Challenge2_MeasurementErrorMitigation

May 5, 2020

1 Exercise 2: Measurement Error Mitigation

Present day quantum computers are subject to noise of various kinds. The principle behind error mitigation is to reduce the effects from a specific source of error. Here we will look at mitigating measurement errors, i.e., errors in determining the correct quantum state from measurements performed on qubits.

Measurement Error Mitigation

In the above picture, you can see the outcome of applying measurement error mitigation. On the left, the histogram shows results obtained using the device `ibmq_vigo`. The ideal result should have shown 50% counts 00000 and 50% counts 10101. Two features are notable here:

- First, notice that the result contains a skew toward 00000. This is because of energy relaxation of the qubit during the measurement process. The relaxation takes the $|1\rangle$ state to the $|0\rangle$ state for each qubit.
- Second, notice that the result contains other counts beyond just 00000 and 10101. These arise due to various errors. One example of such errors comes from the discrimination after measurement, where the signal obtained from the measurement is identified as either $|0\rangle$ or $|1\rangle$.

The picture on the right shows the outcome of performing measurement error mitigation on the results. You can see that the device counts are closer to the ideal expectation of 50 results in 00000 and 50 results in 10101, while other counts have been significantly reduced.

1.1 How measurement error mitigation works

We start by creating a set of circuits that prepare and measure each of the 2^n basis states, where n is the number of qubits. For example, $n = 2$ qubits would prepare the states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ individually and see the resulting outcomes. The outcome statistics are then captured by a matrix M, where the element M_{ij} gives the probability to get output state $|i\rangle$ when state $|j\rangle$ was prepared. Even for a state that is in an arbitrary superposition $|\psi\rangle = \sum_j \alpha_j |j\rangle$, the linearity of quantum mechanics allows us to write the noisy output state as $|\psi_{noisy}\rangle = M|\psi\rangle$.

The goal of measurement error mitigation is not to model the noise, but rather to apply a classical correction that undoes the errors. Given a noisy outcome, measurement error mitigation seeks to recover the initial state that led to that outcome. Using linear algebra we can see that given a noisy outcome $|\psi_{noisy}\rangle$, this can be done by applying the inverse of the matrix M, i.e., $|\psi\rangle = M^{-1}|\psi_{noisy}\rangle$. Note that the matrix M recovered from the measurements is usually non-invertible, thus requiring a generalized inverse method to solve. Additionally, the noise is not deterministic, and has fluctuations, so this will in general not give you the ideal noise-free state, but it should bring you closer to it.

You can find a more detailed description of measurement error mitigation in Chapter 5.2 of the Qiskit textbook.

The goal of this exercise is to create a calibration matrix M that you can apply to noisy results (provided by us) to infer the noise-free results.

In Qiskit, creating the circuits that test all basis states to replace the entries for the matrix is done by the following code:

[2]:
```
#initialization
%matplotlib inline

# Importing standard Qiskit libraries and configuring account
from qiskit import IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.providers.ibmq import least_busy
from qiskit.tools.jupyter import *
from qiskit.tools.monitor import job_monitor
from qiskit.visualization import *
from qiskit.ignis.mitigation.measurement import complete_meas_cal,␣
 ↪CompleteMeasFitter


provider = IBMQ.load_account() # load your IBM Quantum Experience account
# If you are a member of the IBM Q Network, fill your hub, group, and project␣
 ↪information to
# get access to your premium devices.
# provider = IBMQ.get_provider(hub='', group='', project='')

from may4_challenge.ex2 import get_counts, show_final_answer

num_qubits = 5
meas_calibs, state_labels = complete_meas_cal(range(num_qubits),␣
 ↪circlabel='mcal')
```

Next, run these circuits on a real device! You can choose your favorite device, but we recommend choosing the least busy one to decrease your wait time in the queue. Upon executing the following cell you will be presented with a widget that displays all the information about the least busy backend that was selected. Clicking on the "Error Map" tab will reveal the latest noise information for the device. Important for this challenge is the "readout" (measurement) error located on the left (and possibly right) side of the figure. It is common to see readout errors of a few percent on each qubit. These are the errors we are mitigating in this exercise.

[3]:
```
# find the least busy device that has at least 5 qubits
```

```
backend = least_busy(provider.backends(filters=lambda x: x.configuration().
 ↪n_qubits >= num_qubits and
                                                   not x.configuration().simulator and x.
 ↪status().operational==True))
backend
```

```
VBox(children=(HTML(value="<h1 style='color:#ffffff;background-color:#000000;padding-top: 1%;pa
```

[3]:
```
<IBMQBackend('ibmq_ourense') from IBMQ(hub='ibm-q', group='open',
project='main')>
```

Run the next cell to implement all of the above steps. In order to average out fluctuations as much as possible, we recommend choosing the highest number of shots, i.e., `shots=8192` as shown below.

The call to `transpile` maps the measurement calibration circuits to the topology of the backend being used. `backend.run()` sends the circuits to the IBM Quantum device returning a `job` instance, whereas `%qiskit_job_watcher` keeps track of where your submitted job is in the pipeline.

[4]:
```
# run experiments on a real device
shots = 8192
experiments = transpile(meas_calibs, backend=backend, optimization_level=3)
job = backend.run(assemble(experiments, shots=shots))
print(job.job_id())
%qiskit_job_watcher
```

```
5eb105eef58e720012c4b8d6
```

```
Accordion(children=(VBox(layout=Layout(max_width='710px', min_width='710px')),), layout=Layout
```

```
<IPython.core.display.Javascript object>
```

Note that you might be in the queue for quite a while. You can expand the ‘IBMQ Jobs’ window that just appeared in the top left corner to monitor your submitted jobs. Make sure to keep your job ID in case you ran other jobs in the meantime. You can then easily access the results once your job is finished by running

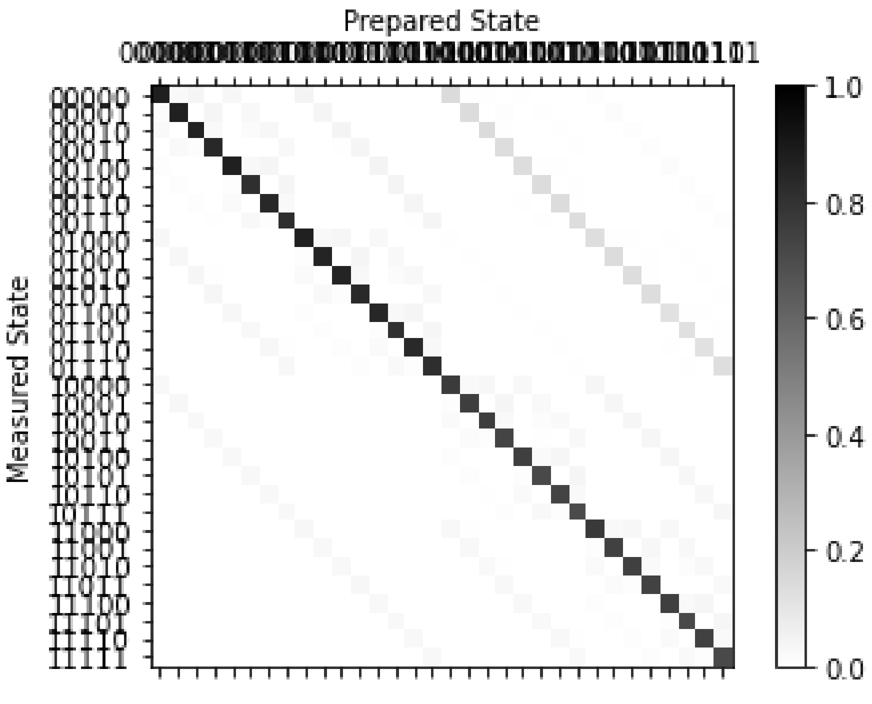
```
job = backend.retrieve_job('YOUR_JOB_ID')
```

Once you have the results of your job, you can create the calibration matrix and calibration plot using the following code. However, as the counts are given in a dictionary instead of a matrix, it is more convenient to use the measurement filter object that you can directly apply to the noisy counts to receive a dictionary with the mitigated counts.

[5]:
```
# get measurement filter
cal_results = job.result()
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')
meas_filter = meas_fitter.filter
print(meas_fitter.cal_matrix)
```

```
meas_fitter.plot_calibration()
```

```
[[8.77563477e-01 2.50244141e-02 3.99169922e-02 … 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [1.29394531e-02 8.68164062e-01 6.10351562e-04 … 6.10351562e-04
  0.00000000e+00 0.00000000e+00]
 [2.57568359e-02 1.58691406e-03 8.62670898e-01 … 2.44140625e-04
  3.66210938e-04 1.22070312e-04]
 …
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 … 7.12890625e-01
  8.54492188e-04 3.67431641e-02]
 [0.00000000e+00 0.00000000e+00 1.22070312e-04 … 3.54003906e-03
  7.40356445e-01 2.64892578e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 … 2.47802734e-02
  9.64355469e-03 7.07641602e-01]]
```



In the calibration plot you can see the correct outcomes on the diagonal, while all incorrect outcomes are off-diagonal. Most of the latter are due to T1 errors depolarizing the states from $|1\rangle$ to $|0\rangle$ during the measurement, which causes the matrix to be asymmetric.

Below, we provide you with an array of noisy counts for four different circuits. Note that as measurement error mitigation is a device-specific error correction, the array you receive depends on the backend that you have used before to create the measurement filter.

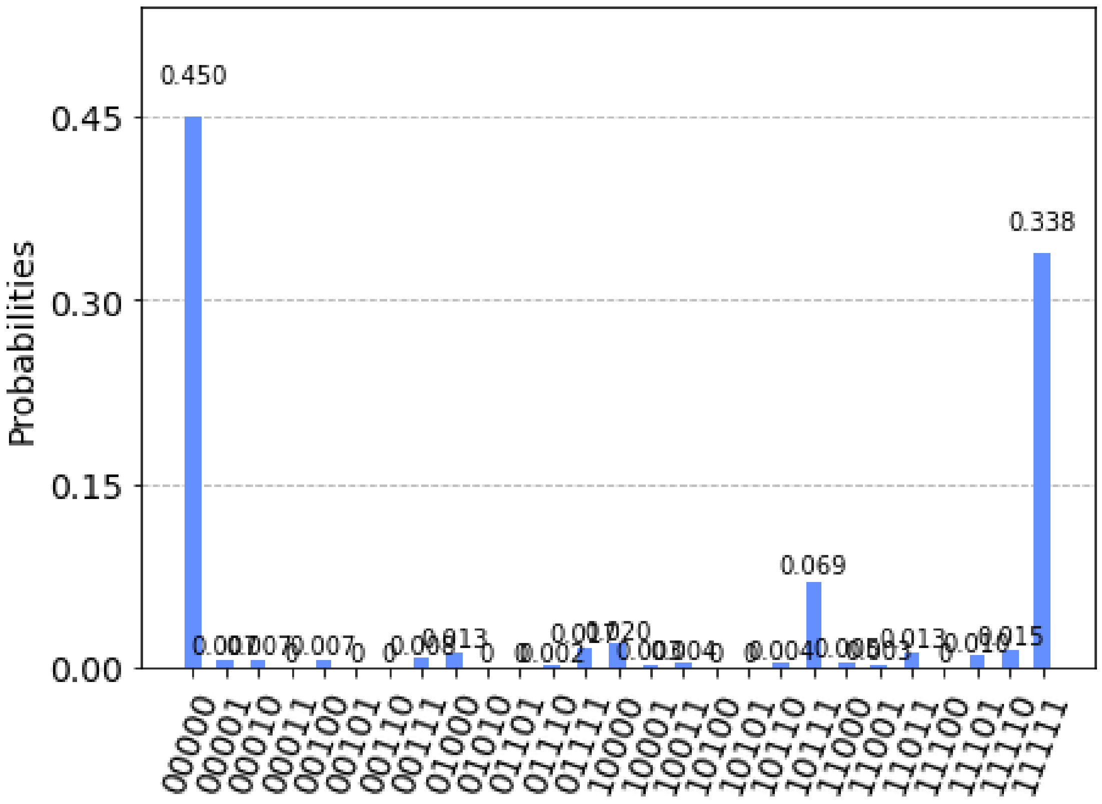
Apply the measurement filter in order to get the mitigated data. Given this mitigated data, choose which error-free outcome would be most likely.

As there are other types of errors for which we cannot correct with this method, you will not get completely noise-free results, but you should be able to guess the correct results from the trend of the mitigated results.

1.2 i) Consider the first set of noisy counts:

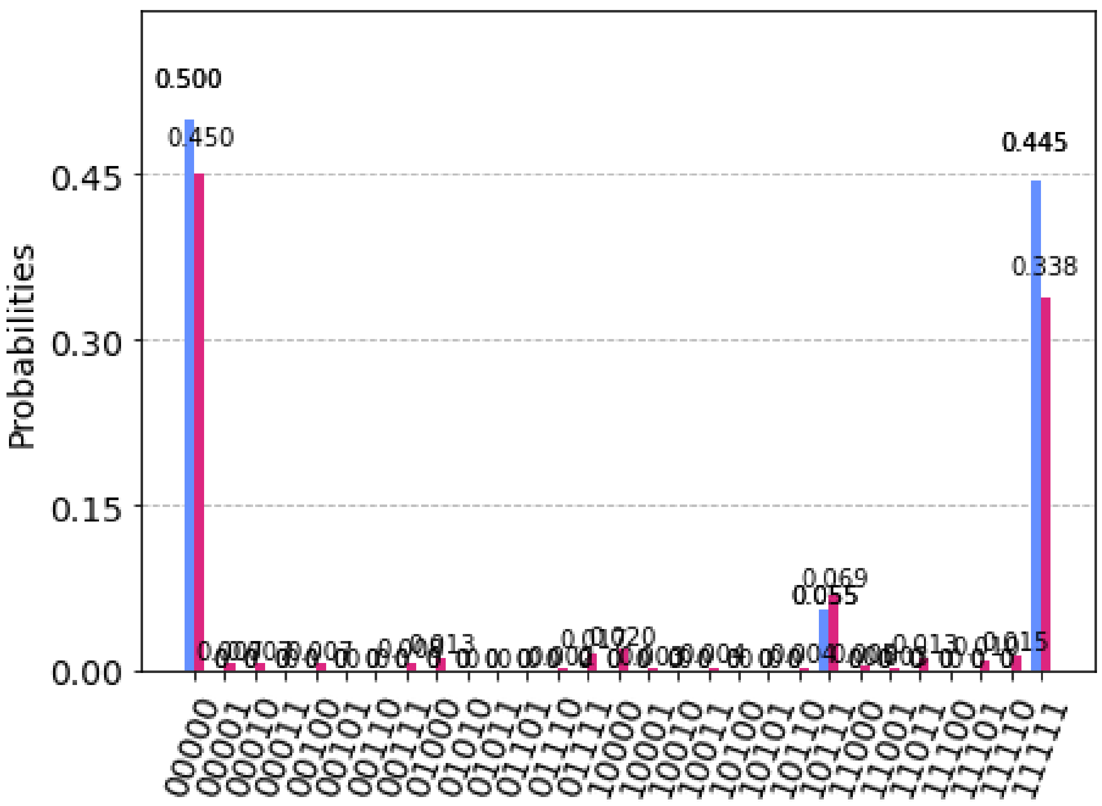
[6]:
```
# get noisy counts
noisy_counts = get_counts(backend)
plot_histogram(noisy_counts[0])
```

[6]:



[7]:
```
# apply measurement error mitigation and plot the mitigated counts
mitigated_counts_0 = meas_filter.apply(noisy_counts[0])
plot_histogram([mitigated_counts_0, noisy_counts[0]])
```

[7]:



1.3 Which of the following histograms most likely resembles the *error-free* counts of the same circuit?

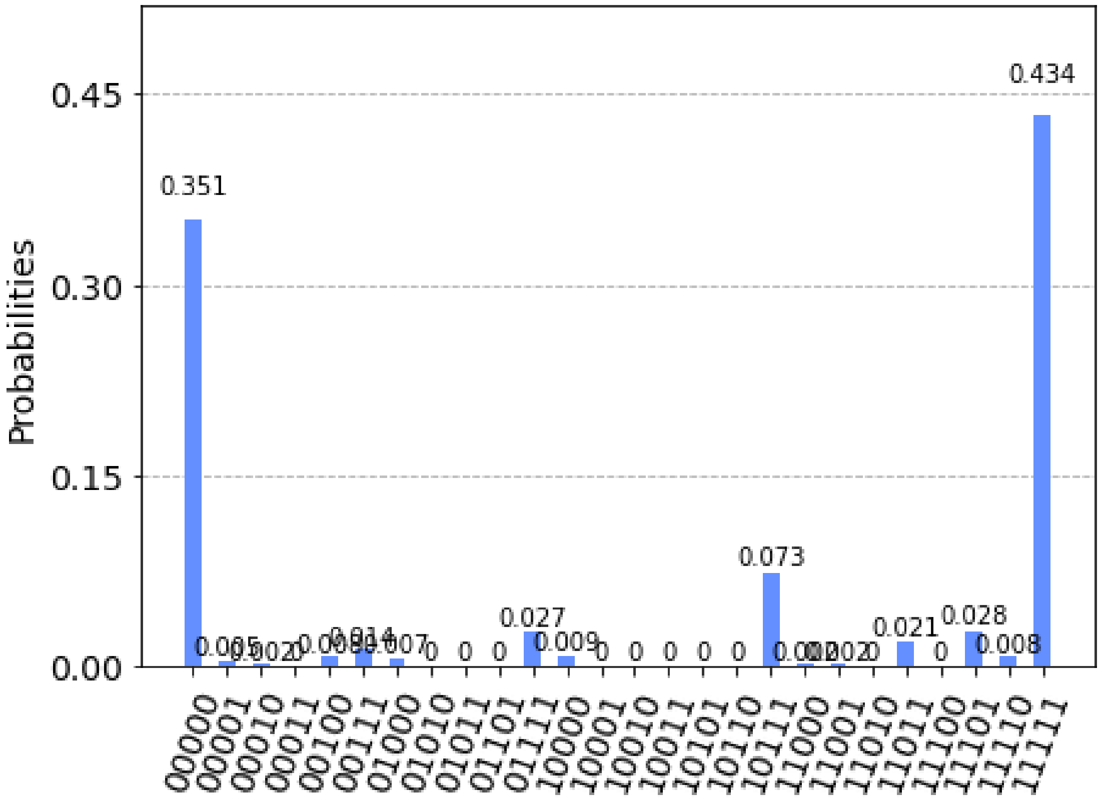
a)
b)
c)
d)

[30]:
```
# uncomment whatever answer you think is correct
#answer1 = 'a'
#answer1 = 'b'
answer1 = 'c'
#answer1 = 'd'
```

1.4 ii) Consider the second set of noisy counts:

[9]:
```
# plot noisy counts
plot_histogram(noisy_counts[1])
```

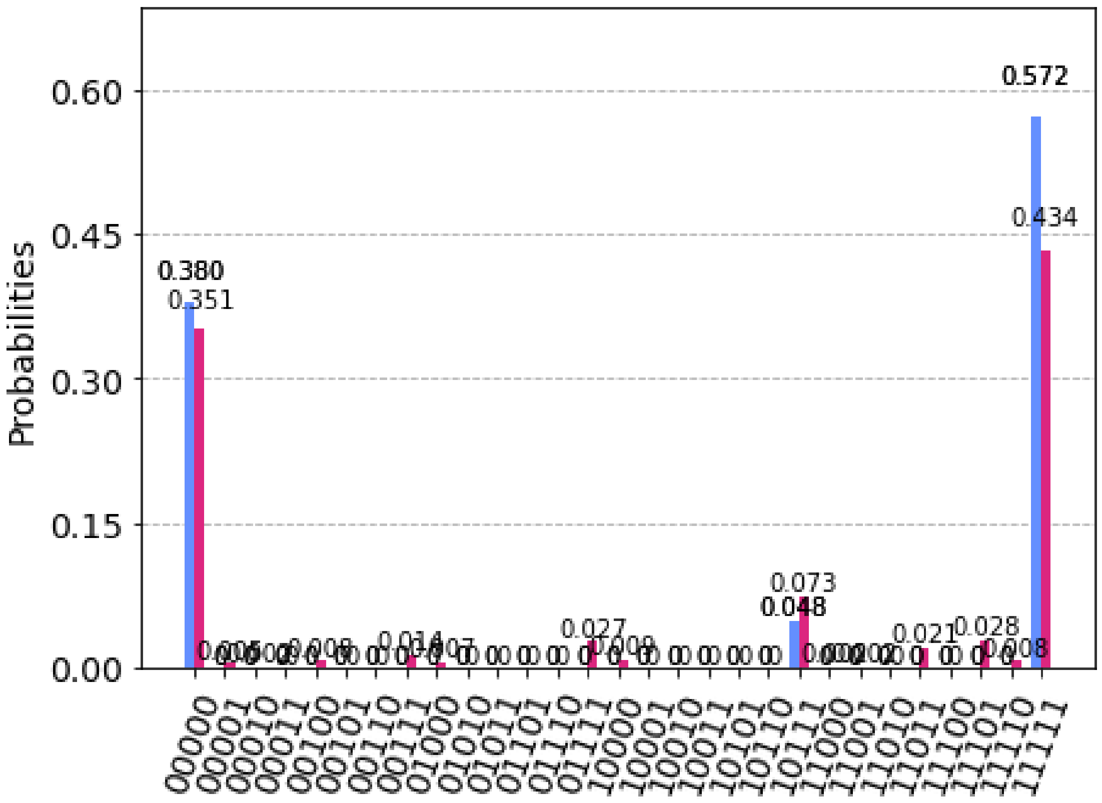
[9]:



[10]:
```
# apply measurement error mitigation
# insert your code here to do measurement error mitigation on noisy_counts[1]
mitigated_counts_1 = meas_filter.apply(noisy_counts[1])

plot_histogram([mitigated_counts_1, noisy_counts[1]])
```

[10]:



1.5 Which of the following histograms most likely resembles the *error-free* counts of the same circuit?

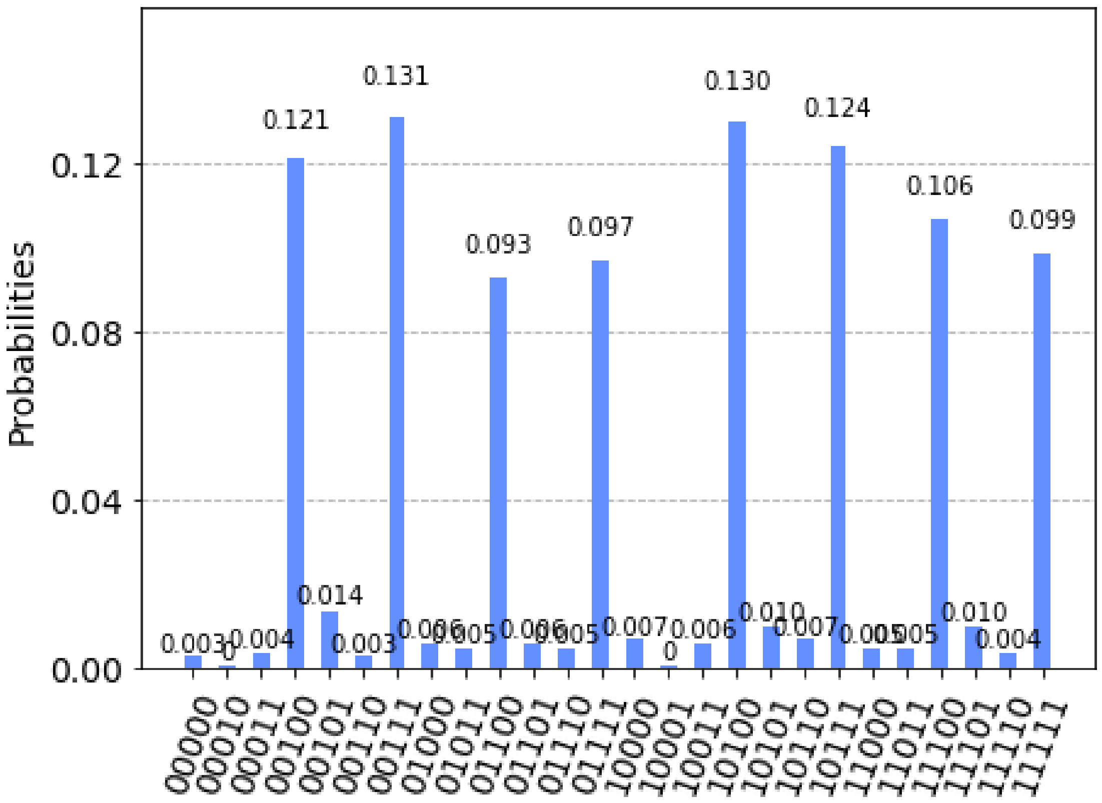
a)
b)
c)
d)

[31]:
```
# uncomment whatever answer you think is correct
#answer2 = 'a'
#answer2 = 'b'
#answer2 = 'c'
answer2 = 'd'
```

1.6 iii) Next, consider the third set of noisy counts:

```
[12]: # plot noisy counts
      plot_histogram(noisy_counts[2])
```

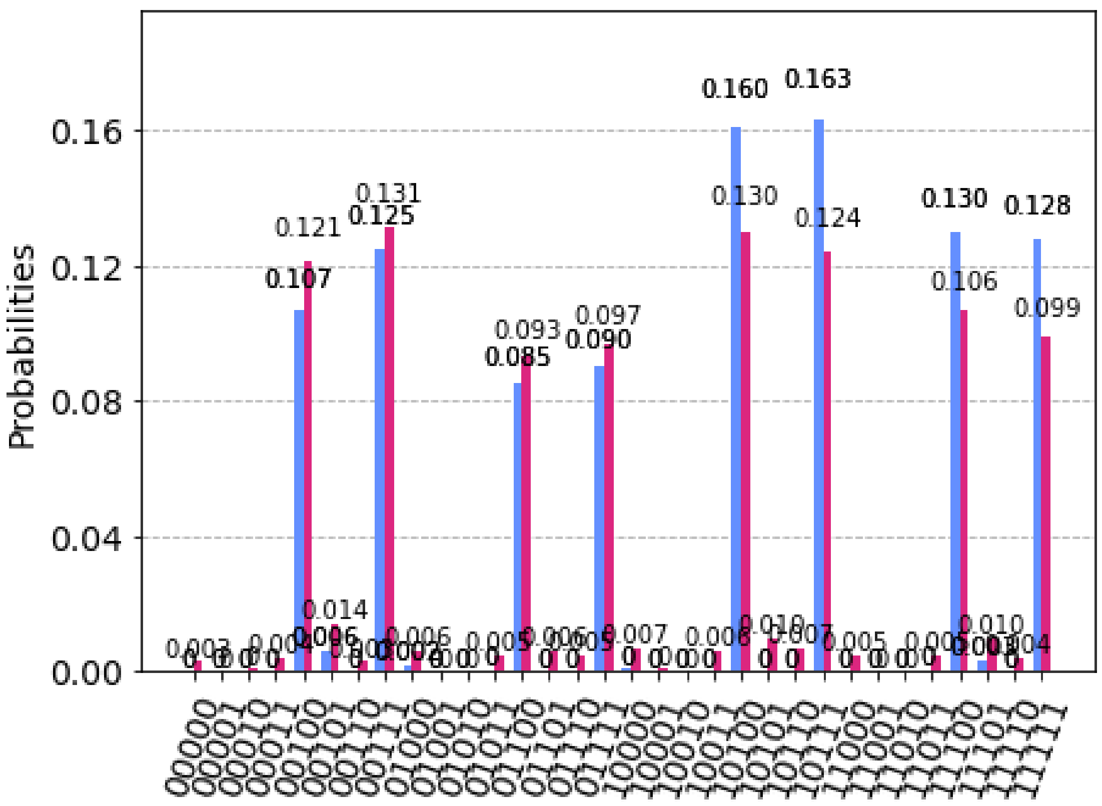
[12]:



```
[13]: # apply measurement error mitigation
      # insert your code here to do measurement error mitigation on noisy_counts[2]
      mitigated_counts_2 = meas_filter.apply(noisy_counts[2])

      plot_histogram([mitigated_counts_2, noisy_counts[2]])
```

[13]:



1.7 Which of the following histograms most likely resembles the *error-free* counts of the same circuit?

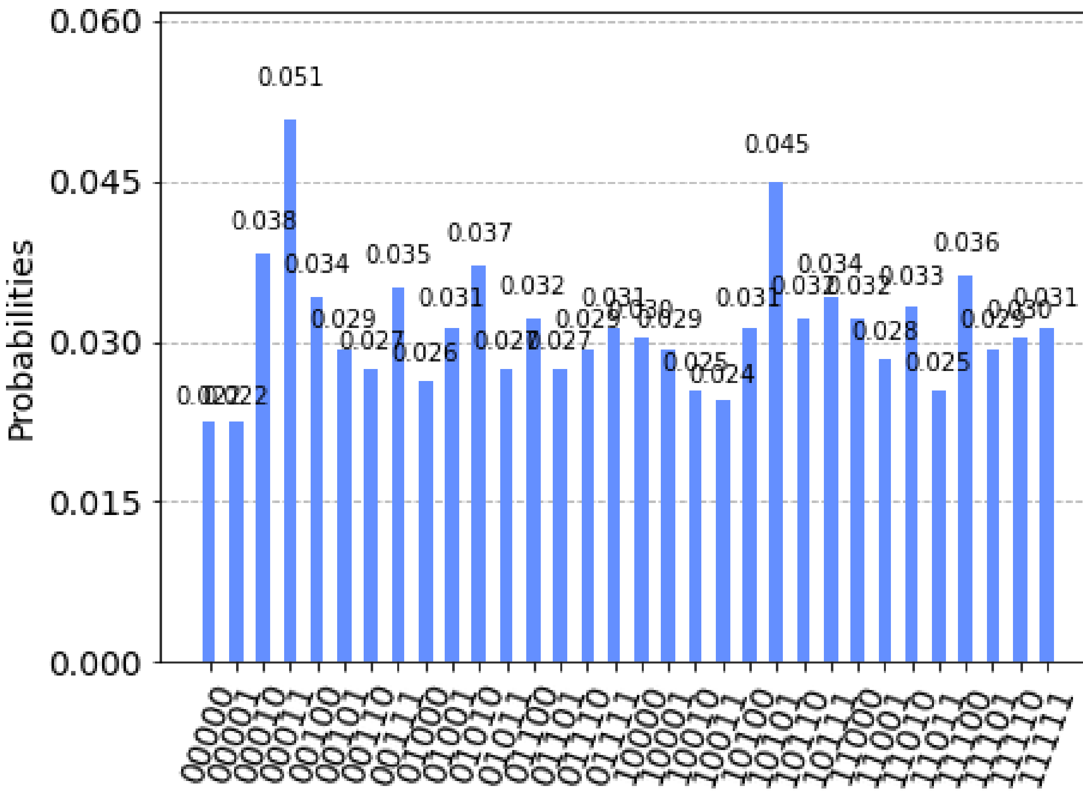
a)
b)
c)
d)

```
[32]: # uncomment whatever answer you think is correct
#answer3 = 'a'
answer3 = 'b'
#answer3 = 'c'
#answer3 = 'd'
```

1.8 iv) Finally, consider the fourth set of noisy counts:

[15]:
```
# plot noisy counts
plot_histogram(noisy_counts[3])
```

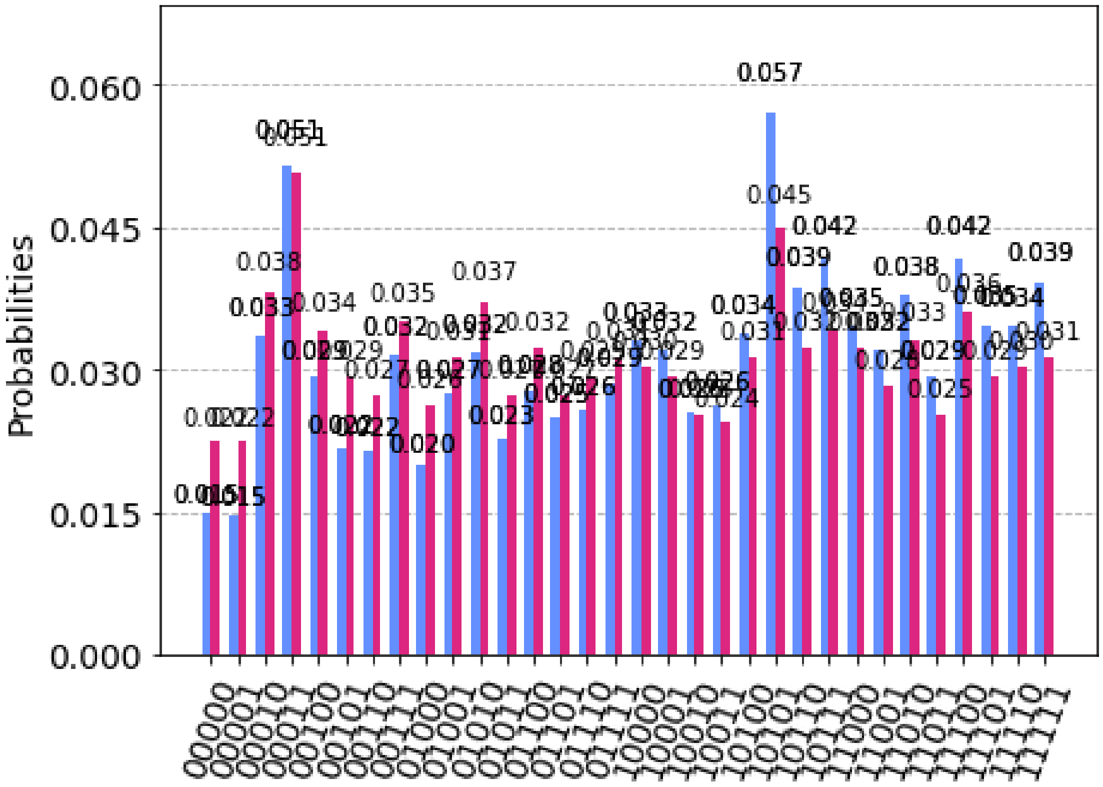
[15]:



[16]:
```
# apply measurement error mitigation
# insert your code here to do measurement error mitigation on noisy_counts[3]
mitigated_counts_3 = meas_filter.apply(noisy_counts[3])

plot_histogram([mitigated_counts_3, noisy_counts[3]])
```

[16]:



1.9 Which of the following histograms most likely resembles the *error-free* counts of the same circuit?

a)
b)
c)
d)

```
[33]: # uncomment whatever answer you think is correct
      #answer4 = 'a'
      answer4 = 'b'
      #answer4 = 'c'
      #answer4 = 'd'
```

The answer string of this exercise is just the string of all four answers. Copy and paste the output of the next line on the IBM Quantum Challenge page to complete the exercise and track your progress.

```
[34]: # answer string
      show_final_answer(answer1, answer2, answer3, answer4)
```

```
<IPython.core.display.HTML object>
```

Now that you are done, move on to the next exercise!

```
[ ]:
```