Why do we do statistical inference?

- To draw probabilistic conclusions about what we might expect if we collected the same data again
- To draw actionable conclusions from data
- To draw more general conclusions from relatively few data or observations
- All of these

- Correct! Statistical inference involves taking your data to probabilistic conclusions about what you would expect if you took even more data, and you can make decisions based on these conclusions.

Which of the following is *not* a reason why we use probabilistic language in statistical inference?

- Probability provides a measure of uncertainty
- Probabilistic language is not very precise
- Data are almost never exactly the same when acquired again, and probability allows us to say how much we expect them to vary.

- Correct. Probabilistic language is in fact very precise. It precisely describes uncertainty.

Generating random numbers using the np.random module

- We will be hammering the np.random module for the rest of this course and its sequel. Actually, you will probably call functions from this module more than any other while wearing your hacker statistician hat. Let's start by taking its simplest function, np.random.random() for a test spin. The function returns a random number between zero and one. Call np.random.random() a few times in the IPython shell. You should see numbers jumping around between zero and one.

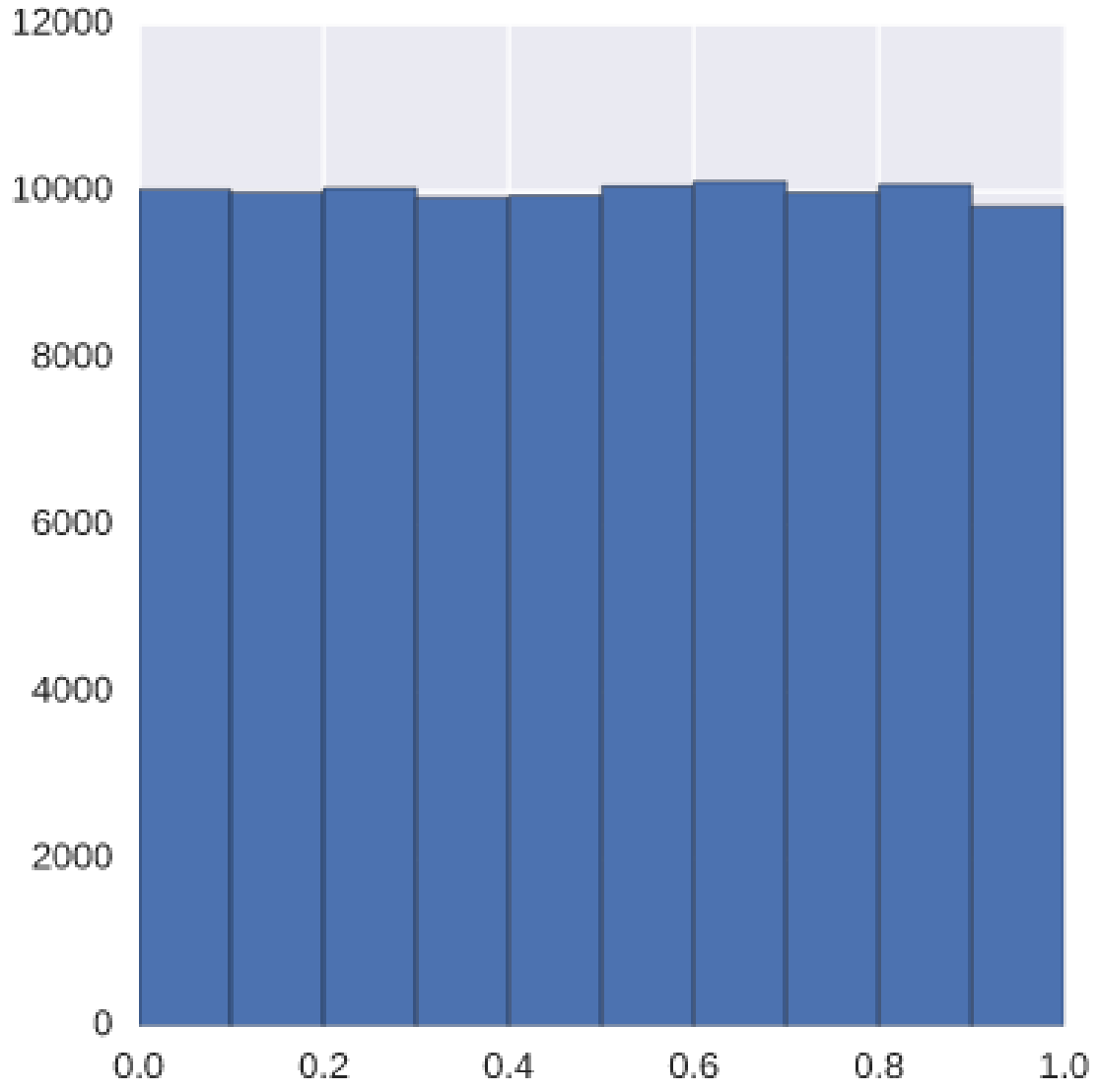
- In this exercise, we'll generate lots of random numbers between zero and one, and then plot a histogram of the results. If the numbers are truly random, all bars in the histogram should be of (close to) equal height.

- You may have noticed that, in the video, Justin generated 4 random numbers by passing the keyword argument size=4 to np.random.random(). Such an approach is more efficient than a for loop: in this exercise, however, you will write a for loop to experience hacker statistics as the practice of repeating an experiment over and over again.

Instructions

- Seed the random number generator using the seed 42.
- Initialize an empty array, random_numbers, of 100,000 entries to store the random numbers. Make sure you use np.empty(100000) to do this.
- Write a for loop to draw 100,000 random numbers using np.random.random(), storing them in the random_numbers array. To do so, loop over range(100000).
- Plot a histogram of random_numbers. It is not necessary to label the axes in this case because we are just checking the random number generator. Hit 'Submit Answer' to show your plot.

```
# Seed the random number generator
np.random.seed(42)

# Initialize random numbers: random_numbers
random_numbers = np.empty(100000)

# Generate random numbers by looping over range(100000)
for i in range(100000):
    random_numbers[i] = np.random.random()

# Plot a histogram
_ = plt.hist(random_numbers)

# Show the plot
plt.show()
```


12000
10000
8000
6000
4000
2000
0
0.0
0.2
0.4
0.6
0.8
1.0

The np.random module and Bernoulli trials

- You can think of a Bernoulli trial as a flip of a possibly biased coin. Specifically, each coin flip has a probability pp of landing heads (success) and probability 1−p1−p of landing tails (failure). In this exercise, you will write a function to perform n Bernoulli trials, perform_bernoulli_trials(n, p), which returns the number of successes out of n Bernoulli trials, each of which has probability p of success. To perform each Bernoulli trial this, use the np.random.random() function, which returns a random number between zero and one.

Instructions

- Define a function with signature perform_bernoulli_trials(n, p).
- Initialize to zero a variable n_success the counter of Trues, which are Bernoulli trial successes.
- Write a for loop where you perform a Bernoulli trial in each iteration and increment the number of success if the result is True. Perform n iterations by looping over range(n).
- To perform a Bernoulli trial, choose a random number between zero and one using np.random.random(). If the number you chose is less than p, increment n_success (use the += 1 operator to achieve this).
- The function returns the number of successes n_success.

```
def perform_bernoulli_trials(n, p):
    """Perform n Bernoulli trials with success probability p
    and return number of successes."""
    # Initialize number of successes: n_success
    n_success = 0

    # Perform trials
    for i in range(n):
        # Choose random number between zero and one: random_number
        random_number = np.random.random()

        # If less than p, it's a success  so add one to n_success
        if random_number < p:
            n_success += 1

    return n_success
```

```
# Seed random number generator

np.random.seed(42)
# Initialize the number of defaults: n_defaults

n_defaults=np.empty(1000)
# Compute the number of defaults
for i in range(1000):
    n_defaults[i] = perform_bernoulli_trials(100,0.05)



# Plot the histogram with default number of bins; label your
axes
_ = plt.hist(n_defaults, bins=1000, normed=True)
_ = plt.xlabel('number of defaults out of 100 loans')
_ = plt.ylabel('probability')
```

Will the bank fail?

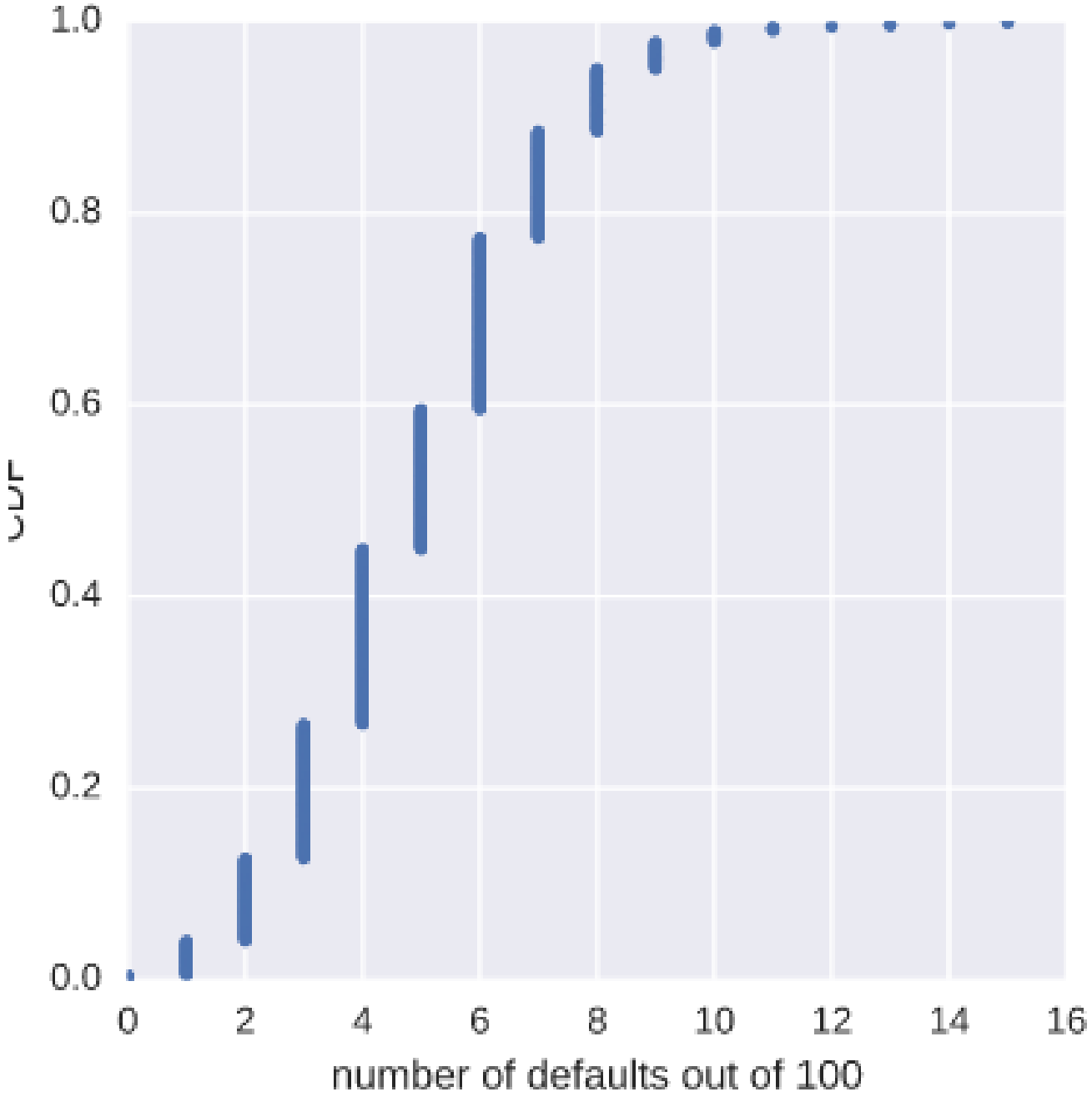
- Plot the number of defaults you got from the previous exercise, in your namespace as n_defaults, as a CDF. The ecdf() function you wrote in the first chapter is available.

- If interest rates are such that the bank will lose money if 10 or more of its loans are defaulted upon, what is the probability that the bank will lose money?

Instructions

- Compute the x and y values for the ECDF of n_defaults.
- Plot the ECDF, making sure to label the axes. Remember to include marker = '.' and linestyle = 'none' in addition to x and y in your call plt.plot().
- Show the plot.
- Compute the total number of entries in your n_defaults array that were greater than or equal to 10. To do so, compute a boolean array that tells you whether a given entry of n_defaults is >= 10. Then sum all the entries in this array using np.sum(). For example, np.sum(n_defaults <= 5) would compute the number of defaults with 5 or fewer defaults.
- The probability that the bank loses money is the fraction of n_defaults that are greater than or equal to 10. Print this result by hitting 'Submit Answer'!

```
# Compute ECDF: x, y
x, y = ecdf(n_defaults)

# Plot the CDF with labeled axes
_ = plt.plot(x, y, marker='.', linestyle='none')
_ = plt.xlabel('number of defaults out of 100')
_ = plt.ylabel('CDF')

# Show the plot
plt.show()

# Compute the number of 100-loan simulations with 10 or more
defaults: n_lose_money
n_lose_money = np.sum(n_defaults >= 10)

# Compute and print probability of losing money
print('Probability of losing money =', n_lose_money / len
(n_defaults))
```


1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
12
14
16
number of defaults out of 100

Sampling out of the Binomial distribution

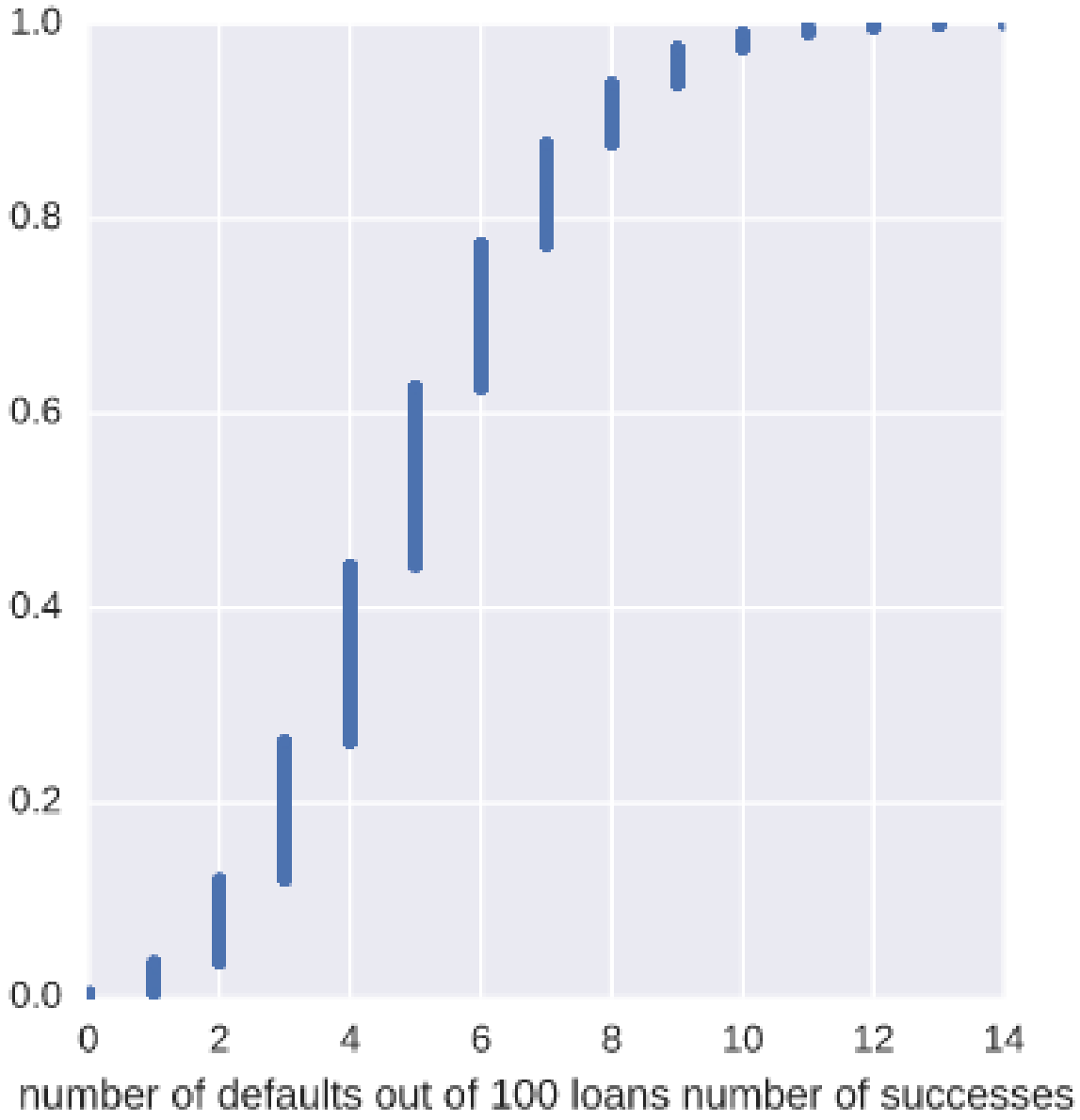
- Compute the probability mass function for the number of defaults we would expect for 100 loans as in the last section, but instead of simulating all of the Bernoulli trials, perform the sampling using np.random.binomial(). This is identical to the calculation you did in the last set of exercises using your custom-written perform_bernoulli_trials() function, but far more computationally efficient. Given this extra efficiency, we will take 10,000 samples instead of 1000. After taking the samples, plot the CDF as last time. This CDF that you are plotting is that of the Binomial distribution.

- Note: For this exercise and all going forward, the random number generator is pre-seeded for you (with np.random.seed(42)) to save you typing that each time.

Instructions

- Draw samples out of the Binomial distribution using np.random.binomial(). You should use parameters n = 100 and p = 0.05, and set the size keyword argument to 10000.
- Compute the CDF using your previously-written ecdf() function.
- Plot the CDF with axis labels. The x-axis here is the number of defaults out of 100 loans, while the y-axis is the CDF.
- Show the plot.

```
# Take 10,000 samples out of the binomial distribution:
n_defaults
n_defaults=np.random.binomial(100,0.05,size = 10000)

# Compute CDF: x, y
x, y = ecdf(n_defaults)

# Plot the CDF with axis labels
_ = plt.plot(x, y, marker = '.', linestyle='none')

_ = plt.xlabel('number of defaults out of 100 loans number
of successes')
_ = plt.ylabel('CDF')


# Show the plot

plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
12
14
number of defaults out of 100 loans number of successes

Plotting the Binomial PMF

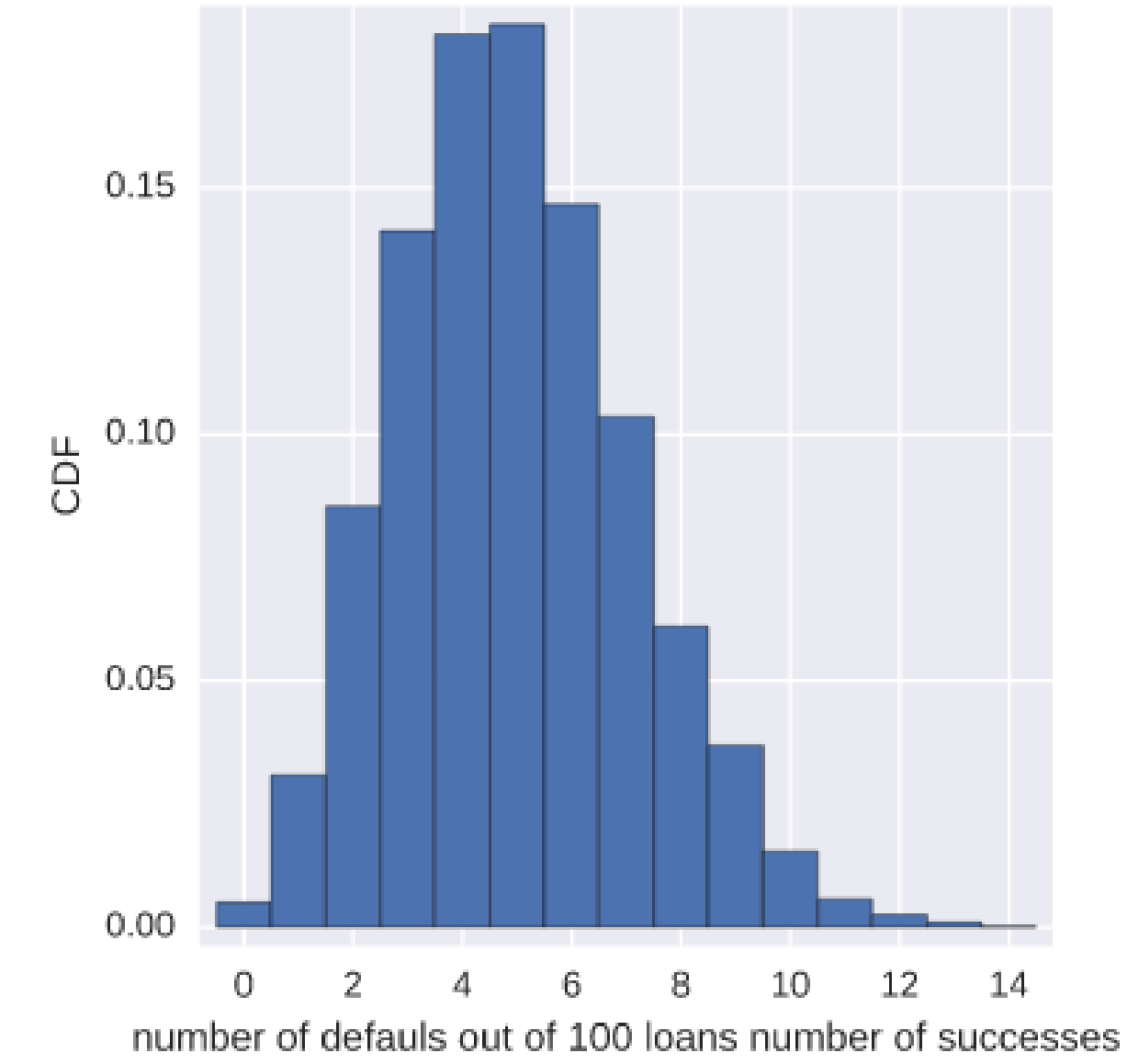
- As mentioned in the video, plotting a nice looking PMF requires a bit of matplotlib trickery that we will not go into here. Instead, we will plot the PMF of the Binomial distribution as a histogram with skills you have already learned. The trick is setting up the edges of the bins to pass to plt.hist() via the bins keyword argument. We want the bins centered on the integers. So, the edges of the bins should be -0.5, 0.5, 1.5, 2.5, ... up to max(n_defaults) + 1.5. You can generate an array like this using np.arange() and then subtracting 0.5 from the array.

- You have already sampled out of the Binomial distribution during your exercises on loan defaults, and the resulting samples are in the NumPy array n_defaults.

Instructions

- Using np.arange(), compute the bin edges such that the bins are centered on the integers. Store the resulting array in the variable bins.
- Use plt.hist() to plot the histogram of n_defaults with the normed=True and bins=bins keyword arguments.
- Leave a 2% margin and label your axes.
- Show the plot.

```
# Compute bin edges: bins
bins = np.arange(0, max(n_defaults)+ 2) - 0.5

# Generate histogram
_ = plt.hist(n_defaults, bins=bins, normed= True)

# Set margins
plt.margins(0.02)

# Label axes
_ = plt.xlabel('number of defauls out of 100 loans number of
successes')
_ = plt.ylabel('CDF')



# Show the plot
plt.show()
```


0.15
0.10
0.05
0.00
CDF
0
2
4
6
8
10
12
14
number of defauls out of 100 loans number of successes

Relationship between Binomial and Poisson distributions

- You just heard that the Poisson distribution is a limit of the Binomial distribution for rare events. This makes sense if you think about the stories. Say we do a Bernoulli trial every minute for an hour, each with a success probability of 0.1. We would do 60 trials, and the number of successes is Binomially distributed, and we would expect to get about 6 successes. This is just like the Poisson story we discussed in the video, where we get on average 6 hits on a website per hour. So, the Poisson distribution with arrival rate equal to npnp approximates a Binomial distribution for nn Bernoulli trials with probability pp of success (with nn large and pp small). Importantly, the Poisson distribution is often simpler to work with because it has only one parameter instead of two for the Binomial distribution.

- Let's explore these two distributions computationally. You will compute the mean and standard deviation of samples from a Poisson distribution with an arrival rate of 10. Then, you will compute the mean and standard deviation of samples from a Binomial distribution with parameters nn and pp such that np=10np=10.

Instructions

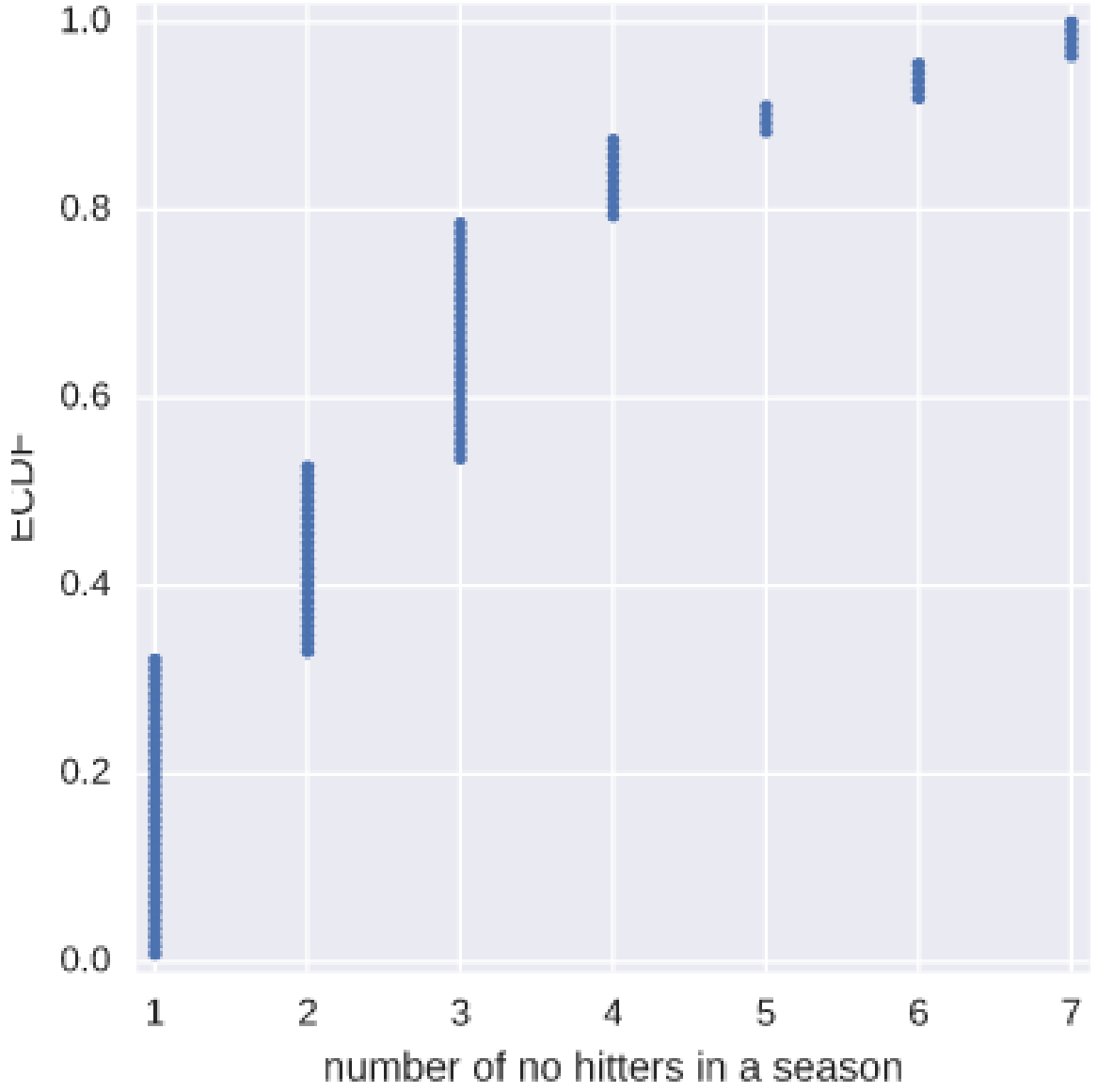
- Using the np.random.poisson() function, draw 10000 samples from a Poisson distribution with a mean of 10.
- Make a list of the n and p values to consider for the Binomial distribution. Choose n = [20, 100, 1000] and p = [0.5, 0.1, 0.01] so that npnp is always 10.
- Using np.random.binomial() inside the provided for loop, draw 10000 samples from a Binomial distribution with each n, p pair and print the mean and standard deviation of the samples. There are 3 n, p pairs: 20, 0.5, 100, 0.1, and 1000, 0.01. These can be accessed inside the loop as n[i], p[i].

```
# Draw 10,000 samples out of Poisson distribution: samples_poisson
samples_poisson = np.random.poisson(10, size=10000)

# Print the mean and standard deviation
print('Poisson:     ', np.mean(samples_poisson),
                       np.std(samples_poisson))

# Specify values of n and p to consider for Binomial: n, p

n=[20, 100, 1000]
p=[0.5, 0.1, 0.01]

# Draw 10,000 samples for each n,p pair: samples_binomial
for i in range(3):
    samples_binomial =np.random.binomial(n[i], p[i], size= 10000)

    # Print results
    print('n =', n[i], 'Binom:', np.mean(samples_binomial),
                                 np.std(samples_binomial))
```

How many no-hitters in a season?

- In baseball, a no-hitter is a game in which a pitcher does not allow the other team to get a hit. This is a rare event, and since the beginning of the so-called modern era of baseball (starting in 1901), there have only been 251 of them through the 2015 season in over 200,000 games. The ECDF of the number of no-hitters in a season is shown to the right. Which probability distribution would be appropriate to describe the number of no-hitters we would expect in a given season?

- Note: The no-hitter data set was scraped and calculated from the data sets available at retrosheet.org (license).


1.0
0.8
0.6
0.4
0.2
0.0
1
2
3
4
5
6
7
number of no hitters in a season

- Discrete uniform
- Binomial
- Poisson
- Both Binomial and Poisson, though Poisson is easier to model and compute.
- Both Binomial and Poisson, though Binomial is easier to model and compute.

- Correct! When we have rare events (low p, high n), the Binomial distribution is Poisson. This has a single parameter, the mean number of successes per time interval, in our case the mean number of no-hitters per season.

Was 2015 anomalous?

- 1990 and 2015 featured the most no-hitters of any season of baseball (there were seven). Given that there are on average 251/115 no-hitters per season, what is the probability of having seven or more in a season?

Instructions

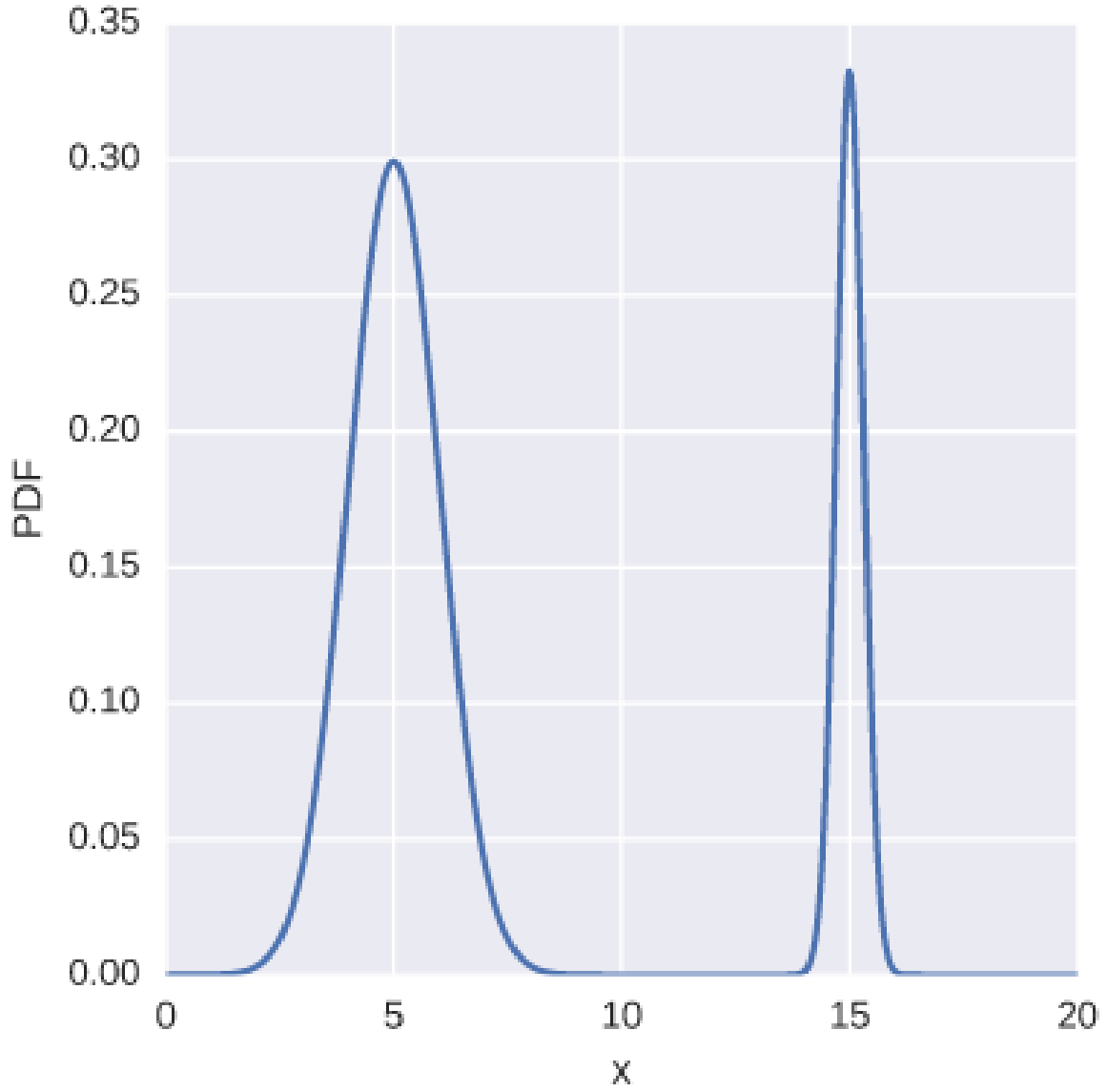
- Draw 10000 samples from a Poisson distribution with a mean of 251/115 and assign to n_nohitters.
- Determine how many of your samples had a result greater than or equal to 7 and assign to n_large.
- Compute the probability, p_large, of having 7 or more no-hitters by dividing n_large by the total number of samples (10000).
- Hit 'Submit Answer' to print the probability that you calculated.

```
# Draw 10,000 samples out of Poisson distribution: n_nohitters
n_nohitters = np.random.poisson(251/115, 10000)

# Compute number of samples that are seven or greater: n_large
n_large = np.sum(n_nohitters >= 7)

# Compute probability of getting seven or more: p_large
p_large=n_large/10000

# Print the result
print('Probability of seven or more no-hitters:', p_large)
```

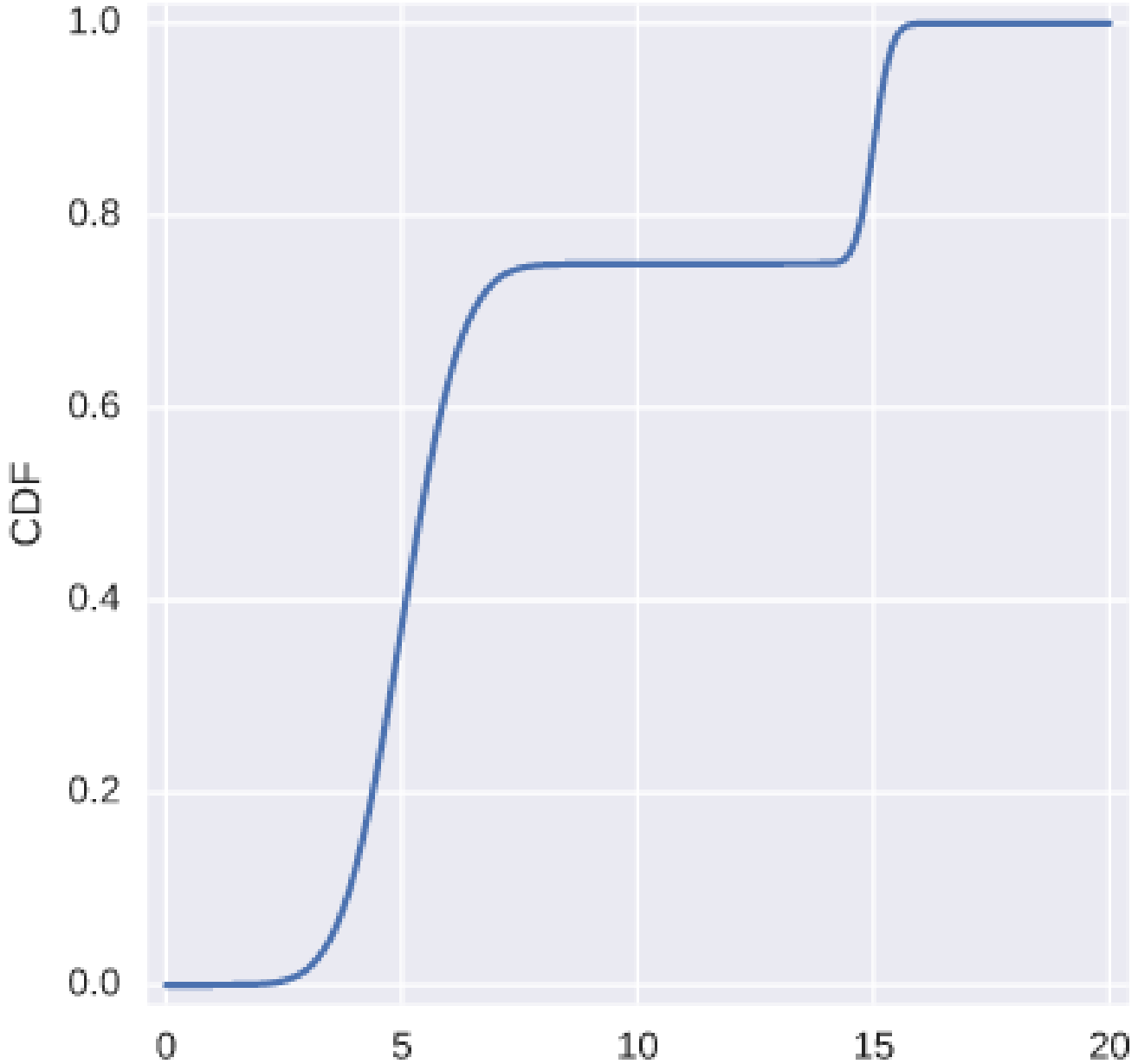
- The result is about 0.007. This means that it is not that improbable to see a 7-or-more no-hitter season in a century. We have seen two in a century and a half, so it is not unreasonable.

Consider the PDF shown to the right. Which of the following is true?

- x is more likely than not less than 10.
- x is more likely than not greater than 10.
- We cannot tell from the PDF if xx is more likely to be greater than or less than 10.
- This is not a valid PDF because it has two peaks.


0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
PDF
0
5
10
15
20
X

- Correct! The probability is given by the *area under the PDF*, and there is more area to the left of 10 than to the right.


1.0
0.8
0.6
0.4
0.2
0.0
CDF
0
5
10
15
20

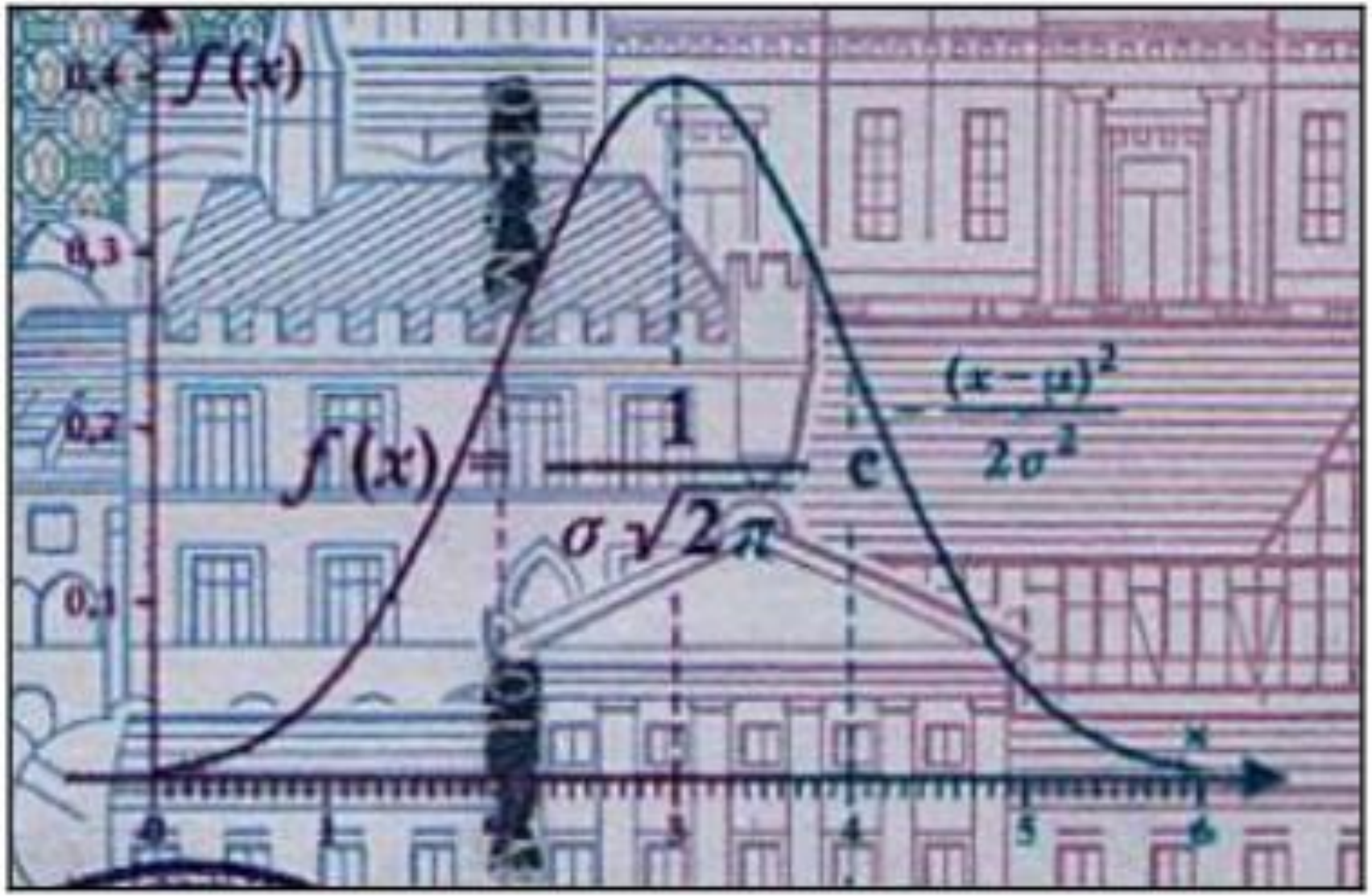
At right is the CDF corresponding to the PDF you considered in the last exercise. Using the CDF, what is the probability that xx is greater than 10?

- 0.25
- 0.75
- 3.75
- 15

- Correct! The value of the CDF at $x = 10$ is 0.75, so the probability that $x < 10$ is 0.75. Thus, the probability that $x > 10$ is 0.25.

Gauss and the 10 Deutschmark banknote

- What are the mean and standard deviation, respectively, of the Normal distribution that was on the 10 Deutschmark banknote, shown to the right?
- mean = 3, std = 1
- mean = 3, std = 2
- mean = 0.4, std = 1
- mean = 0.6, std = 6


0,4
f(x)
0,3
0,2
0,1
f(x) = 1 / (σ√2π) e^(-(x-μ)²/2σ²)
x
0
1
2
3
4
5
6

Are the Belmont Stakes results Normally distributed?

- Since 1926, the Belmont Stakes is a 1.5 mile-long race of 3-year old thoroughbred horses. Secretariat ran the fastest Belmont Stakes in history in 1973. While that was the fastest year, 1970 was the slowest because of unusually wet and sloppy conditions. With these two outliers removed from the data set, compute the mean and standard deviation of the Belmont winners' times. Sample out of a Normal distribution with this mean and standard deviation using the np.random.normal() function and plot a CDF. Overlay the ECDF from the winning Belmont times. Are these close to Normally distributed?

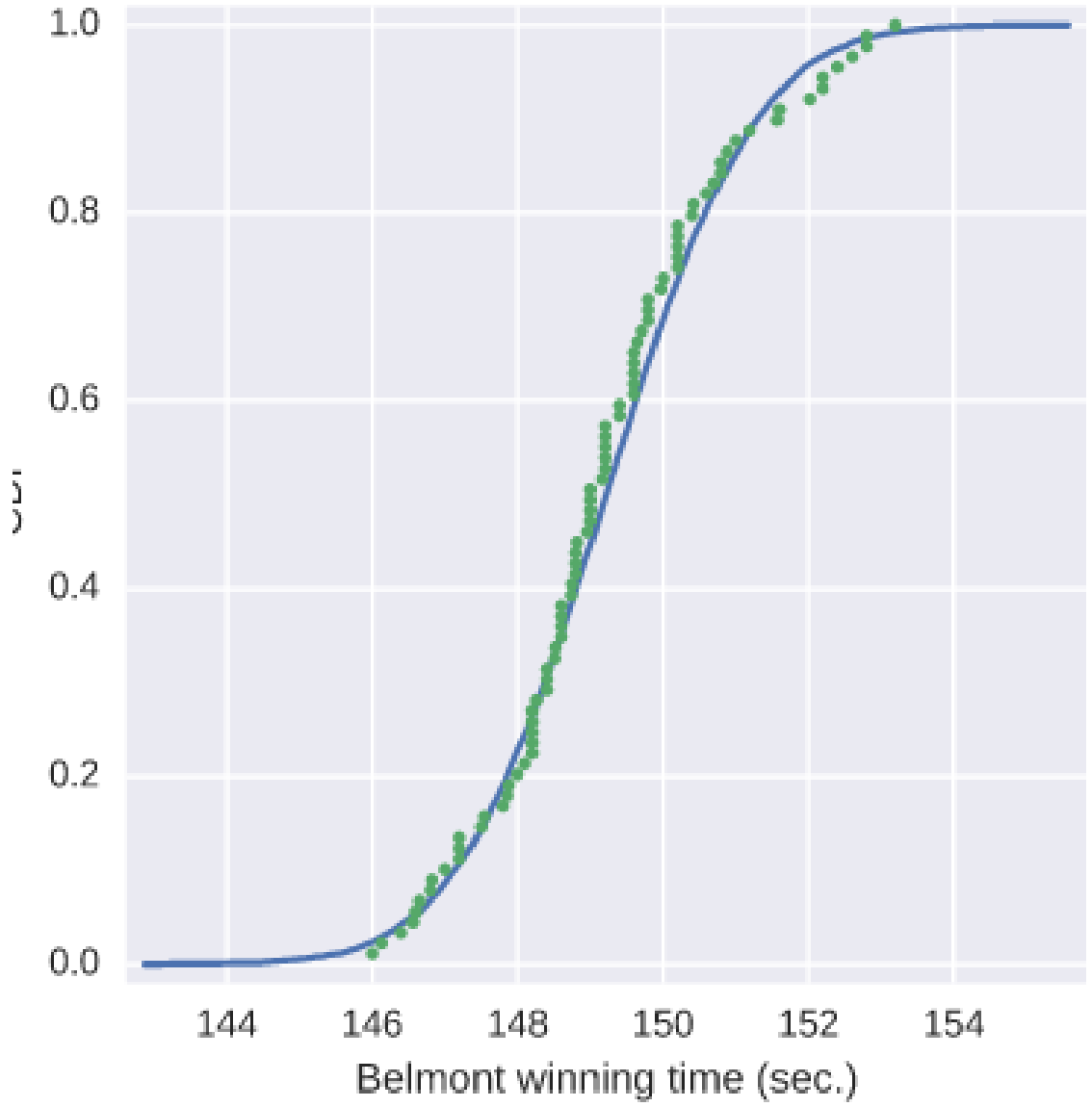
- Note: Justin scraped the data concerning the Belmont Stakes from the Belmont Wikipedia page.

Instructions

- Compute mean and standard deviation of Belmont winners' times with the two outliers removed. The NumPy array belmont_no_outliers has these data.
- Take 10,000 samples out of a normal distribution with this mean and standard deviation using np.random.normal().
- Compute the CDF of the theoretical samples and the ECDF of the Belmont winners' data, assigning the results to x_theor, y_theor and x, y, respectively.
- Hit submit to plot the CDF of your samples with the ECDF, label your axes and show the plot.

```
# Compute mean and standard deviation: mu, sigma
mu = np.mean(belmont_no_outliers)
sigma = np.std(belmont_no_outliers)
# Sample out of a normal distribution with this mu and sigma
: samples
samples = np.random.normal(mu, sigma, size = 10000)
# Get the CDF of the samples and of the data
x_theor, y_theor = ecdf(samples)
x, y = ecdf(belmont_no_outliers)

# Plot the CDFs and show the plot
_ = plt.plot(x_theor, y_theor)
_ = plt.plot(x, y, marker='.', linestyle='none')
plt.margins(0.02)
_ = plt.xlabel('Belmont winning time (sec.)')
_ = plt.ylabel('CDF')
plt.show()
```

- The theoretical CDF and the ECDF of the data suggest that the winning Belmont times are, indeed, Normally distributed. This also suggests that in the last 100 years or so, there have not been major technological or training advances that have significantly affected the speed at which horses can run this race.


1.0
0.8
0.6
0.4
0.2
0.0
144
146
148
150
152
154
Belmont winning time (sec.)

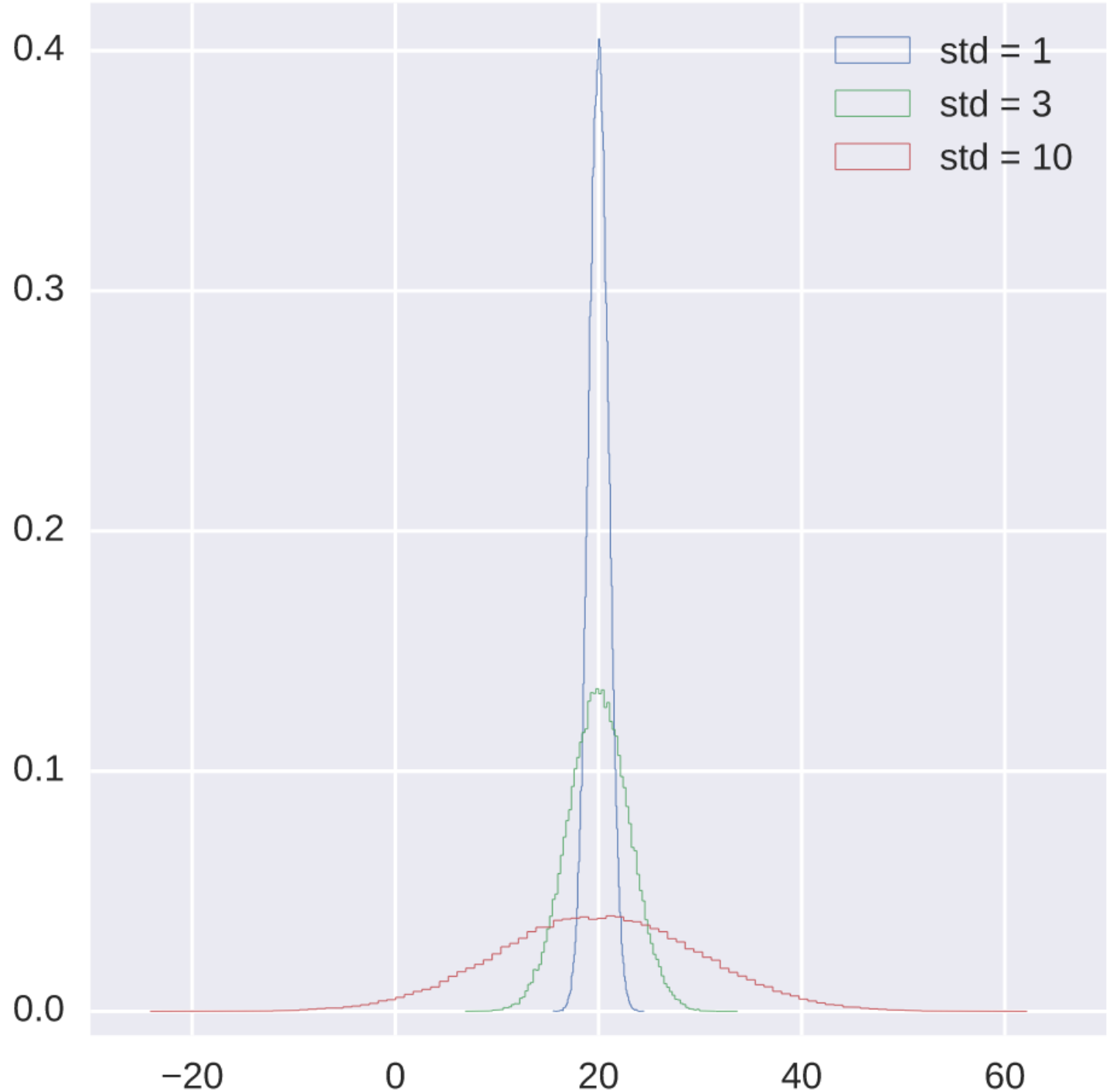
The Normal PDF

- In this exercise, you will explore the Normal PDF and also learn a way to plot a PDF of a known distribution using hacker statistics. Specifically, you will plot a Normal PDF for various values of the variance.

Instructions

- Draw 100,000 samples from a Normal distribution that has a mean of 20 and a standard deviation of 1. Do the same for Normal distributions with standard deviations of 3 and 10, each still with a mean of 20. Assign the results to samples_std1, samples_std3 and samples_std10, respectively.
- Plot a histograms of each of the samples; for each, use 100 bins, also using the keyword arguments normed=True and histtype='step'. The latter keyword argument makes the plot look much like the smooth theoretical PDF. You will need to make 3 plt.hist() calls.
- Hit 'Submit Answer' to make a legend, showing which standard deviations you used, and show your plot! There is no need to label the axes because we have not defined what is being described by the Normal distribution; we are just looking at shapes of PDFs.

```
# Draw 100000 samples from Normal distribution with stds of
interest: samples_std1, samples_std3, samples_std10
samples_std1= np.random.normal(20, 1, size = 100000)
samples_std3= np.random.normal(20, 3, size = 100000)
samples_std10= np.random.normal(20, 10, size = 100000)
# Make histograms
_= plt.hist(samples_std1, bins = 100, normed= True, histtype
='step' )
_= plt.hist(samples_std3, bins = 100, normed= True, histtype
='step' )
_= plt.hist(samples_std10, bins = 100, normed= True,
histtype='step' )

# Make a legend, set limits and show plot
_ = plt.legend(('std = 1', 'std = 3', 'std = 10'))
plt.ylim(-0.01, 0.42)
plt.show()
```

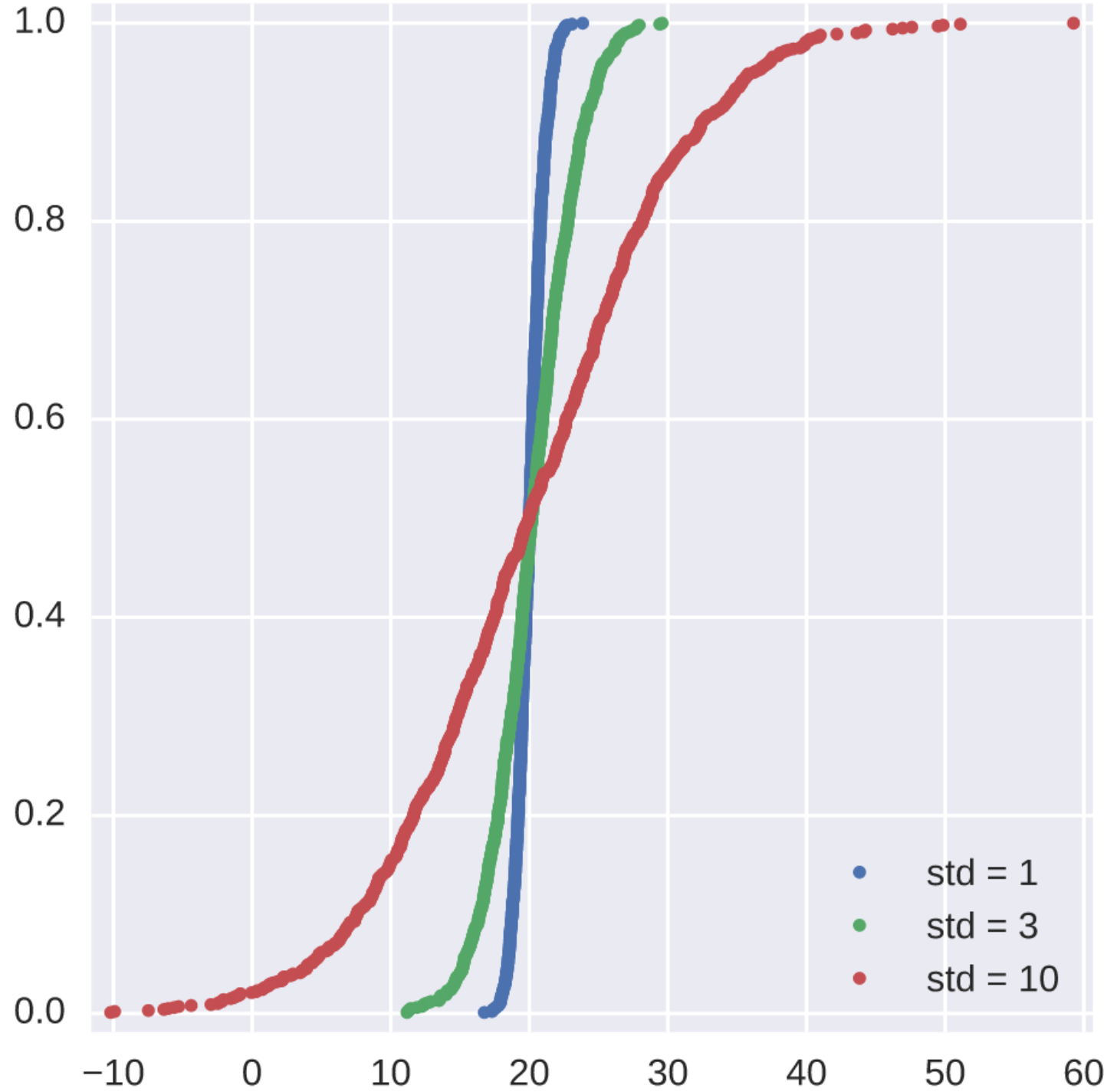

0.4
0.3
0.2
0.1
0.0
−20
0
20
40
60
std = 1
std = 3
std = 10

- Great work! You can see how the different standard deviations result in PDFS of different widths. The peaks are all centered at the mean of 20.

- Now that you have a feel for how the Normal PDF looks, let's consider its CDF. Using the samples you generated in the last exercise (in your namespace as samples_std1, samples_std3, and samples_std10), generate and plot the CDFs.

- Use your ecdf() function to generate x and y values for CDFs: x_std1, y_std1, x_std3, y_std3 and x_std10, y_std10, respectively.
- Plot all three CDFs as dots (do not forget the marker and linestyle keyword arguments!).
- Make a 2% margin in your plot.
- Hit submit to make a legend, showing which standard deviations you used, and to show your plot. There is no need to label the axes because we have not defined what is being described by the Normal distribution; we are just looking at shapes of CDFs.

```
# Generate CDFs
x_std1, y_std1 = ecdf(samples_std1)
x_std3, y_std3 = ecdf(samples_std3)
x_std10, y_std10 = ecdf(samples_std10)



# Plot CDFs
_= plt.plot(x_std1, y_std1, marker='.', linestyle='none')
_= plt.plot(x_std3, y_std3, marker='.', linestyle='none')
_= plt.plot(x_std10, y_std10, marker='.', linestyle='none')
# Make 2% margin
plt.margins(0.02)

# Make a legend and show the plot
_ = plt.legend(('std = 1', 'std = 3', 'std = 10'), loc
='lower right')
plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
−10
0
10
20
30
40
50
60
std = 1
std = 3
std = 10

- Great work! The CDFs all pass through the mean at the 50th percentile; the mean and median of a Normal distribution are equal. The width of the CDF varies with the standard deviation.

Matching a story and a distribution.

- How might we expect the time between Major League no-hitters to be distributed? Be careful here: a few exercises ago, we considered the probability distribution for the number of no-hitters in a season. Now, we are looking at the probability distribution of the time between no hitters.
- Normal
- Exponential
- Poisson
- Uniform

Waiting for the next Secretariat

- Unfortunately, Justin was not alive when Secretariat ran the Belmont in 1973. Do you think he will get to see a performance like that? To answer this, you are interested in how many years you would expect to wait until you see another performance like Secretariat's. How is the waiting time until the next performance as good or better than Secretariat's distributed? Choose the best answer.

- Normal, because the distribution of Belmont winning times are Normally distributed.
- Normal, because there is a most-expected waiting time, so there should be a single peak to the distribution.
- Exponential: It is very unlikely for a horse to be faster than Secretariat, so the distribution should decay away to zero for high waiting time.
- Exponential: A horse as fast as Secretariat is a rare event, which can be modeled as a Poisson process, and the waiting time between arrivals of a Poisson process is Exponentially distributed.

- Correct! The Exponential distribution describes the waiting times between rare events, and Secretariat is *rare*!

If you have a story, you can simulate it!

- Sometimes, the story describing our probability distribution does not have a named distribution to go along with it. In these cases, fear not! You can always simulate it. We'll do that in this and the next exercise.

- In earlier exercises, we looked at the rare event of no-hitters in Major League Baseball. Hitting the cycle is another rare baseball event. When a batter hits the cycle, he gets all four kinds of hits, a single, double, triple, and home run, in a single game. Like no-hitters, this can be modeled as a Poisson process, so the time between hits of the cycle are also Exponentially distributed.

- How long must we wait to see both a no-hitter and a batter hit the cycle? The idea is that we have to wait some time for the no-hitter, and then after the no-hitter, we have to wait for hitting the cycle. Stated another way, what is the total waiting time for the arrival of two different Poisson processes? The total waiting time is the time waited for the no-hitter, plus the time waited for the hitting the cycle.

- Now, you will write a function to sample out of the distribution described by this story.

Instructions

- Define a function with call signature successive_poisson(tau1, tau2, size=1) that samples the waiting time for a no-hitter and a hit of the cycle.
- Draw waiting times tau1 (size number of samples) for the no-hitter out of an exponential distribution and assign to t1.
- Draw waiting times tau2 (size number of samples) for hitting the cycle out of an exponential distribution and assign to t2.
- The function returns the sum of the waiting times for the two events.

```
def successive_poisson(tau1, tau2, size=1):
    # Draw samples out of first exponential distribution: t1
    t1 = np.random.exponential(tau1, size=size)

    # Draw samples out of second exponential distribution: t2
    t2 = np.random.exponential(tau2, size=size)

    return t1 + t2
```

Distribution of no-hitters and cycles

- Now, you'll use your sampling function to compute the waiting time to observer a no-hitter and hitting of the cycle. The mean waiting time for a no-hitter is 764 games, and the mean waiting time for hitting the cycle is 715 games.

Instructions

- Use your successive_poisson() function to draw 100,000 out of the distribution of waiting times for observing a no-hitter and a hitting of the cycle.
- Plot the PDF of the waiting times using the step histogram technique of a previous exercise. Don't forget the necessary keyword arguments. You should use bins=100, normed=True, and histtype='step'.
- Label the axes.
- Show your plot.

```
# Draw samples of waiting times: waiting_times
waiting_times = successive_poisson(764, 715, size = 100000)

# Make the histogram

_= plt.hist(waiting_times,bins=100,normed=True,histtype
='step')

# Label axes

_ = plt.xlabel('')
_ = plt.ylabel('')

# Show the plot
plt.show()
```

What are the chances of a horse matching or beating Secretariat's record?

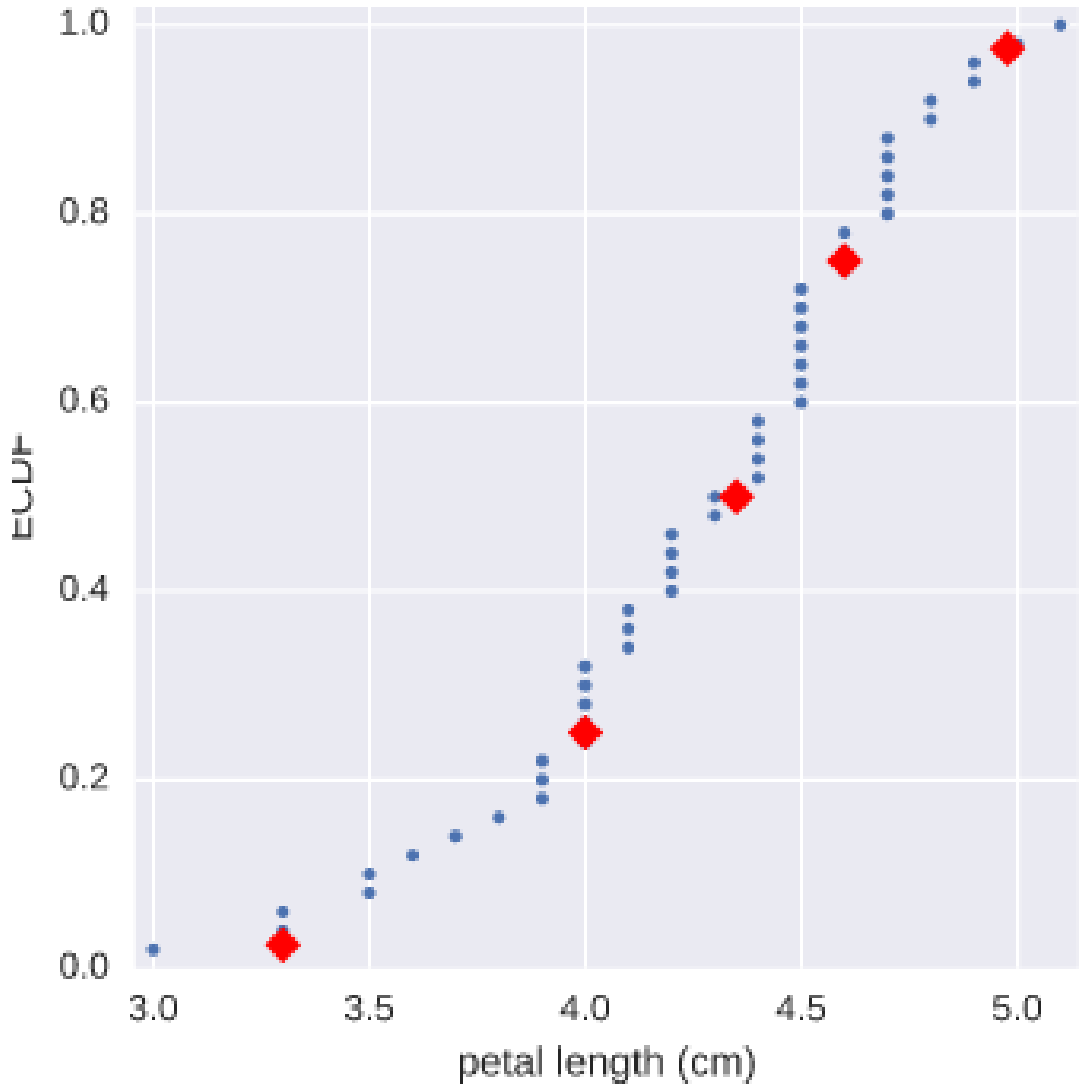
- Assume that the Belmont winners' times are Normally distributed (with the 1970 and 1973 years removed), what is the probability that the winner of a given Belmont Stakes will run it as fast or faster than Secretariat?

Instructions

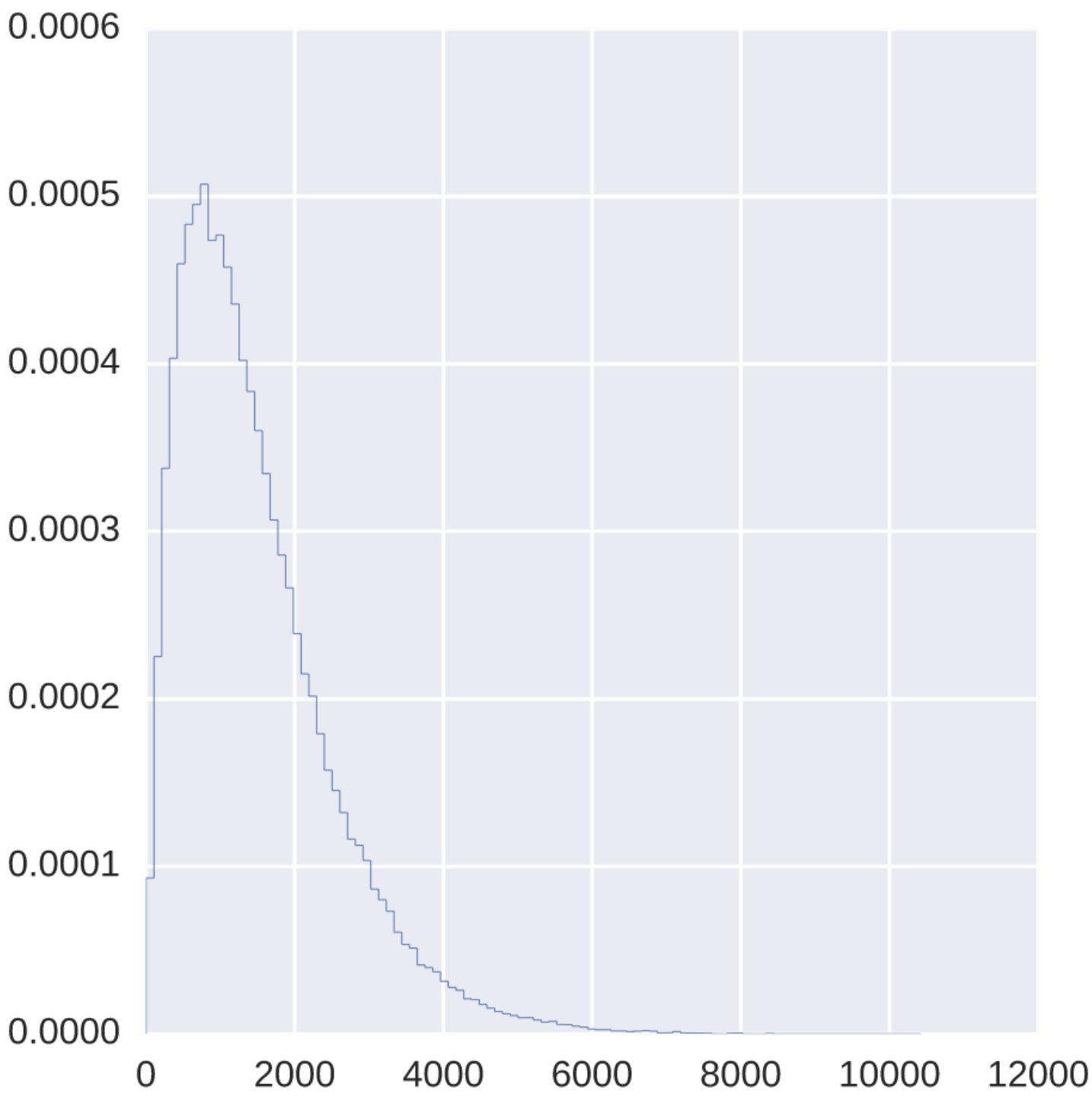
- Take 1,000,000 samples from the normal distribution using the np.random.normal() function. The mean mu and standard deviation sigma are already loaded into the namespace of your IPython instance.
- Compute the fraction of samples that have a time less than or equal to Secretariat's time of 144 seconds.
- Print the result.

```
# Take a million samples out of the Normal distribution: samples
samples = np.random.normal(mu, sigma, size=1000000)

# Compute the fraction that are faster than 144 seconds: prob
prob = np.sum(samples <= 144) / len(samples)

# Print the result
print('Probability of besting Secretariat:', prob)
```

```
# Plot the ECDF
_ = plt.plot(x_vers, y_vers, '.')
plt.margins(0.02)
_ = plt.xlabel('petal length (cm)')
_ = plt.ylabel('ECDF')

# Overlay percentiles as red diamonds.
_ = plt.plot(ptiles_vers, percentiles/100, marker='D', color
='red',
         linestyle='none')

# Show the plot
plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
3.0
3.5
4.0
4.5
5.0
petal length (cm)

- Great work! Notice that the PDF is peaked, unlike the waiting time for a single Poisson process. For fun (and enlightenment), I encourage you to also plot the CDF.


0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
0
2000
4000
6000
8000
10000
12000