on coefficients and gams we focus on fitted functions
- dotted red lines are error bars
- partial dependence plot - The partial dependence plot (short PDP or PD plot) shows the marginal effect one or two features have on the predicted outcome of a machine learning model (J. H. Friedman 200130). A partial dependence plot can show whether the relationship between the target and a feature is linear, monotonic or more complex. https://christophm.github.io/interpretable-ml-book/pdp.html

What's wrong with cookbooks - https://www.youtube.com/watch?v=CuWvPFTwUIs

Types of Regression

- More Regression Notebook

Concepts vs Facts

Concepts are an efficient way to store knowledge. Oncel you have abstracted the common features of a concept such as chair, you will recognize objects you have never seen before as a type of chair. This is because you hold in memory a prototypic representation of the concept - an abstraction of "chairness." Unfortunately, facts must each be held in memory individually. This is because they have no common group features. By definition, a fact is a unique piece of information that must be individually held in memory to be known. Compared to concepts, facts are a much less efficient form of knowledge.

Developing Technical Training: A Structured Approach for Developing Classroom and Computer-Based Instructional Materials 3rd Edition by Ruth Clark https://www.amazon.com/Developing-Technical-Training-Computer-based-Instructional/dp/0787988464

Logarithms and Euler's Number

Logarithms

- Logarithms are the inverse of exponentials
- $10^3 = 1000$ and $log_{10}(1000) = 3$
- When data units need scaling
- Expressing large numbers
- $y = ax^n$ is like $log(y) = nlog(x) + log(a)$
- Reduces finding n to a linear equation
- $y = ae^{bx}$ is like $log(y) = bx + log(a)$
- Addition and subtraction
- Multiplication and division

Types of logarithms

- Binary: computer science (0s and 1s), music theory (distance between stacked notes)
- Natural (Euler): chemistry, economics, comparing age of a dog to human
- Common (10): sound, pH, electricity, light

Sources

- https://math.stackexchange.com/questions/35810/intuitive-use-of-logarithms
- https://www.snexplores.org/article/explainer-what-are-logarithms-exponents

Examples:

- $log_{10}(10) = 1$
- $log_{10}(100) = 2$

- etc
- numpy log10
- numpy log default is natural log

Euler's Number (e)

- $e = 1 + 1/1 + 1/2 + 1/3! + 1/4! + ... + 1/n!$
- Compound interest (like exponential growth) $FV = PV e^{rt}
- where FV is Future Value, PV is Present Value, e is Euler's Number, r is Interest Rate Compounded, and t is time

```
# $1000 investment with 2% interest over 3 years
import numpy as np

1000 * np.exp(0.02 * 3)
```

```
# Investing $1000 with 10 percent continuous compound
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(1, 40, 40)

# continuous
y = 1000 * np.exp(.1*x)
plt.scatter(x, y, label='continuous')
print(max(y))

# discrete
y = 1000 * (1 + .1)**(x)
```

```
plt.scatter(x, y, label='discrete')
print(max(y))
plt.legend();
```

```
# 10 percent compound interest over 50 years and annu
# https://math.stackexchange.com/questions/4475485/co
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(1, 40, 40)

y = (1000*(1.1)**x) + (2400/.1)*((1.1)**x - 1)
plt.scatter(x, y);
print((1000*(1.1)**40) + (2400/.1)*((1.1)**40 - 1))
```

Functions

Examples

- See Functions Notebook

Why We Use General Linear Models

Let's look at some distributions and think about pvalues and confidence intervals

Distribution Plots and Shapiro-Wilk Test (Normality Test)

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6676026/#:~:text=The%20purpose%20of%20the%20t,essential%20in%20the%20t%2Dtest

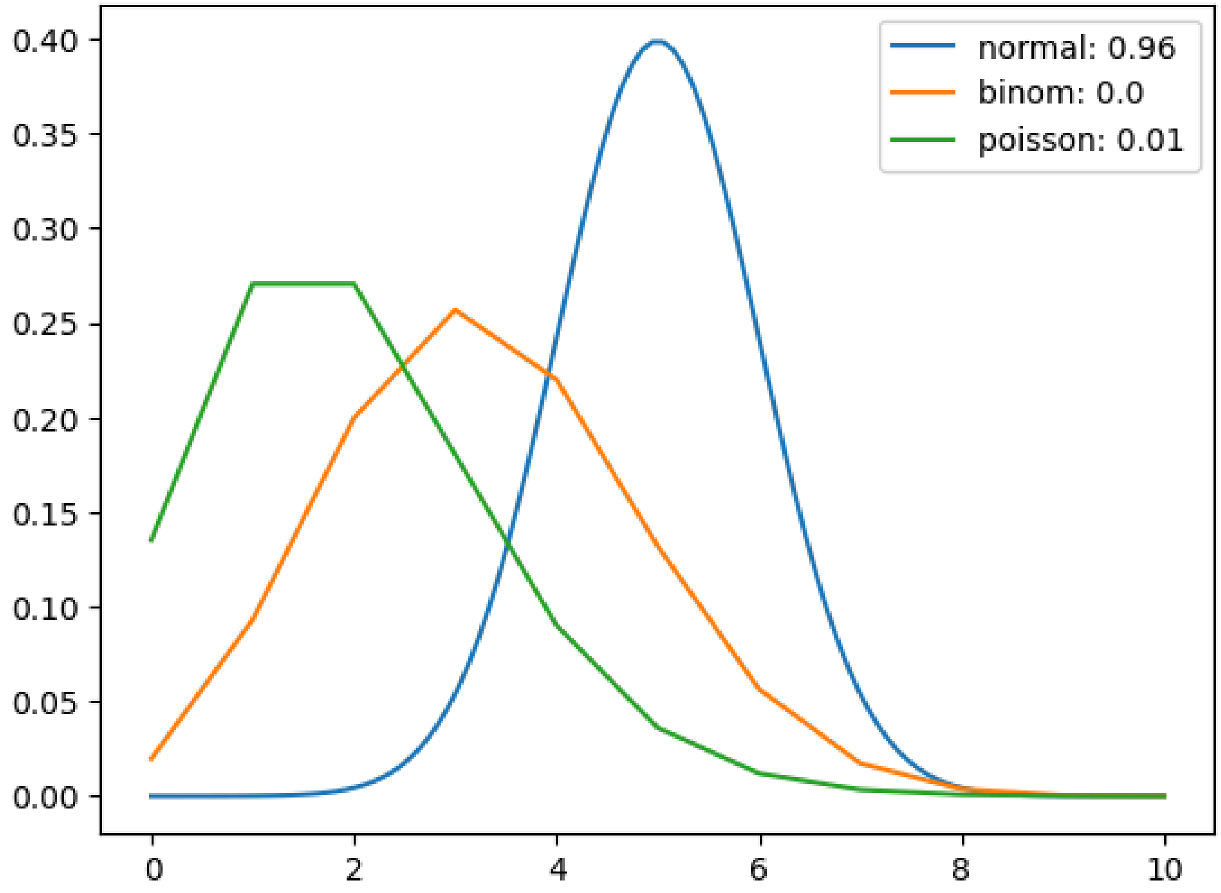
The Shapiro–Wilk test is a test of normality. It was published in 1965 by Samuel Sanford Shapiro and Martin Wilk.

https://en.wikipedia.org/wiki/Shapiro%E2%80%93Wilk_test

If the p-value of the test is greater than α = .05, then the data is assumed to be normally distributed.

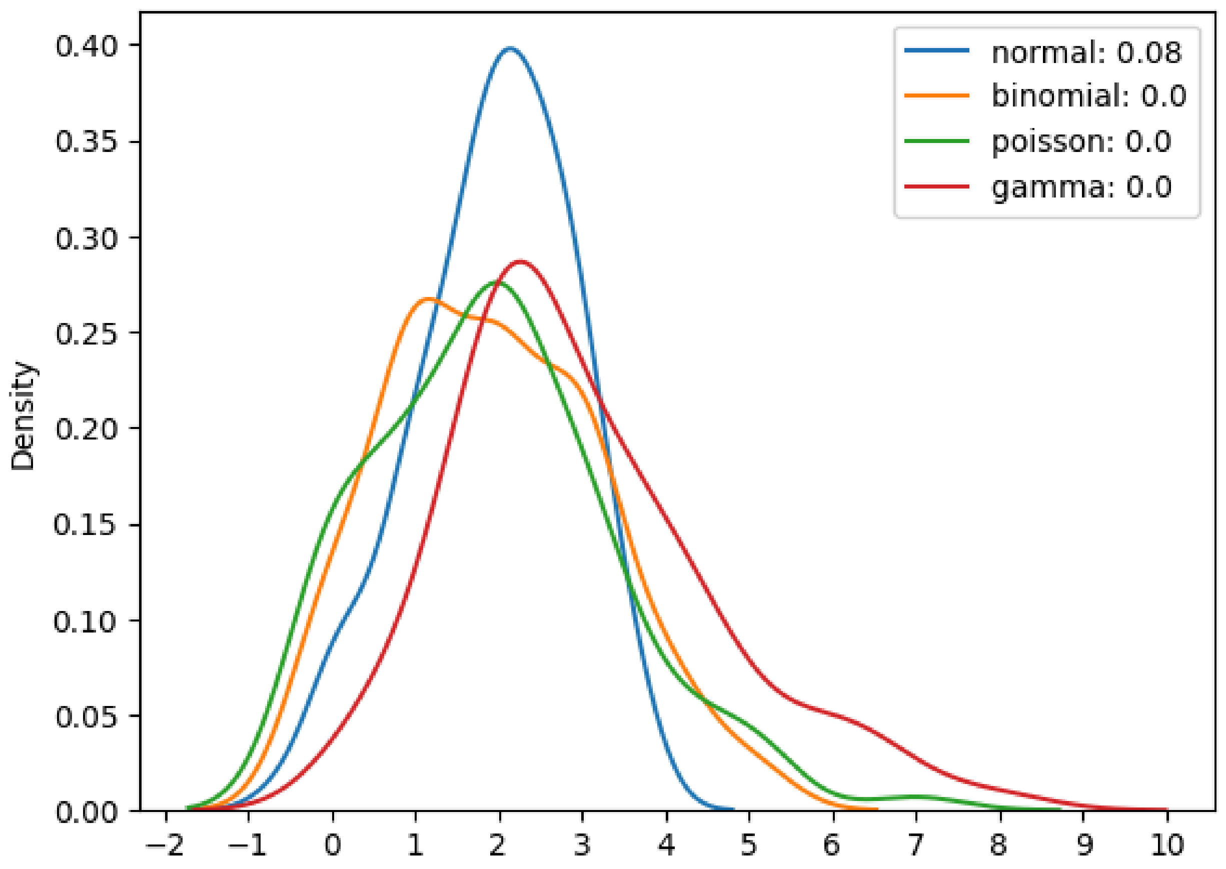
```
# https://levelup.gitconnected.com/probability-distri
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats

# normal
n = 100
x = np.linspace(0, 10, n)
y = stats.norm.pdf(x, 5, 1)
dist = stats.norm().rvs(1000)
plt.plot(x, y, label=f'normal: {round(stats.shapiro(d

# binom
x = np.arange(0,11)
y = stats.binom(n=len(x), p=.3).pmf(x)
dist = stats.binom(n=len(x), p=.3).rvs(1000)
plt.plot(x, y, label=f'binom: {round(stats.shapiro(di


# poisson
x = np.arange(0,11)
y = stats.poisson(mu=2).pmf(x)
dist = stats.poisson(mu=2).rvs(1000)
```

```
plt.plot(x, y, label=f'poisson: {round(stats.shapiro(

plt.legend();
```



```
from numpy.lib import bincount
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import seaborn as sns
import scipy.stats as stats

n = 100
fig, ax = plt.subplots()
```

```
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))

# normal
norm = np.random.normal(2, 1, size=n)
sns.kdeplot(norm, label=f'normal: {round(stats.shapir

# binomial
bino = np.random.binomial(10, .2, size=n)
sns.kdeplot(bino, label=f'binomial: {round(stats.shap

# poisson
pois = np.random.poisson(2, n)
sns.kdeplot(pois, label=f'poisson: {round(stats.shapi

# gamma
gamma = np.random.gamma(3, 1, n)
sns.kdeplot(gamma, label=f'gamma: {round(stats.shapir

plt.legend();
```



Probability Density Function

- When normal, we get significance, confidence, p-values, power, alpha, z-scores, etc
- When not normal, normal pdf won't work, e.g., for a Poisson distribution use the Poisson pdf (stats.poisson.pdf) (cdf, ppf, pmf, etc)
- What is a distribution appropriate for social media when trying to make inferences back the whole population?

- The probabilities associated with extreme values on a normal distribution will probably not be the same for other distributions (beta, poisson, gamma, etc)

Practical Examples of Distributions

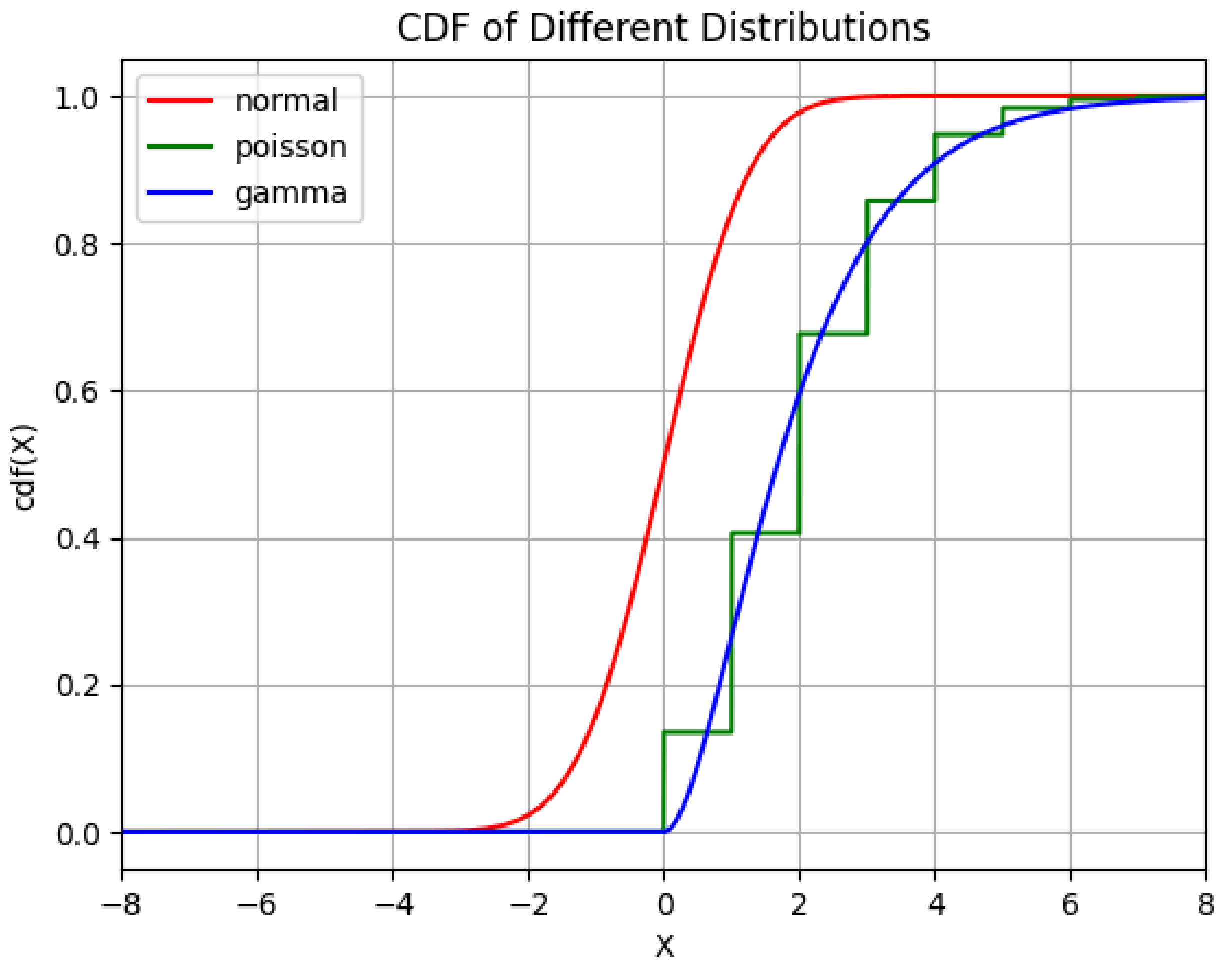
- Normal: natural phenomena of one variate
- Binomial: counting, gambling
- Poisson: events happening in a period of time
- Gamma: time to failure in mechanics, reliability in a piece of technology

S-Curve and CDF

```
# plot the cdf
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

n = 10000
bound = 8
x = np.linspace(-bound, bound, n)
plt.plot(x, stats.norm.cdf(x), 'r', label='normal')

plt.plot(x, stats.poisson.cdf(x, 2), 'g', label='pois
print(stats.poisson.ppf(.9, 2))
plt.plot(x, stats.gamma.cdf(x, 2), 'b', label='gamma'
plt.title('CDF of Different Distributions')
plt.xlabel('X')
plt.xlim(-bound, bound)
plt.ylabel('cdf(X)')
```

```
    plt.grid(True)
    plt.legend();
```

```
4.0
```



Imagine you're driving a car. Let's break down how first and second derivatives relate to your trip:

1. First Derivative: Velocity (Speed and Direction)

- **What it tells you:** How fast your position is changing at a specific moment and in what direction (are you going forwards or backwards?).
- **Example:** If your first derivative is 30 mph, you're moving forward at 30 mph. If it's -20 mph, you're moving backward at 20 mph.
- **In a graph:** The first derivative represents the *slope* of the line tangent to the curve at any point. A steep uphill slope means a large positive first derivative (high speed in the positive direction). A flat line means a first derivative of zero (no change in position/stopped).

2. Second Derivative: Acceleration

- **What it tells you:** How fast your *velocity* is changing. Are you speeding up, slowing down, or staying at a constant speed?
- **Example:** If your second derivative is 5 mph per second, you're increasing your speed by 5 mph every second. If it's -10 mph per second, you're braking and decreasing your speed by 10 mph every second.
- **In a graph:** The second derivative tells you the *concavity* of the curve. A curve that's bending upwards (like a smile) has a positive second derivative (accelerating). A curve bending downwards (like a frown) has a negative second derivative (decelerating).

Difference in a nutshell:

- **First derivative:** Measures the *rate of change* of a function (like position).
- **Second derivative:** Measures the *rate of change of the rate of change* of a function (like the change in velocity).

Here's a table summarizing the key differences:

Feature	First Derivative	Second Derivative
Measures	Rate of change	Rate of change of the rate of change
In motion	Velocity	Acceleration
In a graph	Slope	Concavity

Why are they important?

Derivatives help us understand how things change over time or with respect to other variables. They are fundamental to many areas of math, science, and engineering, including:

- **Physics:** Understanding motion, forces, and energy.
- **Economics:** Analyzing market trends and optimizing production.
- **Machine learning:** Training models and optimizing algorithms.

Fun with Expressions

$$\underset{g \in S}{minimize} \sum_{i=1}^{n} (y_i - g(x_i))^2 + \lambda \int g''(t)^2 dt$$

From the video / book

- The first term is RSS, and tries to make g(x) match the data at each x_i.
- The second term is a roughness penalty and controls how wiggly g(x) is. It is modulated byt the tuning parameter $\lambda \geq 0$
- The smaller λ, the more wiggly the function, eventually interpolating y_i when $\lambda = 0$
- As $\lambda \to \infty$, the function g(x) becomes linear

From Gemini, October 23 2024

1. `minimize`

This indicates we're looking for a function `g` within a set of possible functions `S` that minimizes the value of the expression. In machine learning, this "function" is often called a model, and we're trying to find the best model to fit our data.

2. `Σ(yi - g(xi))²`

- This is the **sum of squared errors (SSE)**.
    - `yi` represents the actual observed value for the i-th data point.
    - `g(xi)` represents the predicted value for the i-th data point using the function `g`.
    - `(yi - g(xi))` is the error, or the difference between the actual and predicted values.
    - We square the error to penalize larger errors more heavily and to ensure all errors are positive.

    - The summation (`∑`) adds up these squared errors for all `n` data points.

   This part of the formula aims to find a function `g` that fits the data closely, minimizing the difference between predicted and actual values.

3. **`λ ∫ g''(t)² dt`**
   - This is the **regularization term**.
     - `g''(t)` represents the second derivative of the function `g`. The second derivative measures the rate of change of the slope of `g`. In other words, it measures how "curvy" the function `g` is.
     - `∫ g''(t)² dt` calculates the integral of the squared second derivative over the domain of the function. This essentially quantifies the overall "curviness" or complexity of `g`.
     - `λ` (lambda) is a **regularization parameter** that controls the balance between fitting the data and keeping the function smooth. A larger `λ` puts more emphasis on smoothness, potentially leading to a simpler model that may underfit the data. A smaller `λ` prioritizes fitting the data closely, which could lead to a more complex model that may overfit the data.

   This part of the formula aims to prevent overfitting by penalizing functions that are overly complex or "wiggly."

Overfitting occurs when the model learns the training data too well and fails to generalize to new, unseen data.

In essence, this objective function seeks to find a balance between:

- **Accuracy:** Fitting the given data points well (minimizing SSE).
- **Simplicity:** Avoiding excessive complexity in the function `g` (minimizing the regularization term).

This balance is crucial in machine learning to create models that are both accurate and generalizable to new data.

Example:

Imagine you're trying to predict house prices based on their size. You could have a simple linear function, or a very complex curve that goes through every single data point. The objective function helps you find a function that captures the general trend of the data without being overly influenced by individual data points or noise.

Second Take on t and dt

$$\underset{g \in S}{minimize} \sum_{i=1}^{n} (y_i - g(x_i))^2 + \lambda \int g''(t)^2 dt$$

Understanding the Context

This equation represents a way to find the best fit for a set of data points. It's trying to find a function `g` that balances two things:

1. **Fitting the data closely:** This is represented by the first part of the equation (the sum of squared errors).
2. **Keeping the function smooth:** This is where the second part of the equation with the integral comes in.

The Role of the Integral

The integral (`∫`) is a mathematical operation that essentially calculates the area under a curve. In this case, the "curve" is the square of the second derivative of the function `g`.

- **`t`:** This is a dummy variable used within the integral. It represents the input to the function `g`. You can think of it as a placeholder that takes on different values along the curve.
- **`dt`:** This indicates that we're integrating with respect to `t`. It represents an infinitesimally small change in `t`.

Why the Second Derivative and Integral?

- **Second derivative (`g''(t)`):** The second derivative of a function measures its "curvature" or how much its slope is changing. A large second derivative means the function is changing direction rapidly (it's "wiggly").
- **Integral of the squared second derivative:** By integrating the squared second derivative, the equation is essentially measuring the total "wiggliness" of the function `g` over its entire domain.

Putting it Together

The second part of the equation (`λ ∫ g''(t)² dt`) is called a "regularization term." It penalizes functions that are too "wiggly" or complex.

- `λ` **(lambda):** This is a parameter that controls how much weight is given to the smoothness penalty.

In simple terms:

Imagine you're trying to draw a smooth curve through a set of points. You could draw a very wiggly line that goes through every point exactly, but that might not be the best representation of the overall trend. This equation helps you find a balance between fitting the points and keeping the curve smooth. The `dt` and `t` within the integral are the mathematical tools that allow you to measure the "wiggliness" of the curve.

Differention and Integration

Differentiation and integration are two fundamental concepts in calculus that are closely related, yet they represent opposite operations. Here's a comparison to help you understand their key differences:

Differentiation

- **What it does:** Finds the instantaneous rate of change of a function. Think of it as finding the slope of a curve at a specific point.

- **Geometric interpretation:** The derivative at a point represents the slope of the tangent line to the curve at that point.
- **Applications:**
    - Finding velocities and accelerations in physics.
    - Analyzing rates of growth or decay in biology and economics.
    - Optimizing functions in machine learning and engineering.
- **Example:** If you have a function describing the position of an object over time, the derivative gives you its velocity at any given time.

Integration

- **What it does:** Finds the area under a curve. Think of it as accumulating small changes over an interval.
- **Geometric interpretation:** The definite integral represents the signed area between the curve and the x-axis over a given interval.
- **Applications:**
    - Calculating areas and volumes of irregular shapes.
    - Finding the total displacement of an object given its velocity.
    - Determining the work done by a force.
    - Calculating probabilities in statistics.

- **Example:** If you have a function describing the velocity of an object over time, the integral gives you its total displacement over a period.

Relationship:

- **Inverse operations:** Differentiation and integration are inverse operations. This means that if you differentiate a function and then integrate the result, you get back the original function (with a possible constant term added).

Analogy:

Imagine you're building a Lego tower:

- **Differentiation:** Like taking the tower apart brick by brick. You're analyzing the individual components and how they contribute to the overall structure.
- **Integration:** Like putting the bricks together to build the tower. You're accumulating the individual components to create the whole.

Here's a table summarizing the key differences:

Feature	Differentiation	Integration
Operation	Finding rate of change	Finding area
Geometric interpretation	Slope of tangent line	Area under curve
Application	Velocity, optimization	Displacement, volume
Relationship to the other	Inverse operation	Inverse operation

By understanding the differences between differentiation and integration, you can gain a deeper understanding of calculus and its applications in various fields.

Univariate Spline

`scipy.interpolate.UnivariateSpline` is a powerful tool in SciPy for creating smooth curves that fit your data. It's particularly useful when you want to:

- **Smooth noisy data:** If your data has some noise or variability, `UnivariateSpline` can help you find a smooth curve that captures the underlying trend without being overly influenced by individual data points.
- **Interpolate between data points:** You can use it to estimate values between your existing data points, creating a continuous curve.
- **Approximate a function:** Even if you don't have an explicit function, you can use `UnivariateSpline` to create a spline representation that approximates the relationship between your variables.

Here's a breakdown of how it works and how to use it:

1. What is a spline?

A spline is a piecewise polynomial function. Imagine taking a curve and dividing it into smaller segments. In each segment, the spline is represented by a different polynomial function (like a line, quadratic, or cubic curve). These polynomial pieces are joined together smoothly at points called "knots."

2. How `UnivariateSpline` works:

- **Input:** You provide the function with your x and y data points.
- **Fitting:** `UnivariateSpline` finds the best spline (with a specified degree) that fits your data by adjusting the coefficients of the polynomials and the positions of the knots.
- **Smoothing:** It uses a smoothing factor (`s`) to control how closely the spline follows the data points. A higher `s` leads to a smoother curve that may not pass through all the points, while a lower `s` results in a more "wiggly" curve that closely follows the data.
- **Output:** You get a spline object that you can use to evaluate the spline at any x-value (for interpolation or prediction).

3. Using `UnivariateSpline`:

```
from scipy.interpolate import UnivariateSpline
import numpy as np

# Sample data
x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 1, 3, 5, 4])

# Create a spline object (cubic spline by default)
spline = UnivariateSpline(x, y, s=1)  # s is the smoothir

# Evaluate the spline at new x-values
```

```
x_new = np.linspace(1, 5, 100)  # 100 points between 1 ar
y_new = spline(x_new)

# Plot the results
import matplotlib.pyplot as plt
plt.scatter(x, y, label='Data')
plt.plot(x_new, y_new, label='Spline')
plt.legend()
plt.show()
```

Key parameters:

- `x`: The independent variable data points.
- `y`: The dependent variable data points.
- `s`: The smoothing factor.
- `k`: The degree of the spline (e.g., 1 for linear, 3 for cubic).

Advantages of `UnivariateSpline`:

- **Flexibility:** You can control the smoothness of the fit.
- **Efficiency:** It's generally efficient for moderate-sized datasets.
- **Easy to use:** The interface is relatively straightforward.

When to use it:

- When you need a smooth curve to represent your data.
- When you want to interpolate between data points.
- When you need to approximate a function from data.

Both `UnivariateSpline` and Kernel Density Estimation (KDE) are techniques for creating smooth curves from data, but they serve different purposes and work in distinct ways. Here's a comparison:

UnivariateSpline

- **Purpose:** To fit a smooth curve that represents the relationship between two variables. It's focused on approximating the underlying function or trend in the data.
- **How it works:** Fits a piecewise polynomial function (a spline) to the data points, with a smoothing factor controlling how closely the spline follows the data.
- **Output:** A spline object that can be used to estimate values at any point along the curve (interpolation or prediction).
- **Focus:** Representing the relationship between variables.

KDE

- **Purpose:** To estimate the probability density function (PDF) of a single variable. It's focused on visualizing the distribution of the data and identifying areas of high and low density.
- **How it works:** Places a kernel (a small, smooth function) at each data point and sums them up to

create a smooth density estimate. The bandwidth of the kernel controls the smoothness of the density estimate.
- **Output:** A density estimate that shows the probability of observing a value within a given range.
- **Focus:** Representing the distribution of a single variable.

Here's an analogy:

Imagine you have data on the heights of students in a class.

- `UnivariateSpline`: Like drawing a smooth curve that shows the general trend of height as a function of age (or some other variable). You're trying to capture the relationship between height and age.
- **KDE:** Like creating a histogram with smooth curves instead of bars. It shows you the distribution of heights in the class, highlighting where most students' heights cluster.

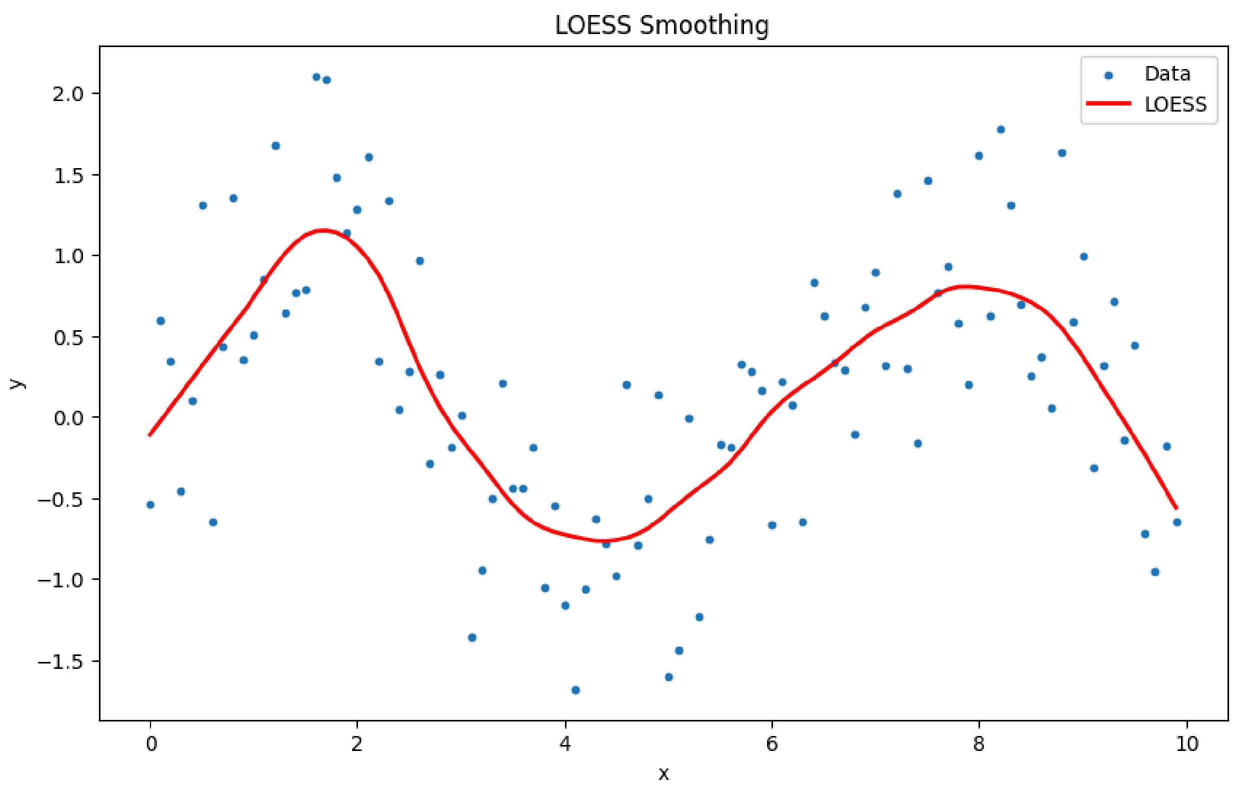
Key differences in a table:

Feature	`UnivariateSpline`	KDE
Purpose	Fit a curve to data	Estimate probability den
Method	Piecewise polynomial fitting	Kernel smoothing
Output	Spline object	Density estimate
Focus	Relationship between variables	Distribution of a single va
Smoothing parameter	`s` (smoothing factor)	Bandwidth

When to use which:

- `UnivariateSpline`: When you want to smooth noisy data, interpolate between points, or approximate a function from data.
- **KDE:** When you want to visualize the distribution of a single variable, identify clusters or outliers, or estimate probabilities.

```
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.nonparametric.smoothers_lowess impor

# Generate data
np.random.seed(123)
x = np.arange(0, 10, 0.1)
y = np.sin(x) + np.random.normal(0, 0.5, len(x))

# Apply LOESS smoothing
smoothed = lowess(y, x, frac=0.2)  # frac controls th

# Plot the results
plt.figure(figsize=(10, 6))
plt.scatter(x, y, label='Data', s=10)
plt.plot(smoothed[:, 0], smoothed[:, 1], label='LOESS
plt.xlabel('x')
plt.ylabel('y')
plt.title('LOESS Smoothing')
plt.legend()
plt.show()
```



LOESS

LOESS (locally estimated scatterplot smoothing), sometimes called LOWESS (locally weighted scatterplot smoothing), is a method used to create a smooth curve that fits a set of data points without assuming a specific functional form like a line or a parabola. It's particularly helpful when you have data with a complex relationship that doesn't follow a simple pattern.

Here's a breakdown of how LOESS works:

1. Local Regression:

- Imagine you have a set of data points scattered on a plot. LOESS works by fitting a separate regression model to localized subsets of the data.
- It's like sliding a small window across your data points. For each window, LOESS fits a simple model (usually a low-degree polynomial like a line or a quadratic curve) to the points within that window.

2. Weighted Regression:

- Not all points within the window are treated equally. LOESS assigns weights to the points based on their distance from the center of the window.
- Points closer to the center have higher weights, meaning they have more influence on the local regression. Points farther away have lower weights and less influence.
- This weighting ensures that the local regression is most influenced by the points closest to the point being estimated.

3. Combining Local Models:

- After fitting local regressions to each window, LOESS combines the results to create a smooth curve that passes through the entire dataset.
- The final curve is a weighted average of the predictions from all the local models.

Key Features and Benefits of LOESS:

- **Non-parametric:** LOESS doesn't assume any specific functional form for the relationship between the variables. It's flexible and can capture a wide range of patterns.
- **Robust to outliers:** The weighting scheme reduces the influence of outliers, making LOESS more robust compared to methods like ordinary least squares regression.
- **Control over smoothness:** You can control how smooth the curve is by adjusting the size of the window (bandwidth) and the degree of the polynomial used in the local regressions.

When is LOESS Useful?

- **Complex relationships:** When the relationship between variables is non-linear and difficult to model with a simple function.
- **Noisy data:** When your data has a lot of noise or variability, LOESS can help smooth out the noise and

reveal the underlying trend.

- **Data exploration:** LOESS is a great tool for exploring data and visualizing patterns without making strong assumptions about the data.

In simpler terms:

Imagine you're trying to draw a smooth line through a set of scattered points. LOESS is like using a small, flexible ruler that you move along the points, fitting it to small sections of the data at a time. The final curve is a combination of all these local "fits," creating a smooth and flexible representation of the overall trend.

LOESS and KDE are both smoothing techniques, but they approach the problem from different perspectives and have distinct characteristics. Here's how they relate and differ:

Similarities:

- **Non-parametric:** Both LOESS and KDE are non-parametric methods, meaning they don't assume a specific functional form (like a linear or polynomial equation) for the relationship between variables. They let the data "speak for itself" to determine the shape of the smooth curve.
- **Local smoothing:** Both methods focus on local neighborhoods of data points to create the smooth curve. LOESS fits local regressions, while KDE uses kernels to estimate local density.

- **Smoothing parameter:** Both have a parameter that controls the degree of smoothing. In LOESS, it's the bandwidth or window size, while in KDE, it's the bandwidth of the kernel.

Differences:

- **Purpose:**
    - **LOESS:** Primarily used for smoothing and visualizing the relationship between two variables. It focuses on estimating the conditional mean of the response variable (y) given the predictor variable (x).
    - **KDE:** Primarily used for estimating the probability density function of a single variable. It focuses on visualizing the distribution of the data and identifying areas of high and low density.
- **Method:**
    - **LOESS:** Uses locally weighted polynomial regression. It fits a low-degree polynomial to a subset of data points within a local window, with weights decreasing as you move away from the center of the window.
    - **KDE:** Uses kernel smoothing. It places a kernel function (a small, smooth function like a Gaussian) at each data point and sums them up to create a smooth density estimate.

- **Output:**
    - **LOESS:** Produces a smooth curve that represents the trend in the data.
    - **KDE:** Produces a density estimate that shows the probability of observing a value within a given range.

Analogy:

Imagine you're trying to map a mountain range:

- **LOESS:** Like drawing a smooth road that follows the general terrain of the mountains, focusing on the elevation (y) at different locations (x).
- **KDE:** Like creating a topographic map that shows the density of elevation changes, highlighting peaks and valleys.

In essence:

LOESS and KDE are both valuable tools for smoothing data, but they serve different purposes. LOESS is more focused on capturing the relationship between variables, while KDE is more focused on visualizing the distribution of a single variable. The choice of which method to use depends on the specific goals of your analysis.

Maximum Likelihood Estimation (MLE)

Imagine you're a detective at a crime scene. You find a footprint, but you don't know who it belongs to. You have a list of suspects, and you know the shoe size of each suspect.

Maximum likelihood estimation (MLE) is like trying to find the suspect whose shoe size would most likely produce the footprint you found.

Here's how it works:

1. **Assume a model:** You assume that the footprint was made by a shoe, and that shoe sizes follow a certain distribution (e.g., normal distribution).
2. **Calculate the likelihood:** For each suspect, you calculate how likely it is that their shoe size would produce the observed footprint. This likelihood is based on your assumed model (the distribution of shoe sizes) and the evidence (the footprint size).
3. **Find the maximum:** You choose the suspect whose shoe size has the *highest* likelihood of producing the observed footprint. This is your maximum likelihood estimate.

In simpler terms:

MLE is like finding the best guess for an unknown value (like the shoe size of the person who left the footprint) by finding the value that makes the observed data (the footprint) most probable.

Here's an example with coins:

You flip a coin 10 times and get 7 heads. You want to estimate the probability of getting heads (let's call it 'p').

- **Assume a model:** You assume the coin flips follow a Bernoulli distribution (each flip has two possible outcomes: heads or tails).
- **Calculate the likelihood:** For different values of 'p' (e.g., 0.1, 0.2, ..., 0.9), you calculate how likely it is to get 7 heads out of 10 flips.
- **Find the maximum:** The value of 'p' that gives you the highest likelihood is your maximum likelihood estimate. In this case, it would be around 0.7.

You're right to ask for more clarification! Let's break down that statement and why the maximum likelihood estimate (MLE) for the coin flip example is around 0.7.

Recap of the scenario:

- You flipped a coin 10 times and got 7 heads.
- You want to estimate the *true* probability of getting heads with this coin (which we'll call 'p').

What we're doing with MLE:

MLE aims to find the value of 'p' that makes the observed outcome (7 heads in 10 flips) the *most likely* outcome. It's like saying, "If the true probability of heads was actually [some value], how likely would it be to get the results I got?" We try different values of 'p' and see which one gives us the highest probability of observing our data.

Why around 0.7?

Think about it intuitively:

- If 'p' was very low (like 0.1), getting 7 heads out of 10 flips would be very unlikely.
- If 'p' was very high (like 0.9), getting 7 heads would also be less likely (you'd expect more heads).
- A value of 'p' around 0.7 seems to fit the observed data best. It makes sense that a coin with a 70% chance of landing heads would be more likely to produce 7 heads in 10 flips.

How do we actually find the MLE?

1. **Likelihood function:** We use a formula called the likelihood function to calculate the probability of observing our data (7 heads) for different values of 'p'. For a Bernoulli distribution (like coin flips), the likelihood function looks like this:

   ```
   Likelihood(p) = p^k * (1-p)^(n-k)
   ```

   Where:

   - `p` is the probability of heads
   - `k` is the number of heads (7 in our case)
   - `n` is the total number of flips (10 in our case)

2. **Maximizing the likelihood:** We can try different values of 'p' (between 0 and 1) and calculate the likelihood for

each. The value of 'p' that gives us the *highest* likelihood is our MLE.

In practice, we often use calculus (finding the derivative and setting it to zero) or numerical methods to find the exact value of 'p' that maximizes the likelihood function. In this case, the MLE turns out to be 0.7.

Important note:

The MLE is just an *estimate* based on the observed data. It's not necessarily the *true* value of 'p', but it's the value that best explains the data we have.

Key takeaways:

- MLE is a method for estimating unknown parameters (like probabilities, averages, etc.) from data.
- It finds the parameter value that makes the observed data most likely.
- It relies on an assumed model for the data.

MLE is a powerful tool used in many areas, from statistics and machine learning to economics and engineering.

It's important to understand how Maximum Likelihood Estimation (MLE) and Bayesian approaches differ, especially in how they incorporate prior knowledge.

MLE (the 0.7 estimate):

- **Focus:** MLE only considers the observed data (7 heads in 10 flips). It tries to find the parameter value

(probability of heads, 'p') that maximizes the likelihood of observing that data.
- **No prior information:** MLE doesn't incorporate any prior beliefs or information about the coin. It acts as if we know nothing about the coin before the experiment.
- **Result:** In this case, MLE suggests that the most likely value of 'p' is 0.7, based purely on the observed data.

Bayesian approach (the 0.5 estimate):

- **Prior knowledge:** Bayesian methods allow us to incorporate prior beliefs or information about the parameter. In your example, you're suggesting a prior belief that the coin is fair (p=0.5). This could be based on our general experience with coins or knowing that the coin is supposed to be standard.
- **Updating beliefs:** The Bayesian approach combines the prior information with the observed data to update our belief about 'p'.
- **Result:** With a strong prior belief that p=0.5, even after observing 7 heads in 10 flips, the Bayesian estimate might be closer to 0.5 than 0.7. The observed data influences the estimate, but the prior still pulls it towards 0.5.

How they compare:

- **Data vs. prior:** MLE relies solely on the data, while the Bayesian approach balances data with prior knowledge.

- **Point estimate vs. distribution:** MLE gives you a single point estimate (0.7), while the Bayesian approach provides a posterior distribution over possible values of 'p'. This distribution reflects the remaining uncertainty after observing the data.

Which is better?

It depends on the situation and how much prior information you have:

- **Limited data:** If you have very little data, a Bayesian approach with a reasonable prior can be helpful.
- **Strong prior:** If you have strong prior knowledge (e.g., you know the coin is from a reputable mint), a Bayesian approach can incorporate that.
- **No prior/objective analysis:** If you want an objective estimate purely based on the data, MLE is a good choice.

In the coin flip example:

If you have reason to believe the coin is fair (e.g., it's a standard coin used in a magic trick), the Bayesian approach with a prior of 0.5 might be more reasonable. However, if you're testing a coin you found on the street and have no reason to assume it's fair, MLE might be a better starting point.

In this example, the MLE of 0.7 for the coin flip probability is a frequentist interpretation.

Here's why:

- **Frequentist interpretation:** Frequentists view probability as the long-run frequency of an event. They would say that if you flip the coin many, many times, the proportion of heads will approach the true probability 'p'. MLE aligns with this view because it seeks the value of 'p' that would make the observed data (7 heads in 10 flips) most likely in the long run.
- **No prior beliefs:** As we discussed, MLE doesn't incorporate any prior beliefs about the coin's fairness. It only considers the observed data, which is a key characteristic of frequentist methods.
- **Contrast with Bayesian:** Bayesian methods, on the other hand, allow for the incorporation of prior beliefs. They treat 'p' as a random variable with a prior distribution, which is updated based on the observed data to obtain a posterior distribution.

In essence:

The MLE approach in the coin flip example embodies the frequentist perspective by:

1. **Estimating 'p' based on the frequency of heads in the observed data.**
2. **Not considering any prior information or beliefs about the coin.**

This contrasts with the Bayesian approach, which would incorporate a prior distribution for 'p' (such as assuming a fair coin with p=0.5) and update that distribution based on the observed data.

In machine learning, we often deal with complex loss functions (functions that measure the error of our model). These loss functions can have many "hills and valleys," and our goal is to find the lowest point in this landscape, which represents the best set of parameters for our model.

Here's how local and global minima fit into this picture:

Local Minimum

- **What it is:** A point in the "landscape" of the loss function where the function has the lowest value in its immediate neighborhood. Think of it as the bottom of a valley.
- **Problem:** If your optimization algorithm (like gradient descent) gets stuck in a local minimum, it might think it has found the best solution, even though there might be a much deeper valley (a better solution) somewhere else.

Global Minimum

- **What it is:** The absolute lowest point in the entire landscape of the loss function. This represents the best possible solution (the set of parameters that minimizes the error the most).

- **Goal:** The ultimate goal of optimization in machine learning is to find the global minimum of the loss function.

Analogy:

Imagine you're trying to find the lowest point in a mountain range:

- **Local minimum:** You might find yourself at the bottom of a small valley, thinking you've reached the lowest point.
- **Global minimum:** However, there might be a much deeper valley somewhere else in the mountain range that you haven't discovered yet.

Challenges:

- **Finding the global minimum can be very difficult,** especially with complex loss functions that have many local minima.
- **Optimization algorithms can get trapped in local minima,** preventing them from finding the best solution.

Techniques to address the issue:

- **Different optimization algorithms:** Some algorithms, like stochastic gradient descent or simulated annealing, are designed to help escape local minima.
- **Momentum:** Adding momentum to the optimization process can help "overshoot" local minima.

- **Multiple starting points:** Running the optimization from different starting points can increase the chances of finding the global minimum.

Understanding the difference between local and global minima is crucial for effectively training machine learning models and avoiding suboptimal solutions.

Generalized Linear Models

- https://www.youtube.com/watch?v=SqN-qlQOM5A&t=18s

Model Responses

Quantitative Response

- $y = \theta^T X + \epsilon_i$ (matrix form)
- where $\epsilon \overset{iid}{\sim} \mathcal{N}(0, \sigma^2)$
- and ϵ is model noise and generally unknown

Binary Response

- $p(y=1) = \frac{1}{1+e^{-(\theta^T X)}}$

Generalized Linear Model

- Other responses include non-negative, skewed, etc
- Framework for different response types
- Useful when models violate assumptions

- See Assumptions Notebook

```
# test of normality
from sklearn.datasets import make_regression
import scipy.stats as stats

for _ in range(0, 10):
  X, y = make_regression(n_samples=1000, n_features=1
  print(stats.shapiro(y))

cnt = 0
for _ in range(0, 100):
  X, y = make_regression(n_samples=1000, n_features=1
  if stats.shapiro(y).pvalue >= .05 and stats.shapiro
    cnt += 1

print()
print(cnt)
```

```
ShapiroResult(statistic=0.9992840886116028, pvalue=0.9
ShapiroResult(statistic=0.9991645812988281, pvalue=0.9
ShapiroResult(statistic=0.9987448453903198, pvalue=0.7
ShapiroResult(statistic=0.9987128973007202, pvalue=0.6
ShapiroResult(statistic=0.9975616931915283, pvalue=0.1
ShapiroResult(statistic=0.9968883395195007, pvalue=0.0
ShapiroResult(statistic=0.998943030834198, pvalue=0.84
ShapiroResult(statistic=0.9990555047988892, pvalue=0.9
ShapiroResult(statistic=0.9987570643424988, pvalue=0.7
ShapiroResult(statistic=0.9987488389015198, pvalue=0.7

94
```

The General Linear Model

$y = \beta_0 + \beta_1 X + e$ or Outcome = Intercept + Slope x Predictor + e

- $y = \beta_0 + e$: one sample t test
- $y = \beta_0 + \beta_1 Group + e$: independent t
- $y1 - y2 = \beta_0 + e$: paired
- $y = \beta_0 + \beta_1 Group_1 + \beta_2 Group_2 + e$: anova

Intercepts and Slopes vs Means and Mean Differences

- Intercept $\sim$ Mean
- Slope $\sim$ Mean Differences

Generalized Linear Model Components

- Random Component - probability distribution
- Systematic Component - coefficients + explanatory variables
- Link Function - transforms (relates) the mean to the explanatory variables

Model Parameters

- Gaussian (mean (μ) and variance (σ^2))
- Binomial (independent trials (n) and success (p))
- Poisson Regression Model (mean and variance are equal)
- Gamma (α = shape, β = rate)

- Negative-Binomial (similar to Poisson but uses α as a shape parameter)
- Inverse Gaussian (mean (μ) and shape (λ))

Link Functions

https://www.statisticshowto.com/link-function/

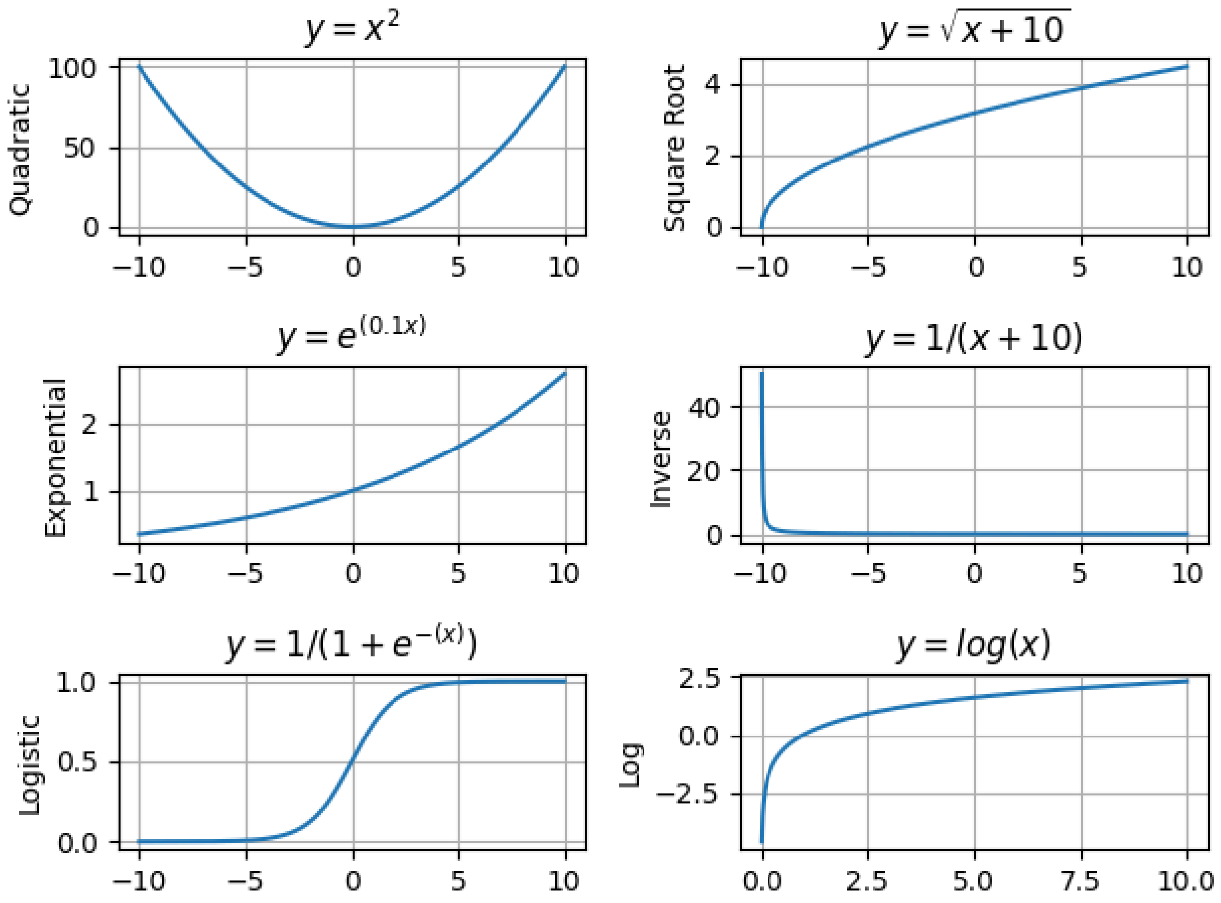
Allows the linear model to be related to the response variable by transforming the output of the response to fit the model boundaries.

η = link

- linear: $\eta(\mu) = \mu$
- logistic: $\eta(\mu) = log(\mu/(1-\mu))$
- poisson: $\eta(\mu) = log(\mu)$

```
# link functions
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")

fig = plt.figure()
ax1 = fig.add_subplot(321)
ax2 = fig.add_subplot(322)
ax3 = fig.add_subplot(323)
ax4 = fig.add_subplot(324)
ax5 = fig.add_subplot(325)
ax6 = fig.add_subplot(326)

x = np.linspace(-10, 10, 1000)
```

```
    y = x ** 2
    ax1.grid()
    ax1.plot(x, y)
    ax1.title.set_text(r'$y=x^2$')
    # ax1.title.set_text('y=x^2')
    ax1.set_ylabel('Quadratic')

    y = np.sqrt(x+10)
    ax2.grid()
    ax2.plot(x, y)
    ax2.title.set_text(r'$y=\sqrt{x+10}$')
    # ax2.title.set_text('y=sqrt(x)')
    ax2.set_ylabel('Square Root')

    y = np.e**(0.1*x)
    ax3.grid()
    ax3.plot(x, y)
    ax3.title.set_text(r'$y=e^{(0.1x)}$')
    # ax3.title.set_text('y=e^(0.1x)')
    ax3.set_ylabel('Exponential')

    y = 1/(x+10)
    ax4.grid()
    ax4.plot(x, y)
    ax4.title.set_text(r'$y=1/(x+10)$')
    # ax4.title.set_text('y=1/(x+10)')
    ax4.set_ylabel('Inverse')

    y = 1/(1 + np.e**-(x))
    ax5.grid()
    ax5.plot(x, y)
    ax5.title.set_text(r'$y=1/(1+e^{-(x)})$')
    # ax5.title.set_text('y=1/(1+e^-(x)')
    ax5.set_ylabel('Logistic')

    y = np.log(x)
    ax6.grid()
    ax6.plot(x, y)
    ax6.title.set_text(r'$y=log(x)$')
    # ax6.title.set_text('y=log(x)')
    ax6.set_ylabel('Log')

    plt.tight_layout();
```



Simple Linear Regression

The link function for simple linear regression is the **identity function**.

Here's why:

- **GLMs and Link Functions:** Generalized Linear Models (GLMs) extend traditional linear regression to accommodate various types of response variables (continuous, binary, counts, etc.). A key component of a GLM is the *link function*, which establishes a connection between the linear predictor (the familiar $mx + b$ in simple linear regression) and the expected value of the response variable.
- **Identity Function:** The identity function is simply a function that returns its input unchanged. Mathematically, it's represented as $f(x) = x$.
- **Simple Linear Regression:** In simple linear regression, we're modeling a continuous response variable with a normal distribution. The expected value of the response variable is directly equal to the linear predictor. In other words, we don't need any transformation to map the linear predictor to the expected value. This "no transformation" is precisely what the identity function accomplishes.

In summary: The identity link function in simple linear regression implies that the predicted value from the linear equation ($mx + b$) is directly used as the expected value of the response variable.

Mixed Models

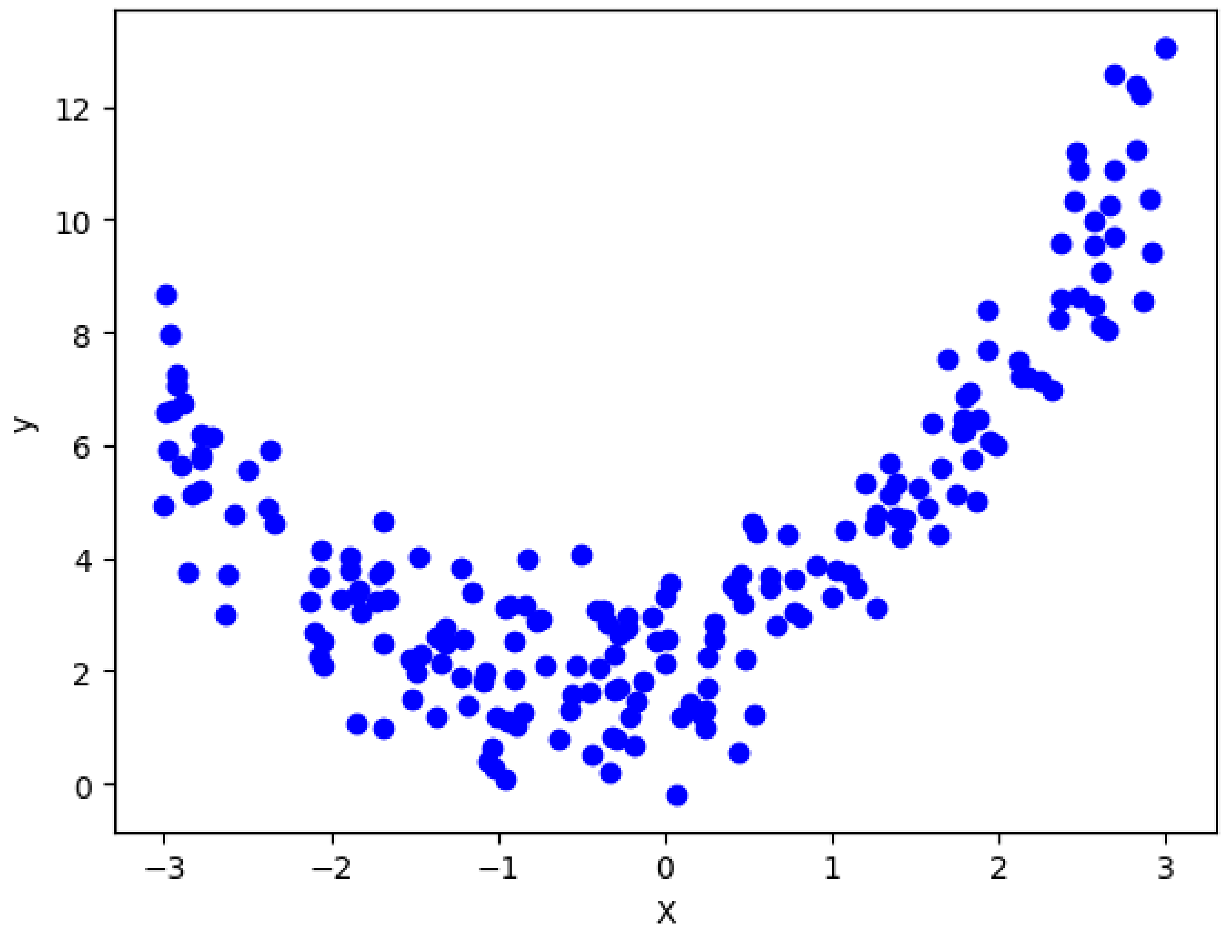
Non Linear Models

The truth is never linear!

Polynomial Regression

- See More Regressions Notebook

```
import numpy as np
import matplotlib.pyplot as plt

X = 6 * np.random.rand(200, 1) - 3
noise = np.random.normal(0, 1, X.shape)
y = 0.8*X**2 + 0.9*X + 2 + noise

plt.scatter(X, y, color='blue')
plt.xlabel("X")
plt.ylabel("y")
plt.show()
```

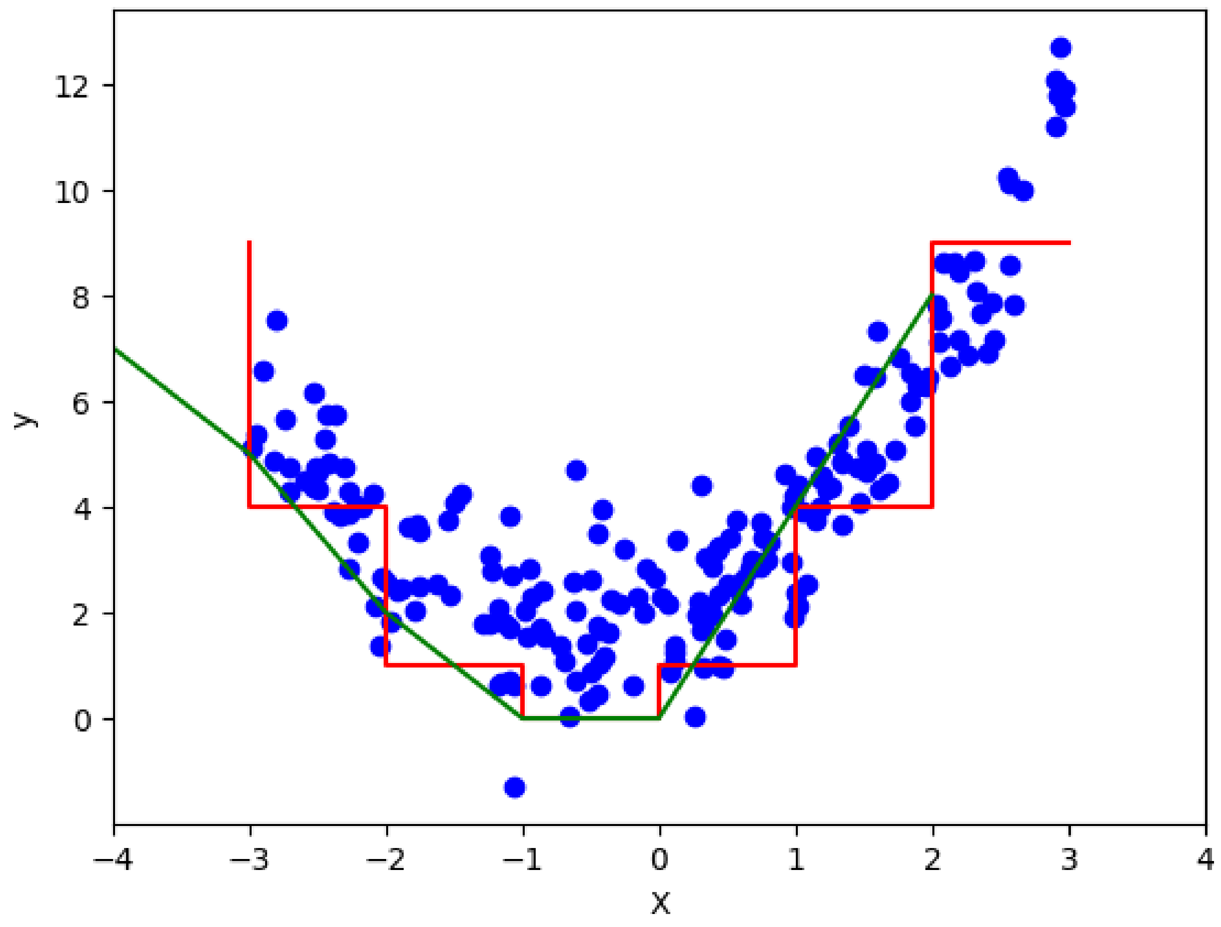


Step Functions / Piecewise Linear Fit

Cutting a variable into distinct regions

```
# polynomial regression and step functions
# https://stackoverflow.com/questions/29382903/how-to
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
```

```
X = 6 * np.random.rand(200, 1) - 3
noise = np.random.normal(0, 1, X.shape)
y = 0.8*X**2 + 0.9*X + 2 + noise

plt.scatter(X, y, color='blue')



def piecewise_linear(x, x0, y0, k1, k2):
  return np.piecewise(x, [x < x0], [lambda x:k1*x + y

sX = [-3, -2, -1, 0, 0, 1, 2, 3]
sy = [i**2 for i in sX]
plt.step(sX, sy, where='pre', color='red')

# Fit the piecewise linear function using shifted sX
sX_shifted = np.array(sX) - 1
p, e = optimize.curve_fit(piecewise_linear, sX_shifte
plt.plot(sX_shifted, piecewise_linear(sX_shifted, *p)

# p , e = optimize.curve_fit(piecewise_linear, sX, sy
# plt.plot(sX, piecewise_linear(sX, *p), color='green

plt.xlabel("X")
plt.xlim(-4, 4)
plt.ylabel("y")
plt.show()
```



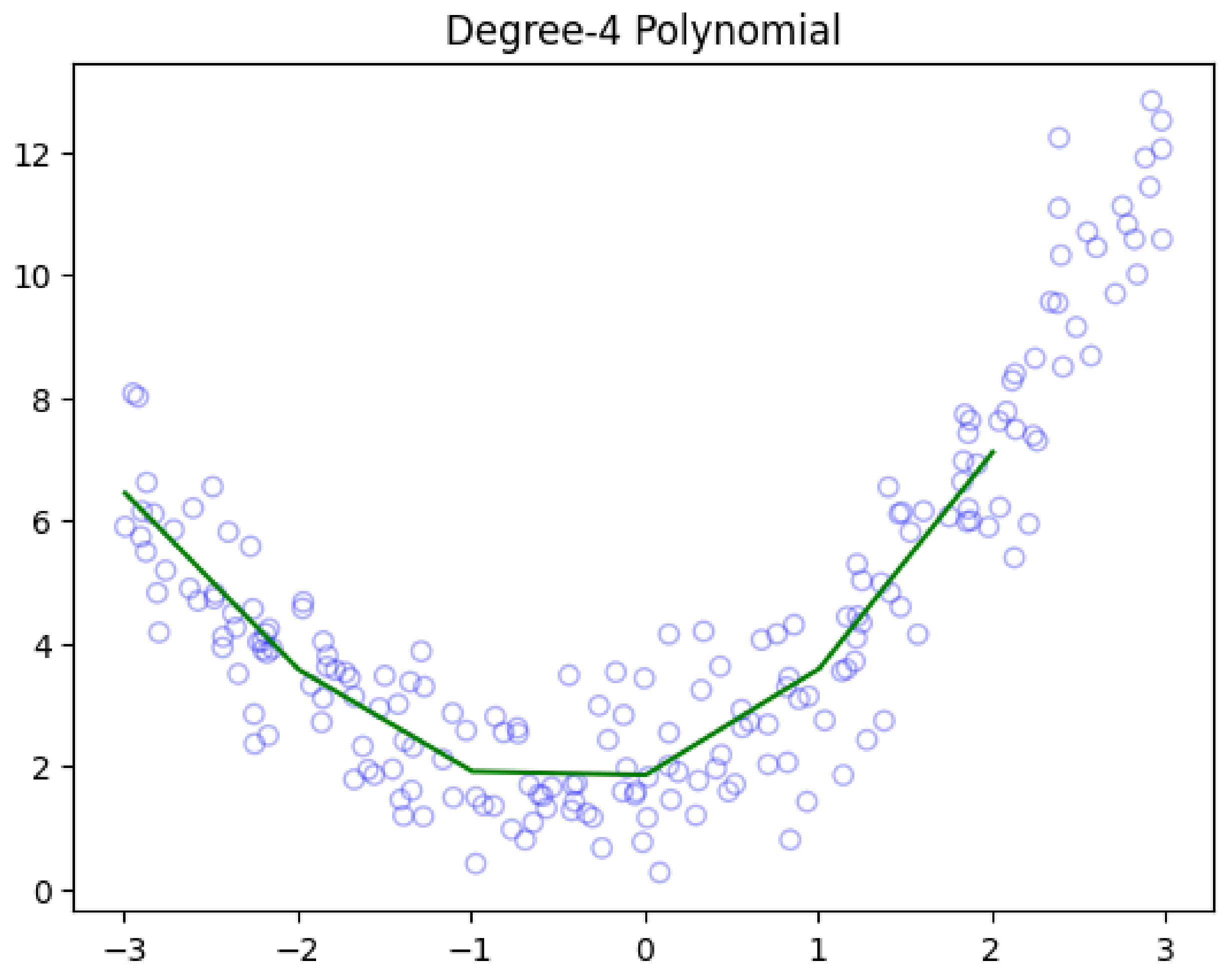
```
# polynomial regression and step functions
# https://www.science.smith.edu/~jcrouser/SDS293/labs
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

from sklearn.preprocessing import PolynomialFeatures
import statsmodels.api as sm

X = 6 * np.random.rand(200, 1) - 3
noise = np.random.normal(0, 1, X.shape)
```

```
y = 0.8*X**2 + 0.9*X + 2 + noise

d = {'X': X.flatten(), 'y': y.flatten()}
df = pd.DataFrame(d)

X1 = PolynomialFeatures(1).fit_transform(df.X.values.
X2 = PolynomialFeatures(2).fit_transform(df.X.values.
X3 = PolynomialFeatures(3).fit_transform(df.X.values.
X4 = PolynomialFeatures(4).fit_transform(df.X.values.
X5 = PolynomialFeatures(5).fit_transform(df.X.values.

fit1 = sm.GLS(y, X4).fit()
print(fit1.summary())
```

```
                            GLS Regression Results
======================================================================
Dep. Variable:                      y   R-squared:
Model:                            GLS   Adj. R-squared
Method:                 Least Squares   F-statistic:
Date:                Fri, 27 Oct 2023   Prob (F-statis
Time:                        17:53:44   Log-Likelihood
No. Observations:                 200   AIC:
Df Residuals:                     195   BIC:
Df Model:                           4
Covariance Type:            nonrobust
======================================================================
                 coef    std err          t      P>|t|
----------------------------------------------------------------------
const          1.8469      0.126     14.654      0.000
x1             0.8036      0.096      8.348      0.000
x2             0.9027      0.085     10.667      0.000
x3             0.0167      0.016      1.030      0.304
x4            -0.0079      0.010     -0.764      0.446
======================================================================
Omnibus:                        1.920   Durbin-Watson:
Prob(Omnibus):                  0.383   Jarque-Bera (J
Skew:                           0.139   Prob(JB):
Kurtosis:                       3.331   Cond. No.
```

```
==============================================

Notes:
[1] Standard Errors assume that the covariance matrix
```

```
# Create response matrix
import pandas as pd

resp_mx = (df.y > 3).map({False:0, True:1})

# GLM families comprise a link function as well as a
clf = sm.GLM(resp_mx, X4, family=sm.families.Binomial
res = clf.fit()

# Generate a sequence of X values spanning the range
X_grid = np.arange(df.X.min(), df.X.max()).reshape(-1

# Generate test data
X_test = PolynomialFeatures(4).fit_transform(X_grid)

# Predict the value of the generated X
pred1 = fit1.predict(X_test)

# Create plots
plt.title('Degree-4 Polynomial')

# Scatter plot with polynomial regression line
plt.scatter(df.X, df.y, facecolor='None', edgecolor='
plt.plot(X_grid, pred1, color = 'g')
plt.show()
```



```
# generalized least squares
fit_1 = fit = sm.GLS(df.X, X1).fit()
fit_2 = fit = sm.GLS(df.X, X2).fit()
fit_3 = fit = sm.GLS(df.X, X3).fit()
fit_4 = fit = sm.GLS(df.X, X4).fit()
fit_5 = fit = sm.GLS(df.X, X5).fit()

print(sm.stats.anova_lm(fit_1, fit_2, fit_3, fit_4, f
```

```
   df_resid           ssr  df_diff       ss_diff
0     198.0  2.023937e-28      0.0           NaN
1     197.0  1.270389e-28      1.0  7.535477e-29    0.
```

```
2      196.0  9.103060e-29      1.0  3.600829e-29    0.
3      195.0  3.414588e-27      1.0 -3.323557e-27   -0.
4      194.0  1.191946e-24      1.0 -1.188532e-24 -193.
```

```
# step functions
df_cut, bins = pd.cut(df.X, 4, retbins = True, right
df_cut.value_counts(sort = False)

df_steps = pd.concat([df.X, df_cut, df.y], keys = ['X

# Create dummy variables for the age groups
df_steps_dummies = pd.get_dummies(df_steps['X_cuts'])

# Statsmodels requires explicit adding of a constant
df_steps_dummies = sm.add_constant(df_steps_dummies)

# Drop the (17.938, 33.5] category
df_steps_dummies = df_steps_dummies.drop(df_steps_dum

df_steps_dummies.head(5)
```

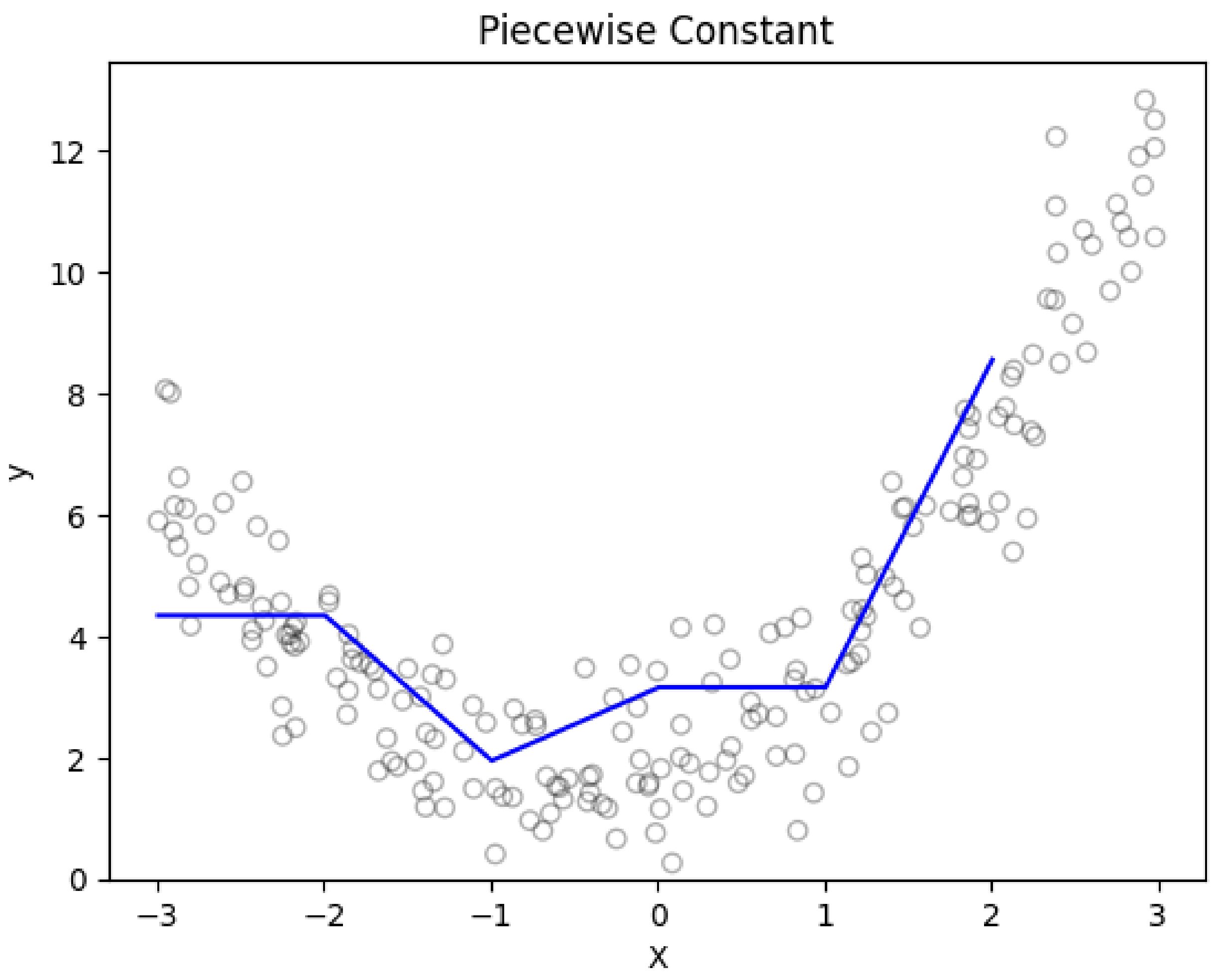
	const	(-1.497, -0.00348]	(-0.00348, 1.49]	(1.49, 2.984]
0	1.0	0	0	0
1	1.0	0	0	1
2	1.0	0	0	0
3	1.0	0	0	0
4	1.0	0	0	1

```
fit2 = sm.GLM(df_steps.y, df_steps_dummies).fit()
print(fit2.summary())
```

```
                 Generalized Linear Model Regression R
==============================================================
Dep. Variable:                          y   No. Observatio
Model:                                GLM   Df Residuals:
Model Family:                    Gaussian   Df Model:
Link Function:                   Identity   Scale:
Method:                              IRLS   Log-Likelihood
Date:                    Fri, 27 Oct 2023   Deviance:
Time:                            17:53:45   Pearson chi2:
No. Iterations:                         3   Pseudo R-squ.
Covariance Type:                nonrobust
==============================================================
                         coef    std err          z
--------------------------------------------------------------
const                  4.3446      0.214     20.349
(-1.497, -0.00348]    -2.3929      0.308     -7.768
(-0.00348, 1.49]      -1.1868      0.303     -3.912
(1.49, 2.984]          4.2113      0.313     13.445
==============================================================
```

```
# Put the test data in the same bins as the training
bin_mapping = np.digitize(X_grid.ravel(), bins)

# Get dummies, drop first dummy category, add constan
X_test2 = sm.add_constant(pd.get_dummies(bin_mapping)

# Predict the value of the generated ages using the l
predictions = fit2.predict(X_test2)

# # And the logistic model
# clf2 = sm.GLM(y, df_steps_dummies, family=sm.famili
# res2 = clf2.fit()
# pred3 = res2.predict(X_test2)

# Plot
plt.title('Piecewise Constant')
```

```
    # Scatter plot with polynomial regression line
    plt.scatter(df.X, df.y, facecolor = 'None', edgecolor
    plt.plot(X_grid, predictions, c = 'b')

    plt.xlabel('X')
    plt.ylabel('y')
    plt.ylim(ymin = 0)
    plt.show()
```



Basis Functions

- Family of functions or transformations that can be applied to X
- Examples include polynomials, piece-wise constant functions, wavelets for Fourier series, and regression splines

Splines

Splines create smooth curves out of irregular data points

Source

- https://towardsdatascience.com/data-science-deciphered-what-is-a-spline-18632bf96646

Piecewise Polynomials, Knots, Splines, and Interpolation

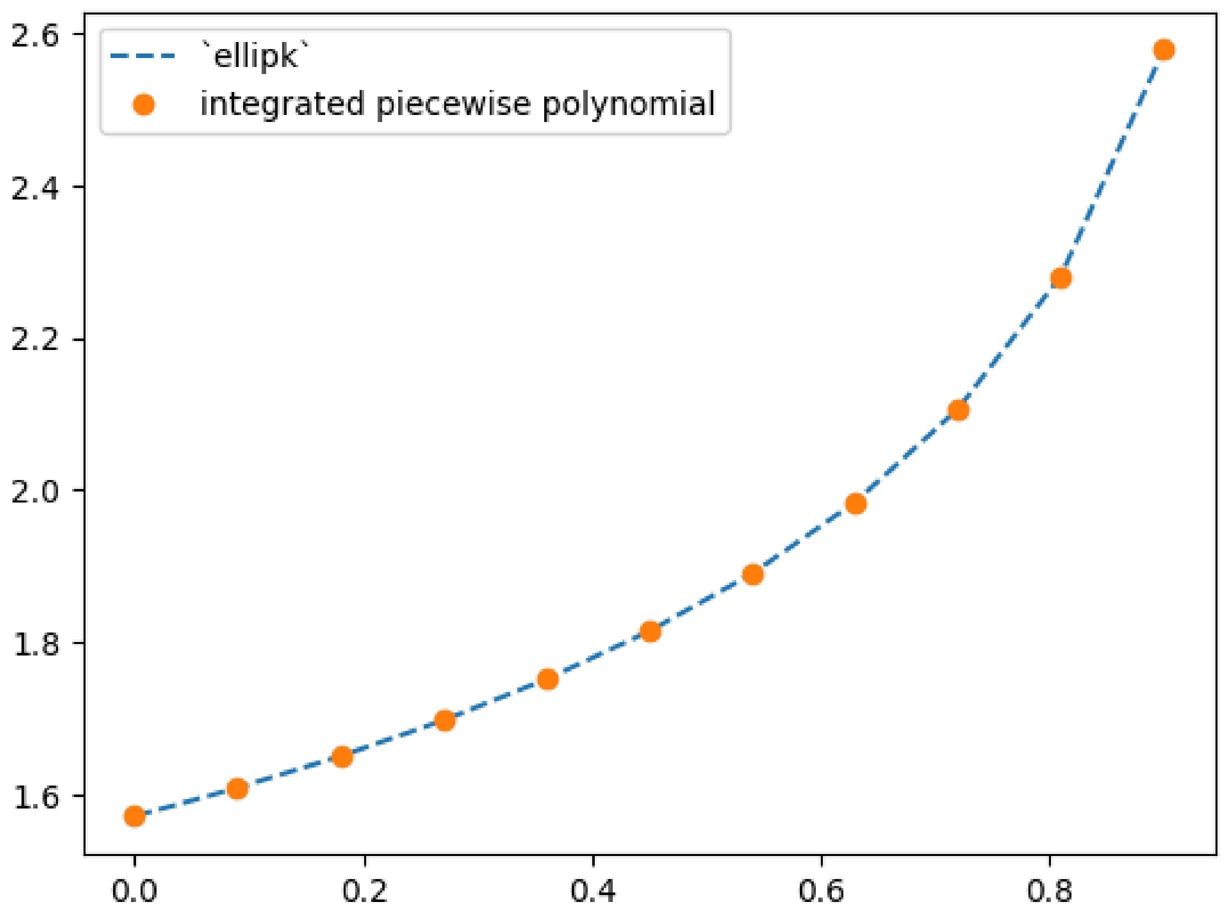
- An alternative to fit all data points with a single polynomial curve, is to fit segments to different parts of the data, with breakpoints (knots) at pre-determined places
- Connect the knots with lines (polynomial lines if needed)
- Smooth out the fitted curve

- In essence, splines are piecewise polynomials, joined at points called knots

Source

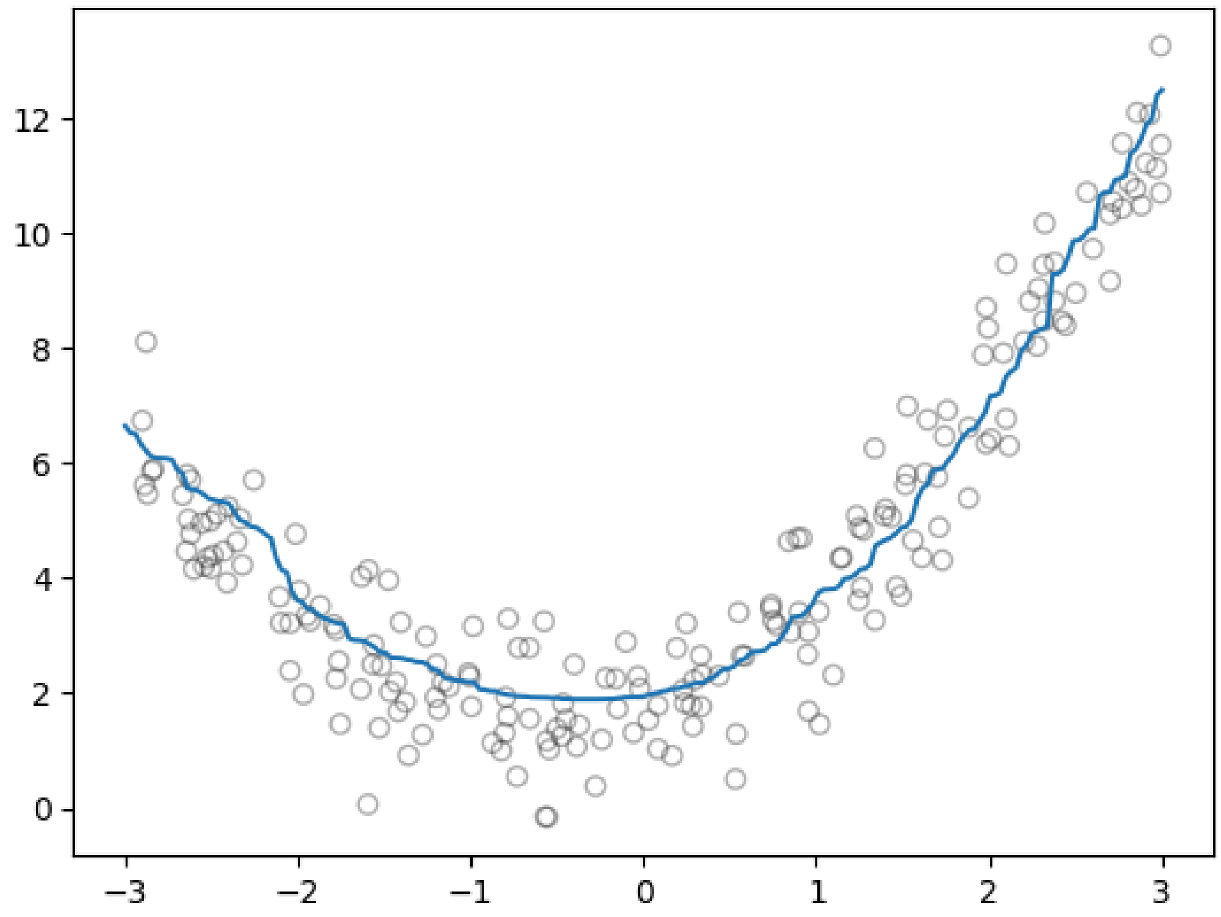
- https://bookdown.org/tpinto_home/Beyond-Linearity/piecewise-regression-and-splines.html

```
# https://docs.scipy.org/doc/scipy/tutorial/interpola
import matplotlib.pyplot as plt
from scipy.interpolate import PchipInterpolator
from scipy.special import ellipk

m = np.linspace(0, 0.9, 11)
x = np.linspace(0, np.pi/2, 70)
y = 1 / np.sqrt(1 - m[:, None]*np.sin(x)**2)

spl = PchipInterpolator(x, y, axis=1)  # the default

plt.plot(m, spl.integrate(0, np.pi/2), '--')
plt.plot(m, ellipk(m), 'o')
plt.legend(['`ellipk`', 'integrated piecewise polynom
plt.show()
```



```
import matplotlib.pyplot as plt
from scipy.interpolate import PchipInterpolator

m = np.linspace(-3, 3, 200)
x = 6 * np.random.rand(200, 1) - 3
noise = np.random.normal(0, 1, x.shape)
x = np.sort(x.flatten())
y = [0.8*i**2 + 0.9*i + 2 + noise for i in x]

X1 = 6 * np.random.rand(200, 1) - 3
y1 = 0.8*X1**2 + 0.9*X1 + 2 + noise

spl = PchipInterpolator(x, y, axis=1)  # the default
```

```
    plt.scatter(X1, y1, facecolor = 'None', edgecolor = '
    plt.plot(m, spl.integrate(0, np.pi/3))

    plt.show()
```



Interpolation is a method of estimating unknown data points in a known range to make the curve smoother. Univariate interpolation is a type of curve fitting that seeks the curve that best fits a set of two-dimensional data points. Since the data points are sampled from a single variable function, it is called univariate interpolation.

SciPy API provides several functions to implement the interpolation method for a given data. In this tutorial, you'll learn how to apply interpolation for a given data by using interp1d, CubicSpline, PchipInterpolator, and Akima1DInterplator methods in Python. The tutorial covers;

- Preparing test data
- interp1d method
- CubicSpline method
- PchipInterpolator method
- Akima1DInterpolator method
- Source code listing

Source

- https://www.datatechnotes.com/2022/12/univariate-interpolation-examples-in.html

Smoothing Splines

- For the interpolation problem, the task is to construct a curve which passes through a given set of data points. This may be not appropriate if the data is noisy: we then want to construct a smooth curve, g(x), which approximates the input data without passing through each point exactly. To this end, scipy.interpolate allows constructing smoothing splines, based on the Fortran library FITPACK by P. Dierckx.

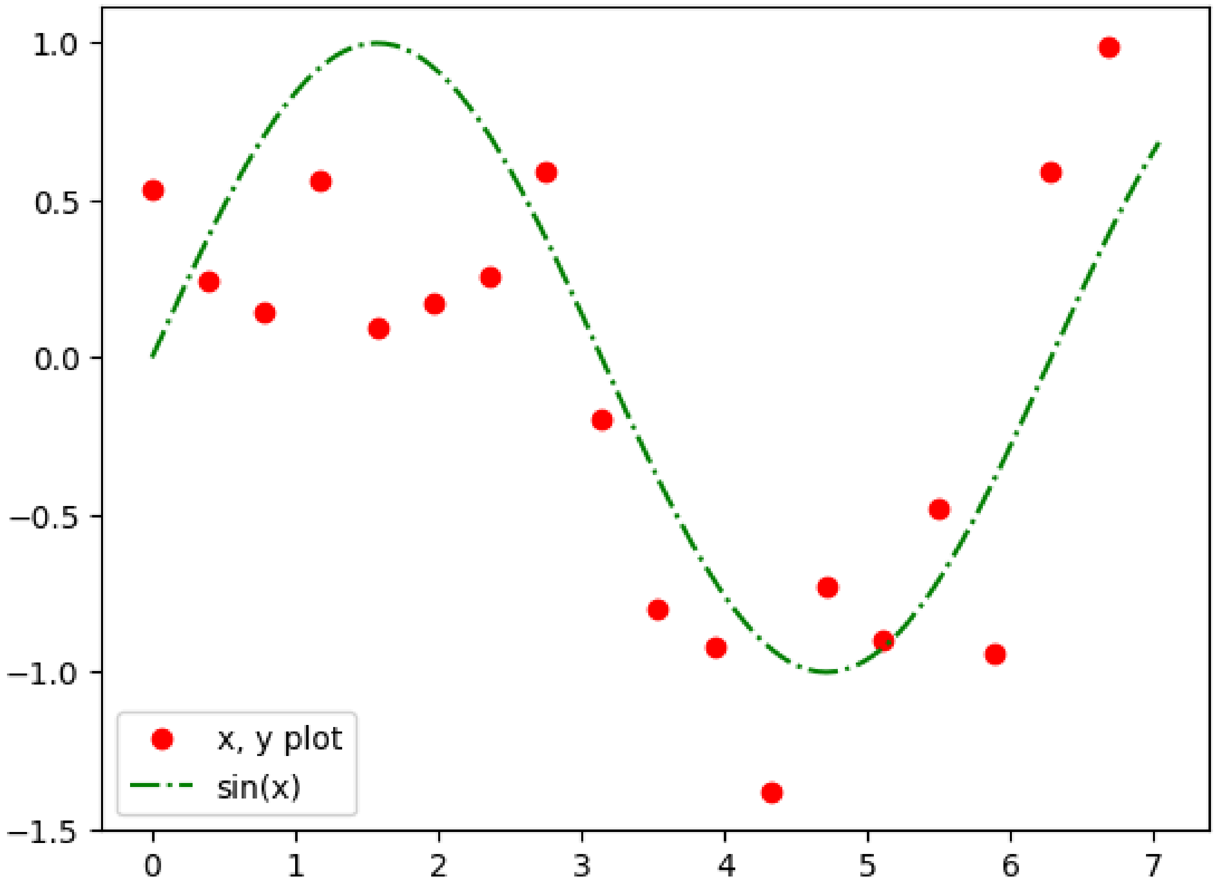
- splrep: Spline interpolation requires two essential steps: (1) a spline representation of the curve is computed, and (2) the spline is evaluated at the desired points.
- BSpline: A basis spline is a nonlinear function constructed of flexible bands that pass through control points to create a smooth curve.

Sources

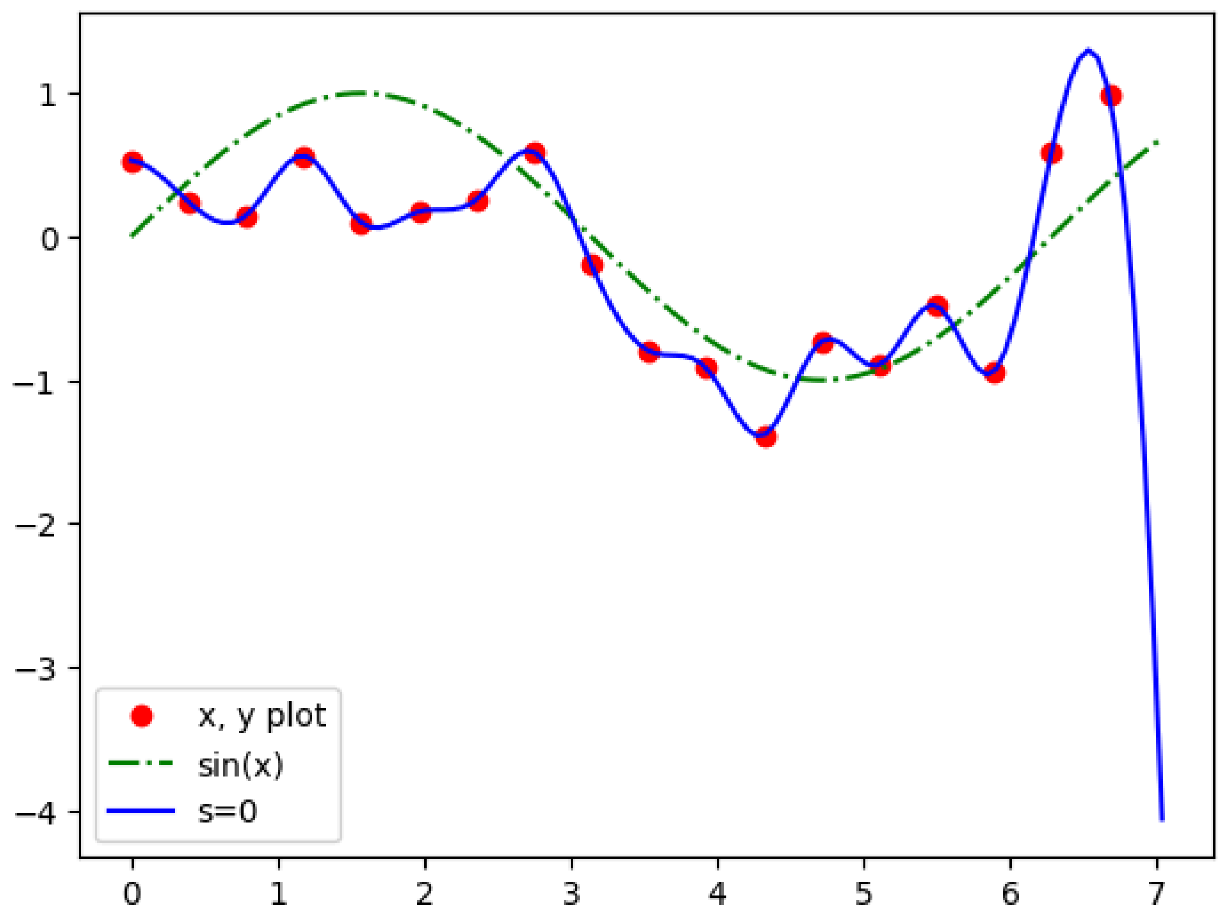
- https://docs.scipy.org/doc/scipy/tutorial/interpolate/smoothing_splines.html
- https://apmonitor.com/wiki/index.php/Main/ObjectBspline

```
# https://docs.scipy.org/doc/scipy/tutorial/interpola
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import splrep, BSpline

x = np.arange(0, 2*np.pi+np.pi/4, 2*np.pi/16)
rng = np.random.default_rng()
y =  np.sin(x) + 0.4*rng.standard_normal(size=len(x))
```

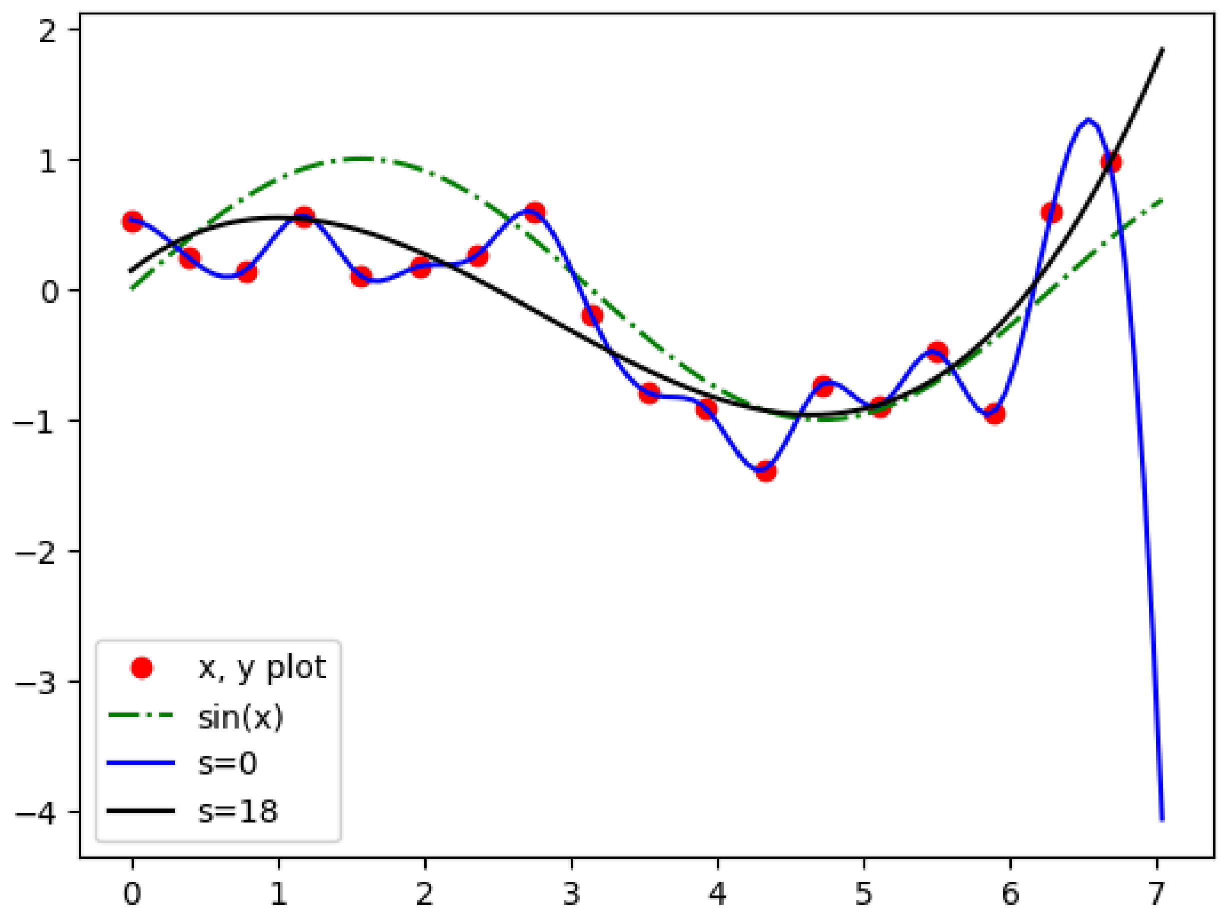
```
tck = splrep(x, y, s=0) # s is used to specify the am
tck_s = splrep(x, y, s=len(x))

xnew = np.arange(0, 9/4, 1/50) * np.pi
plt.plot(x, y, 'o', label = 'x, y plot', color='r') #
plt.plot(xnew, np.sin(xnew), '-.', label='sin(x)', co
# plt.plot(xnew, BSpline(*tck)(xnew), '-', label='s=0
# plt.plot(xnew, BSpline(*tck_s)(xnew), '-', label=f'

plt.legend()
plt.show()
```



```
plt.plot(x, y, 'o', label = 'x, y plot', color='r') #
plt.plot(xnew, np.sin(xnew), '-.', label='sin(x)', co
plt.plot(xnew, BSpline(*tck)(xnew), '-', label='s=0',
# plt.plot(xnew, BSpline(*tck_s)(xnew), '-', label=f'

plt.legend()
plt.show()
```



```
plt.plot(x, y, 'o', label = 'x, y plot', color='r') #
plt.plot(xnew, np.sin(xnew), '-.', label='sin(x)', co
plt.plot(xnew, BSpline(*tck)(xnew), '-', label='s=0',
plt.plot(xnew, BSpline(*tck_s)(xnew), '-', label=f's=

plt.legend()
plt.show()
```



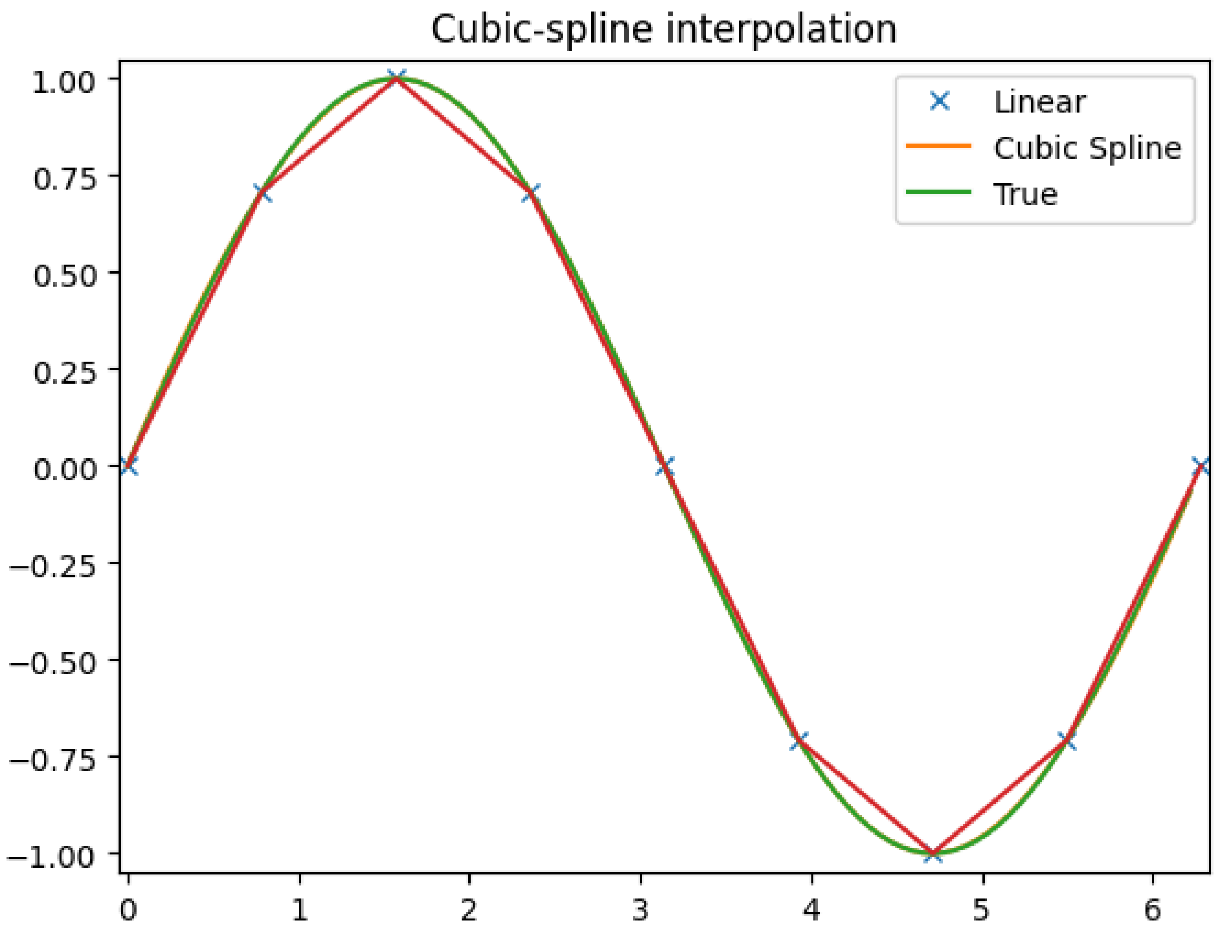
Cubic Spline

- Cubic spline interpolation is a way of finding a curve that connects data points with a degree of three or less

Source

- https://www.geeksforgeeks.org/cubic-spline-interpolation/

```
# https://docs.scipy.org/doc/scipy/tutorial/interpola
import matplotlib.pyplot as plt
from scipy import interpolate

x = np.arange(0, 2*np.pi+np.pi/4, 2*np.pi/8)
y = np.sin(x)
tck = interpolate.splrep(x, y, s=0)
xnew = np.arange(0, 2*np.pi, np.pi/50)
ynew = interpolate.splev(xnew, tck, der=0)

plt.figure()
plt.plot(x, y, 'x', xnew, ynew, xnew, np.sin(xnew), x
plt.legend(['Linear', 'Cubic Spline', 'True'])
plt.axis([-0.05, 6.33, -1.05, 1.05])
plt.title('Cubic-spline interpolation')
plt.show()
```

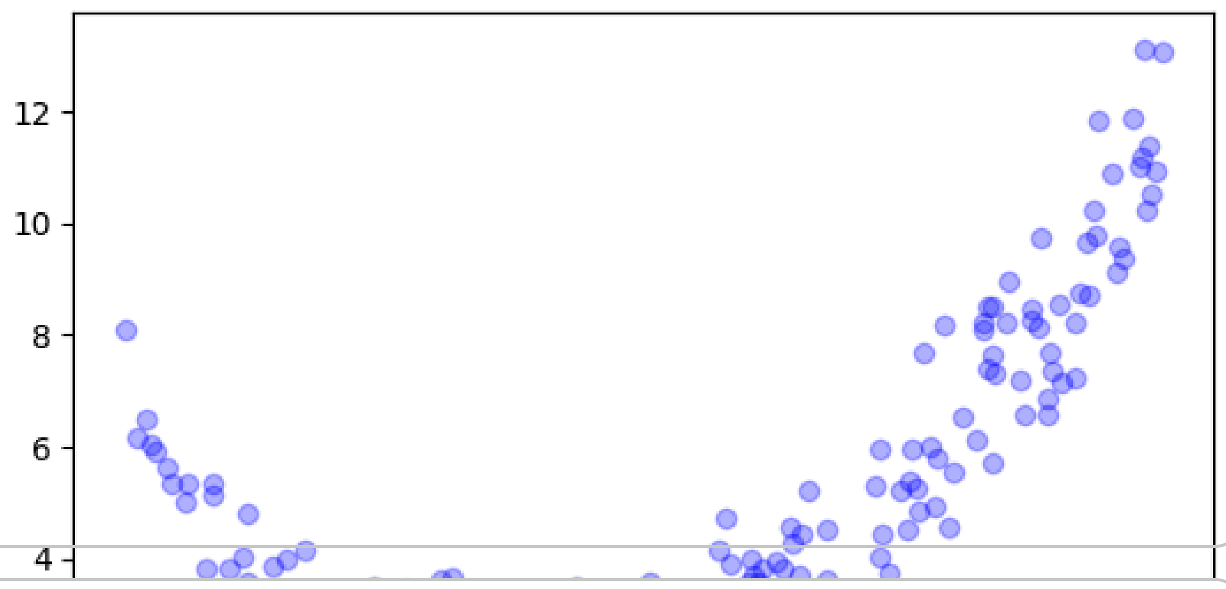


Local Regression

- Local regression is a diferent approach for ftting fexible non-linear func- local regression tions, which involves computing the ft at a target point x0 using only the nearby training observations
- Check out Python LOWESS (Locally Weighted Scatterplot Smoothing)

- https://www.google.com/imgres?imgurl=https%3A%2F%2Fmiro.medium.com%2Fv2%2Fresize%3Afit%3A928%2F1*H3QSO5Q1GJtY-tiBLOOiug.png&tbnid=zODKwyhWlDC45M&vet=12ahUKEwiOna6PwJSCAxUH3MkDHWOIA8oQMygAegQIARBW..i&imgrefurl=https%3A%2F%2Ftowardsdatascience.com%2Flocally-weighted-linear-regression-in-python-3d324108efbf&docid=nFD3cy7xVIYQVM&w=928&h=704&q=local%20regression%20python%20example&ved=2ahUKEwiOna6PwJSCAxUH3MkDHWOIA8oQMygAegQIARBW

```
import numpy as np
import matplotlib.pyplot as plt

X = 6 * np.random.rand(200) - 3
noise = np.random.normal(0, 1, X.shape)
y = 0.8*X**2 + 0.9*X + 2 + noise

plt.scatter(X, y, color='blue', alpha=0.3);
```



```
# https://james-brennan.github.io/posts/lowess_conf/
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.pyplot as plt
import scipy.stats

x = 6 * np.random.rand(200) - 3
```