Feature Selection with the D-Wave System

This notebook demonstrates the formulation of an $\binom{n}{k}$ optimization problem for solution on a D-Wave quantum computer. The method used in the example problem of this notebook—feature-selection for machine learning— is applicable to problems from a wide range of domains; for example financial portfolio optimization.

1. What is Feature Selection? defines and explains the feature-selection problem.
2. Feature Selection by Mutual Information describes a particular method of feature selection that is demonstrated in this notebook.
3. Solving Feature Selection on a Quantum Computer shows how such optimization problems can be formulated for solution on a D-Wave quantum computer.
4. Example Application: Predicting Survival of Titanic Passengers demonstrates the use of *Kerberos*, an out-of-the-box classical-quantum hybrid sampler, to select optimal features for a public-domain dataset.

This notebook should help you understand both the techniques and Ocean software tools used for solving optimization problems on D-Wave quantum computers.

New to Jupyter Notebooks? JNs are divided into text or code cells. Pressing the **Run** button in the menu bar moves to the next cell. Code cells are marked by an "In: []" to the left; when run, an asterisk displays until code completion: "In: [*]".

What is Feature Selection?

Statistical and machine-learning models use sets of input variables ("features") to predict output variables of interest. Feature selection can be part of the model design process: selecting from a large set of potential features a highly informative subset simplifies the model and reduces dimensionality.

For example, to build a model that predicts the ripening of hothouse tomatoes, Farmer MacDonald daily records the date, noontime temperature, daylight hours, degree of cloudiness, rationed water and fertilizer, soil humidity, electric-light hours, etc. These measurements constitute a list of potential features. After a growth cycle or two, her analysis reveals some correlations between these features and crop yields:

- fertilizer seems a strong predictor of fruit size
- cloudiness and daylight hours seem poor predictors of growth
- water rations and soil humidity seem a highly correlated pair of strong predictors of crop rot

Farmer MacDonald suspects that her hothouse's use of electric light reduces dependency on seasons and sunlight. She can simplify her model by discarding date, daylight hours, and cloudiness. She can record just water ration or just soil humidity rather than both.

For systems with large numbers of potential input information—for example, weather forecasting or image recognition—model complexity and required compute resources can be daunting. Feature selection can help make such models tractable.

However, optimal feature selection can itself be a hard problem. This example introduces a powerful method of optimizing feature selection based on a complex probability calculation. This calculation is

submitted for solution to a quantum computer.

Illustrative Toy Problem

This subsection illustrates the use of feature selection with a simple example: a toy system with a single output generated from three inputs.

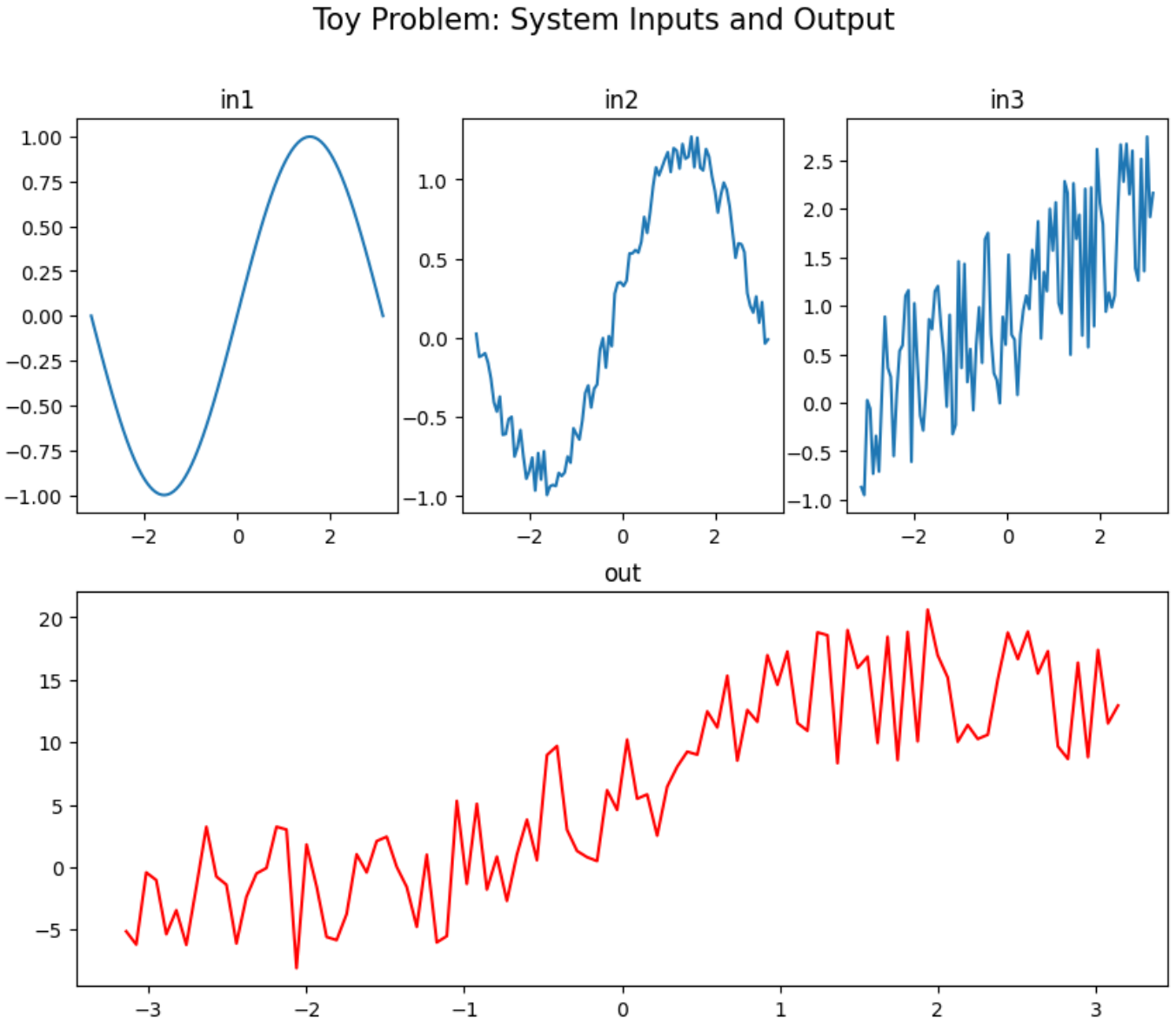
The model built to predict the system's output is even simpler: it uses just two of three possible features (inputs). You can expect it to perform better when the selected two features are more independent, assuming all three contribute somewhat commensurately to the system output (if an independent feature contributes less than the difference between two dependent ones, this might not be true). In the case of Farmer MacDonalds's tomatoes, a model using rationed water and fertilizer should perform better than one using rationed water and soil humidity.

The code cell below uses the NumPy library to define three inputs, the first two of which are very similar: a sine, a noisy sine, and a linear function with added random noise. It defines an output that is a simple linear combination of these three inputs.

In [1]:
```
import numpy as np

sig_len = 100
# Three inputs: in1 & in2 are similar
in1 = np.sin(np.linspace(-np.pi, np.pi, sig_len)).reshape(sig_len, 1)
in2 = np.sin(np.linspace(-np.pi+0.1, np.pi+0.2, sig_len)).reshape(sig_len, 1) + 0.3*np.rand
in3 = np.linspace(-1, 1, sig_len).reshape(sig_len,1) + 2*np.random.rand(sig_len, 1)

out = 2*in1 + 3*in2 + 6*in3
```

Plot the features and variable of interest (the output). In this and other cells below, graphics code is imported from a `helpers` module. To see this code, navigate to the `helpers` folder of this repository.

In [2]:
```
from helpers.plots import plot_toy_signals
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd

# Store problem in a pandas DataFrame for later use
toy = pd.DataFrame(np.hstack((in1, in2, in3, out)), columns=["in1", "in2", "in3", "out"])

plot_toy_signals(toy)
```



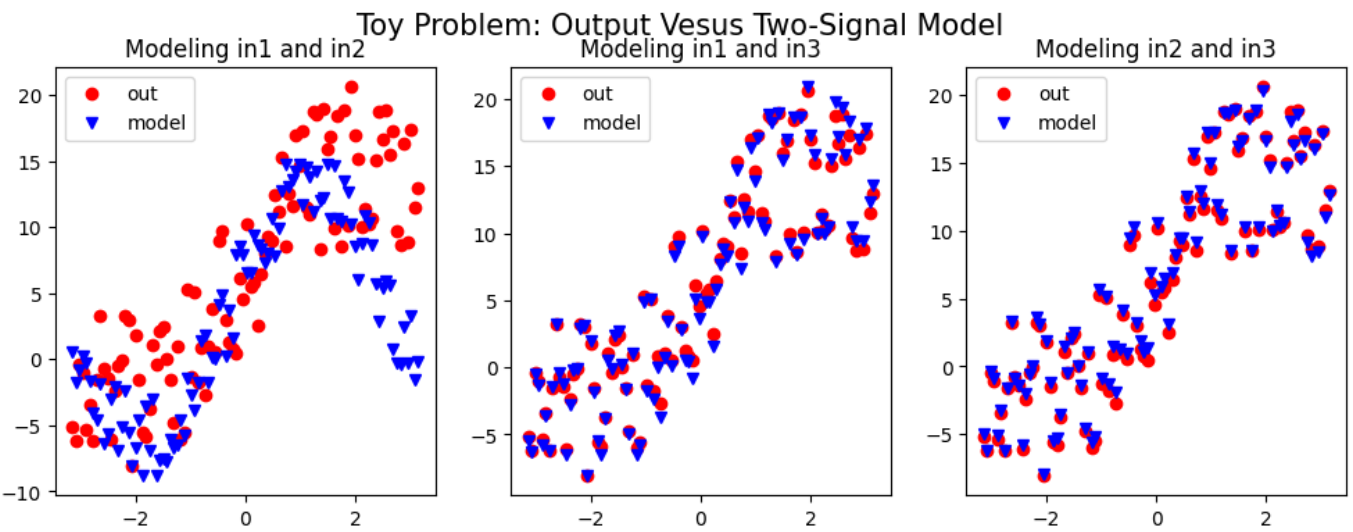
The `two_var_model` function in the next cell defines a linear model of two variables.

In [3]:
```
def two_var_model(in_tuple, a, b):
    ina, inb = in_tuple
    return a*ina + b*inb
```

Use the SciPy library's `curve_fit` function to try to predict the variable of interest from two of three features. The model with less-correlated features performs better.

In [4]:
```
from helpers.plots import plot_two_var_model
import itertools
from scipy.optimize import curve_fit

model = []
two_vars = []
for f0, f1 in itertools.combinations(['in1', 'in2', 'in3'], 2):
    two_vars.append((f0, f1))
    popt, pcov = curve_fit(two_var_model, (toy[f0].values, toy[f1].values), toy['out'].valu
    model.append(two_var_model((toy[f0].values, toy[f1].values), popt[0], popt[1]).reshape(
    print("Standard deviation for selection of features {} and {} is {:.4f}.".format(f0, f1
model_df = pd.DataFrame(np.hstack(model), columns=two_vars)

plot_two_var_model(model_df, toy)
```

```
Standard deviation for selection of features in1 and in2 is 3.2464.
Standard deviation for selection of features in1 and in3 is 0.0820.
Standard deviation for selection of features in2 and in3 is 0.0561.
```



The following section introduces a method for optimizing feature selection that can be useful in modeling complex systems.

Feature Selection by Mutual Information

There are various methods for feature selection. If you are building a machine-learning model, for example, and have six potential features, you might naively consider training it first on each of the features by itself, then on all 15 combinations of subsets of two features, then 20 combinations of subsets of three features, and so on. However, statistical methods are more efficient.

One statistical criterion that can guide this selection is mutual information (MI). The following subsections explain information and MI with some simple examples.

If you already understand MI and Shannon entropy, please skip ahead to section Solving Feature Selection on a Quantum Computer (and then from the menu bar, click the *Run All Above* option).

Quantifying Information: Shannon Entropy

Shannon entropy, $H(X)$, mathematically quantifies the information in a signal:

$$H(X) = -\sum_{x \in X} p(x) \log p(x)$$

where $p(x)$ represents the probability of an event's occurrence. The Shannon Entropy (SE) formula can be understood as weighing by an event's probability a value of $\log \frac{1}{p(x)}$ for the event, where the reciprocal is due to the minus sign. This value means that the less likely the occurrence of an event, the more information is attributed to it (intuitively, when a man bites a dog it's news).

To calculate SE, the `prob` function defined below calculates probability for a dataset representing some variables (a training set in a machine learning context) by dividing it into bins as a histogram using the NumPy library's `histogramdd` function.

In [5]:
```
def prob(dataset, max_bins=10):
    """Joint probability distribution P(X) for the given data."""

    # bin by the number of different values per feature
    num_rows, num_columns = dataset.shape
    bins = [min(len(np.unique(dataset[:, ci])), max_bins) for ci in range(num_columns)]

    freq, _ = np.histogramdd(dataset, bins)
```

```
    p = freq / np.sum(freq)
    return p

def shannon_entropy(p):
    """Shannon entropy H(X) is the sum of P(X)log(P(X)) for probabilty distribution P(X)."""
    p = p.flatten()
    return -sum(pi*np.log2(pi) for pi in p if pi)
```

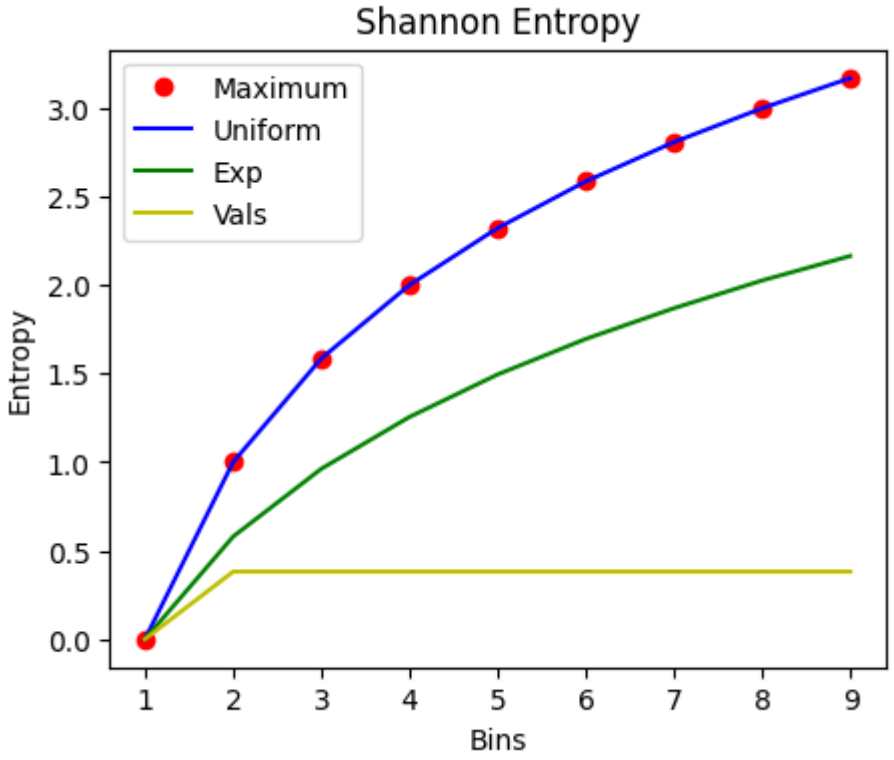
Illustration of Shannon Entropy

For an intuitive example of measuring SE, this subsection applies the `shannon_entropy` function to three signals with defined distributions:

- Uniform: this distribution maximizes values of SE because all outcomes are equally likely, meaning every outcome is equally unpredictable. $H(X) = log(N)$ for uniform distribution, where N is the number of possible outcomes, $x_1, x_2, \ldots x_N$.
- Exponential: the steeper the curve, the more outcomes are in the "tail" part (have higher probability) with lower information value.
- Binomial: the stronger this signal is biased to one outcome, the more predictable its values, the lower its information value. $H(X) = -p\log(p) - (1-p)\log(1-p)$ for binomial distribution; for $p = 0.1$, for example, $H(X) = 0.468$.

Define the three signals and plot the SE. The red dots show the maximal values of SE for different numbers of bits (Shannon developed the formula to calculate channel bandwidth, which for digital communications is measured in bits) or, as here, the bins into which the signals' possible values are divided.

In [6]:
```
from helpers.plots import plot_se

max_bins = 10

# Signals with defined distributions
x_uniform = np.random.uniform(0, 1, (1000, 1))
x_exp = np.exp(-np.linspace(0, 10, 1000)/2).reshape(1000, 1)
x_vals = np.random.choice([0, 1],(1000, 1), p=[0.1, 0.9])

data = list()
for bins in range(1, max_bins):
    uniform_se = shannon_entropy(prob(x_uniform, bins))
    exp_se = shannon_entropy(prob(x_exp, bins))
    vals_se = shannon_entropy(prob(x_vals, bins))
    data.append({'Bins': bins, 'Uniform': uniform_se, 'Maximum': np.log2(bins), 'Exp': exp_

plot_se(data)
```



Conditional Shannon Entropy

Conditional SE (CSE) measures the information in one signal, X, when the value of another signal, Y, is known:

$$\begin{aligned} H(X|Y) &= H(X,Y) - H(Y) \\ &= -\sum_{x \in X} p(x,y) \log p(x,y) - H(Y) \end{aligned}$$

where joint SE, $H(X,Y)$, measures the information in both signals together, with $p(x,y)$ being their joint probability. For example, knowing that it's winter reduces the information value of news that it is raining.

In [7]:
```
def conditional_shannon_entropy(p, *conditional_indices):
    """Shannon entropy of P(X) conditional on variable j"""

    axis = tuple(i for i in np.arange(len(p.shape)) if i not in conditional_indices)

    return shannon_entropy(p) - shannon_entropy(np.sum(p, axis=axis))
```

Illustration of CSE

Apply CSE to the toy problem. Because signals `in1` and `in2` are similar, knowing the value of one provides a good estimate of the other; in contrast, the value of signal `in3` is less good for estimating the first two.

In [8]:
```
print("H(in1) = {:.2f}".format(shannon_entropy(prob(toy[["in1"]].values))))
print("H(in1|in3) = {:.2f}".format(conditional_shannon_entropy(prob(toy[["in1", "in3"]].val
print("H(in1|in2) = {:.2f}".format(conditional_shannon_entropy(prob(toy[["in1", "in2"]].val
```

```
H(in1) = 3.16
H(in1|in3) = 2.21
H(in1|in2) = 1.19
```

Mutual Information

Mutual information between variables X and Y is defined as

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}$$

where $p(x)$ and $p(y)$ are marginal probabilities of X and Y, and $p(x,y)$ the joint probability. Equivalently,

$$I(X;Y) = H(Y) - H(Y|X)$$

where $H(Y)$ is the SE of Y and $H(Y|X)$ is the CSE of Y conditional on X.

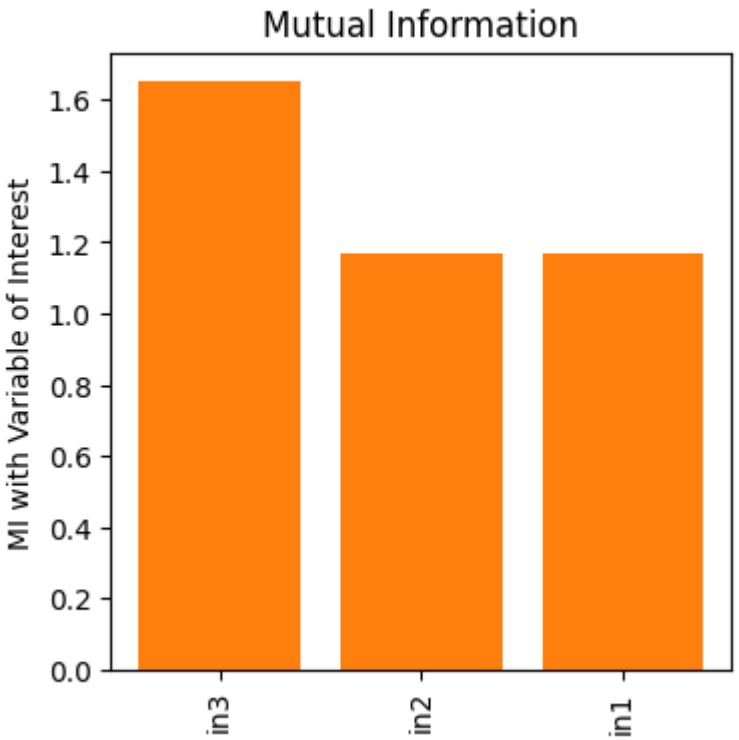
Mutual information (MI) quantifies how much one knows about one random variable from observations of another. Intuitively, a model based on just one of a pair of features (e.g., farmer MacDonald's water rations and soil humidity) will better reproduce their combined contribution when MI between them is high.

In [9]:
```
def mutual_information(p, j):
    """Mutual information between all variables and variable j"""
    return shannon_entropy(np.sum(p, axis=j)) - conditional_shannon_entropy(p, j)
```

Mutual Information on the Toy Problem

Calculate and plot MI between the output of the toy problem and its three input signals. This measures the suitability of each on its own as a feature in a model of the system, or how much each shapes the output.

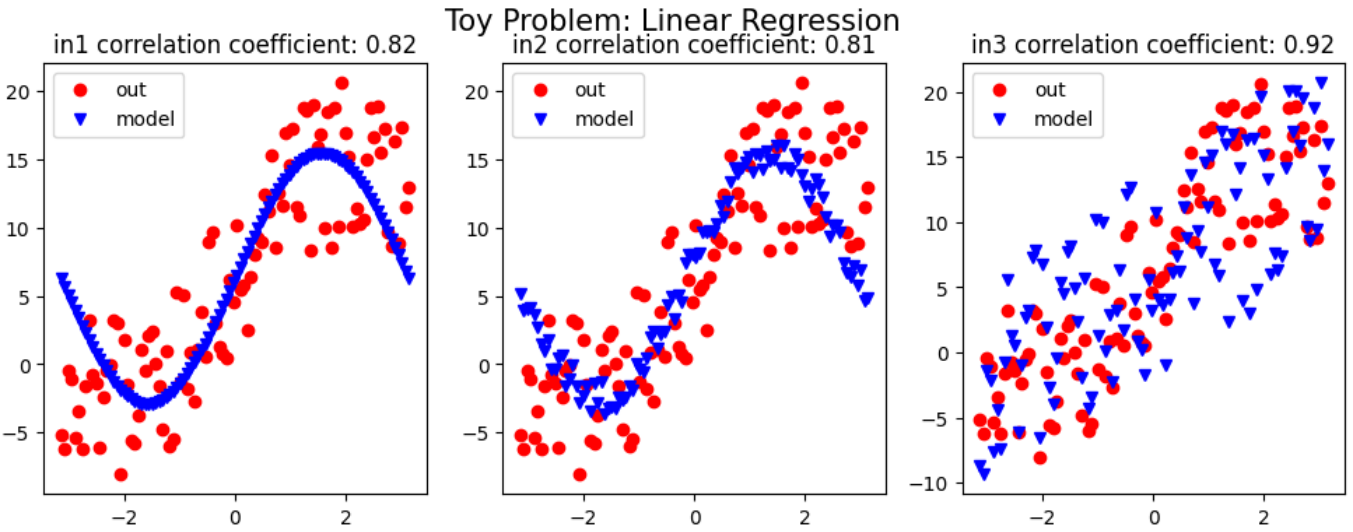
In [10]:
```
from helpers.plots import plot_mi

mi = {}
for column in toy.columns:
    if column == 'out':
        continue
    mi[column] = mutual_information(prob(toy[['out', column]].values), 1)

plot_mi(mi)
```



The plot of input and output signals in the first section might give an impression that the toy model's

output is closer to the two sine signals than to `in3`, but the linear regression below confirms the MI result.

In [11]:
```
from scipy.stats import linregress
from helpers.plots import plot_lingress

model = []
var_rval = []
for column in toy.columns:
    if column == 'out':
        continue
    slope, intercept, rvalue, pvalue, stderr = linregress(toy[column].values, toy['out'].va
    model.append((slope*toy[column].values + intercept).reshape(len(toy), 1))
    var_rval.append((column, rvalue))

plot_lingress(pd.DataFrame(np.hstack(model), columns=var_rval), toy)
```



The result should in fact be expected given an output defined as $out = 2 \times in_1 + 3 \times in_2 + 6 \times in_3$ with the sixfold multiplier on in_3 greater than the sum of multipliers on the other signals, all three of which have an amplitude of 1.

Conditional Mutual Information

Conditional mutual information (CMI) between a variable of interest, X, and a feature, Y, given the selection of another feature, Z, is given by

$$I(X;Y|Z) = H(X|Z) - H(X|Y,Z)$$

where $H(X|Z)$ is the CSE of X conditional on Z and $H(X|Y,Z)$ is the CSE of X conditional on both Y and Z.

In this code cell, because marginalization over j removes a dimension, any conditional indices pointing to subsequent dimensions are decremented by 1.

In [12]:
```
def conditional_mutual_information(p, j, *conditional_indices):
    """Mutual information between variables X and variable Y conditional on variable Z."""

    marginal_conditional_indices = [i-1 if i > j else i for i in conditional_indices]

    return (conditional_shannon_entropy(np.sum(p, axis=j), *marginal_conditional_indices)
            - conditional_shannon_entropy(p, j, *conditional_indices))
```

Apply `conditional_mutual_information` to the toy problem to find CMI between `out` and either `in2` or `in3`, conditional on `in1`.

In [13]:
```
print("I(out;in2|in1) = {:.2f}".format(conditional_mutual_information(prob(toy[['out', 'in2
print("I(out;in3|in1) = {:.2f}".format(conditional_mutual_information(prob(toy[['out', 'in3
```

```
I(out;in2|in1) = 0.59
I(out;in3|in1) = 1.72
```

Given the sine signal, `in1`, if you try to predict the output from one of the remaining signals, you find that the linear function, `in3` contributes more information than the noisy sine, `in2`.

Ideally, to select a model's k most relevant of n features, you could maximize $I(X_k; Y)$, the MI between a set of k features, X_k, and variable of interest, Y. This is a hard calculation because $\binom{n}{k}$ grows rapidly in real-world problems.

Solving Feature Selection on a Quantum Computer

There are different methods of approximating the hard calculation of optimally selecting $\binom{n}{k}$ features to maximize MI. The approach followed here assumes conditional independence of features and limits CMI calculations to permutations of three features. The optimal set of features, S, is then approximated by:

$$\arg\max_S \sum_{i \in S} \left\{ I(X_i; Y) + \sum_{j \in S, j \neq i} I(X_j; Y | X_i) \right\}$$

The left-hand component, $I(X_i; Y)$, represents MI between the variable of interest and a particular feature; maximizing selects features that best predict the variable of interest. The right-hand component, $I(X_j; Y | X_i)$, represents conditional MI between the variable of interest and a feature given the prior selection of another feature; maximizing selects features that complement information about the variable of interest rather than provide redundant information.

This approximation is still a hard calculation. The following subsection demonstrates a method for formulating it for solution on the D-Wave quantum computer. The method is based on the 2014 paper, Effective Global Approaches for Mutual Information Based Feature Selection, by Nguyen, Chan, Romano, and Bailey published in the Proceedings of the 20th ACM SIGKDD international conference on knowledge discovery and data mining.

MIQUBO: QUBO Representation of Feature Selection

D-Wave systems solve binary quadratic models (BQM)—the Ising model traditionally used in statistical mechanics and its computer-science equivalent, the quadratic unconstrained binary optimization (QUBO) problem. Given N variables $x_1, \ldots, x_N$, where each variable x_i can have binary values 0 or 1, the system finds assignments of values that minimize,

$$\sum_i^N q_i x_i + \sum_{i<j}^N q_{i,j} x_i x_j,$$

where q_i and $q_{i,j}$ are configurable (linear and quadratic) coefficients. To formulate a problem for the D-Wave system is to program q_i and $q_{i,j}$ so that assignments of $x_1, \ldots, x_N$ also represent solutions to the problem.

For feature selection, the Mutual Information QUBO (MIQUBO) method formulates a QUBO based on the approximation above for $I(X_k;Y)$, which can be submitted to the D-Wave quantum computer for solution.

The reduction of scope to permutations of three variables in this approximate formulation for MI-based optimal feature selection makes it a natural fit for reformulation as a QUBO:

	Optimization	Linear Terms	Quadratic Terms	Formula
Feature Selection	Maximize	$I(X_i;Y)$	$I(X_j;Y\|\|X_i)$	$\sum_{i\in S}\left\{I(X_i;Y)+\sum_{j\in S,j\neq i}I(X_j;Y\|\|X_i)\right\}$
QUBO	Minimize	$q_i x_i$	$q_{i,j}x_i x_j$	$\sum_i^N q_i x_i+\sum_{i<j}^N q_{i,j}x_i x_j$

You can represent each choice of $\binom{n}{k}$ features as the value of solution $x_1,\ldots,x_N$ by encoding $x_i=1$ if feature X_i should be selected and $x_i=0$ if not. With solutions encoded this way, you can represent the QUBO in matrix format, $\mathbf{x}^T\mathbf{Q}\mathbf{x}$, where $\mathbf{Q}$ is an n x n matrix and $\mathbf{x}$ is an n x 1 matrix (a vector) that should have k ones representing the selected features.

To map the feature-selection formula to a QUBO, set the elements of $\mathbf{Q}$ such that

- diagonal elements (linear coefficients) represent MI: $Q_{ii}\leftarrow -I(X_i;Y)$
- non-diagonal elements (quadratic elements) represent CMI: $Q_{ij}\leftarrow -I(X_j;Y|X_i)$

These QUBO terms are negative because the quantum computer seeks to minimize the programmed problem while the feature-selection formula maximizes. The following subsection codes this and then completes the formulation by adding the $\binom{n}{k}$ constraint to the QUBO.

MIQUBO on the Toy Problem

This subsection applies the MIQUBO formulation to the toy problem by configuring the QUBO in three parts: (1) linear biases that maximize MI between the variable of interest and each feature (2) quadratic biases that maximize CMI between the variable of interest and each feature, given the prior choice of another feature (3) selection of just k features.

Create a BQM and set the linear coefficients as the MI between `out` and each potential feature.

In [14]:
```
import dimod

bqm = dimod.BinaryQuadraticModel.empty(dimod.BINARY)

for column in toy.columns:

    if column == 'out':
        continue

    mi = mutual_information(prob(toy[['out', column]].values), 1)
    bqm.add_variable(column, -mi)

for item in bqm.linear.items():
    print("{}: {:.3f}".format(item[0], item[1]))
```

```
in1: -1.168
in2: -1.171
in3: -1.650
```

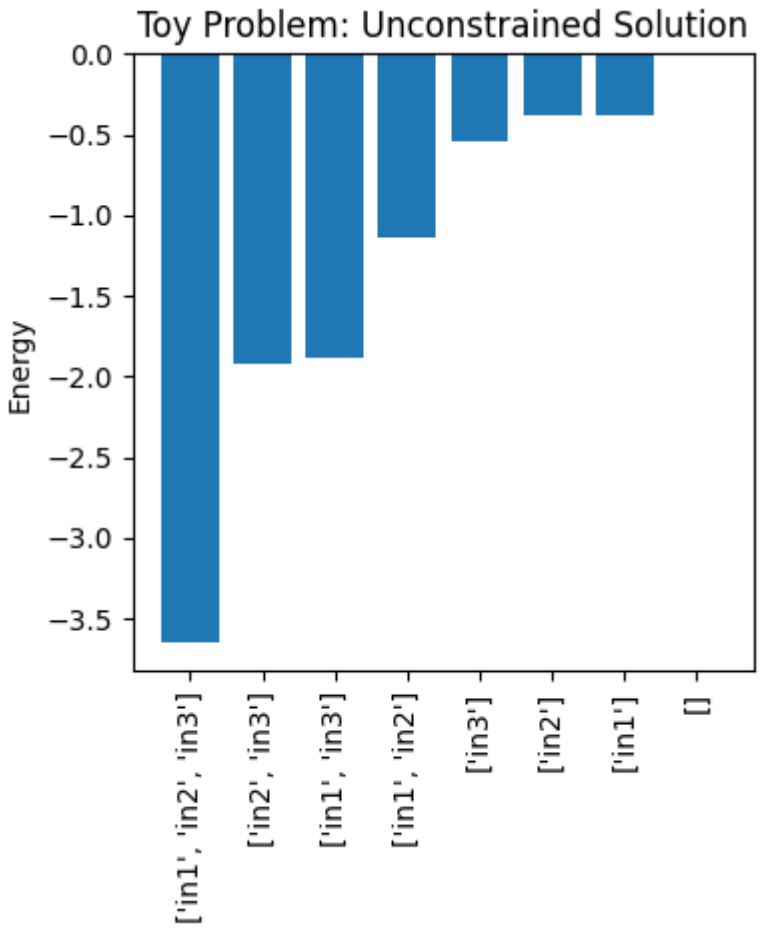
Set the quadratic coefficients as the MI between `out` and each potential feature conditional on the other features.

In [15]:
```
for f0, f1 in itertools.combinations(['in1', 'in2', 'in3'], 2):
    cmi_01 = conditional_mutual_information(prob(toy[['out', f0, f1]].values), 1, 2)
    cmi_10 = conditional_mutual_information(prob(toy[['out', f1, f0]].values), 1, 2)
    bqm.add_interaction(f0, f1, -cmi_01)
    bqm.add_interaction(f1, f0, -cmi_10)

bqm.normalize()     # scale the BQM to (-1, 1) biases

for item in bqm.quadratic.items():
    print("{}: {:.3f}".format(item[0], item[1]))
```

```
('in2', 'in1'): -0.383
('in3', 'in1'): -0.961
('in3', 'in2'): -1.000
```

Use Ocean software's dimod ExactSolver() to find an exact solution of the BQM, which currently represents a minimization of MI and CMI, and plot the results.

In [16]:
```
from helpers.plots import plot_solutions

sampler = dimod.ExactSolver()

result = sampler.sample(bqm)

plot_solutions(result)
```



Unsurprisingly, the best solution (lowest-energy solution for the minimized QUBO) employs all three input signals because the current QUBO mapping does not constrain the number of selected features.

In those solutions where only two are selected, models that select `in3` are better than the one that selects just `in1` and `in2`.

Penalizing Non-k Selections

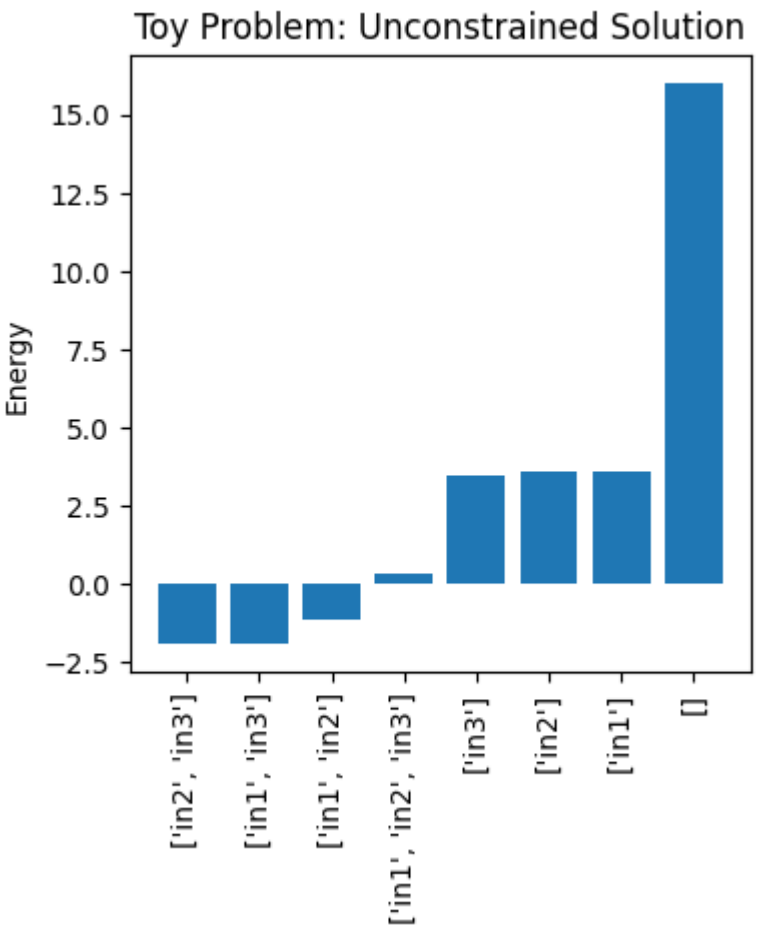
How do you program on the quantum computer a constraint that exactly k features be selected? By penalizing solutions that select greater or fewer than k features. If you add

$$P = \alpha(\sum_{i=1}^{n} x_i - k)^2$$

to the QUBO, where penalty P is positive whenever the number of 1s in solution $x_1, \ldots, x_N$ is not k, a large enough α can ensure that such solutions are no longer minima of the problem.

Set set a constraint that $k = 2$ with a penalty amplitude of 4 (you can rerun this cell with varying values of `strength` to see the penalty range from ineffective to overshadowing the problem) and plot the solution.

In [17]:
```
k = 2
bqm.update(dimod.generators.combinations(bqm.variables, k, strength=4))

result = sampler.sample(bqm)

plot_solutions(result)
```



Example Application: Predicting Survival of Titanic Passengers

This example illustrates the MIQUBO method by finding an optimal feature set for predicting survival of Titanic passengers. It uses records provided in file `formatted_titanic.csv`, which is a feature-

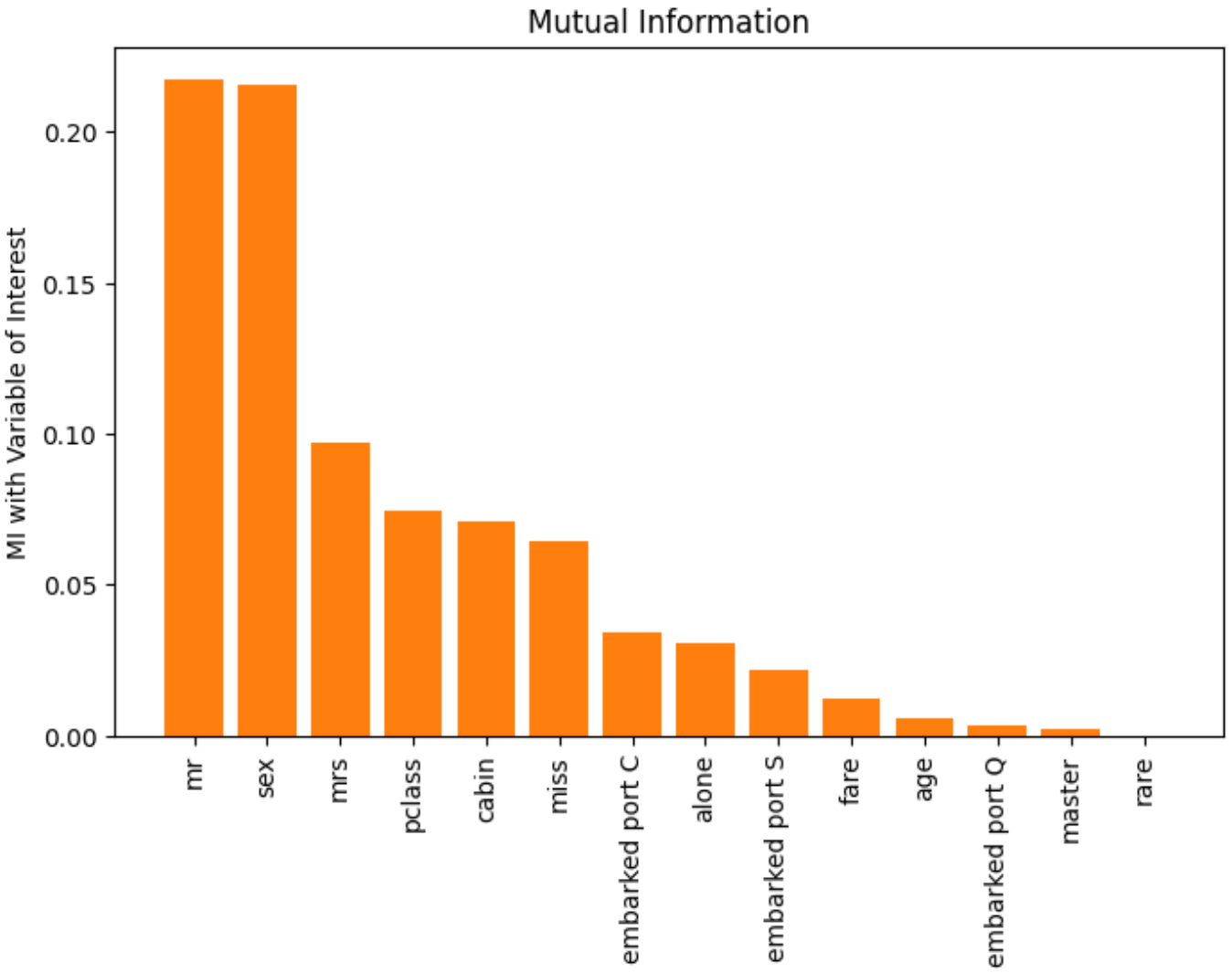
engineered version of a public database of passenger information recorded by the ship's crew. In addition to a column showing survival for each passenger, its columns record gender, title, class, port of embarkation, etc.

The next cell reads the file into a pandas DataFrame.

In [18]:
```
titanic = pd.read_csv("data/formatted_titanic.csv") # To see the data folder's contents, se
```

Plot a ranking of MI between each feature and the variable of interest (survival).

In [19]:
```
mi = {}
features = list(set(titanic.columns).difference(('survived',)))

for feature in features:
    mi[feature] = mutual_information(prob(titanic[['survived', feature]].values), 1)

plot_mi(mi)
```



Exact Versus Good Solutions: a Note on this Dataset

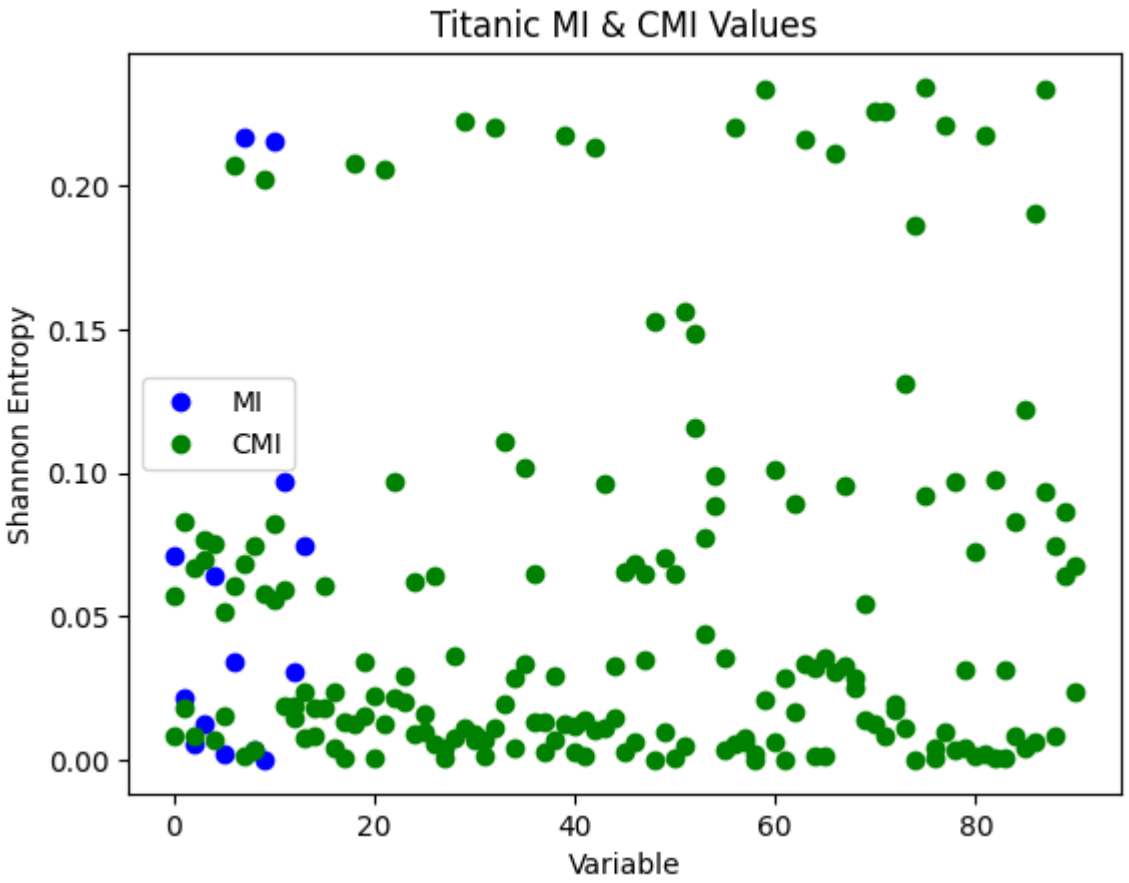
This example demonstrates a technique for solving a type of optimization problem on a quantum computer. The Titanic dataset provides a familiar, intuitive example available in the public domain. In itself, however, it is not a good fit for solving by sampling.

The next cell plots the values of MI and CMI for all features and three-variable permutations. Notice the clustering of values in tight ranges of values.

In [20]:
```
plt.plot(range(len(features)), [mutual_information(prob(titanic[['survived', feature]].valu
```

```
plt.plot(range(len([x for x in itertools.combinations(features, 2)])), [conditional_mutual_
plt.plot(range(len([x for x in itertools.combinations(features, 2)])), [conditional_mutual_

plt.title("Titanic MI & CMI Values")
plt.ylabel("Shannon Entropy")
plt.xlabel("Variable")
plt.legend(["MI", "CMI"])
```

Out[20]: <matplotlib.legend.Legend at 0x7572c5920350>



The plot below, obtained by exploiting the problem's small size and brute-force solving for all possible values, shows the solution space for a couple of choices of k. The left side shows the resulting energy for all possible assignments of values to $x_1 \dots x_N$ (yellow) and those that satisfy the requirement of $\binom{n}{k}$ (blue); the right side focuses on only those that satisfy $\binom{n}{k}$ and highlights the optimal solution (red).

Notice the high number of valid solutions that form a small cluster (the energy difference between the five best solutions in the depicted graph is in the fourth decimal place). The quantum computer's strength is in quickly finding diverse good solutions to hard problems, it is not best employed as a double-precision numerical calculator. Run naively on this dataset, it finds numerous good solutions but is unlikely to find the exact optimal solution.

There are many techniques for reformulating problems for the D-Wave system that can improve performance on various metrics, some of which can help narrow down good solutions to closer approach an optimal solution. These are out of scope for this example. For more information, see Leap's other Jupyter Notebooks, the D-Wave Problem-Solving Handbook, and examples in the Ocean software documentation.

The remainder of this section solves the problem for just the highest-scoring features.

Building the MI-Based BQM

Select 8 features with the top MI ranking found above.

In [21]:
```
keep = 8

sorted_mi = sorted(mi.items(), key=lambda pair: pair[1], reverse=True)
```

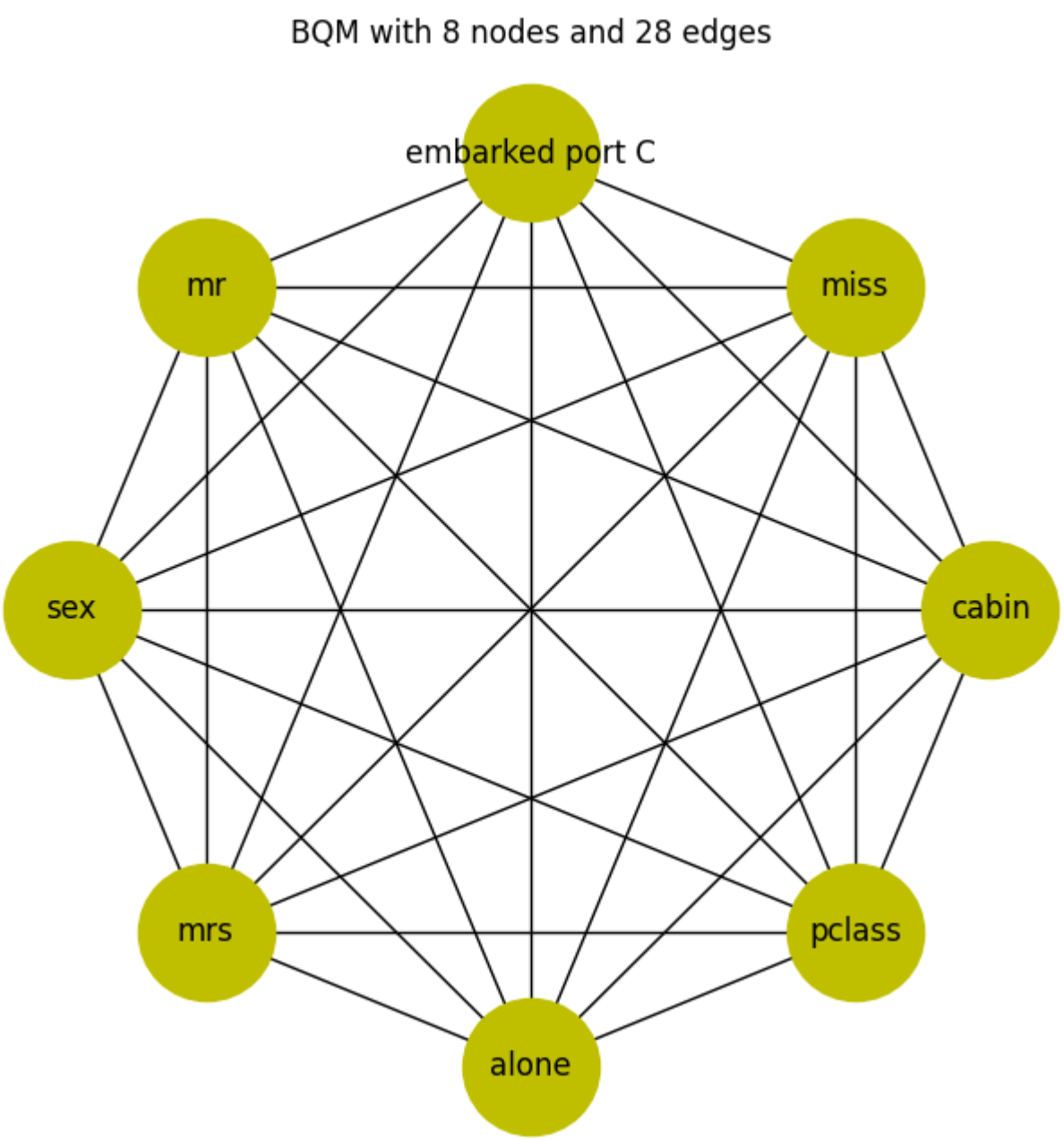
```
titanic = titanic[[column[0] for column in sorted_mi[0:keep]] + ["survived"]]
features = list(set(titanic.columns).difference(('survived',)))

print("Submitting for {} features: {}".format(keep, features))
```

```
Submitting for 8 features: ['cabin', 'miss', 'embarked port C', 'mr', 'sex', 'mrs', 'alone',
'pclass']
```

Calculate a BQM based on the problem's MI and CMI as done previously for the toy problem.

In [22]:

```
from helpers.draw import plot_bqm

bqm = dimod.BinaryQuadraticModel.empty(dimod.BINARY)

# add the features
for feature in features:
    mi = mutual_information(prob(titanic[['survived', feature]].values), 1)
    bqm.add_variable(feature, -mi)

for f0, f1 in itertools.combinations(features, 2):
    cmi_01 = conditional_mutual_information(prob(titanic[['survived', f0, f1]].values), 1,
    cmi_10 = conditional_mutual_information(prob(titanic[['survived', f1, f0]].values), 1,
    bqm.add_interaction(f0, f1, -cmi_01)
    bqm.add_interaction(f1, f0, -cmi_10)

bqm.normalize()

plot_bqm(bqm)
```



Setting Up a QPU as a Solver

Set up a D-Wave system as your solver in the standard way described in the Ocean documentation's Configuring Access to D-Wave Solvers.

minor-embedding, the mapping between the problem's variables to the D-Wave QPU's numerically indexed qubits, can be handled in a variety of ways and this affects solution quality and performance. Ocean software provides tools suited for different types of problems; for example, dwave-system EmbeddingComposite() has a heuristic for automatic embedding. This example uses FixedEmbeddingComposite() with the embedding found using an algorithm tuned for cliques (complete graphs).

The code below creates a *dwave-networkx* graph that represents the *working graph* of the QPU selected by DWaveSampler, such as a Pegasus graph with the same sets of nodes (qubits) and edges (couplers) as the QPU. Ocean software's *minorminer* finds an embedding for the required clique size in the working graph.

In [23]:
```
from dwave.system import DWaveSampler, FixedEmbeddingComposite
from minorminer.busclique import find_clique_embedding

qpu = DWaveSampler()
```

```
qpu_working_graph = qpu.to_networkx_graph()
embedding = find_clique_embedding(bqm.variables, qpu_working_graph)

qpu_sampler = FixedEmbeddingComposite(qpu, embedding)

print("Maximum chain length for minor embedding is {}.".format(max(len(x) for x in embeddin
```

```
Maximum chain length for minor embedding is 2.
```

This problem is small enough to be solved in its entirety on an Advantage QPU. For datasets with higher numbers of features, D-Wave Ocean's dwave-hybrid tool can be used to break the BQM into smaller pieces for serial submission to the QPU and/or parallel solution on classical resources. Here, an out-of-the-box hybrid sampler, Kerberos is used.

In [24]:
```
from hybrid.reference.kerberos import KerberosSampler

kerberos_sampler = KerberosSampler()
```

Submit the Problem for All k Values

For all numbers of selected features, k, set a $\binom{n}{k}$ penalty, submit an updated BQM for solution, and at the end plot the selected features.

In [25]:
```
from helpers.draw import plot_feature_selection

selected_features = np.zeros((len(features), len(features)))
for k in range(1, len(features) + 1):
    print("Submitting for k={}".format(k))
    kbqm = dimod.generators.combinations(features, k, strength=6)
    kbqm.update(bqm)
    kbqm.normalize()

    best = kerberos_sampler.sample(kbqm,
                                   qpu_sampler=qpu_sampler,
                                   qpu_reads=5000,
                                   max_iter=1,
                                   qpu_params={'label': 'Notebook - Feature Selection'}
                                  ).first.sample

    for fi, f in enumerate(features):
        selected_features[k-1, fi] = best[f]

plot_feature_selection(features, selected_features)
```

```
Submitting for k=1
Submitting for k=2
Submitting for k=3
Submitting for k=4
Submitting for k=5
Submitting for k=6
Submitting for k=7
Submitting for k=8
```