Homework_05_WI25

March 7, 2025

1 Homework 05: Confidence Intervals and Sensitivity

1.1 Problem 1

Write a program to calculate an confidence limits at arbitrary confidence levels for a given number of observed counts (remember those are integers). Calculate upper and lower limits (if possible) at 68% and 95% CL for observed counts between 0 and 10.

Hint: Poisson distribution and stats.xxx.ppf may help

[1]:
```python
import scipy.stats as stats
```

[2]:
```python
def confidence_limits(observed_count, confidence_level):
    """
    Calculate the confidence limits for a given observed count and confidence␣
 ↪level.

    :param observed_count: Number of observed events (integer).
    :param confidence_level: Desired confidence level (e.g., 0.68 for 68% CL, 0.
 ↪95 for 95% CL).

    :return: Tuple (lower_limit, upper_limit).
    """
    # Implement your code below
    lower_prob = (1 - confidence_level) / 2
    upper_prob = 1 - lower_prob

    if observed_count == 0:
        # When no events are observed, we define the lower limit as 0.
        lower_limit = 0.0
        # For the upper limit we use the Garwood interval which amounts to:
        #   0.5 * chi2.ppf(upper_prob, 2*(observed_count+1))
        upper_limit = 0.5 * stats.chi2.ppf(upper_prob, 2 * (observed_count + 1))
    else:
        lower_limit = stats.poisson.ppf(lower_prob, observed_count)
        upper_limit = stats.poisson.ppf(upper_prob, observed_count)

    return lower_limit, upper_limit
```

```
[3]: assert confidence_limits(100, 0.68) == (90.0, 110.0)
     assert confidence_limits(100, 0.95) == (81.0, 120.0)
```

1.2 Problem 2

In this problem we will do a simplified version of searching for rare event physics signals. In this set up, the background is a flat background in energy space, and the signal we are searching for is a monoenergetic gaussian peak (monoenergetic with energy resolution results in a gaussian peak). We don't consider any efficiency loss (assuming we can detect all signal and background events), and what we are trying to do is to define the models, simulate the "fake" events, then fit the signal rate, and find the upperlimit on the signal rate.

1.2.1 Part 1

Defining the signal and background models. In those kind of fit (assuming we are using χ^2 fit), binning is important. Suppose you are expecting 1000 flat bakcground events between 100 and 300 GeV, with a 10 GeV binning, how many events are you expecting in each bin?

Note: Carefully deal with bin_edges and bin_centers, len(bin_edges) = len(bin_centers)+1 = n_bins+1

```
[4]: import numpy as np
     import math
     import matplotlib.pyplot as plt
     from iminuit import Minuit
```

```
[5]: def background(E_min, E_max, n_bins, B):
         '''
         Input:
             E_min, E_max: min and max of the energy range
             n_bins: number of bins within the energy range
             B: background event rate, unit: number of events
         Output:
             bin_centers: array contains the central energy of the energy bins
             bkg_counts: array contains the expected number of backgrounds in each␣
      ↪energy bin
         '''
         # Implement your code below
         # Calculate bin edges and centers
         bin_edges = np.linspace(E_min, E_max, n_bins + 1)
         energies = (bin_edges[:-1] + bin_edges[1:]) / 2.0
         # Distribute background events evenly over the bins
         bkg_counts = np.full(n_bins, B / n_bins)
         return energies, bkg_counts
```

```
[6]: E_min = 100
     E_max = 300
     n_bins = 40
```

```
B = 2000
```

[7]:
```
energies, bkg_counts = background(E_min, E_max, n_bins, B)
assert len(energies) == n_bins
assert math.isclose(np.sum(bkg_counts), B, abs_tol=1.0)
```

[8]:
```
from scipy.stats import norm
def signal(E_min, E_max, n_bins, E_0, sigma, S):
    '''
    Input:
        E_min, E_max: min and max of the energy range
        n_bins: number of bins within the energy range
        B: background event rate, unit: number of events
        E_0: the energy of the monoenergetic signal
        sigma: the width of the signal energy spectrum (assuming gaussian)
        S: total number of signal events
    Output:
        bin_centers: array contains the central energy of the energy bins
        signal_counts: array contains the expected number of signals in each␣
 ↪energy bin
    '''
    # Implement your code below
    bin_edges = np.linspace(E_min, E_max, n_bins + 1)
    energies = (bin_edges[:-1] + bin_edges[1:]) / 2.0
    # Compute the probability for a signal event to fall in each bin using the␣
 ↪Gaussian CDF.
    bin_probs = norm.cdf(bin_edges[1:], loc=E_0, scale=sigma) - norm.
 ↪cdf(bin_edges[:-1], loc=E_0, scale=sigma)
    signal_counts = S * bin_probs
    return energies, signal_counts
```

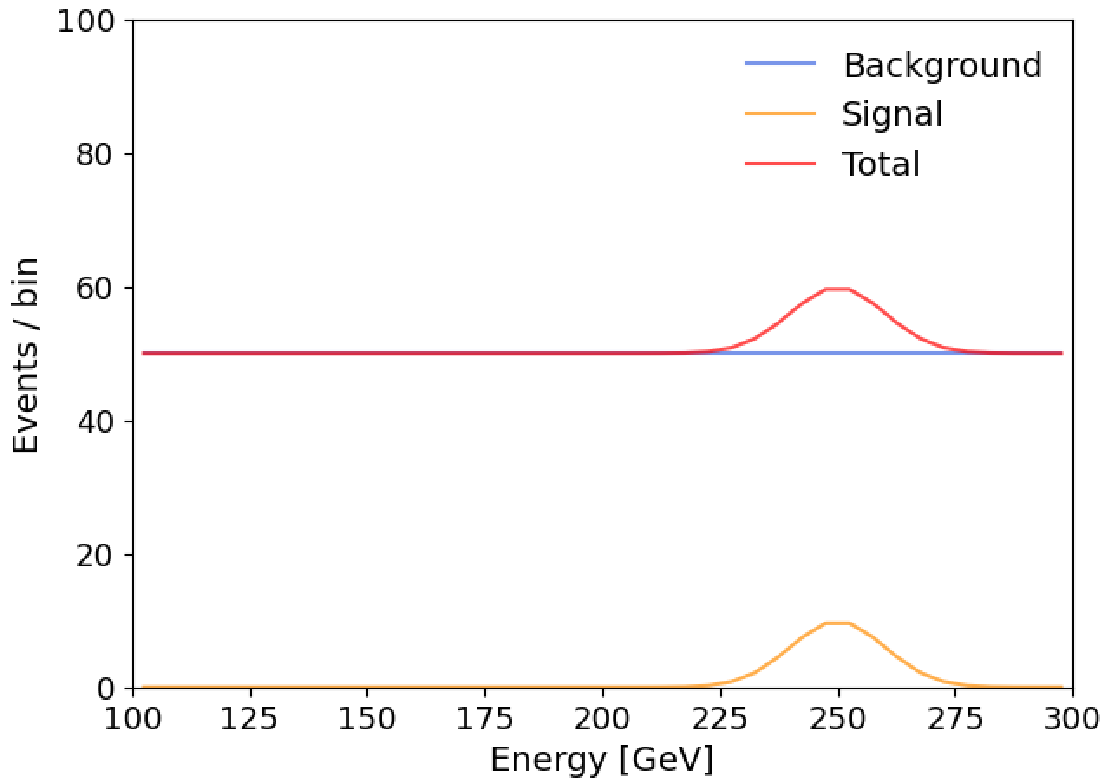
[9]:
```
E_0 = 250
sigma = 10
S = 50
```

[10]:
```
energies, signal_counts = signal(E_min, E_max, n_bins, E_0, sigma, S)
assert len(energies) == n_bins
assert math.isclose(np.sum(signal_counts), S, abs_tol=0.1)
```

Now we can plot the signal and background model and see how they look like

[11]:
```
plt.figure(figsize=(7, 5))
plt.plot(energies, bkg_counts, alpha=0.7, color='royalblue', label='Background')
plt.plot(energies, signal_counts, alpha=0.7, color='darkorange', label='Signal')
plt.plot(energies, signal_counts+bkg_counts, alpha=0.7, color='red',␣
 ↪label='Total')
plt.xlabel('Energy [GeV]', fontsize=14)
```

```
plt.ylabel('Events / bin', fontsize=14)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)
plt.legend(frameon=False, fontsize=14)
plt.xlim(100, 300)
plt.ylim(0, 100)
plt.show()
```



1.2.2 Part 2

Based on the signal and background model we have defined above, we are able to define our simulation for the "fake" events. Assume for each energy bin the observed number of events follows Poisson(expected), where expected is the number of expected events in the bin.

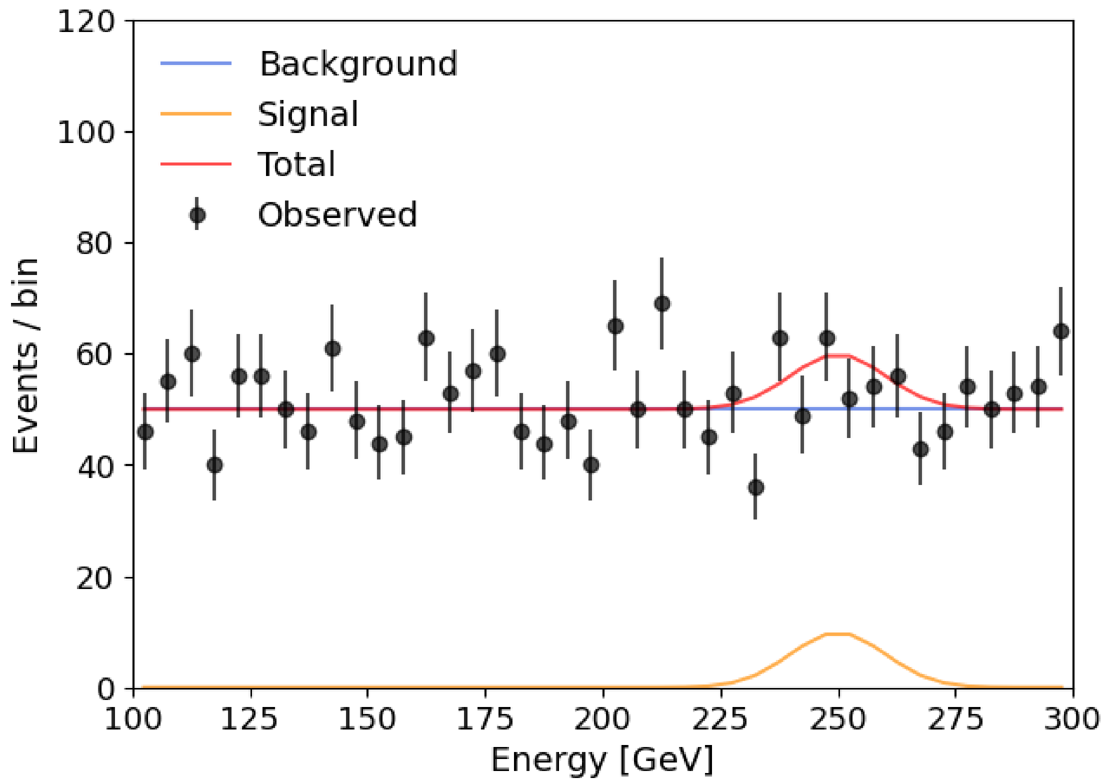
[12]:
```
def generate_toy_mc(E_min, E_max, n_bins, E_0, sigma, S, B):
    '''
    Input:
        E_min, E_max: min and max of the energy range
        n_bins: number of bins within the energy range
        E_0: energy of the monoenergetic signal
        sigma: width of the signal energy spectrum (assuming Gaussian)
        S: total number of signal eventss
```

```
        B: total number of background events
    Output:
        toy_data: simulated events in each bin, signal_observed +␣
↪background_observed
    '''
    # Implement your code below
    energies, bkg_counts = background(E_min, E_max, n_bins, B)
    energies, signal_counts = signal(E_min, E_max, n_bins, E_0, sigma, S)

    # Draw signal counts first, then background counts
    signal_observed = np.random.poisson(signal_counts)
    bkg_observed = np.random.poisson(bkg_counts)


    toy_data = signal_observed + bkg_observed
    return toy_data
```

[13]:
```
# Set random seed for reproducibility
np.random.seed(42)

observed_counts = generate_toy_mc(E_min, E_max, n_bins, E_0, sigma, S, B)
expected_counts = signal_counts+bkg_counts
assert observed_counts[0] == 46
assert observed_counts[-1] == 64
assert np.sum(observed_counts) == 2087
```

Now you can plot the "observed" data and compare with the theoretical distribution. The uncertainty (error bar) for each bin count N_i is $\sqrt{N_i}$.

[14]:
```
plt.figure(figsize=(7, 5))
plt.plot(energies, bkg_counts, alpha=0.7, color='royalblue', label='Background')
plt.plot(energies, signal_counts, alpha=0.7, color='darkorange', label='Signal')
plt.plot(energies, expected_counts, alpha=0.7, color='red', label='Total')
plt.errorbar(energies, observed_counts, yerr=np.sqrt(observed_counts),
             fmt='o', linestyle='None',
             alpha=0.7, color='black', label='Observed')
plt.xlabel('Energy [GeV]', fontsize=14)
plt.ylabel('Events / bin', fontsize=14)
plt.xticks(fontsize=13)
plt.yticks(fontsize=13)
plt.legend(loc='upper left', frameon=False, fontsize=14)
plt.xlim(100, 300)
plt.ylim(0, 120)
plt.show()
```



1.2.3 Part 3

Now let's define the χ-square function and perform the fit!

[15]:
```python
# Define the chi-square function, make use of the functions you defined␣
↪previously
def chi2(S_fit, B_fit):
    '''
    Input:
        S: total number of signal events
        B: total number of background events
    Output:
        Chi square
    '''
    # Implement your code below
    energies, bkg_fit = background(E_min, E_max, n_bins, B_fit)
    energies, signal_fit = signal(E_min, E_max, n_bins, E_0, sigma, S_fit)
    expected_fit = bkg_fit + signal_fit

    # Compute the chi-square value:
    chi2_value = np.sum((observed_counts - expected_fit)**2 / expected_fit)
```

```
    return chi2_value
```

[26]:
```
chi2_test = chi2(50, 2000)
assert math.isclose(chi2_test, 48.4744, abs_tol=1e-2), f"chi2_test failed:␣
↪chi2_test = {chi2_test}"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[26], line 2
      1 chi2_test = chi2(50, 2000)
----> 2 assert math.isclose(chi2_test, 48.4744, abs_tol=1e-2), f"chi2_test␣
↪failed: chi2_test = {chi2_test}"

AssertionError: chi2_test failed: chi2_test = 48.07842429971355
```

[19]:
```
# Initial guess of the parameters
S_init = 100
B_init = np.sum(observed_counts)
# Implement your code below (only one line)
minuit = Minuit(chi2, S_fit=S_init, B_fit=B_init)
minuit.limits = [(0, None), (0, None)]
minuit.migrad()  # Minimization

# Print the fit results
best_fit_S = minuit.values["S_fit"]
best_fit_B = minuit.values["B_fit"]

print(f"Best-fit S: {best_fit_S:.2f}")
print(f"Best-fit B: {best_fit_B:.2f}")

# Print the best-fit chi-square value
chi2_min = minuit.fval

print (f"Best-fit chi-square: {chi2_min:.2f}")
```

```
Best-fit S: 15.60
Best-fit B: 2093.16
Best-fit chi-square: 43.72
```

[23]:
```
assert math.isclose(best_fit_S, 15.99, abs_tol=1e-2), f"Assertion failed:␣
↪best_fit_S = {best_fit_S}"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[23], line 1
```

```
----> 1 assert math.isclose(best_fit_S, 15.99, abs_tol=1e-2), f"Assertion failed:
 ↪ best_fit_S = {best_fit_S}"

AssertionError: Assertion failed: best_fit_S = 15.597740352899509
```

[24]:
```
assert math.isclose(best_fit_B, 2092.99, abs_tol=1e-2), f"Assertion failed:␣
 ↪best_fit_B = {best_fit_B}"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[24], line 1
----> 1 assert math.isclose(best_fit_B, 2092.99, abs_tol=1e-2), f"Assertion␣
 ↪failed: best_fit_B = {best_fit_B}"

AssertionError: Assertion failed: best_fit_B = 2093.1610536493317
```

[25]:
```
assert math.isclose(chi2_min, 44.14, abs_tol=1e-2), f"Assertion failed:␣
 ↪chi2_min = {chi2_min}"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[25], line 1
----> 1 assert math.isclose(chi2_min, 44.14, abs_tol=1e-2), f"Assertion failed:␣
 ↪chi2_min = {chi2_min}"

AssertionError: Assertion failed: chi2_min = 43.71791780976758
```

1.2.4 Part 4

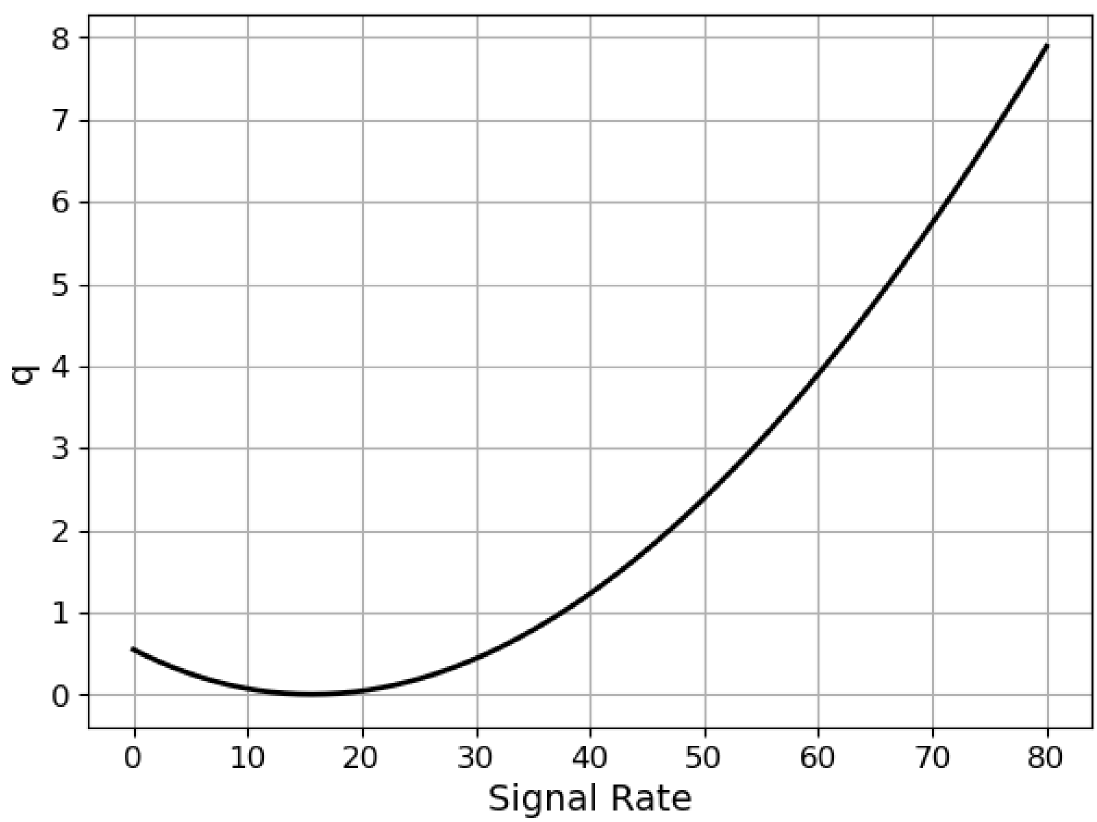
As we can see, although we set S=50 and B=2000, after simulating the "observed" data we fit the observed with our signal and background models, the best-fit values we get is not very close to the values we set.

Now, let's go ahead and find the upperlimit on the signal rate at 90% confidence level.

The way we will do this is as follows: 1. Fix a S_{fit} (or S_{scan}) value 2. Perform the fit again (here it will be fitting B_fit only 3. Get the best-fit $\chi^2_{fix\ S}$, calculate $q = \chi^2_{fix\ S} - \chi^2_{best-fit}$ 4. Look for the S_{scan} value that makes its corresponding χ^2 value larger than best-fit $\chi^2_{best-fit}$ by 2.71 (based on approximation you learned from lecture), i.e. $q = 2.71$.

[27]:
```
# Array to store q values for different signal strengths
S_scan = np.linspace(0, 80, 200)  # Test range of signal strengths
q_values = []

for s in S_scan:
    # Implement your code below (replace "..." to your code)
```

```
    # Perform a minimization for each fixed S_scan
    # Hint: try Minuit(lambda B_fit: ... ) for a minuit fit with a fixed S_fit␣
↪parameter
    m_fixed = Minuit(lambda B_fit: chi2(s, B_fit), B_fit=np.
↪sum(observed_counts))
    m_fixed.limits = [(0, None)]
    m_fixed.migrad()
    chi2_fixed = m_fixed.fval

    # Calculate q for this signal rate
    q = chi2_fixed - chi2_min
    q_values.append(q)

# Plot q vs. signal rate
plt.figure(figsize=(7, 5))
plt.plot(S_scan, q_values, color='black', linewidth=2)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.xlabel("Signal Rate", fontsize=14)
plt.ylabel("q", fontsize=14)
plt.grid()
plt.show()
```



Now to find the S_{scan} value corresponding to $q = 2.7$, in principle one can fit the curve q(S), but to be simple as we have fairly fine scanning values, we can just find the closet one.

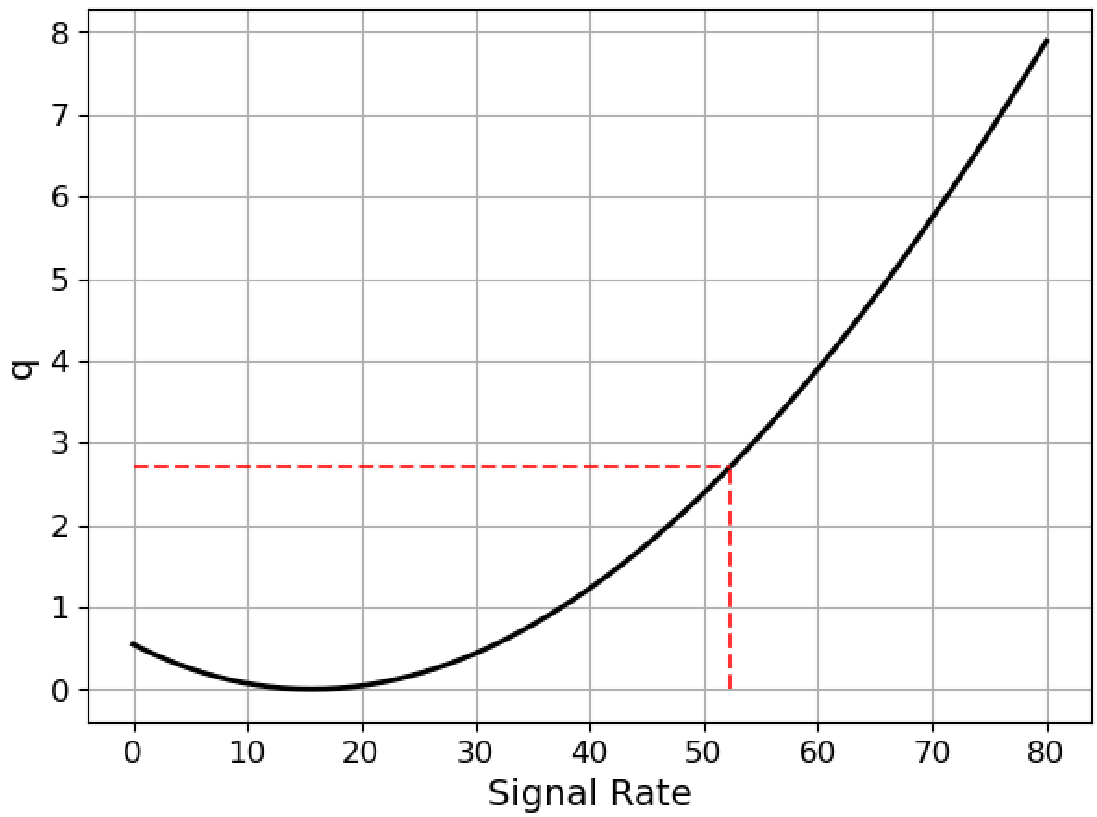
[28]:
```
closest_diff = float('inf')
closest_rate = float('inf')
for i in range(len(S_scan)):
    if abs(q_values[i]-2.71) < closest_diff and S_scan[i]>best_fit_S: # Note␣
↪that the upperlimit has to be greater than S_best-fit
        closest_diff = abs(q_values[i]-2.71)
        closest_rate = S_scan[i]
```

[29]:
```
print (f"Upper-limit on Signal Rate: {closest_rate:.2f}")
assert math.isclose(closest_rate, 52.66, abs_tol=1e-1), f"closest_rate:␣
↪{closest_rate}"
```

```
Upper-limit on Signal Rate: 52.26
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[29], line 2
      1 print (f"Upper-limit on Signal Rate: {closest_rate:.2f}")
----> 2 assert math.isclose(closest_rate, 52.66, abs_tol=1e-1), f"closest_rate:
↪{closest_rate}"

AssertionError: closest_rate: 52.26130653266332
```

[30]:
```
# Plot q vs. signal rate and the critial signal rate
plt.figure(figsize=(7, 5))
plt.plot(S_scan, q_values, color='black', linewidth=2)
plt.hlines(2.71, 0, closest_rate, linestyle='--', color='red', alpha=0.8)
plt.vlines(closest_rate, 0, 2.71, linestyle='--', color='red', alpha=0.8)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.xlabel("Signal Rate", fontsize=14)
plt.ylabel("q", fontsize=14)
plt.grid()
plt.show()
```



q
Signal Rate
0
1
2
3
4
5
6
7
8
0
10
20
30
40
50
60
70
80


```
[ ]:
```

```
[ ]:
```