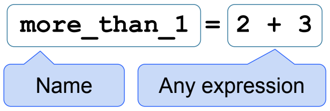
Data 8 Midterm Study Guide — Page 1

- Group by some *treatment* and measure some *outcome*
- Simplest setting: a *treatment group* and a *control group*
- If the *outcome* differs between these two groups, that's evidence of an *association* (or *relation*)
  - E.g., the top-tier chocolate eaters died of heart disease at a lower rate (12%) than chocolate abstainers (17%)
- If the two groups are similar in all ways but the *treatment* itself, a difference in the *outcome* is evidence of *causality*
- When a group is divided *randomly*, it's unlikely that there are systematic differences between sub-groups

Operation	Operator	Example	Value
Addition	+	2 + 3	5
Subtraction	-	2 - 3	-1
Multiplication	*	2 * 3	6
Division	/	7 / 3	2.66667
Remainder	%	7 % 3	1
Exponentiation	**	2 ** 0.5	1.41421



`more_than_1 = 2 + 3`

- Statements don't have a value; they perform an action
- An assignment statement changes the meaning of the name to the left of the = symbol
- The name is bound to a value (not an equation)

Growth rate: The rate of increase per unit time

- After one time unit, a quantity **`x`** growing at rate **`g`** will be

  `x * (1 + g)`

- After **`t`** time units, a quantity **`x`** growing at rate **`g`** will be

  `x * (1 + g) ** t`

- If **`after`** and **`before`** are measurements of the same quantity taken **`t`** time units apart, then the growth rate is

  `(after/before) ** (1/t) - 1`

A **range** is an array of consecutive numbers

- **`np.arange(end)`**: An array of increasing integers from 0 up to **`end`**
- **`np.arange(start, end)`**: An array of increasing integers from **`start`** up to **`end`**
- **`np.arange(start, end, step)`**: A range with **`step`** between consecutive values

The range always includes **`start`** but excludes **`end`**

The **`group`** method aggregates all rows with the same value for a column into a single row in the result

- First argument: Which column to group by
- Second argument: How to combine values
  - **`len`** — number of grouped values (default)
  - **`sum`** — total of all grouped values
  - **`list`** — list of all grouped values

The **`pivot`** method aggregates by a combination of columns **and** organizes the result into a grid

- First argument: Which column gives the pivot columns
- Second argument: Which column gives the pivot rows
- Third argument: (Optional) Column of pivot values
- Fourth argument: (Optional) How to combine values

`max, min`	Maximum or minimum of a sequence
`str, float, int`	Convert a value to a string, number, or integer
`np.prod`	Multiply all elements together
`sum, np.sum`	Add all elements together
`all, np.all`	Test whether all elements are true values (non-zero numbers are true)
`any, np.any`	Test whether any elements are true values (non-zero numbers are true)
`np.count_nonzero`	Count the number of non-zero elements. `False` is considered zero and `True` non-zero.
`np.diff`	Difference between adjacent elements
`round, np.round`	Round each number to the nearest integer (whole number)
`np.cumprod`	A cumulative product: for each element, multiply all elements so far
`np.cumsum`	A cumulative sum: for each element, add all elements so far
`np.exp`	Exponentiate each element
`np.log`	Take the natural logarithm of each element
`np.abs`	Take the absolute value of each element
`np.sqrt`	Take the square root of each element
`np.sort`	Sort the elements

- < and > mean what you expect (less than, greater than)
- <= means "less than or equal"; likewise for >=
- == means "equal"; != means "not equal"
- Both numbers and strings can be compared, but a string cannot be compared to a number
- Comparing strings compares their alphabetical order

A table can be created using...

- **`Table.read_table(`*`url`*`)`** reads an existing table
- **`Table(`*`labels`*`)`** is a table with empty columns
- **`Table()`** is an empty table with no columns

A table **`t`** can be extended using...

- **`t.with_column(`*`label, values`*`)`**
- **`t.with_columns(`*`labels_and_values`*`)`**
- **`t.with_row(`*`row`*`)`** and **`t.with_rows(`*`rows`*`)`**

- **`t.select(`*`columns`*`)`** keeps only certain columns
- **`t.drop(`*`columns`*`)`** keeps all but certain columns
- **`t.relabeled(`*`a, b`*`)`** substitutes label **`a`** with **`b`**
- Each **`column`** has a label and an index (starting at 0)
  - The resulting array has one **`item`** per row

- **`t.where(`*`column, value`*`)`** keeps all rows containing a certain value in a column
- **`t.where(`*`condition`*`)`** keeps all rows for which the condition is true, where ***`condition`*** is a boolean array
- **`t.take(row_numbers)`** keeps the numbered rows
- **`t.sort(column)`** sorts the rows in increasing order
- Each **`row`** has an index (starting at 0)
  - The resulting row value has one **`item`** per column
  - Row items can be selected by column index or label

Tables can generate charts using these methods:

- **`barh`** draw a horizontal bar chart
- **`hist`** draw a histogram (later)
- **`scatter`** draw a scatter diagram (later)

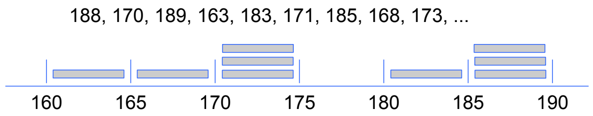
Data 8 Midterm Study Guide — Page 2

Types of values: All values in a column should have the same type **and** be comparable to each other in some way

- **Categorical** — Each value is from a fixed inventory
  - May or may not have an ordering
  - Categories are the same or different
  - Allows grouping by value (`group, pivot, join`)
- **Numerical** — Each value is from a fixed scale
  - Numerical measurements are ordered
  - Allows binning by value (`bin, hist`)

Binning is counting the number of numerical values that lie within ranges, called bins.

- Bins are defined by their lower bounds (inclusive)
- The upper bound is the lower bound of the next bin



A **histogram** has two defining properties:

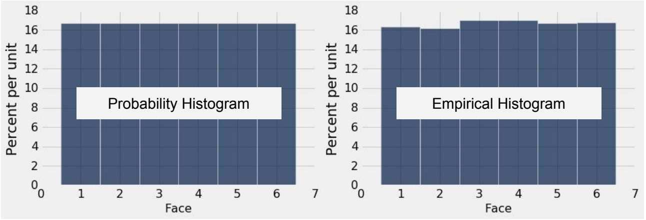
- The bins are contiguous (though some might be empty) and are drawn to scale
- The **area** of each bar is proportional to the number of entries in the bin

A histogram drawn *on the density scale* has total area 1

Vertical axis units: Proportion / Unit on the horizontal axis

Height of a bar: (proportion of entries in bin) / (width of bin)

- A histogram of proportions of all possible outcomes of a *known* random process is called a *probability histogram*
- A histogram is a summary visualization of a *distribution*
- A histogram of proportions of actual outcomes generated by sampling is called an *empirical histogram*
- When a histogram is created from data, it is empirical



Sampling from a population means selecting a subset

Probability sample: Use a random procedure for which you *know in advance* the chance that each subset of the population will enter the sample (called the *distribution*)

Uniform random sample example: choose each movie at random with equal chance

Each item has equal chance of being drawn

- **With replacement**: Items can be drawn more than once, and no draw affects the chance of any other draw
- **Without replacement**: Once an item is selected, it cannot be selected again

Name

Argument names (parameters)

Return expression

Body

```
def spread(values):
    return max(values) - min(values)
```

```
for _i_ in ______np.arange(12)______:
    print(i)
```

The body is executed **for** every item in a sequence

The body of the statement can have multiple lines

The body should do something: print, assign, hist, etc.

Law of averages: If a chance experiment is **repeated independently** under identical conditions, in the long run:

The proportion of times that an event occurs gets closer and closer to the theoretical probability of the event

Biased estimate: On average across all possible samples, the estimate is either too high or too low.

- Bias creates a systematic error in one direction.
- Good estimates typically have low bias.
- The value of an estimate **varies** across samples.
- High variability makes it hard to estimate accurately.
- Good estimates typically have low variability.

Mean (or average): Balance point of the histogram

- ***Not*** the "half-way point" of the data; the mean is not the median unless the histogram is symmetric
- If the histogram is skewed, then the mean is pulled away from the median in the direction of the tail

Standard deviation (SD) =

root 5	mean 4	square of 3	deviations from 2	average 1

Measures roughly how far off the values are from average

"average ± *z* SDs"

- *z* measures "how many SDs above average"
- If *z* is negative, the value is below average
- *z* is called **standard units**
- Almost all standard units are in the range (-5, 5)
- Convert a value to standard units: *z* = (value - average) / SD

Percent in Range	**All Distributions**	**Normal Distribution**
average ± 1 SD	at least 0%	about 68%
average ± 2 SDs	at least 75%	about 95%
average ± 3 SDs	at least 88.888...%	about 99.73%

Correlation Coefficient (*r*) =

average of	product of	x in standard units	and	y in standard units

Measures how clustered the scatter is around a straight line

- $-1 \le r \le 1$
- r = ± 1 if the scatter is a perfect straight line
- *r* is a pure number, with no units
- *r* is not affected by changing units of measurement
- *r* is not affected by switching the horizontal and vertical axes

Regression to the mean: a statement about x and y pairs

- Measured in *standard units*
- Describing the deviation of x from 0 (the average of x's)
- And the deviation of y from 0 (the average of y's)

On average, y deviates from 0 less than x deviates from 0

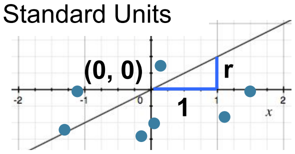
Regression Line

$$y_{(\mathrm{su})} = \underbrace{r}_{\text{Correlation}} \times x_{(\mathrm{su})}$$

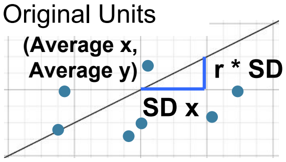
In original units, the regression line has this equation:

$$\underbrace{\frac{\text{estimate of } y \ - \ \text{average of } y}{\text{SD of } y}}_{\text{y in standard units}} = r \times \underbrace{\frac{\text{the given } x \ - \ \text{average of } x}{\text{SD of } x}}_{\text{x in standard units}}$$

Standard Units



Original Units



```
census = Table.read_table(census_url)
Census.labels       # Returns a tuple of labels
Census.num_columns  # Returns an integer number of columns
Census.num_rows     # Returns an integer number of rows
census.row(5)       # Returns a row, which has items
census.column(2)    # Returns a column, which is an array
census.row(0).item(2) == census.column(2).item(0)

# Filtering on one or more conditions
census.where('AGE', 40)
census.where(census.column(2) < 40)
census.where(np.logical_or(census.column('AGE') == 18,
                           census.column('AGE') == 19))
census.where(np.logical_and(census.column('AGE') >= 18,
                            census.column('AGE') <= 19))
def age_group(age):
    if age < 2:
        return 'Baby'
    elif age < 13:
        return 'Child'
    elif age < 20:
        return 'Teen'
    else:
        return 'Adult'
population = census.with_columns([
    'Age Group', census.apply(age_group, 'AGE'),
    'Male/Female', census.column('SEX'),
    'Population', census.column('2014')])

population.group('Age Group')
# Creates a table with labels: Age Group | count

sums = population.select([0, 2]).group('Age Group', sum)
# Creates a table with labels: Age Group | Population sum

population.pivot('Age Group', 'Male/Female')
# Counts labeled: Male/Female | Adult | Baby | Child | Teen

population.join('Age Group', sums)
# Adds a column for the Population sum of each age group

top = Table.read_table('top_movies.csv')
top.take([3, 18, 100])
top.take(np.arange(0, top.num_rows, 40))
np.random.randint(3, 8)  # select randomly from 3, 4, 5, 6, 7
start = np.random.randint(0, 10)
top.take(np.arange(start, top.num_rows, 10))

die = Table().with_column('Face', [1, 2, 3, 4, 5, 6])
dice_bins = np.arange(0.5, 7, 1)
die.hist(bins=dice_bins)
die.sample(10, with_replacement=True)

for i in np.arange(5):
    print(i)

# Generating a table of dice rolls
def roll_once(n):
    return np.random.randint(1, n+1)
rolls = Table(['First roll', 'Second roll'])
for i in np.arange(1000):
    row = [roll_once(6), roll_once(6)]
    rolls.append(row)
rolls.hist(bins=dice_bins)
```

`np.average, np.mean`	Average of values
`np.std`	Standard deviation of values
`np.var`	Variance of values (standard deviaion squared)
`Table.append`	Append a row or all rows of a table.
`Table.append_column`	Appends a column to the table.
`Table.apply`	Returns an array where a function is applied to each item in a column.
`Table.bin`	Group values by bin and compute counts per bin by column.
`Table.column`	Return the values of a column as an array.
`Table.drop`	Return a table with only columns other than selected label or labels.
`Table.group`	Group rows by unique values in a column; count or aggregate other values.
`Table.join`	Generate a table with the columns of self and other, containing rows for all values of a column that appear in both tables.
`Table.pivot`	Group rows by unique values in two columns; count or aggregate values from a third column.
`Table.relabel`	Change the label of a column
`Table.relabeled`	Return a new table with a label changed.
`Table.sample`	Returns a new table where k rows are randomly sampled from the original table.
`Table.select`	Return a table with only the selected columns.
`Table.sort`	Return a table of rows sorted according to the values in a column.
`Table.take`	Return a table of the rows taken by index.
`Table.where`	Return a table of the rows for which the column matches a value or is a non-zero value.
`Table.with_column` `Table.with_columns`	Return a table with an additional or replaced column or columns.
`Table.with_row` `Table.with_rows`	Return a table with an additional row or rows.

```
annual_growth_rate = 0.035
ten_year_growth_rate = (1 + annual_growth_rate) ** 10 - 1
ten_year_growth_rate # evaluates to: 0.410598760621121

baseline_high = 14.48
highs = np.array([baseline_high - 0.880, baseline_high - 0.093,
                  baseline_high + 0.105, baseline_high + 0.684])
highs # evaluates to: array([ 13.6   ,  14.387,  14.585,  15.164])
(9/5) * highs + 32 # array([ 56.48  ,  57.89,  58.253 ,  59.295])

def percent(x):
    if x < 1e-8:
        return 0.0
    elif x < 0.00005:
        return 100 * x
    else:
        return round(100 * x, 2)
percent(1/6) # returns 16.67
percent(1/6000) # returns 0.02
percent(1/60000) # returns 0.0016666666666666668
percent(1/60000000000) # returns 0.0

Table(['Odd', 'Even']).with_row([3, 4])
Table(['Odd', 'Even']).with_rows([[3, 4], [5, 6], [7, 8]])
Table().with_column('Odd', [3, 5, 7])
Table().with_columns(['Odd',  [3, 5, 7], 'Even', [4, 6, 8]])
```

```python
def standard_units(any_numbers):
    return (any_numbers - np.mean(any_numbers))/np.std(any_numbers)

def correlation(t, x, y):
    return np.mean(standard_units(t.column(x))*standard_units(t.column(y)))

def slope(table, x, y):
    r = correlation(table, x, y)
    return r * np.std(table.column(y))/np.std(table.column(x))

def intercept(table, x, y):
    a = slope(table, x, y)
    return np.mean(table.column(y)) - a * np.mean(table.column(x))

def fit(table, x, y):
    a = slope(table, x, y)
    b = intercept(table, x, y)
    return a * table.column(x) + b

# Regression effect
heights = Table.read_table('heights.csv')
six_foot_fathers = heights.where(np.round(heights.column('father')) == 72)
six_foot_fathers.hist('son', bins=np.arange(55.5, 80, 1))
np.count_nonzero(six_foot_fathers.column('son') < 72) / six_foot_fathers.num_rows
np.mean(six_foot_fathers.column('son'))

# Graph of averages
fathers = heights.column('father')
sons = heights.column('son')
heights_su = Table().with_columns([
        'father (su)', standard_units(fathers),
        'son (su)', standard_units(sons)])
rounded = heights_su.with_column('father (su)', np.round(father_su))
rounded.join('father (su)', rounded.group(0, np.average)).scatter(0, s=80)

# Regression model
def signal_and_noise(x, r):
    return r * x + np.random.normal() * (1-r**2)**0.5

def regression_model(r, sample_size):
    pairs = Table(['x', 'y'])
    for i in np.arange(sample_size):
        x = np.random.normal()
        y = signal_and_noise(x, r)
        pairs.append([x, y])
    return pairs

regression_model(1/2, 1000).scatter('x', 'y')

def compare(true_r, sample_size):
    pairs = regression_model(true_r, sample_size)
    estimated_r = correlation(pairs, 'x', 'y')
    pairs.scatter('x', 'y', fit_line=True, s=10)
    plt.plot([-3, 3], [-3 * true_r, 3 * true_r], color='g', lw=4)
    print("The true r is ", true_r, " and the estimated r is ", estimated_r)

compare(0.5, 1000) # Compare true line to regression line
```

Data 8 Final Study Guide — Page 1

Higher-order function: A function that returns a function

```
def <name>(x):
    def <inner name>(y):
        return <some expression involving x and y>
    return <inner name>
```

The returned *function* takes one argument (y) but computes its result from two values (x and y).

The regression line is the one that minimizes the (root) mean squared error of a collection of paired values

- The slope and intercept are unique for linear regression

```
def mean_squared_error(table, x, y):
    def for_line(a, b):
        estimate = (a * table.column(x) + b)
        return np.average((table.column(y) - estimate) ** 2)
    return for_line
long = faithful.where('eruptions', are.above(3))
mse_long = mean_squared_error(long, 1, 0)
a, b = minimize(mse_long) # regression line is a * x + b
```

Simple regression: one input ➔ one output
Multiple regression: many inputs ➔ one output

GPA = a_{days} * days + $a_{contributions}$ * contributions + b

We could find a's and b by minimizing mean squared error

Inference: Making conclusions from random samples
Population: The entire set that is the subject of interest
Parameter: A quantity computed for the entire population
Sample: A subset of the population

In a **Random Sample**, we know the chance that any subset of the population will enter the sample, in advance

Statistic: A quantity computed for a particular sample

Estimation is a process with a random outcome

Population (fixed) → Sample (random) → Statistic (random)

A 95% **Confidence Interval** is an interval constructed so that it will contain the parameter for 95% of samples

For a particular sample, the interval either contains the parameter or it doesn't; the process works 95% of the time

Resampling: When we wish we could sample again from the population, instead sample from the sample

The 80th percentile is the value in a set that is at least as large as 80% of the elements in the set

For `s = [1, 7, 3, 9, 5]`, `percentile(80, s)` is 7

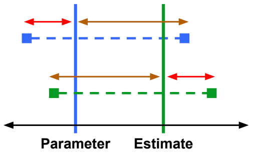
The 80th percentile is ordered element 4: `(80/100) * 5`

Percentile | Size of set

For a percentile that does not exactly correspond to an element, take the next greater element instead

`percentile(10, s)` is 1 `percentile(20, s)` is 1
`percentile(21, s)` is 3 `percentile(40, s)` is 3

If an interval **around the parameter** contains the estimate, then a (reflected) interval of the same width **around the estimate** contains the parameter (and vis versa)



K-nearest Neighbor classifier

Goal: Using a set of labeled examples, define a classifier that can predict the label of an unlabeled example
Example: A set of features (i.e., a row in a table of features)
Data preparation: *Randomly* divide all available examples into three sets: training, validation, and test
Classification: Find the K examples in the training set that are most similar to the example and average the labels
Accuracy: The proportion of examples in a set (e.g., the test set or validation set) that are classified with the correct label

```
def distance(pt1, pt2):
    tot = 0
    for i in range(len(pt1)):
        tot = tot + (pt1[i] - pt2[i])**2
    return math.sqrt(tot)

def computetablewithdists(training, p):
    dists = np.zeros(training.num_rows)
    attributes = training.drop('Class')
    for i in np.arange(training.num_rows):
        dists[i] = distance(attributes.row(i), p)
    return training.with_column('Distance', dists)

def closest(training, p, k):
    withdists = computetablewithdists(training, p)
    sortedbydist = withdists.sort('Distance')
    topk = sortedbydist.take(np.arange(k))
    return topk

def majority(topk):
    if topk.where('Class',1).num_rows>topk.where('Class',0).num_rows:
        return 1
    else:
        return 0

def classify(training, p, k):
    closestk = closest(training, p, k)
    topkclasses = closestk.select('Class')
    return majority(topkclasses)

def evaluate_accuracy(training, test, k):
    testattrs = test.drop('Class')
    numcorrect = 0
    for i in range(test.num_rows):
        # Run the classifier on the ith patient in the test set
        c = classify(training, testattrs.rows[i], k)
        # Was the classifier's prediction correct?
        if c == test.column('Class').item(i):
            numcorrect = numcorrect + 1
    return numcorrect / test.num_rows
```

Feature Selection Using the Validation Set

Goal: Select features and K to maximize accuracy
Overfitting: Accuracy is higher on the set used for feature selection than another set because of peculiarities of the set
Selecting Features: Maintain a set of selected features, which starts out empty; Each added feature is the feature that improves validation set accuracy the most, in combination with the features already added

Inferential idea: The variability of a statistic for different samples is similar to the variability of that statistic for resamples

Computing a confidence interval for an estimate from a sample:

- Collect a random sample
- Compute your estimate (e.g., sample average)
- Resample K samples from the sample, with replacement.
    - Compute the same statistic for each resampled sample
    - Take *percentiles* of the deviations around the estimate
    - 95%: (estimate - 97½% deviation, estimate - 2½% deviation)

```
def confidence_interval_95_percent(sample, label, f):
    deviations = Table(['Resample #', 'Deviation'])
    n = sample.num_rows
    stat = f(sample.column(label))
    for i in np.arange(400):
        resample = sample.sample(n, with_replacement=True)
        dev = f(resample.column(label)) - stat
        deviations.append([i, dev])
    return (stat - percentile(97.5, deviations.column(1)),
            stat - percentile( 2.5, deviations.column(1)))
```

Data 8 Final Study Guide — Page 2

Total Variation Distance (TVD):

- For each category, compute the difference in proportions between two distributions
- Take the absolute value of each difference
- Sum and divide by 2

```
def total_variation_distance(column, other):
    return sum(np.abs(column - other)) / 2
```

`np.random` is a collection of sampling procedures

`np.random.randint(lower, upper)`

An integer lower bound (inclusive)

An integer upper bound (exclusive)

`np.random.multinomial(n, distribution)`

The number of samples to draw

An array or list containing the chance for each category

Does a sample look like a random sample from a distribution?

- E.g., Do the counts of races on a jury panel look like a random sample from the proportion of races in a county?
- Sample many times from the distribution and compute the TVD of each sample from the original distribution
    - The result is called an *empirical distribution* of a statistic
    - It approximates the *sampling distribution* of the statistic
- Compare the observed TVD between the sample and the original distribution to this *empirical distribution*

```
def empirical_distribution(table, label, size, k, f):
    stats = Table(['Sample #', 'Statistic'])
    for i in np.arange(k):
        sample = table.sample_from_distribution(label, size)
        statistic = f(sample)
        stats.append([i, statistic])
    return stats
```

Step 1: The Hypotheses

- A test chooses between two views of how data were generated
- *Null hypothesis* proposes that data were generated at random
- *Alternative hypothesis* proposes some effect other than chance

Step 2: The Test Statistic

- A value that can be computed for the data and for samples

Step 3: The Sampling Distribution of the Test Statistic

- What the test statistic might be if the null hypothesis were true
- Approximate the sampling distribution by an empirical distribution

Step 4: Resolve choice between null and alternative hypotheses

- Compare observed test statistic to its empirical distribution under the null hypothesis
- If the observed value is **consistent** with the distribution, then the test *does not* support the alternative hypothesis

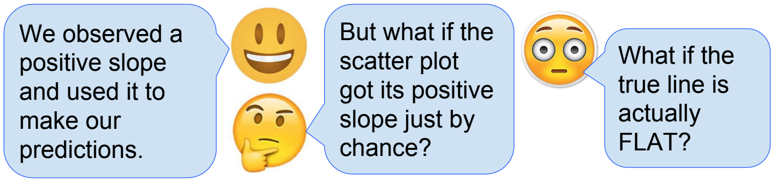
P-Value: The chance, under the null hypothesis, that the test statistic is equal to the value that was observed or is even further in the direction of the alternative.

Statistically Significant: The P-value is less than 5%

Highly Statistically Significant: The P-value is less than 1%

Bootstrap A/B test using means

- *n* individuals in Category A; *m* in Category B
- Under the null hypothesis, samples in both categories are drawn from the same underlying distribution.
- So draw (*n*+*m*) times at random with replacement from the entire sample. Assign *n* of the draws to Category A and *m* to Category B.
- Find the difference between the means of these two new groups.
- Repeat the last two steps many times and compare with the difference between the two original sample means.

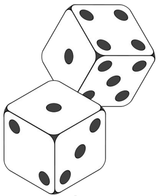

We observed a positive slope and used it to make our predictions.
But what if the scatter plot got its positive slope just by chance?
What if the true line is actually FLAT?

- **Bootstrap the scatter plot & find the slope of the regression line through the bootstrapped plot many times.**
- Draw the empirical histogram of all the resampled slopes.
- Get the "middle 95%" interval: that's an approximate 95% confidence interval for the slope of the true line.
- **Null hypothesis:** The slope of the true line is 0.
    - Construct a bootstrap confidence interval for the true slope.
    - If the interval doesn't contain 0, reject the null hypothesis.
    - If the interval does contain 0, there isn't enough evidence to reject the null hypothesis.

- **Residual**: Difference between y and regression line height at x.
- **Regression Model**: y is a linear function of x + normal "noise"
- Residual plot looks like a formless "noise" cloud under this model
- Average of residuals is always 0 for any scatter diagram

$$|r| \;=\; \frac{\text{SD of fitted values of } y}{\text{SD of observed values of } y} \qquad \text{SD of residuals} \;=\; \sqrt{1-r^2} \times \text{SD of } y$$

- **Experiment**: An occurrence with an uncertain outcome
- **Outcome**: The result of an experiment
- **Sample Space**: The set of all possible outcomes for the experiment
- **Event**: A subset of possible outcomes that have some property of interest
- **Probability**: The proportion of experiments for which the event occurs
- **Distribution**: The probability of all events



Multinomial distributions have a finite sample space

- Distribution with a fixed number of possible outcomes
- All outcomes are mutually exclusive for an experiment
- The sum of the chances of the outcomes is 1
- The probability of an event is the sum of the chances of the outcomes in which the event occurs

Conditional Probability: The chance of an event B **given** an event A

Both events describe the same experiment & sample space

- Find all outcomes in which A occurs
- Divide the chances of A's outcomes by P(A)

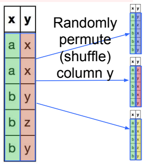
Conditional Distribution

- Find all outcomes in which B also occurs
- Sum the updated chances of these outcomes: P(B | A) = P(A, B) / P(A)
- Bayes' Rule: P(A | B) = P(B | A) * P(A) / P(B)

Comparing samples: Do two groups differ in some attribute?

- **E.g.**: Among U.S. men who are in heterosexual relationships, is there an association between marriage and employment status?
- **Null hypothesis**: In the United States, the distribution of employment status among married men is the same as among unmarried men who live with their partners.
- **Inferential Idea:** If marital status and employment status were not connected in any way, then we could replicate the sampling process by replacing each man's employment status by a randomly picked employment status from among all the men
    - Permute (shuffle) the outcome column K times
    - A shuffled table pairs each example with a random outcome
    - Compute the total variation distance between the two treatment conditions to create an empirical distribution

```
repetitions = 500
tvds = Table().with_column("TVD between married and partnered men", [])
for i in np.arange(repetitions):
    # Construct a permuted table
    shuffled = males.select('Employment Status').sample()
    combined = Table().with_columns([
          "Marital Status", males.column('Marital Status'),
          "Employment Status", shuffled.column('Employment Status')])
    employment_shuffled = combined.pivot('Marital Status','Employment Status')
    # Compute TVD
    e_s = employment_shuffled
    married = e_s.column('married')/sum(e_s.column('married'))
    partner = e_s.column('partner')/sum(e_s.column('partner'))
    permutation_tvd = 0.5*sum(abs(married - partner))
    tvds.append([permutation_tvd])
```



Exam Scratch Paper

Exam Scratch Paper

Exam Scratch Paper

Exam Scratch Paper