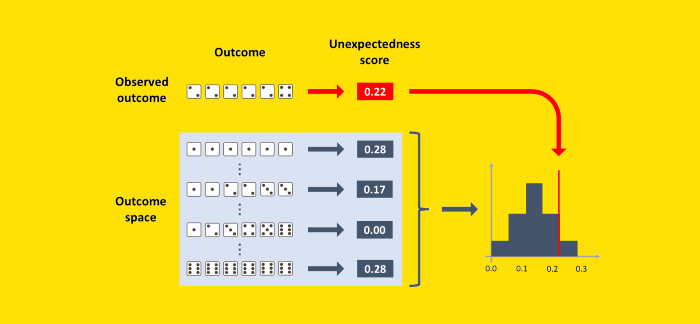
**Data Scientists Need to Know Just One Statistical Test**

After you read this, you will be able to test any possible statistical hypothesis. With a unique algorithm.



[Image by Author]

Asof today, Wikipedia counts a total of [104 statistical tests](https://en.wikipedia.org/wiki/Category:Statistical_tests). As a consequence, data scientists may feel overwhelmed and ask themselves:

“Should I know all of them? And how will I know when to use one over the other?”

I am here to reassure you: as a data professional, there is only one test that you need to know. Not because 1 test is important and the other 103 are negligible. But because:

All the statistical tests are in reality the same one test!

And once you really grasp how this one test work, you will be able to test any hypothesis you will ever need.

Want proof? In this article, we will solve 4 very diverse statistical problems. And we will solve them always using the same exact algorithm.

1. You have thrown a die 10 times. You got [1, 1, 1, 1, 1, 2, 2, 2, 3, 3]. Is the die loaded?
2. Your friend claims that some Scrabble tiles fell out of the bag and, coincidentally, the letters formed a real word: “F-E-A-R”. You suspect that your friend is just trying make fun of you. Is your friend lying?
3. In a customer satisfaction survey, 100 customers gave an average rating of 3.00 to product A and 2.63 to product B. Is this difference significant?
4. You trained a binary classification model. It has an area under the ROC curve of 70% on your test set (made of 100 observations). Is the model significantly better than random?

Before delving into the answers to these questions, let’s try to get to the essence of what statistical testing is.

**What is the profound meaning of any statistical test?**

I will try to answer this question with the least original example in statistics: the throw of a die.

Imagine you have thrown a die six times, and you got [2, 2, 2, 2, 2, 4]. A bit suspect, isn’t it? You don’t expect to get the same number 5 out of 6 times. At least, you don’t expect it to happen *if the die is fair*.

That’s exactly the point of statistical testing.

You have a hypothesis — called “null hypothesis” — and you want to put it to the test. Thus, you ask yourself:

“If the hypothesis was true, **how often would I get an outcome as suspect as the outcome that I actually had?”**

In the example of the die, the question becomes: “If the die was fair, how often would I get a sequence as unexpected as [2, 2, 2, 2, 2, 4]?” Since you are asking “how often”, the answer must necessarily be a number between 0 and 1, where 0 means never and 1 means always.

**In statistics, this “how often” is called “p-value”.**

At this point, the line of reasoning is pretty trivial: **if the p-value is very low, then it means that your original hypothesis is likely to be wrong.**

Note that the concept of “unexpectedness” depends closely on the specific hypothesis that you are testing. For instance, the outcome [2, 2, 2, 2, 2, 4] is pretty weird if you think that the die is fair. However, it’s little surprising if you think that the die is loaded to get the number “2” 75% of the time.

**The ingredients of statistical testing**

Reading the previous paragraph, you may have guessed that we need two ingredients:

1. The distribution of the possible outcomes, depending on the null hypothesis.
2. A measure of the “unexpectedness” of any outcome.

Regarding the 1st ingredient, it’s not always straightforward to get the full distribution of outcomes. Often, it’s more convenient (and easier) to **randomly simulate a high number of outcomes**: this is a good approximation of the true distribution.

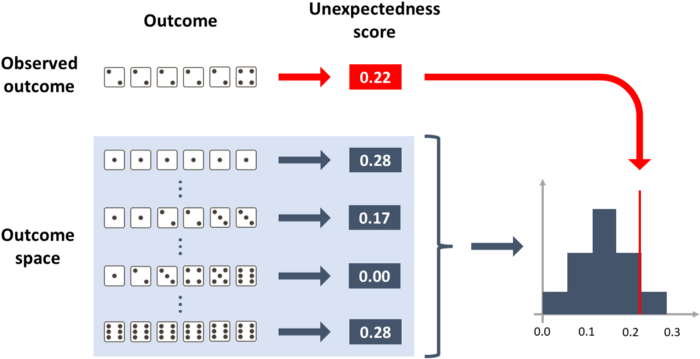
About the 2nd ingredient, we need to define **a function that maps each possible outcome into a single number**. This number must express how unexpected the outcome is, provided that the null hypothesis is true: the more unexpected the outcome, the higher this score.

Once we have these two ingredients, the job is basically done. In fact, it’s enough to calculate the unexpectedness score of each outcome in the distribution and the unexpectedness score of the observed outcome.

**The p-value is the percentage of random scores that are higher than the observed score.**

That’s it. This is how every single statistical test works under the hood.

And here is a graphical representation of the process we just described:



How to calculate the p-value of outcome [2, 2, 2, 2, 2, 4] under the null hypothesis of a fair die. [Image by Author]

**A unique statistical test**

But how do we do it in Python? The algorithm is the following:

1. Define a function draw\_random\_outcome. This function should return the outcome of a random trial, given that the null hypothesis is true. It may be a single number, an array, a list of arrays, an image, practically anything: it depends on the specific case.
2. Define a function unexp\_score (which stands for “unexpectedness score”). The function should take an experiment outcome as input, and return a single number. This number must be a score of how unexpected the outcome is, assuming it was generated under the null hypothesis. The score may be positive, negative, integer, or float, it doesn’t matter. The only property it must have is the following: **the unlikelier the outcome is, the higher this score must be**.
3. Run many times (e.g. 10,000 times) the function draw\_random\_outcome (defined at point 1) and, for each random outcome, compute its unexp\_score (defined at point 2). Store all the scores in an array called random\_unexp\_scores.
4. Compute unexp\_score of the observed outcome, and call it observed\_unexp\_score.
5. Compute how many random outcomes are more unexpected than the observed outcome. That is to say, count how many elements of random\_unexp\_scores are higher than observed\_unexp\_score. This is the p-value.

**The first two steps are the only ones that require a bit of creativity**, depending on the specific case, while steps 3, 4, and 5 are purely mechanical.

Now, to make it more concrete, let’s go through the examples.

**Example 1. Rolling A Die**

We have launched a die 10 times and obtained this outcome:

observed\_outcome = np.array([1,1,1,1,1,2,2,2,3,3])

The null hypothesis is that the die is fair. Under this hypothesis, it’s easy to extract random outcomes: it’s enough to use Numpy’s random.choice. So, this is the first step of our algorithm:

**# step 1**def draw\_random\_outcome(): return np.random.choice([1,2,3,4,5,6], size=10)

The second step is to define a function called unexp\_score that should assign a score of unexpectedness to each possible outcome.

If the die is fair, we expect each face to appear on average one sixth of the time. So we should check the distance between the observed frequency of each face and 1/6. Then, to obtain a single score, we should take the average. In this way, the higher the average distance from one sixth, the more unexpected is the outcome.

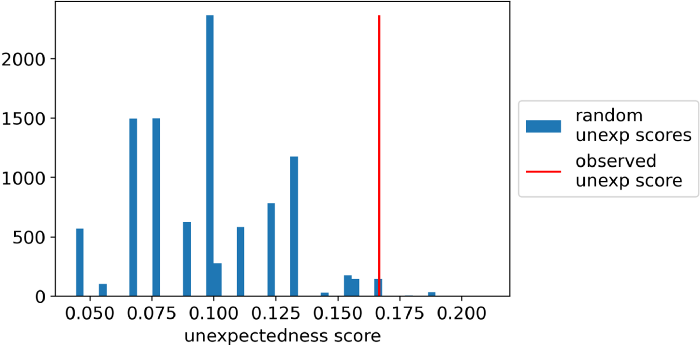
**# step 2**def unexp\_score(outcome): outcome\_distribution = np.array([np.mean(outcome == face) for face in [1,2,3,4,5,6]]) return np.mean(np.abs(outcome\_distribution - 1/6))

At this point, the hard part is done: as I said before, steps 3, 4, and 5 of the algorithm are totally mechanical.

**# step 3**n\_iter = 10000random\_unexp\_scores = np.empty(n\_iter)for i in range(n\_iter):  
 random\_unexp\_scores[i] = unexp\_score(draw\_random\_outcome())**# step 4**observed\_unexp\_score = unexp\_score(observed\_outcome)**# step 5**pvalue = np.sum(random\_unexp\_scores >= observed\_unexp\_score) / n\_iter

The resulting p-value is 1.66%, which means that under the null hypothesis only 1.66% of the outcomes are as unexpected as [1,1,1,1,1,2,2,2,3,3].

Out of curiosity, this is a histogram that shows the distribution of unexpectedness scores and where the observed score falls exactly.



Unexpectedness scores of 10 throws of a die. Null hypothesis: the die is fair. Observed outcome: [1,1,1,1,1,2,2,2,3,3]. [Image by Author]

**Example 2. Scrabble mystery**

Your friend claims that some Scrabble tiles fell out of the bag and, coincidentally, the letters formed a real word: “F-E-A-R”. You suspect that your friend is teasing you. How to check statistically if your friend is lying?

First of all, the observed outcome is a sequence of letters, therefore a string:

observed\_outcome = 'FEAR'

Suppose the bag contained the 26 letters of the alphabet. The null hypothesis is that a random number of letters (between 1 and 26) fell out of the bag in random order. So, we will have to use Numpy’s random for both the number of letters and the choice of the letters:

**# step 1**def draw\_random\_outcome(): size=np.random.randint(low=1, high=27) return ''.join(np.random.choice(list(string.ascii\_uppercase), size=size, replace=False))

Now, how to evaluate unexpectedness in this scenario?

In general, it’s reasonable to expect that the more letters fall from the bag, the less likely is to obtain a real word.

Therefore, we can use this rule: if the string is an existing word, then its score will be the length of the word. If the string is not a real word, then its score will be minus the length of the word.

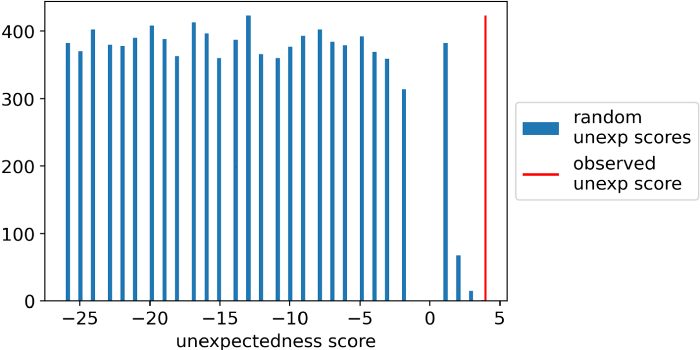
So this is step 2 of the algorithm (note: you will have to pip install english-words to make the code below work).

**# step 2**from english\_words import english\_words\_setenglish\_words\_set = [w.upper() for w in english\_words\_set]def unexp\_score(outcome): is\_in\_dictionary = outcome in english\_words\_set return (1 if is\_in\_dictionary else -1) \* len(outcome)

Steps 3, 4, 5 are always the same, so we will just copy and paste them from the previous example:

**# step 3**n\_iter = 10000random\_unexp\_scores = np.empty(n\_iter)for i in range(n\_iter):  
 random\_unexp\_scores[i] = unexp\_score(draw\_random\_outcome())**# step 4**observed\_unexp\_score = unexp\_score(observed\_outcome)**# step 5**pvalue = np.sum(random\_unexp\_scores >= observed\_unexp\_score) / n\_iterThis is the result:

This is the result:



Unexpectedness scores of random strings from the alphabet. Null hypothesis: a random number of letters are selected, and the order of letters is random as well. Observed outcome: “FEAR”. [Image by Author]

In this case, the p-value is 0.0, since no random score was higher than the observed score. So, according to this statistical test, your friend was lying!

**Example 3. Difference between two means**

You asked 100 of your customers to rate two of your products: product A and product B. You obtained the following ratings:

product\_a = np.repeat([1,2,3,4,5], 20)product\_b = np.array([1]\*27+[2]\*25+[3]\*19+[4]\*16+[5]\*13)observed\_outcome = np.mean(product\_a) - np.mean(product\_b)

The outcome is calculated as the difference between the mean rating of product A (3.0) and the mean rating of product B (2.63), in this case 0.37.

Since you want to test whether the difference between the average ratings is significant, the null hypothesis is that there is no difference between product A and product B. If that is true, we could shuffle the ratings between the two products.

So, the function unexp\_score will just take all the 200 ratings, shuffle them, and assign 100 at random to A and the remaining 100 to B. Then, it will compute the difference between the two means.

**# step 1**def draw\_random\_outcome(): pr\_a, pr\_b = np.random.permutation(np.hstack([product\_a, product\_b])).reshape(2,-1) return np.mean(pr\_a) - np.mean(pr\_b)

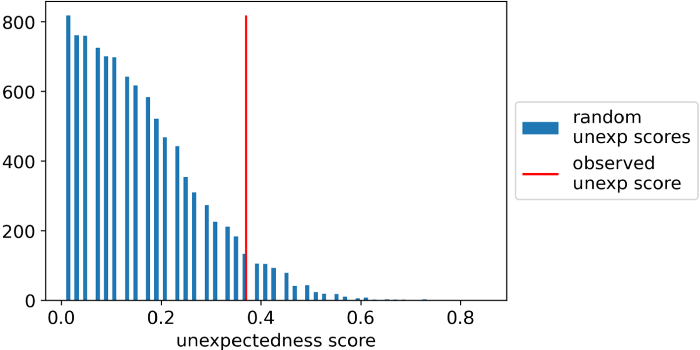
Following the hypothesis of no difference between the two products, the observed difference between the means should be small. Therefore, values of outcomes that are further from zero are more unexpected. So we can take the absolute value of the outcome as the unexpectedness score:

**# step 2**def unexp\_score(outcome): return np.abs(outcome)

Steps 3, 4, 5 are always the same, so we will just copy and paste them from the previous example:

**# step 3**n\_iter = 10000random\_unexp\_scores = np.empty(n\_iter)for i in range(n\_iter):  
 random\_unexp\_scores[i] = unexp\_score(draw\_random\_outcome())**# step 4**observed\_unexp\_score = unexp\_score(observed\_outcome)**# step 5**pvalue = np.sum(random\_unexp\_scores >= observed\_unexp\_score)/ n\_iter

This is the result:



Unexpectedness scores of difference between means. Null hypothesis: there is no difference between the means. Observed outcome: 0.37. [Image by Author]

The p-value in this case is 6.82%. So, in traditional statistical testing, if you set the threshold at 1% or at 5%, you should conclude that there is no significant difference between the average ratings.

**Example 4. Area under the ROC curve**

Imagine you have trained a predictive model, and it has 70% area under ROC on the test dataset. Good news, right? But the test set is made of only 100 observations. So how can you prove whether this result is significant?

Imagine these are your initial elements:

y\_test = np.random.choice([0,1], size=100, p=[.9,.1])proba\_test = np.random.uniform(low=0, high=1, size=100)observed\_outcome = .7

The null hypothesis is that your model is no different from random, so if you just shuffle your predictions and compute the roc\_auc\_score, you have an outcome generated under the null hypothesis:

**# step 1**def draw\_random\_outcome(): return roc\_auc\_score(y\_test, np.random.permutation(proba\_test))

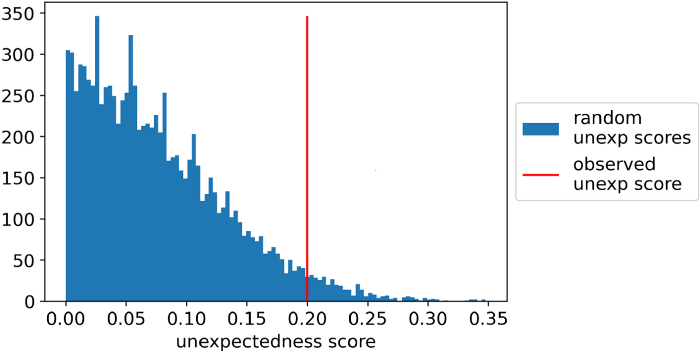
How unexpected is an outcome? Since we are dealing with the area under the ROC curve, where 50% identifies a random classifier, then an outcome is more unexpected as it gets further away from 50%, so our function unexp\_score could be the absolute difference between the roc\_auc\_score and 50%:

**# step 2**def unexp\_score(outcome):  
 return np.abs(outcome - .5)

Steps 3, 4, 5 are always the same, so we will just copy and paste them from the previous example:

**# step 3**n\_iter = 10000random\_unexp\_scores = np.empty(n\_iter)for i in range(n\_iter):  
 random\_unexp\_scores[i] = unexp\_score(draw\_random\_outcome())**# step 4**observed\_unexp\_score = unexp\_score(observed\_outcome)**# step 5**pvalue = np.sum(random\_unexp\_scores >= observed\_unexp\_score) / n\_iter

This is the result:



Unexpectedness scores of area under ROC curve. Null hypothesis: the predictive model is not different from random. Observed outcome: 70%. [Image by Author]

The p-value is 2.25%, so the observed outcome was not so “surprising” as we may have thought initially. If you set the threshold at 1%, following the classical rule of statistical testing, you couldn’t reject the hypothesis that your model is not better than random.

**So why all those statistical tests?**

If you have come this far, you may wonder: if it’s so easy, why do so many tests exist? The answer is mostly “historical”.

There was a time when computation was much more expensive than now, so “statistical tests” were basically shortcuts to compute p-values efficiently. And since there are so many possibilities to choose step 1 and step 2 of the algorithm we have seen, the tests proliferated.

If you want to deepen this topic, you can read the classic [“There is only one test”](http://allendowney.blogspot.com/2011/05/there-is-only-one-test.html) by Allen Downey, which inspired this article.