CSCI2202: Lecture 10
Regression

Finlay Maguire (finlay.maguire@dal.ca)
TA: Ehsan Baratnezhad (ethan.b@dal.ca)
TA: Precious Osadebamwen (precious.osadebamwen@dal.ca)

Overview

- Linear Regression
- Matplotlib Pyplot
- Cost/Error Functions
- Basic Model Fitting
  - Brute-Force/Naive
  - Gradient Descent
  - Analytical Solution
  - Matrix Formulation
- Regression Variants
  - Multiple regression
  - Polynomial regression
  - Regularisation
- Limitations of Linear Regression

Regression

Technique used for the modeling and analysis of numerical data

• Exploits the relationship between two or more variables so that we can gain information about one of them through knowing values of the other

• Regression can be used for prediction, estimation, hypothesis testing, and modeling causal relationships

Advertising Data Set

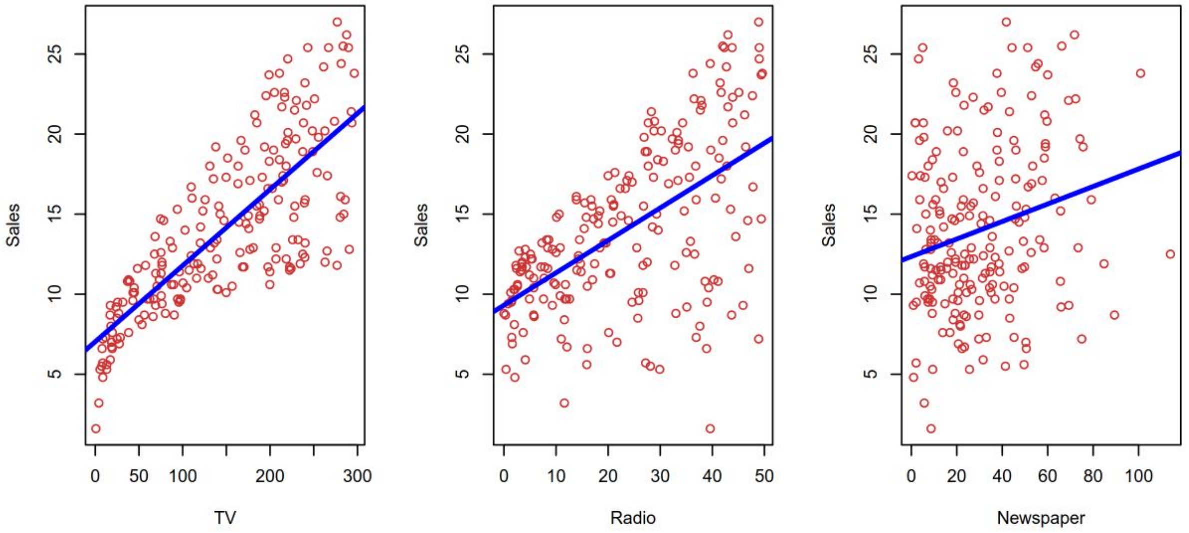
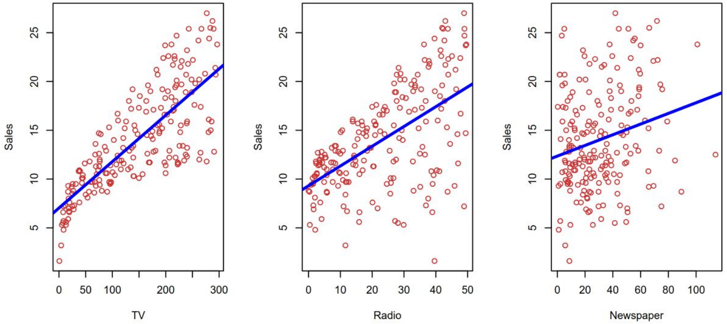


FIGURE 2.1. *The* Advertising *data set. The plot displays* sales*, in thousands of units, as a function of* TV*,* radio*, and* newspaper *budgets, in thousands of dollars, for 200 different markets. In each plot we show the simple least squares fit of* sales *to that variable, as described in Chapter 3. In other words, each blue line represents a simple model that can be used to predict* sales *using* TV*,* radio*, and* newspaper*, respectively.*

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2023). *An introduction to statistical learning Python 2nd Edition*

Advertising Research Questions

- Is there a relationship between advertising budget and sales?
- How strong is the relationship between advertising budget and sales?
- Which media are associated with sales?
- How large is the association between each medium and sales?
- How accurately can we predict future sales?
- Is the relationship linear?
- Is there synergy among the advertising media?



James, G., Witten, D., Hastie, T., & Tibshirani, R. (2023). *An introduction to statistical learning Python 2nd Edition*

Simple Linear Regression

Predict a quantitative response **Y** on the basis of a single predictor variable **X** i.e., regressing Y on/onto X

$$\mathbf{Y} \sim \beta_0 + \beta_1 \mathbf{X}$$

Y names:
Dependent Variable
Outcome Variable
Response Variable
Label

β names:
Intercept / Slope
Coefficients
Weights

X names:
Independent Variable
Predictor Variable
Explanatory Variable
Regression
Feature

Simple Linear Regression

Predict a quantitative response **Y** on the basis of a single predictor variable **X** i.e., regressing Y on/onto X

$$\mathbf{Y} \sim \boldsymbol{\beta}_0 + \boldsymbol{\beta}_1 \mathbf{X}$$

$$\text{sales} \sim \boldsymbol{\beta}_0 + \boldsymbol{\beta}_1 * \text{TV}$$

Y names:
Dependent Variable
Outcome Variable
Response Variable
Label

$\boldsymbol{\beta}$ names:
Intercept / Slope
Coefficients
Parameters
Weights

X names:
Independent Variable
Predictor Variable
Explanatory Variable
Regression
Feature

Simple Linear Regression

Predict a quantitative response **Y** on the basis of a single predictor variable **X** i.e., regressing Y on/onto X

$$\mathbf{Y \sim \beta_0 + \beta_1 X}$$

$$\text{sales} \sim \boldsymbol{\beta_0} + \boldsymbol{\beta_1} * \text{TV}$$

Once we estimate $\hat{\beta}_0 + \hat{\beta}_1$ we can predict future sales on the basis of a particular value of TV advertising by computing:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$$

hat symbol, ˆ , denotes the estimated value for an unknown value

Estimating Coefficients

In practice β_0 + β_1 are unknown so we have to try and find the values that best "fit" our data:

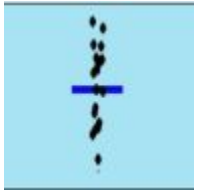
$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$$

Where n is 200 and each x is a TV advertising budget in a specific market and each y is the sales in that market.

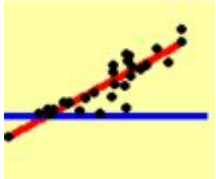
Specifically, which coefficients draw a line that is as close as possible to as many of the points as possible.

Aside: common statistical tests are just linear models

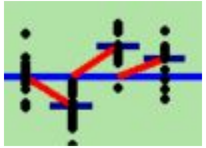
- **y independent of x** = one number (intercept i.e., mean) predicts **y**
  - t-test
  - wilcoxon signed-rank (non-parametric - predicts RANK instead of number)



- **y has linear relationship with x** = intercept + **x** * slope predicts **y**
  - Pearson correlation
  - Spearman correlation (non-parametric - rank)



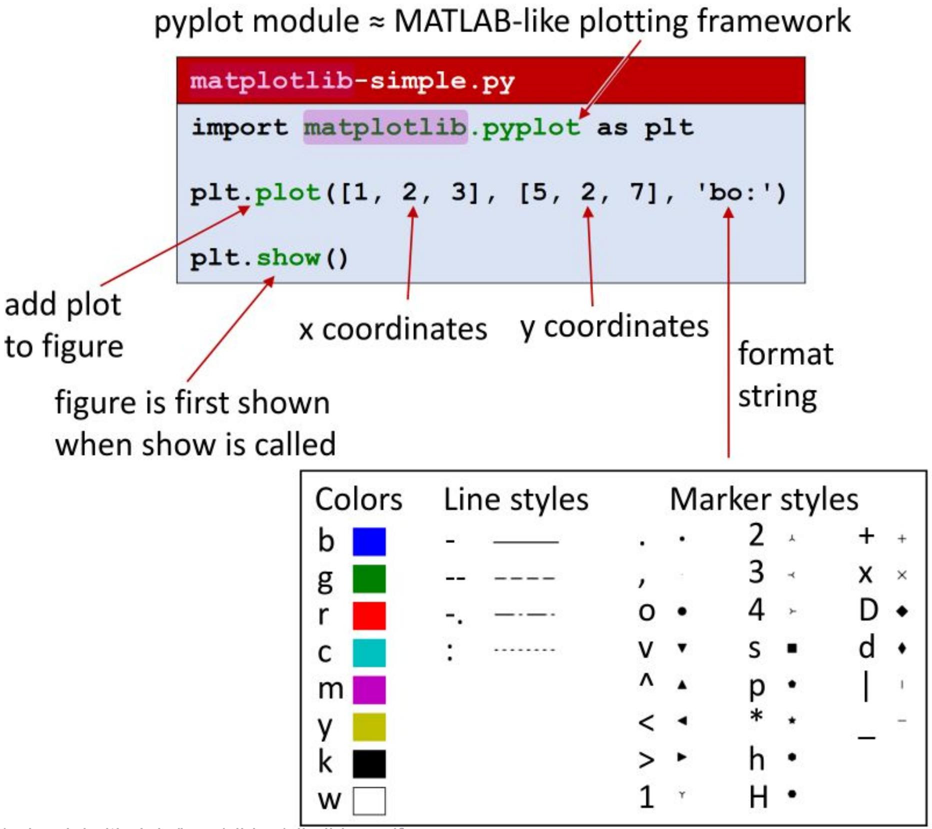
- **y of groups of x are different**: intercept for group 1's x predicts y
  - ANOVA
  - Kruskal-Wallis (non-parametric - rank)

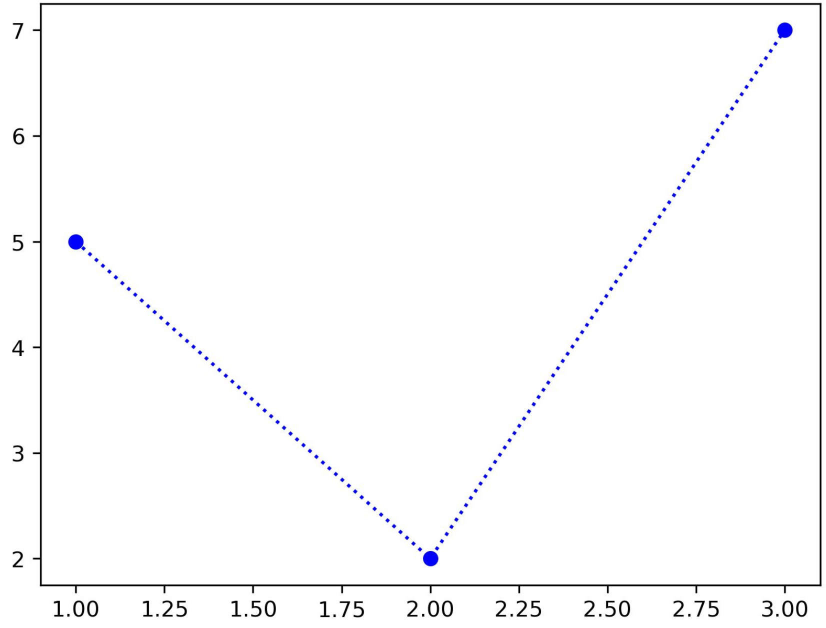


https://lindeloev.github.io/tests-as-linear/

Plotting your data is vital to regression!

Matplotlib pyplot





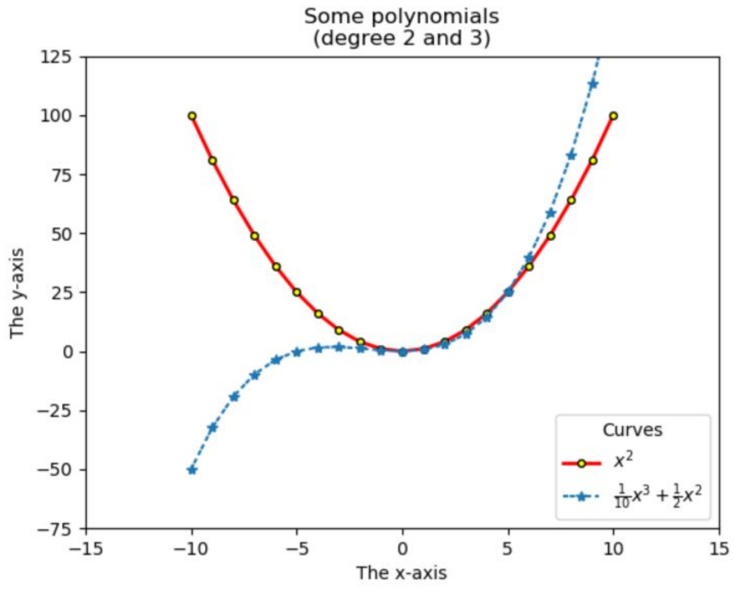
https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

Pyplot offer lots of control of your figure appearances

matplotlib-plot.py

```python
import matplotlib.pyplot as plt
X = range(-10, 11)
Y1 = [x ** 2 for x in X]
Y2 = [x ** 3 / 10 + x ** 2 / 2 for x in X]
plt.plot(X, Y1, color='red', label='$x^2$',
    linestyle='-', linewidth=2,
    marker='o', markersize=4,
    markeredgewidth=1,
    markeredgecolor='black',
    markerfacecolor='yellow')
plt.plot(X, Y2, '*', dashes=(2, 0.5, 2, 1.5),
    label=r'$\frac{1}{10}x^3+\frac{1}{2}x^2$')
plt.xlim(-15, 15)
plt.ylim(-75, 125)
plt.title('Some polynomials\n(degree 2 and 3)')
plt.xlabel('The x-axis')
plt.ylabel('The y-axis')
plt.legend(title='Curves')
plt.show()  # finally show figure
```

LaTeX



matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html

Colors: matplotlib.org/gallery/color/named_colors.html

Linear regression needs more than lines: scatter plots

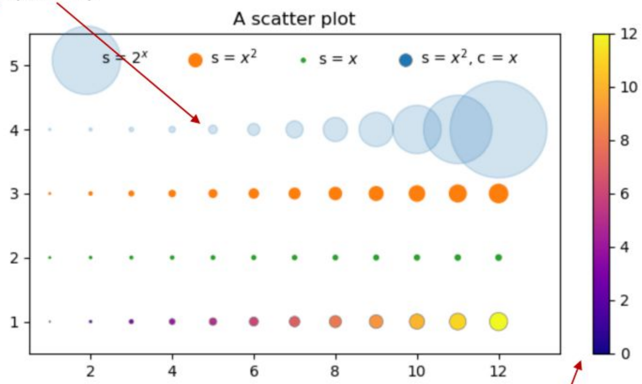
matplotlib-scatter.py

```
import matplotlib.pyplot as plt
n = 13
X = range(n)
S = [x ** 2 for x in X]
E = [2 ** x for x in X]
plt.scatter(X, [4] * n, s=E, label='s = $2^x$', alpha=.2)
plt.scatter(X, [3] * n, s=S, label='s = $x^2$')
plt.scatter(X, [2] * n, s=X, label='s = $x$')
plt.scatter(X, [1] * n, s=S, c=X, cmap='plasma',
    label='s = $x^2$, c = $x$',
    edgecolors='gray', linewidth=0.5)
plt.colorbar()
plt.ylim(0.5, 5.5)
plt.xlim(0.5, 13.5)
plt.title('A scatter plot')
plt.legend(loc='upper center', frameon=False, ncol=4,
           handletextpad=0)
plt.show()
```

colormap (predefined)
color of each point
size ≈ area of each point
point boundary width
point boundary color

manual placement of legend box (default automatic); remove frame; place legends in 4 columns (default 1); reduce space between marks and label

transparency



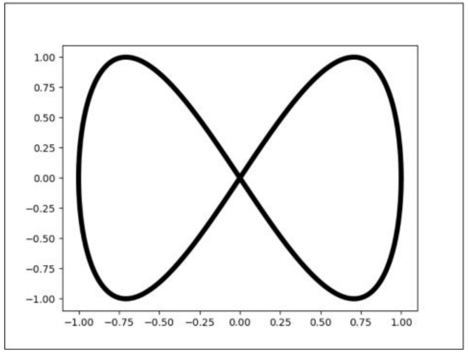
colorbar
(of most recently used colormap)

matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html
matplotlib.org/tutorials/colors/colormaps.html

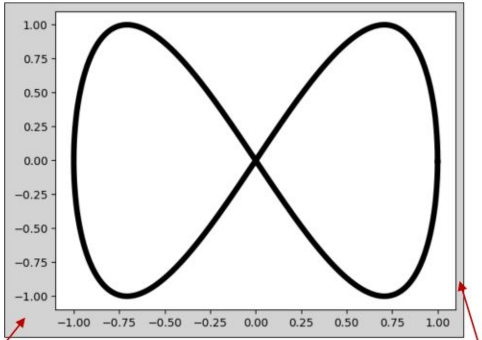
Saving your figures in pyplot

matplotlib-savefig.py

```python
import matplotlib.pyplot as plt
from math import pi, sin, cos

n = 1000
points = [(cos(2 * pi * i / n),
           sin(4 * pi * i / n)) for i in range(n)]
x, y = zip(*points)
plt.plot(x, y, 'k-', linewidth=5)

plt.savefig('butterfly.png')  # save plot as PNG

plt.savefig('butterfly-grey.png',
    dpi=100,                  # dots per inch
    bbox_inches='tight',      # crop to bounding box
    pad_inches=0.1,           # space around figure
    facecolor='lightgrey',    # background color
    format='png')             # optional if file extension

plt.savefig('butterfly.pdf')  # save plot as PDF

plt.show()                    # interactive viewer
```



butterfly.png



butterfly-grey.png

matplotlib.org/api/_as_gen/matplotlib.pyplot.savefig.html

So, how do we fit linear models?

Measuring “Closeness” using least squares

We can measure closeness between a line and points several ways but most common/simplest: least squares

Let $y_i = \beta_0 + \beta_1 x_i$ then $e_i = y_i - y_i$ (or the difference/residual between the ith observed response value and the ith predicted response value) then we can calculate closeness:

Residual sum of squares (RSS) = $e_1^2 + e_2^2 + \ldots + e_n^2$

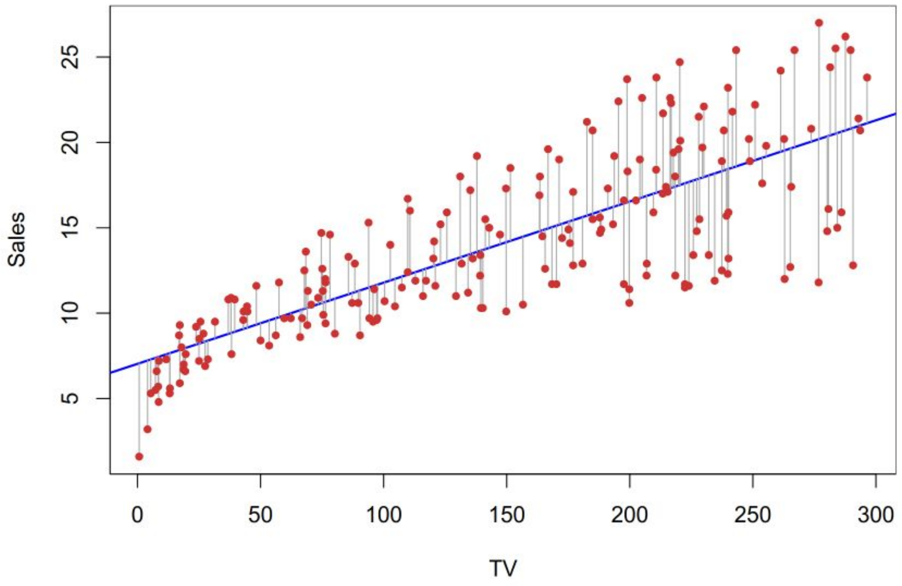
$$\text{RSS} = (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1)^2 + (y_2 - \hat{\beta}_0 - \hat{\beta}_1 x_2)^2 + \cdots + (y_n - \hat{\beta}_0 - \hat{\beta}_1 x_n)^2.$$

$$\min_{\beta_0, \beta_1} \mathbb{E}\left[(y - \beta_0 - \beta_1 x)^2\right]$$

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2023). *An introduction to statistical learning Python 2nd Edition*

RSS as a cost/loss/fit function

Residual sum of squares (RSS) = $\min_{\beta_0, \beta_1} \mathbb{E}\left[(y - \beta_0 - \beta_1 x)^2\right]$




```
def calculate_rss(beta0, beta1, x, y):
    '''beta0 and beta1 - floats
    x, y - np.arrays
    returns float'''
    y_pred = beta0 + beta1 * x
    residuals = y - y_pred
    return np.sum(residuals**2)
```

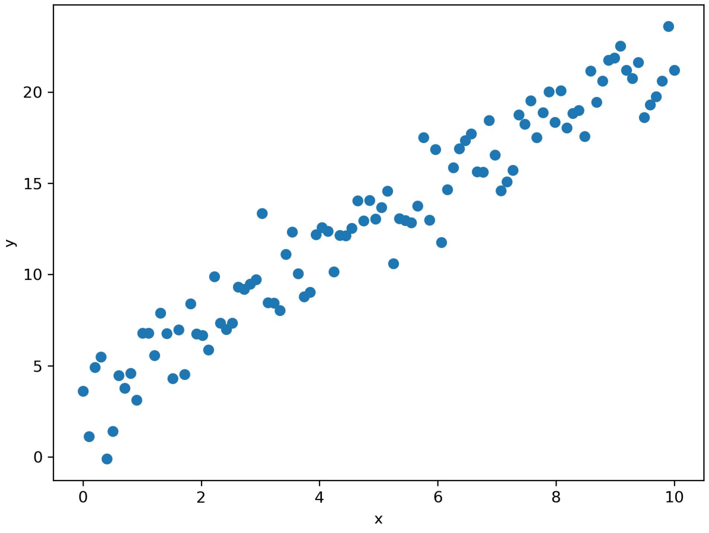
Assume we have done import numpy as np

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2023). *An introduction to statistical learning Python 2nd Edition*

Let’s simulate a simple dataset and start using this `calculate_rss` function to fit linear models

Finding linear parameters (slope + intercept) in python

There are several approaches we can take to finding the optimal parameters values for a model.

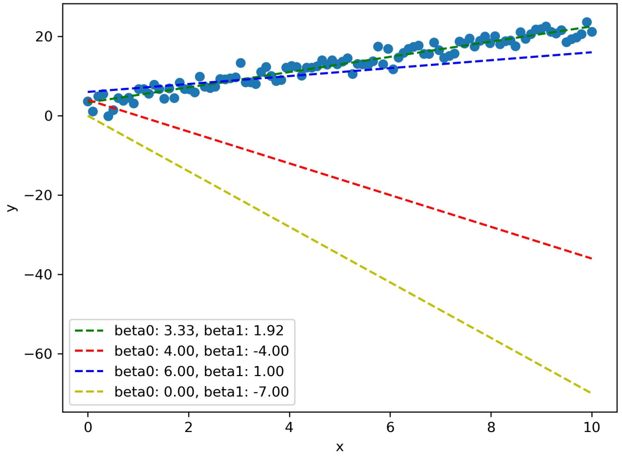



```
rng = np.random.default_rng(42)

x = np.linspace(0, 10, 100)

y_true = 3 + 2 * x

# True relationship: y = 3 + 2x

y = y_true + rng.normal(0, 2, size=len(x))
# Add some noise
```

Brute-force naive approach

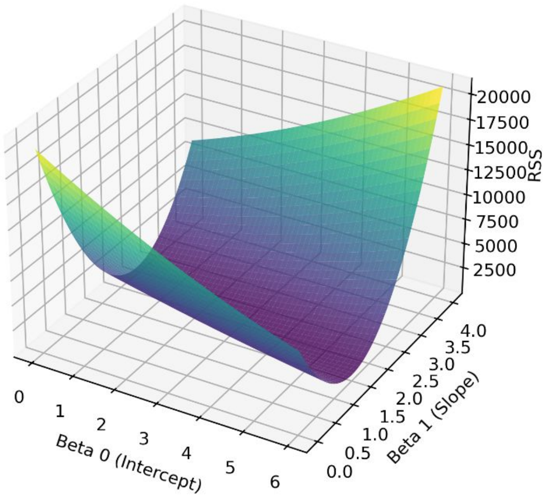
There are several approaches we can take to finding the optimal parameters values for a model.

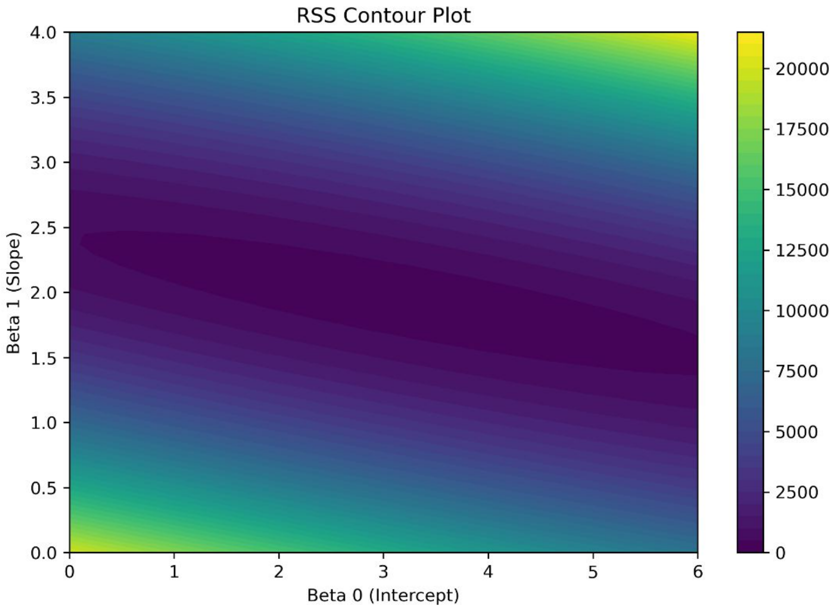
- Naive grid (or random) search
  - Try a bunch of values and pick the best
  - Pros: always works eventually
  - Cons: **eventually** can be infinite




```python
beta_0_values = np.linspace(-10, 10, 100)
beta_1_values =  np.linspace(-10, 10, 100)

min_rss = float('inf')
best_beta_0 = None
best_beta_1 = None

# Nested loop to try all combinations
for beta_0 in beta_0_values:
    for beta_1 in beta_1_values:
        current_rss = calculate_rss(beta_0,
                                    beta_1,
                                    x, y)
        if current_rss < min_rss:
            min_rss = current_rss
            best_beta_0 = beta_0
            best_beta_1 = beta_1
```

Relationship between RSS and parameter values





Find lowest RSS: go down this surface until we get to the bottom.

But how to calculate slope without calculating every possible value?

Gradient descent using partial derivatives of RSS

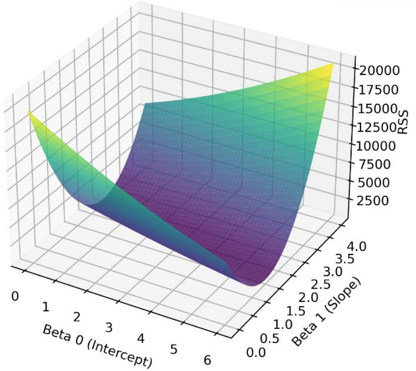
$RSS = \Sigma(y_i - (\beta_0 + \beta_1 x_i))^2 = \Sigma(y_i - \beta_0 - \beta_1 x_i)^2$

2 parameters so need to calculate derivative with respect to β_0 and β_1 i.e., partial derivatives using chain rule.

Chain rule: $(f(g(x)))' = f'(g(x)) \times g'(x)$

$$\partial RSS/\partial\beta_0 = \Sigma\, 2(y_i - \beta_0 - \beta_1 x_i) \times (-1)$$
$$= -2\Sigma(y_i - (\beta_0 + \beta_1 x_i))$$

$$\partial RSS/\partial\beta_1 = \Sigma\, 2(y_i - \beta_0 - \beta_1 x_i) \times (-x_i)$$
$$= -2\Sigma(y_i - \beta_0 - \beta_1 x_i)x_i$$




```
beta_0, beta_1 = 0, 0

learning_rate = 0.0001

prev_rss = calculate_rss(beta_0, beta_1, x, y)

for i in range(10000):

        y_pred = beta_0 + beta_1 * x

        grad_beta_0 = -2 * np.sum(y - y_pred)

        grad_beta_1 = -2 * np.sum((y - y_pred) * x)

        beta_0 = beta_0 - learning_rate * grad_beta_0

        beta_1 = beta_1 - learning_rate * grad_beta_1

        current_rss = calculate_rss(beta_0, beta_1, x, y)

        if abs(prev_rss - current_rss) < 1e-8:

            break

        prev_rss = current_rss
```

Learning Rate is an important parameter

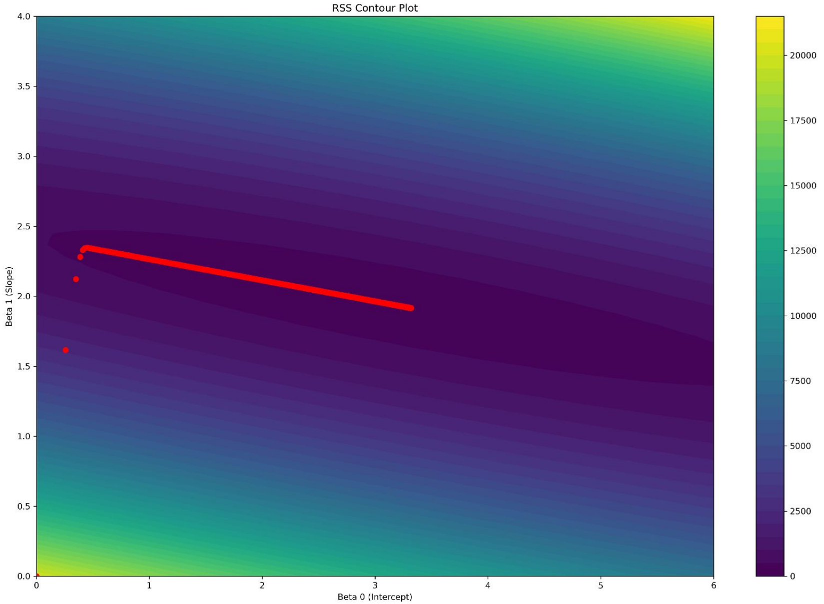
Learning rate is essentially relative “step” size when sliding down the gradient.

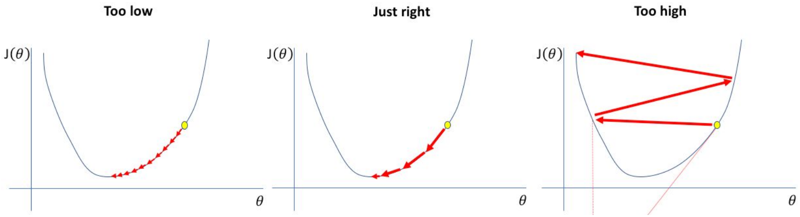
Learning rate too small:

- Convergence becomes extremely slow
- Stuck in local minima
- Can run out of iterations!

Learning rate too large:

- Overshoot optimal value and oscillate
- Failure to converge
- Float overflows





Analytical approach: finding an exact closed-form solution

Can directly calculate solution by solving for $\partial RSS/\partial\beta_0 = 0$ and $\partial RSS/\partial\beta_1 = 0$

$\partial RSS/\partial\beta_0 = -2\Sigma(y_i - \beta_0 - \beta_1 x_i) = 0$

$0 = -2[\Sigma y_i - n\beta_0 - \beta_1\Sigma x_i]$: expand

$n\beta_0 = \Sigma y_i - \beta_1\Sigma x_i$: rearrange

$\beta_0 = (\Sigma y_i - \beta_1\Sigma x_i)/n$: divide by n

$\beta_0 = \bar{y} - \beta_1\bar{x}$: sub $\Sigma y_i/n = \bar{y}$ and $\Sigma x_i/n = \bar{x}$

Special case for OLS - not possible for all models

$\partial RSS/\partial\beta_1 = -2\Sigma(y_i - \beta_0 - \beta_1 x_i)x_i$

$0 = \Sigma x_i y_i - \beta_0\Sigma x_i - \beta_1\Sigma x_i^2$: expand

$\Sigma x_i y_i - (\bar{y} - \beta_1\bar{x})\Sigma x_i - \beta_1\Sigma x_i^2 = 0$: substitute $\beta_0 = \bar{y} - \beta_1\bar{x}$

$\Sigma x_i y_i - \bar{y}\Sigma x_i + \beta_1\bar{x}\Sigma x_i - \beta_1\Sigma x_i^2 = 0$: simplify

$\Sigma x_i y_i - n\bar{y}\bar{x} + \beta_1 n\bar{x}^2 - \beta_1\Sigma x_i^2 = 0$: $\Sigma x_i = n\bar{x}$

$\beta_1(\Sigma x_i^2 - n\bar{x}^2) = \Sigma x_i y_i - n\bar{y}\bar{x}$: rearrange to isolate β_1

$\beta_1 = (\Sigma x_i y_i - n\bar{y}\bar{x})/(\Sigma x_i^2 - n\bar{x}^2)$

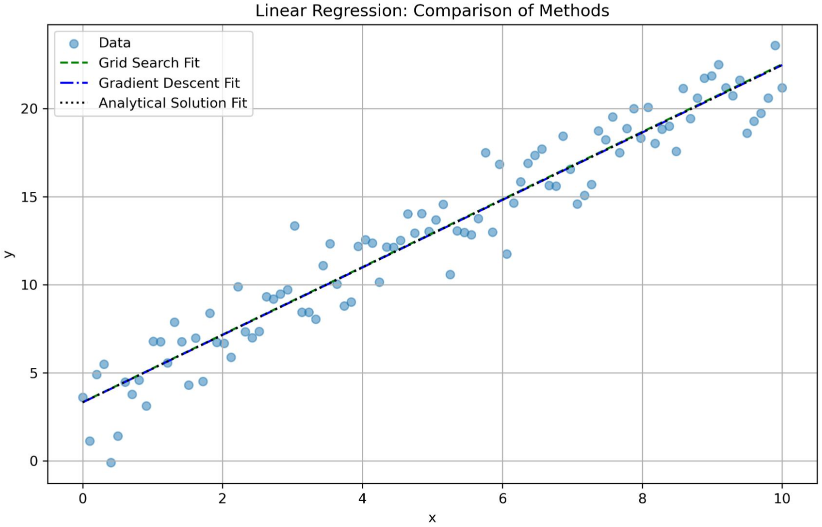
$\beta_1 = \Sigma(x_i - \bar{x})(y_i - \bar{y})/\Sigma(x_i - \bar{x})^2$

Analytical approach: finding an exact closed-form solution

Can directly calculate solution by solving for $\partial RSS/\partial\beta_0 = 0$ and $\partial RSS/\partial\beta_1 = 0$

$\beta_0 = \bar{y} - \beta_1\bar{x}$

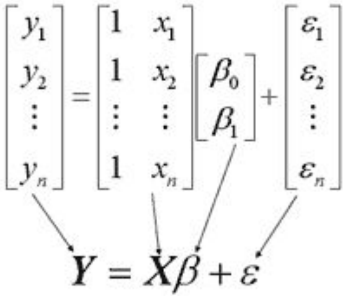
$\beta_1 = \Sigma((x - \bar{x})(y - \bar{y})) / \Sigma((x - \bar{x})^2)$




```
x_mean = np.mean(x)
y_mean = np.mean(y)


numerator = np.sum((x - x_mean) * (y - y_mean))
denominator = np.sum((x - x_mean)**2)
beta_1 = numerator / denominator


beta_0 = y_mean - beta_1 * x_mean


rss = calculate_rss(beta_0, beta_1, x, y)
```

Matrix formulation of analytical solution

Simplify $y = \beta_0 + \beta_1 x + \varepsilon$ to $y = X\beta + \varepsilon$ by adding column of 1s to x.



RSS(β) = $(y - X\beta)^T(y - X\beta)$

Repeating the partial derivative solution in matrix form:

$\beta = (X^TX)^{-1}X^Ty$

```
X = np.column_stack((np.ones(n), x)) # add ones

beta = np.linalg.inv(X.T @ X) @ X.T @ y

beta0, beta1 = beta
```

$$\vec{\beta} = (X^T X)^{-1} X^T \vec{y}$$

$$\begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = \begin{bmatrix} N & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum y \\ \sum x_i y_i \end{bmatrix}$$

Not doing everything by hand!

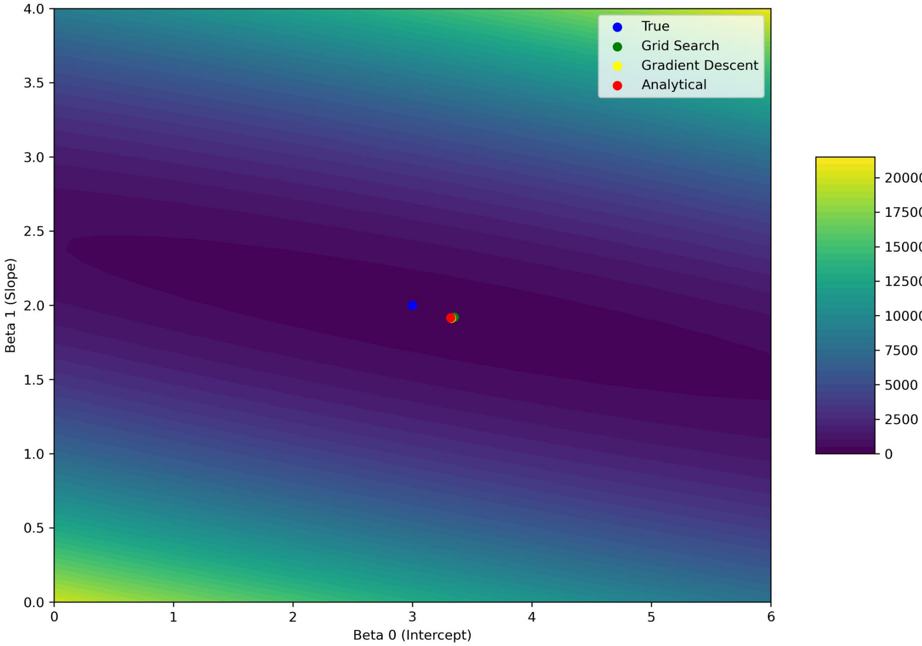
- Several libraries in python that can fit simple linear models

- `numpy` most basic and all interpretation (e.g., calculating RSS, applying values, calculating p-values etc is manual)
- `scipy` provides a more intuitive interface but still relatively simple
- `statsmodels` - python's main statistical modelling library that does regression in a more traditional "statistical" manner (e.g., regression table etc).
- `scikit-learn` - more on that next week!

```
beta0, beta1 = np.polyfit(x, y, 1)


from scipy import stats

result = stats.linregress(x, y)

beta_0 = result.intercept

beta_1 = result.slope


import statsmodels.api as sm

X = sm.add_constant(x)  # add column of 1s for intercept to simplify

model = sm.OLS(y, X)

results = model.fit()

beta_0 = results.params[0]

beta_1 = results.params[1]
```

Summary of ways to fit linear models

- Grid Search: great if you've got nothing better!
    a. **Pros**: simple, easily implemented, works when there is no gradient, often works eventually
    b. **Cons**: slow, expensive, can miss optimal values
- Gradient Descent: workhorse
    a. **Pros**: highly flexible, works well
    b. **Cons**: hyperparameters, issues with convergence/getting stuck, no guarantee of optimal value
- Analytical Solutions: usually best if it exists!
    a. **Pros**: exact solution in one step with no hyperparameters
    b. **Cons:** often doesn't exist and can struggle with large datasets due to matrix inversion and multicollinearity
- Too little data, biases in data, very big or small values - never find exact true parameter values with any method



Regression is multiple courses!

Many variations on simple linear regression

What if we have more than 1 column of data in x?

Generalise to multiple linear regression:

$y = \beta_0 * x_0 + \beta_1 * x_1 + \beta_2 * x_2 + \ldots + \beta_p * x_p + \epsilon$

What if the relationship is not a straight line?

$y = \beta_0 + \beta_1 * x + \beta_2 * x^2 + \ldots + \beta_n * x^n + \epsilon$

What if I want to find the SIMPLEST model?

Change cost function to penalise big parameters

L1 Regularisation: LASSO

Breaks analytical solution => gradient descent

$$\min_{\beta} \left\{ \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 + \alpha \sum_{j=1}^{p} |\beta_j| \right\}$$

```
# multiple regression

beta = np.linalg.inv(X.T @ X) @ X.T @ y


# add polynomial values to X

X_poly = np.ones((len(x), 1))

for d in range(1, degree + 1):

    X_poly = np.column_stack((X_poly, x**d))

beta = np.linalg.inv(X_poly.T @ X_poly) @ X_poly.T @ y

# or polyfit

np.polyfit(x, y, degree)
```

Limitations of linear regression

Linear regression has several limitations:

- **Linearity Assumption**: Fails with highly nonlinear relationships, even with polynomial features.
  - Tree-based methods or neural networks.
- **Outlier Sensitivity**: Coefficients highly influenced by outliers.
  - Robust regression methods like Huber regression.
- **Multicollinearity**: Unstable when predictors are highly correlated.
  - Use regularization or principal component regression.
- **Heteroscedasticity**: When error variance isn't constant.
  - Weighted least squares or transformations.
- **Non-normal Errors**: Affects inference validity.
  - Generalized linear models.

Summary

- Linear Regression
  - *core method for exploring data relationships*
- Matplotlib Pyplot
  - *basic python plotting functions*
- Cost/Error Functions:
  - *Function that measures closeness to known data - form basis of model fitting*
  - *Residual sum of squares i.e., Ordinary Least Squares common in linear regression*
- Basic Model Fitting
  - Brute-Force/Naive: *try lots of values and pick the best*
  - Gradient Descent: *use gradients for direction of parameter updates (needs learning rate)*
  - Analytical Solution: *can use calculus to directly solve OLS*
  - Matrix Formulation: *matrix formulation of analytical solution makes calculation more convenient in numpy*
- Regression Variants
  - Multiple regression: *fit a separate beta to each column of your data*
  - Polynomial regression: *use higher-order combinations of your features*
  - Regularisation: *penalise your cost function based on numbers of parameters*
- Challenges of Linear Regression
  - *Assumes linearity, minimal outliers, constant normal error variance, independence of predictors*