NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Applications of the Theory of Quantitative Information Flow

Michail K. Vargiamis

Supervisor: Kostas Chatzikokolakis, Associate Professor

ATHENS

JULY 2021

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Εφαρμογές της Θεωρίας του Quantitative Information Flow

Μιχαήλ Κ. Βαργιάμης

Επιβλέπων: **Κωνσταντίνος Χατζηκοκολάκης,** Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2021

BSc THESIS

Applications of the Theory of Quantitative Information Flow

Michail K. Vargiamis
S.N.: 1115201300018

SUPERVISOR: **Kostas Chatzikokolakis,** Associate Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Εφαρμογές της Θεωρίας του Quantitative Information Flow

Μιχαήλ Κ. Βαργιάμης
Α.Μ.: 1115201300018

ΕΠΙΒΛΕΠΩΝ: **Κωνσταντίνος Χατζηκοκολάκης,** Αναπληρωτής Καθηγητής

ABSTRACT

In this thesis we explore different applications of the theory of Quantitative Information Flow (QIF). We present concepts and definitions having in mind readers somewhat familiar with the subject or currently learning about it. The original form of this thesis is that of Jupyter Notebooks and the goal was to create a more interactive and approachable way of studying the theory of QIF. The Jupyter Notebooks can be found at https://github.com/damik3/qif-notebooks.

SUBJECT AREA: Security and Privacy

KEYWORDS: security, privacy, quantitative information flow

ΠΕΡΙΛΗΨΗ

Σε αυτήν την εργασία εξερευνούμε διάφορες εφαρμογές της θεωρίας του Quantitative Information Flow (QIF). Παρουσιάζουμε έννοιες και ορισμούς έχοντας κατά νού αναγνώστες σχετικά οικείους με το αντικείμενο ή που επί του παρόντος μαθαίνουν για αυτό. Η αρχική μορφή αυτής της εργασίας ήταν σε Jupyter Notebooks με σκοπό να δημιουργήσουμε ένα πιο διαδραστικό και εφικτό τρόπο μελέτης και εξοικίωσης με την θεωρία του QIF. Τα Jupyter Notebooks υπάρχουν διαθέσιμα στο https://github.com/damik3/qif-notebooks.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ασφάλεια και Ιδιωτικότητα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ασφάλεια, ιδιωτικότητα, quantitative information flow

To my family

ACKNOWLEDGEMENTS

I would like to deeply thank my supervisor, Prof. Kostas Chatzikokolakis for giving me the chance to work on this very interesting topic for my undergraduate thesis and for being an amazing professor and mentor.

CONTENTS

1. INTRODUCTION **14**

2. THEORY **16**

2.1 Secrets And Vulnerability . . . 16
2.1.1 Probability distribution of X . . . 16
2.1.2 Bayes vulnerability . . . 17
2.1.3 Guessing entropy . . . 17
2.1.4 Shannon entropy . . . 17
2.1.5 So how vulnerable is Evil-Eye Henry's secret? . . . 18

2.2 g-vulnerability . . . 19
2.2.1 Defining g . . . 19
2.2.2 Calculating g-vulnerability . . . 20
2.2.3 Setting a threshold . . . 20

2.3 Channels and Posterior Vulnerability . . . 22
2.3.1 Channel matrix . . . 22
2.3.2 Prior distribution . . . 23
2.3.3 Joint Matrix . . . 23
2.3.4 Posterior Distributions . . . 24
2.3.5 Finally solving the problem . . . 24

2.4 Channels and Posterior Vulnerability (part 2) . . . 27
2.4.1 What if the warden uses a biased coin to answer the question? . . . 27
2.4.2 Biased coin with $p = \frac{2}{3}$ . . . 28
2.4.3 Biased coin with $p = \frac{3}{4}$ . . . 29

2.5 Channels and Posterior Vulnerability (part 3) . . . 34
2.5.1 What if the pardoned prisoner is not uniformly chosen at random? . . . 34
2.5.2 Prior vunlerability . . . 35
2.5.3 Posterior vunlerability . . . 36
2.5.4 Multiplicative leakage . . . 40
2.5.5 Generalizing over any prior distribution . . . 41

2.6 Refinement . . . 42
2.6.1 Producing the actual location with probability $p = 0.7$ . . . 42
2.6.2 Producing the actual location with probability $p = 0.6$ . . . 43
2.6.3 Comparing C_1 and C_2 . . . 44
2.6.4 Different p for some rows . . . 46

3. CASE STUDIES **50**

3.1 Voting systems . . . 50

3.1.1 Modeling elections as channels 50
3.1.2 Computing the vulnerability of W 52
3.1.3 Computing the vulnerability of C 53
3.1.4 Comparing the two channels 54
3.1.5 Different Adversarial Models 56
3.1.5.1 Adversary 1 56
3.1.5.2 Adversary 2 60

3.2 Differential Privacy 63
3.2.1 An example scenario 63
3.2.2 Assesing information leakage through *QIF* 65
3.2.3 Assesing information leakage through *Differential Privacy* 67
3.2.4 Comparing the two approaches 67

4. CONCLUSIONS 70

ABBREVIATIONS - ACRONYMS 71

REFERENCES 72

LIST OF FIGURES

2.1 Posterior vulnerability on uniform prior 30
2.2 Prior vulnerability with original W 35
2.3 Posterior vulnerability with original W 37
2.4 Multiplicative leakage with original W 40
2.5 Posterior Bayes Vulnerability 45
2.6 Posterior Bayes Vulnerability 47

3.1 Posterior Bayes Vulnerability of W and C 55
3.2 Multiplicative Bayes Leakage of W and C 56
3.3 Multiplicative Leakage of W 59
3.4 Multiplicative Leakage of C 59
3.5 Multiplicative Leakages of W and C 61
3.6 Leakage and utility for oblivious mechanisms 65

LIST OF TABLES

1.1 Thesis sections and book chapters matching 15

2.1 g function defined with a matrix 19
2.2 C_1 matrix 42
2.3 C_2 matrix 43
2.4 C_3 matrix 46

3.1 W matrix 51
3.2 C matrix 51
3.3 g_1 function defined with a matrix 57
3.4 f function defined with a matrix 63
3.5 H matrix 64
3.6 C matrix 64

PREFACE

The essential part of this work was developed in Athens, Greece between October 2020 and March 2021. The initial challenge was to study and get familiar with the different concepts of the theory of Quantitative Information Flow (QIF). After that, I had to think of appealing ways for presenting those concepts, write the necessary code and finally create the notebooks.

1. INTRODUCTION

Information Flow is the transfer of information from a source (who already knows the information), to a target (who does not know it yet). In the field of Quantitative Information Flow (QIF) we are working in the same context but we are mainly interested in information leakage and how it can be measured. It is important to notice that through this quantitative approach, we can mark some leaks more serious than others and thus less tolerable. We can also model different adversaries that might want to leak parts of the information but not all of it. And that makes this approach more flexible and more applicable to a wider range of real life scenarios.

The main sections of this work can be split in two parts; theory and case studies. Each section of the first part presents a basic concept of *QIF* using an example scenario, while the second part examines a couple of additional cases using almost all of the concepts presented in the first part. As also mentioned in the abstract, our goal was to provide additional material for individuals who wish to learn more about *QIF* through the interactive nature of Jupyter Noteboooks. At any moment the reader can pause and experiment with different code blocks and observe the results.

In section 2.1 we present the concept of a secret and an adversary who is trying to find out about that secret and how we can measure the secret's vulnerability. In the next section, we give ways in which we can model different types of adversaries that might want to learn different things about the secret. Continuing on with section 2.3 section 2.4 and section 2.5, we introduce the essential concept of channels and posterior vulnerability, which basically gives a measure of leakage for the secret after observing the channel's output. Closing the theory part, we present the notion of refinement, which provides a very useful way for comparing two channels and their leakage.

In chapter 3 we put in use many concepts introduced in the previous part in order to study and analyze elections and voting systems from the point of view of *QIF*. And last but not least, we briefly touch the area of Differential Privacy - a somewhat different and more targeted approach to information leakage - and compare its differences and similarities to *QIF*.

Note that many *QIF* measures are being computed using the `libqif` library available at https://github.com/chatziko/libqif.

This thesis is mainly based on the book *The Science of Quantitative Information Flow* [1] and each section of this work can be matched to a chapter of [1] as below:

Table 1.1: Thesis sections and book chapters matching

Thesis section	Book Chapter
section 2.1	Chapter 2
section 2.2	Chapter 3
section 2.3, section 2.4, section 2.5	Chapter 4, Chapter 5
section 2.6	Chapter 9
section 3.1	Chapter 22
section 3.2	Chapter 23

2. THEORY

Starting off, we are going to use an example with pirates and hidden treasures to present the concept of secrets and vulnerability.

2.1 Secrets And Vulnerability

Evil-Eye Henry was a pirate back in the day. He was not a famous one but he sure accumulated a lot of wealth. He died unexpectedly and his treasure was never found. It was buried in a **secret** spot that he never told anoyone about.

Recently you found an old book mentioning his name. One of his sailors mentioned the following:

> ***Arrrgh!***
>
> *I am 100% sure that Evil-Eye Henry has buried his treasure in one of the following locations:*
>
> 1. *Black Sand Haven*
> 2. *Dead Man's Isle*
> 3. *Isle Of Mermaids*
> 4. *Kraken Reef*
> 5. *Monkey Bay*
> 6. *Old Salt Cavern*
>
> *I am also sure that the probability of the treasure being buried in locations 1 or 2 is the same as the probability of it being buried in locations 3, 4, 5, or 6.*
>
> ***Arrrgh!***

Before finding the book you had no idea where the treasure might be. But now you know something more. How vunerable has Evil-Eye Henry's **secret** become?

2.1.1 Probability distribution of X

Let's call our secret X. It matches the X pirates used to mark treasures on their maps. The possible values for X are $\{1, 2, 3, 4, 5, 6\}$. Each number corresponds to a location in the order that they appear above. The probability distribution of X is given by

$$\pi = \left(\frac{1}{4}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}\right)$$

Why is that? From what the sailor said we can deduce that X is either in location 1 or 2 with probability 50% (uniformly) or in locations 3, 4, 5 or 6 with probability 50% (uniformly again). So we could deduce that $p_X(1) = \frac{1}{4}$ and $p_X(2) = \frac{1}{4}$ because they add up to $\frac{1}{2} = 50\%$. And for the remaining locations uniformly distribute the other $\frac{1}{2}$.

[2]:
```
pi = [1/4, 1/4, 1/8, 1/8, 1/8, 1/8]
print(pi)
```

```
[0.25, 0.25, 0.125, 0.125, 0.125, 0.125]
```

2.1.2 Bayes vulnerability

One way to measure the vulnerability of a probability distribution is Bayes vulnerability. It is basically the answer to the following question:

> What is the probability of guessing X correctly in one try?

For our scenario if we had only **one try** to go and dig up a place in search of the treasure we would naturally pick the one with the highest probability. Which would be either 1 or 2. Either way we would succeed with probability $\frac{1}{4}$.

In our case, Bayes vulnerability is equal to:

[3]:
```
print("Bayes vulnerability:", measure.bayes_vuln.prior(pi))
```

```
Bayes vulnerability: 0.25
```

2.1.3 Guessing entropy

Imagine now that you decide to go and search all of the locations. Eventually you would find the true value of X. But a question naturally arises. Which locations should you visit first?

The logical way of doing it would be to visit the ones with the highest probability first. So there is a higher probability of finding the treasure sooner.

Guessing entropy is the average number of locations we would need to search in order to find the true value of X.

In our case, guessing entropy is equal to:

[4]:
```
print("Guessing entropy:", measure.guessing.prior(pi))
```

```
Guessing entropy: 3.0
```

2.1.4 Shannon entropy

Another way to measure the vulnerability of a probability distribution is Shannon entropy. To better understand it imagine having in front of you someone who knows the true value of X and you ask them questions according to the following train of thought:

> *$X \in \{1, 2\}$ with probability $\frac{1}{2}$ and $X \in \{3, 4, 5, 6\}$ with probability $\frac{1}{2}$. So I ask:*
>
> *Does X belong to $\{1, 2\}$?*

If the answer is yes then $X \in \{1, 2\}$.

$X \in \{1\}$ with probability $\frac{1}{4}$ and $X \in \{2\}$ with probability $\frac{1}{4}$. So I ask:

Does X belong to $\{1\}$?

If the answer is yes then I have found the treasure! It is buried in location 1 and it took me 2 questions to reach to this conclusion! [This scenario happens with probability $p_X(1)$]

If the answer is no then $X \in \{2\}$ and again, I have found the treasure! It is buried in location 2 and it took me 2 questions to reach to this conclusion! [This scenario happens with probability $p_X(2)$]

If the answer is no then $X \in \{3, 4, 5, 6\}$.

$X \in \{3, 4\}$ with probability $\frac{1}{4}$ and $X \in \{5, 6\}$ with probability $\frac{1}{4}$. So I ask:

Does X belong to $\{3, 4\}$?

If the answer is yes then bla bla bla…

Shannon's entropy is basically the average number of questions needed to completely reveal the secret. In general, questions are in the form of *"Does x belong to S?"*.

In our case, Shannon's entropy is equal to:

[5]:
```
print("Shannon entropy:", measure.shannon.prior(pi))
```

```
Shannon entropy: 2.5
```

2.1.5 So how vulnerable is Evil-Eye Henry's secret?

The answer is, well, it depends!

If the person seeking to reveal the secret takes only one guess based on the secret's probability distribution, then Bayes Vulnerability is the measure of vulnerability we should be using.

If the person seeking to reveal the secret tries all possible values of X in decreasing probability order, then Guessing Entropy is the right measure of vulnerability for this case.

And if they can ask questions in the form of *"Does x belong to S?"*, then Shannon Entropy is the right choice.

In general, we need to have a general idea of the adversary who wants to reveal our secret. What is their goal and what methods they can use.

The next step is to find a way for being able to take into account different adversaries with different goals, while assessing the vulnerability of the secret. We are going do that through g-vulnerability.

2.2 g-vulnerability

2.2.1 Defining g

In a more practical, real life scenario, we would have to consider how much we would get from finding the treasure and how much it would costs us searching in each location. When we take a guess and it is the right one, we get rewarded. In our example the reward is the monetary value of the treasure. But if we take a guess and it's the wrong one, then we lose the money we spent traveling back and forth and digging up the place. And that can be expressed with a negative number.

For our example let's say that the treasure is worth $\$1500$. But seraching at each location has a different cost. The matrix below represents that idea.

Table 2.1: g function defined with a matrix

g	True$X=1$	True$X=2$	True$X=3$	True$X=4$	True$X=5$	True$X=6$
Guess $X=1$	$1100	$-\$400$	$-\$400$	$-\$400$	$-\$400$	$-\$400$
Guess $X=2$	$-\$800$	$700	$-\$800$	$-\$800$	$-\$800$	$-\$800$
Guess $X=3$	$-\$100$	$-\$100$	$1400	$-\$100$	$-\$100$	$-\$100$
Guess $X=4$	$-\$200$	$-\$200$	$-\$200$	$1300	$-\$200$	$-\$200$
Guess $X=5$	$-\$300$	$-\$300$	$-\$300$	$-\$300$	$1200	$-\$300$
Guess $X=6$	$-\$400$	$-\$400$	$-\$400$	$-\$400$	$-\$400$	$1100

g's first line corresponds to choosing to dig up location 1.

- $g(1,1)$ means we choose to search location 1 and the treasure is indeed there. So we get our prize of $ $ 1500$ minus the digging expenses for location 1, which are equal to $\$400$. So in total we get $\$1100$.
- $g(1,2)$ means we choose 1 and the treasure is in 2. But we spent $\$400$ for digging.
- $g(1,3)$ means we choose 1 and the treasure is in 3. Again, we spent $\$400$ because we still chose to dig in location 1.
- …

The same logic applies to the rest of g.

What we have just done is define a gain function g. By definition, $g(w,x)$ specifies the gain that the adversary achieves by taking action w when the value of the secret is x. In this example the role of the adversary is basically us trying to guess the secret location X.

[3]:
```
g = np.array([
    [1100, -400, -400, -400, -400, -400],
    [-800, 700, -800, -800, -800, -800],
    [-100, -100, 1400, -100, -100, -100],
    [-200, -200, -200, 1300, -200, -200],
    [-300, -300, -300, -300, 1200, -300],
```

```
    [-400, -400, -400, -400, -400, 1100],
])
```

2.2.2 Calculating g-vulnerability

In general we cannot be sure about what we gain from each action we take. That dependes on the true value of X. But we can make an estimate based on the probability distribution of X by computing the average gain we obtain from each action.

[2]:
```
pi = [1/4, 1/4, 1/8, 1/8, 1/8, 1/8]
```

[4]:
```
exp_gain = np.matmul(g, np.transpose(pi))
for i in range(len(exp_gain)):
    print("Average gain when choosing location %d: $%.2f" % (i+1,␣
 ↪exp_gain[i]))
```

```
Average gain when choosing location 1: $-25.00
Average gain when choosing location 2: $-425.00
Average gain when choosing location 3: $87.50
Average gain when choosing location 4: $-12.50
Average gain when choosing location 5: $-112.50
Average gain when choosing location 6: $-212.50
```

And now what would our best choice be? The one with the highest average winnings of course!

[5]:
```
print("Best choice: Location", np.argmax(exp_gain)+1)
print("Expected winnings: $", max(exp_gain), sep='')
```

```
Best choice: Location 3
Expected winnings: $87.5
```

So the distribution's g vulnerability is equal to 87.5. And it achieved for guessing $X = 3$.

Notice that location 1 has a higher probability of being the true value of X but it costs us more if we are wrong. On the other hand, location 3 has a lower probability of being the true value of X but it costs us less if we are wrong. But that lower probability balances out with the smaller cost for when being wrong. And on average, it makes for a better choice than location 1.

Also, without considering g, our best choice would have been to guess $X = 1$. But given the information g provides us, $X = 3$ is a better guess.

2.2.3 Setting a threshold

Someone could argue that the average winnings, whichever location we chose, might not be good enough for us to take action. For almost all but one location,

the average gain is negative, meaning in total we lose money. And in the one case where the gain is positive, it is just not worth it.

So we might want to set a threshold for our gain. Meaning that, if the average gain of an action is lower than the threshold, we never choose that action.

For our case someone could say that to choose an action, it should give an average of at least $300 in order for it to be worth it.

In order to achieve that we can add an additional row to g like this:

[6]:
```
g = np.array([
    [1100, -400, -400, -400, -400, -400],
    [-800, 700, -800, -800, -800, -800],
    [-100, -100, 1400, -100, -100, -100],
    [-200, -200, -200, 1300, -200, -200],
    [-300, -300, -300, -300, 1200, -300],
    [-400, -400, -400, -400, -400, 1100],
    [300, 300, 300, 300, 300, 300],
])
```

Now, watch what happens when we compute the average gain for each action and then pick the action with the highest gain.

[7]:
```
exp_gain = np.matmul(g, np.transpose(pi))
for i in range(len(exp_gain)):
    print("Average gain when choosing location %d: $%.2f" % (i+1,␣
 ↪exp_gain[i]))
```

```
Average gain when choosing location 1: $-25.00
Average gain when choosing location 2: $-425.00
Average gain when choosing location 3: $87.50
Average gain when choosing location 4: $-12.50
Average gain when choosing location 5: $-112.50
Average gain when choosing location 6: $-212.50
Average gain when choosing location 7: $300.00
```

[8]:
```
print("Best choice: Location", np.argmax(exp_gain)+1)
print("Expected winnings: $", max(exp_gain), sep='')
```

```
Best choice: Location 7
Expected winnings: $300.0
```

Of course there is no Location 7. It just means tha the best action is the last one, which corresponds to:

> *"It's not worth digging any of the other locations up. Just stay home and study QIF."*

In the next three sections we introduce the concept of channels and posterior vulnerability.

2.3 Channels and Posterior Vulnerability

The following is known as *“The Three Prisoners Problem”*.

Three prisoners, Alice, Bob and Charlie are senteced to death, but one of them (uniformly chosen at random) is selected to be pardoned, so that just the two out of the three prisoners will be executed. The warden knows which one will be pardoned, but he is not allowed to tell the prisoners. Alice begs the warden to let her know the identity of one of the others who will be executed saying:

> *“If Bob is pardoned, say Charlie’s name, and if Charlie is pardoned say Bob’s. If I’m pardoned, chose randomly to name Bob or Charlie.”*

We are interested in answering the following two questions: 1. Given the warden’s answer, what is the probability of correctly guessing the pardoned prisoner? 2. Is the warden’s answer useful for Alice?

2.3.1 Channel matrix

Let’s model this problem using a channel W that takes a secret input X (the prisoner to be pardoned) and produces an output Y (the warden’s answer). You can think of W as being the warden in our problem. The possible values for X and Y are A, B, C (short for Alice, Bob and Charlie).

$$W = \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

W’s first row corresponds to $X = A$, meaning the scenario where Alice is chosen to be pardoned. In that case the channel’s output (or the warden’s saying) is not deterministic, but has some degree of randomness. More specifically the warden says Alice with probability 0, and chooses uniformly between Bob and Charlie, that is with probability $\frac{1}{2}$ each.

W’s second row corresponds to $X = B$, meaning the scenario where Bob is chosen to be pardoned. In that case, there is only one possible output, and that is Charlie. In this case, W (the warden) behaves deterministically and this can be seen because the second row of W has 0 everywhere, except for one specific output, which gets probability 1. Same happens with the third row, which corresponds to $X = C$, meaning Charlie is chosen to be pardoned.

Remember that each row of W sums up to 1. This happens because each row defines a probability distribution, which basically says how W (the warden) behaves given a specific input X.

Let’s also define W using python and libqif:

[2]:
```
W = np.array([
    # Y=A   Y=B   Y=C
    [   0, 1/2, 1/2],     # X=A
    [   0,   0,   1],     # X=B
    [   0,   1,   0],     # X=C
])
```

Now, to answer Question 1 using *QIF* terminology, we want to find the **posterior vulnerability** of W. That is, what is the probability of correctly guessing the secret X after observing the channel's output Y. Let's see how we can compute that.

2.3.2 Prior distribution

The problem's description clearly states that the prisoner to be pardoned is uniformly chosen at random so we have

$$p_X(A) = p_X(B) = p_X(C) = \frac{1}{3}$$

or for short

$$\pi = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$$

Let's also define that in python:

[3]:
```
pi = probab.uniform(3)
print(pi)
```

```
[0.33333333 0.33333333 0.33333333]
```

2.3.3 Joint Matrix

The joint matrix J contains the joint probabilities for each combination of X and Y.

$$J = \begin{pmatrix} 0 & \frac{1}{6} & \frac{1}{6} \\ 0 & 0 & \frac{1}{3} \\ 0 & \frac{1}{3} & 0 \end{pmatrix}$$

Notice that J depends on the channel W, **but also** on the distribution π of X. Meaning that it depends on all of the $p_X(A)$, $p_X(B)$, $p_X(C)$. Thus, if the pardoned prisoner were not chosen at random (meaning π was different), then J would be different.

Remember that if we sum all of J's elements they add up to 1. That is excpected because by the defitition of probabiliy, when we sum the probabilities of every possible outcume, they must add up to 1. And that is exactly what happens when we sum J's elements.

For computing its elements we use the rule $p_{Y,X}(y, x) = p_X(x) \cdot p_{Y|X}(y|x)$.

2.3.4 Posterior Distributions

But we are most interested in what does the warden's saying (W's output) tells us about X. The posteriors distribution matrix P helps us with that. It basically tells us the updated probability for X **given that** we've observed the value of Y.

$$P = \begin{pmatrix} 0 & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & \frac{2}{3} \\ 0 & \frac{2}{3} & 0 \end{pmatrix}$$

P's first column gives us the probabilities of each prisoner being pardoned, **given that** the warden has said Alice's name. That is, the probabilities of X being A, B or C **given that** $Y = A$. Here we have 0 everywhere because $Y = A$ never happens. We could basically remove this column because it corresponds to an output of Y which happens with a probability of 0.

P's second column gives us the probabilities of X being A, B or C **given that** $Y = B$. The same happens with the third column.

Remember that each non zero column gives us a new probability distribution (also meaning that each column adds up to 1). That is, it tells us the **updated** probability of each X **after** observing W's output.

P can be computed using the rules $p(y) = \sum_{x \in X} p_{X,Y}(x, y)$ and then $p_{X|Y}(x|y) = \frac{p_{X,Y}(x,y)}{p_Y(y)}$. Notice that we have already calculated all joint probabilities in J.

[4]:
```
P = channel.posteriors(W, pi)
print(P)
```

```
[[       nan 0.33333333 0.33333333]
 [       nan 0.         0.66666667]
 [       nan 0.66666667 0.        ]]
```

2.3.5 Finally solving the problem

Question 2 We are now ready to answer the two questions we set for ourselves. First we will answer the question number 2, which asks

> Is the warden's answer useful for Alice?

Before hearing the warden's saying (W's output), Alice knew that she had a $\frac{1}{3}$ probability of surviving (becuase the pardoned prisoner were chosen uniformly at random).

Now, using matrix P, we know that if the warden says Bob's name (P's second column), then Alice has a $p(X = A|Y = B) = \frac{1}{3}$ probability of surviving.

With the same reasoning, we see that if the warden says Charlie's name (P's third column), then Alice has a $p(X = A|Y = C) = \frac{1}{3}$ probability of surviving.

Notice that the warden never says Alice's name (W never outputs A) and this can be seen by verifying that $p_Y(A) = 0$.

So before the warden's answer, Alice survived with a probability of $\frac{1}{3}$. After the warden's answer, no matter which name the warden says, Alice **again** survives with a probabiliy of $\frac{1}{3}$.

Looks like Alice should have picked her question more carefully, because *the warden's answer is never useful for Alice*.

Note that this question could have been answered by using basic probability theory only. But since this is the original question of the problem and it fits quite well within the *QIF* train of thought, we thought it would be natural to include it.

Question 1 Now we will answer question 1, which asks

> Given the warden's answer, what is the probability of correctly guessing the pardoned prisoner?

Consider the case where the warden says Bob's name (P's second column). Alice has a probability of $\frac{1}{3}$ of being pardoned, Bob 0 and Charlie $\frac{2}{3}$. What would be your best guess for who has been pardoned? The logical answer would be Charlie, because he has the highest probability among the others **in that specific column**. So if the warden says Bob's name, you know that your best guess is Charlie and you have a probability of success $\frac{2}{3}$.

Using the same reasoning, we can see that if the warden says Charlie's name (P's third column), your best guess would Bob because he has the highest probability **in that specific column**. So if the warden says Charlie's name, you know that your best guess is Bob and you have a probability of success $\frac{2}{3}$.

Before the warden's answer our best guess for who is the pardoned prisoner would be, well, anyone since they have the same probability of being pardoned. And our probability of success is $\frac{1}{3}$.

> This is the *prior vulnerability* of W, also called $V(\pi)$. That is, the probability of correctly guessing the secret W *before* observing the channel's output Y.

But after receiving the warden's answer (W's output) we see that we can make a guess with probability of success $\frac{2}{3}$! In the first case we have to guess Charlie and in the second Bob.

> This is the *posterior vulnerability* of W, also called $V(\pi, C)$. That is, the probability of correctly guessing the secret W *after* observing the channel's output Y.

So to answer the question, *given the warden's answer, the probability of correctly guessing the pardoned prisoner is* $\frac{2}{3}$.

Alice's name never pops up so we don't have to deal with that. In fact we could use the same reasoning if we eliminated P's first row, since it has 0 everywhere.

Generalizing this, we can compute the posterior vulnerability of any channel W by first computing its posterior distribution matrix P, removing all 0 columns, then finding for each column, its maximum element (which corresponds to the best guess when

W's output is the one corresponding to that column). In our case the maximum elements were $\frac{2}{3}$ and $\frac{2}{3}$.

But we need a way to combine them and get one value representing the whole channel. We can do that by **weighing** each value with the probability of each column happening, that is by $p_Y(y)$. In our case the second column (W outputs B) happens with probability of $p_Y(B) = \frac{1}{2}$ and the third column (W outputs C) happens with probability of $p_Y(C) = \frac{1}{2}$. So if we combine them we get

$$V(\pi, C) = \frac{1}{2} \cdot \frac{2}{3} + \frac{1}{2} \cdot \frac{2}{3}$$
$$V(\pi, C) = \frac{2}{3}$$

which is what we intuitively got without specifically using the general definition of posterior vulnerability.

We can use the following functions to get the channel's prior or posterior vulnerability with

[5]:
```
print("Prior Bayes vulnerability:", measure.bayes_vuln.prior(pi))
print("Posterior Bayes vulnerability:", measure.bayes_vuln.
 ↪posterior(pi, W))
```

```
Prior Bayes vulnerability: 0.3333333333333333
Posterior Bayes vulnerability: 0.6666666666666666
```

or the best strategy for guessing the pardoned prisoner observing W's output

[6]:
```
print("Best guessing strategy:", measure.bayes_vuln.strategy(pi, W))
```

```
Best guessing strategy: [0 2 1]
```

Here 0 corresponds to Alice, 1 to bob and 2 to Charlie. The first element is for when W's output is A, the second for when W's output is B and the third for C. The first element in our case does not have a meaning because W never outputs A.

Compare libqif's results with the ones we computed ourselves. Did we get everything right?

2.4 Channels and Posterior Vulnerability (part 2)

In this part as well as the next one are going to examine a couple of variations of the original *Three Prisoners* problem. The first one is:

2.4.1 What if the warden uses a biased coin to answer the question?

In the classic version of the problem Alice says to the warden

> …If I'm pardoned, chose randomly to name Bob or Charlie.

which has basically the same meaning as

> …If I'm pardoned, flip a **fair coin** and if it lands on heads says Bob's name otherwise say Charlie's.

But what would happen if the warden flipped a **biased coin** and made his choice according to that?

A biased coin is characterized by a probability p. For example if $p = \frac{2}{3}$ then it means that the coin lands on heads 2 out of 3 times and on tails 1 out of 3.

If you haven't guessed it already, we are going to experiment with a couple different values of p and for that we are going to define the `get_W(p)` and the `get_distribution(p)` function that are going to speed things up a little.

For this variation we are going to keep assuming that **the pardoned prisoner is chosen uniformly**.

[2]:
```
def get_W(p):
    C = np.array([
        [0, p, 1-p],
        [0, 0,   1],
        [0, 1,   0],
    ])
    return C

def get_distribution(p):
    return np.array([p, (1-p)/2,(1-p)/2])
```

`get_W(p)` takes as input a probability p and creates the channel matrix W (as discussed in the previous part) with the only difference that now the warden uses a biased coin characterized by p.

`get_distribution(p)` takes as input a probability p and creates a distribution π where:

- If $p = 1/3$, then you get the uniform distribution with each prisoner being chosen with a probability of $1/3$.
- If $p > 1/3$, then the first prisoner (Alice) is more likely to be pardoned than any of the other two.
- If $p < 1/3$, then the first prisoner (Alice) is the least likely of all to be pardoned.

Now consider the following cases.

2.4.2 Biased coin with $p = \frac{2}{3}$

Let's define W, and the prior distribution π in python.

[3]:
```
W = get_W(2/3)
print("W\n", W)
pi = get_distribution(1/3)
print("pi\n", pi)
```

```
W
 [[0.         0.66666667 0.33333333]
 [0.         0.         1.        ]
 [0.         1.         0.        ]]
pi
 [0.33333333 0.33333333 0.33333333]
```

Now how does this change our answer to the questions we are interested in? Let's quickly remember these questions.

1. Given the warden's answer, what is the probability of correctly guessing the pardoned prisoner?
2. Is the warden's answer useful for Alice?

For question 1, as we said in the previous part, the answer is basically given by the posterior vulnerability of W. Let's also print the prior vulnerability.

[4]:
```
print("Prior Bayes vulnerability:", measure.bayes_vuln.prior(pi))
print("Posterior Bayes vulnerability =", measure.bayes_vuln.
↪posterior(pi, W))
```

```
Prior Bayes vulnerability: 0.33333333333333337
Posterior Bayes vulnerability = 0.6666666666666666
```

We can see that it is exactly the same as in the original problem, where the coin was not biased. Hmm, that looks a bit suspicious…

For question 2, we are also going to need the posteriors distribution matrix P.

[5]:
```
P = channel.posteriors(W, pi)
print(P)
```

```
[[ nan 0.4  0.25]
 [ nan 0.   0.75]
 [ nan 0.6  0.  ]]
```

Here, the answer is basically given by $p(X = A|Y = y)$, by examining it for all values of y. There are two possible values for y and these are B and C. Now, $p(X = A|Y = B)$ corresponds to P's first-row second-column element which is 0.4 and $p(X = A|Y = C)$ corresponds to P's first-row third-column element which is 0.25. So from that we can deduct that if the warden says Bob's name then Alice has

a 0.4 chance of surviving but if the warden says Charlie's then it goes down to 0.25. So here the warden's answer is somewhat useful for Alice, meaning that it updates her knowledge about the probability of her surviving in some cases.

2.4.3 Biased coin with $p = \frac{3}{4}$

For question 1, doing the same thing as before we see that

```
[6]: W = get_W(3/4)
     print("W\n", W, "\n")
     print("Prior Bayes vulnerability:", measure.bayes_vuln.prior(pi))
     print("Posterior Bayes vulnerability =", measure.bayes_vuln.
      ↪posterior(pi, W))
```

```
W
 [[0.   0.75 0.25]
 [0.   0.   1.  ]
 [0.   1.   0.  ]]

Prior Bayes vulnerability: 0.33333333333333337
Posterior Bayes vulnerability = 0.6666666666666666
```

We see that again, the posterior vunlerability of W is the same. This implies that possibly, no matter the value of p, $V(\pi, W)$ stays the same. This indeed can be experimentally verified by trying different values of p and plotting the results. But before that, take a few moments to try yourself a few different values for the `p` parameter of `get_W(p)` in the cell above and rerunning it.

```
[7]: ps = np.linspace(0, 1, 100)
     plt.plot(ps, [measure.bayes_vuln.posterior(get_distribution(1/3),␣
      ↪get_W(p)) for p in ps])
     plt.xlabel('p')
     plt.ylabel('Posterior vulerability on uniform prior')
     None
```

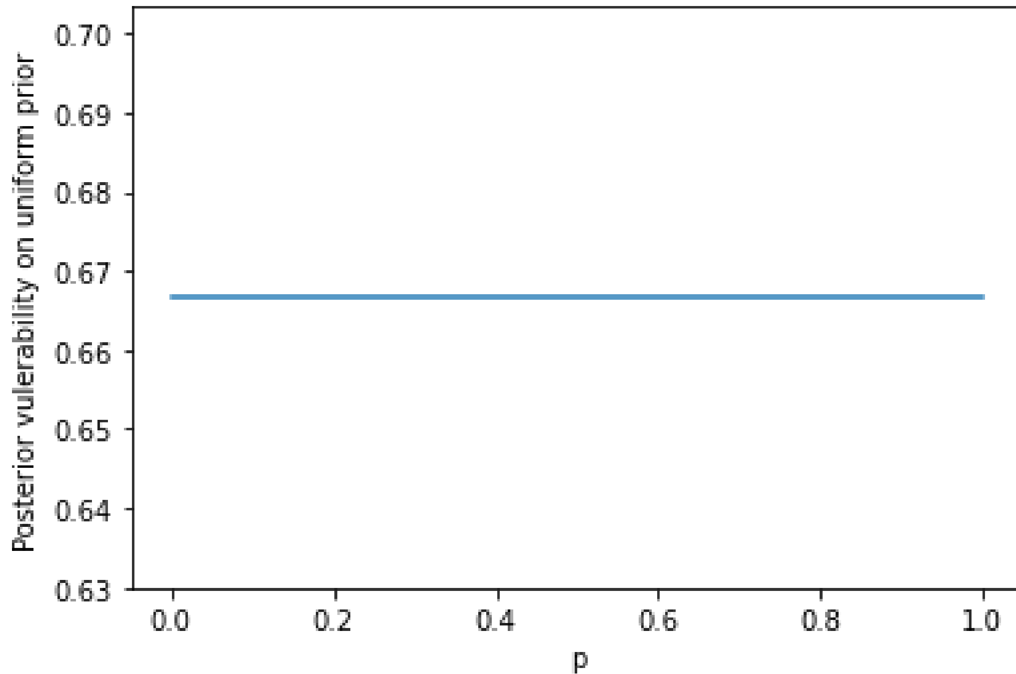

Figure 2.1: Posterior vulnerability on uniform prior

Here we clearly see that $V(\pi, W)$ indeed stays the same for all possible values of p. Which means that no matter what p the warden uses to determine his answer, **our best guess** will always succeed with probability $\frac{2}{3}$. We could say that all W defined this way, have the same vulenrability against someone who tries to guess the pardoned prisoner in one try.

To get a better insight of why this is happening take a look at the following piece of code. For each p from 0 to 1, it prints the p parameter itself, then the distribution $p_Y(y)$ for $y = B$ or C (remember that Alice's name never pops up meaning $p_Y(A) = 0$, thus $y = A$ is omitted) and then array of the posterior distributions. Its output is in the form of

```
-------------------------------
|    p(Y=B)          p(Y=C)    |
-------------------------------
| p(X=A | Y=B)   p(X=A | Y=C) |
| p(X=B | Y=B)   p(X=B | Y=C) |
| p(X=C | Y=B)   p(X=C | Y=C) |
-------------------------------
```

It also marks for each column its maximum element.

[8]:
```
from print_hyper import print_hyper

for k in range(11):
        print("\np=", k, "/10", sep='')
        print_hyper(get_W(k/10), get_distribution(1/3),␣
 ↪highlight_maxima=True)
```

```
p=0/10
-------------------
|    0.33    0.67 |
-------------------
|    0.00    0.50 |
|    0.00 -->0.50 |
| -->1.00    0.00 |
-------------------

p=1/10
-------------------
|    0.37    0.63 |
-------------------
|    0.09    0.47 |
|    0.00 -->0.53 |
| -->0.91    0.00 |
-------------------

p=2/10
-------------------
|    0.40    0.60 |
-------------------
|    0.17    0.44 |
|    0.00 -->0.56 |
| -->0.83    0.00 |
-------------------

p=3/10
-------------------
|    0.43    0.57 |
-------------------
|    0.23    0.41 |
|    0.00 -->0.59 |
| -->0.77    0.00 |
-------------------

p=4/10
-------------------
|    0.47    0.53 |
-------------------
|    0.29    0.38 |
|    0.00 -->0.63 |
| -->0.71    0.00 |
-------------------

p=5/10
-------------------
```

```
|    0.50    0.50 |
-------------------
|    0.33    0.33 |
|    0.00 -->0.67 |
| -->0.67    0.00 |
-------------------

p=6/10
-------------------
|    0.53    0.47 |
-------------------
|    0.38    0.29 |
|    0.00 -->0.71 |
| -->0.63    0.00 |
-------------------

p=7/10
-------------------
|    0.57    0.43 |
-------------------
|    0.41    0.23 |
|    0.00 -->0.77 |
| -->0.59    0.00 |
-------------------

p=8/10
-------------------
|    0.60    0.40 |
-------------------
|    0.44    0.17 |
|    0.00 -->0.83 |
| -->0.56    0.00 |
-------------------

p=9/10
-------------------
|    0.63    0.37 |
-------------------
|    0.47    0.09 |
|    0.00 -->0.91 |
| -->0.53    0.00 |
-------------------

p=10/10
-------------------
|    0.67    0.33 |
-------------------
|    0.50    0.00 |
```

```
|    0.00 -->1.00 |
| -->0.50    0.00 |
-------------------
```

Notice how the maximum elemnts of each column stay at the same positions as p increases. It means that upon observing a specific output y, our best guess always stays the same regardless of the distribution of X.

Remember that the posterior vulnerability in our case is calculated by

$$V(\pi, W) = p_Y(B) \cdot max_B + p_Y(C) \cdot max_C$$

where max_B and max_C are the maximum elements of the first and second columns respectively.

2.5 Channels and Posterior Vulnerability (part 3)

The next variation we are going to examine is:

2.5.1 What if the pardoned prisoner is not uniformly chosen at random?

Here we are going to assume the warden bevahes as in the original description of the problem, but we are going to see what happens when we don't choose the pardoned prisoner uniformly. We are mainly interested in answering the following question:

> Given the warden's answer, what is the probability of correctly guessing the pardoned prisoner?

Remember that answering this questions in terms of *QIF* basically means finding the posterior vulnerability of W, $V(\pi, W)$.

First let's do some basic definitions in python.

[2]:
```
def get_W(p):
    C = np.array([
        [0, p, 1-p],
        [0, 0,   1],
        [0, 1,   0],
    ])
    return C

def get_distribution(p):
    return np.array([p, (1-p)/2,(1-p)/2])
```

Now remember that if you call

[3]:
```
get_distribution(1/3)
```

[3]:
```
array([0.33333333, 0.33333333, 0.33333333])
```

you get the uniform distribution and if you call

[4]:
```
get_W(1/2)
```

[4]:
```
array([[0. , 0.5, 0.5],
       [0. , 0. , 1. ],
       [0. , 1. , 0. ]])
```

you get the original W.

We are going to keep the original W but experiment with `get_distribution(p)` for different values of `p` and for each those values find out its prior and posterior vulnerability.

2.5.2 Prior vunlerability

Take a look at the following graph.

[5]:
```
ps = np.linspace(0, 1, 100)
plt.plot(ps, [measure.bayes_vuln.prior(get_distribution(p)) for p in␣
↪ps])
plt.xlabel('p')
plt.ylabel('Prior vulnerability with original W')
None
```

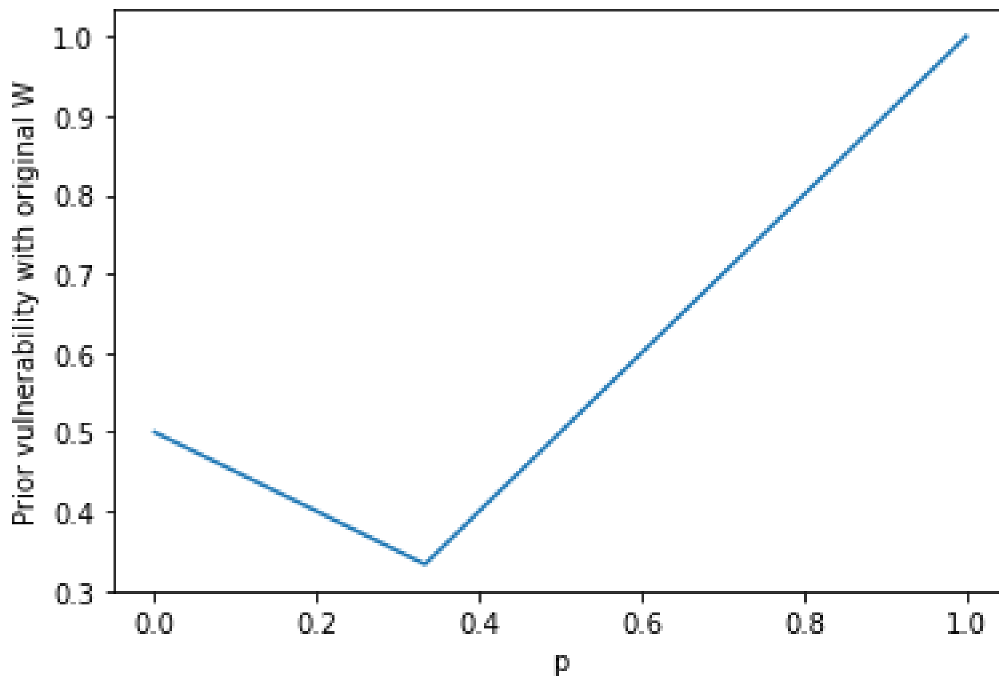

Figure 2.2: Prior vulnerability with original W

The x-axis corresponds to the `p` parameter of `get_distribution(p)` and the y-axis to the vulnerability of the distribution produced by `get_distribution(p)`.

The code below prints for each parameter `p` its corresponding distribution and marks the element with the highest probability which is basically our best guess for that particular distribution. Notice the maximum elemnts of each distribution as `p` increases.

[6]:
```
from print_dist import print_dist

for i in range(11):
    print("get_distribution(%2d" % i , "/10) = ", sep='', end='')
    print_dist(get_distribution(i/10), highlight_maxima=True)
```

```
get_distribution( 0/10) = (    0.00 -->0.50 -->0.50 )
get_distribution( 1/10) = (    0.10 -->0.45 -->0.45 )
get_distribution( 2/10) = (    0.20 -->0.40 -->0.40 )
get_distribution( 3/10) = (    0.30 -->0.35 -->0.35 )
```

```
get_distribution( 4/10) = ( -->0.40    0.30    0.30 )
get_distribution( 5/10) = ( -->0.50    0.25    0.25 )
get_distribution( 6/10) = ( -->0.60    0.20    0.20 )
get_distribution( 7/10) = ( -->0.70    0.15    0.15 )
get_distribution( 8/10) = ( -->0.80    0.10    0.10 )
get_distribution( 9/10) = ( -->0.90    0.05    0.05 )
get_distribution(10/10) = ( -->1.00    0.00    0.00 )
```

Basically what happens is:

- For $p \in \left[0, \frac{1}{3}\right)$ our best guess for the pardoned prisoner is either B or C, and as p gets bigger, our probability of success gets smaller.
- For $p = \frac{1}{3}$ our best guess is either A or B or C, and gives us the minimum chance of success among all possible values of p.
- For $p \in \left(\frac{1}{3}, 1\right)$ our best guess for the pardoned prisoner is always A and as p gets bigger, our probability of success gets bigger as well.
- For $p = 1$ our best guess is A and we succeed with probability of 1. That is, always!

2.5.3 Posterior vunlerability

Now, take a look at the following graph.

[7]:
```
ps = np.linspace(0, 1, 100)
plt.plot(ps, [measure.bayes_vuln.posterior(get_distribution(p),␣
↪get_W(0.5)) for p in ps])
plt.xlabel('p')
plt.ylabel('Posterior vulnerability with original W')
None
```

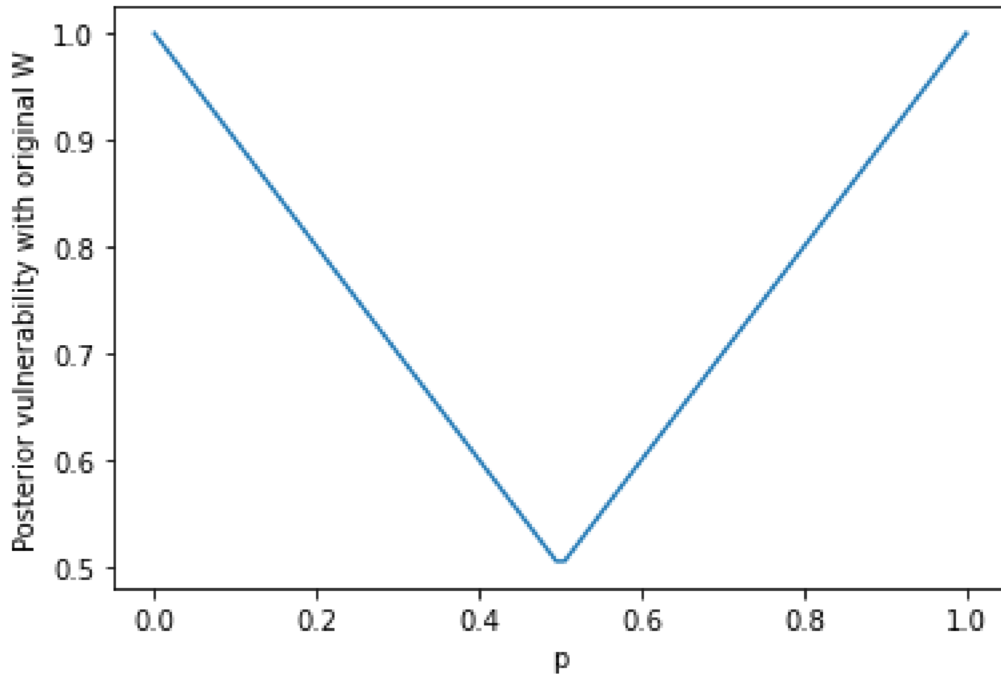

Figure 2.3: Posterior vulnerability with original W

The x-axis again corresponds to the `p` parameter of `get_distribution(p)` but now the y-axis corresponds to the posterior vulnerability of W when based on the distribution produced by `get_distribution(p)`. We can better understand this graph by experimenting with some values of the `p` parameter.

The code below prints for each `p` the `p` parameter itself, then the distribution $p_Y(y)$ for $y = B$ or C (remember that Alice's name never pops up meaning $p_Y(A) = 0$, thus $y = A$ is omitted) and then array of the posterior distributions. Its output is in the form of

```
-------------------------------
|    p(Y=B)          p(Y=C)    |
-------------------------------
| p(X=A | Y=B)   p(X=A | Y=C) |
| p(X=B | Y=B)   p(X=B | Y=C) |
| p(X=C | Y=B)   p(X=C | Y=C) |
-------------------------------
```

It also marks for each column its maximum element. Notice the maximum elemnts of each column as `p` increases.

[8]:
```
from print_hyper import print_hyper

for k in range(11):
        print("\np=", k, "/10", sep='')
        print_hyper(get_W(1/2), get_distribution(k/10),␣
 ↪highlight_maxima=True)
```

```
p=0/10
-------------------
|    0.50    0.50 |
-------------------
|    0.00    0.00 |
|    0.00 -->1.00 |
| -->1.00    0.00 |
-------------------

p=1/10
-------------------
|    0.50    0.50 |
-------------------
|    0.10    0.10 |
|    0.00 -->0.90 |
| -->0.90    0.00 |
-------------------

p=2/10
-------------------
|    0.50    0.50 |
-------------------
|    0.20    0.20 |
|    0.00 -->0.80 |
| -->0.80    0.00 |
-------------------

p=3/10
-------------------
|    0.50    0.50 |
-------------------
|    0.30    0.30 |
|    0.00 -->0.70 |
| -->0.70    0.00 |
-------------------

p=4/10
-------------------
|    0.50    0.50 |
-------------------
|    0.40    0.40 |
|    0.00 -->0.60 |
| -->0.60    0.00 |
-------------------

p=5/10
-------------------
|    0.50    0.50 |
```

```
-------------------
| -->0.50 -->0.50 |
|    0.00 -->0.50 |
| -->0.50    0.00 |
-------------------

p=6/10
-------------------
|    0.50    0.50 |
-------------------
| -->0.60 -->0.60 |
|    0.00    0.40 |
|    0.40    0.00 |
-------------------

p=7/10
-------------------
|    0.50    0.50 |
-------------------
| -->0.70 -->0.70 |
|    0.00    0.30 |
|    0.30    0.00 |
-------------------

p=8/10
-------------------
|    0.50    0.50 |
-------------------
| -->0.80 -->0.80 |
|    0.00    0.20 |
|    0.20    0.00 |
-------------------

p=9/10
-------------------
|    0.50    0.50 |
-------------------
| -->0.90 -->0.90 |
|    0.00    0.10 |
|    0.10    0.00 |
-------------------

p=10/10
-------------------
|    1.00    0.00 |
-------------------
| -->1.00 -->1.00 |
|    0.00    0.00 |
```

```
|     0.00     0.00  |
--------------------
```

Here we see that

- For $p \in \left(0, \frac{1}{2}\right)$ our best guess is C when $y = B$ and B when $y = C$. In both cases we succeed with the same probability.
- For $p = \frac{1}{2}$ our best guess is, well, anyone.
- For $p \in \left(\frac{1}{2}, 1\right)$ our best guess is always A no matter the channels output.

2.5.4 Multiplicative leakage

The multiplicative leakage of a channel W and a distribution π is defined by

$$L^{\times}(\pi, W) = \frac{V(\pi, W)}{V(\pi)}$$

and its main purpose is to provide a measure of leakage with respect to the initial vulnerability the prior distribution had.

[9]:
```
plt.plot(ps, [measure.bayes_vuln.mult_leakage(get_distribution(p),␣
    ↪get_W(0.5)) for p in ps])
plt.xlabel('p')
plt.ylabel('Multiplicative leakage with original W')
None
```

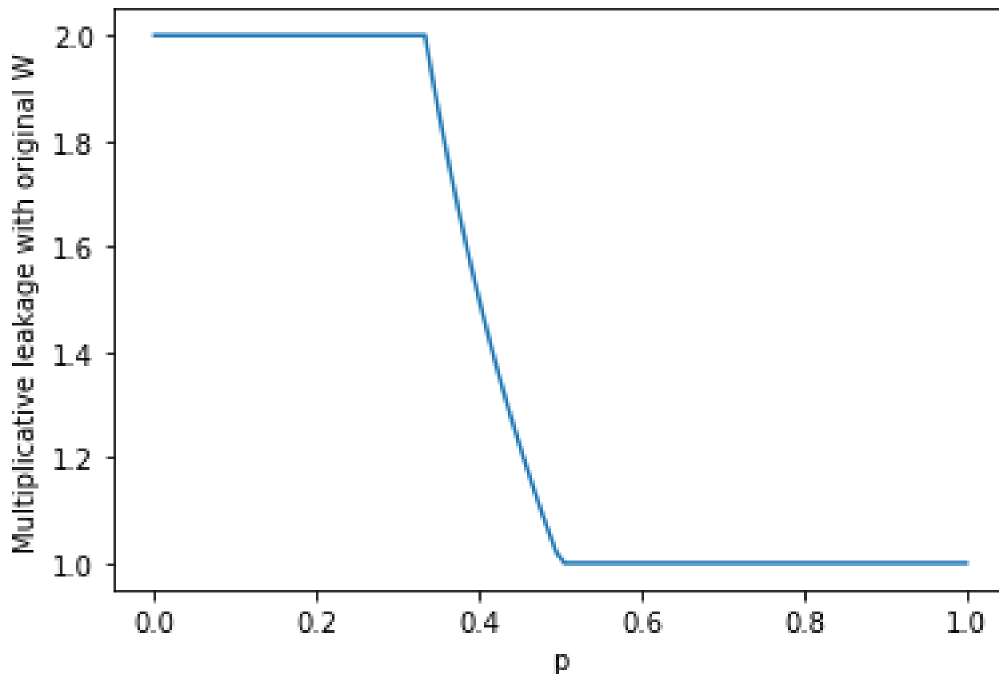

Figure 2.4: Multiplicative leakage with original W

Here we see that:

- For $p \in \left(0, \frac{1}{3}\right)$ W makes the secret more vulnerable by doubling its prior vlunerability.
- For $p \in \left(\frac{1}{3}, \frac{1}{2}\right)$ W makes the secret less and less vulnerable compared to its initial vulnerability.
- For $p \in \left(\frac{1}{2}, 1\right)$ W keeps the secret as vulnerable as it was before entering the channel.

Note that multiplicative leakage equal to 1 does not imply 0 vulnerability. Take for example the prior with generated by `get_distribution(9/10)` fed into W. Both the prior and posterior vulnerability are equal to $\frac{9}{10}$ which agrees to the fact that the multiplicative leakage is 1. But the secret is still quite vunerable. It can be guessed correctly in one try with probability of success $\frac{9}{10}$.

It is also interesting to notice that $L^{\times}(\pi, W)$ has an inflexion point at $p = \frac{1}{3}$, the same point where the prior vulnerability makes a turn. It also has another one at $p = \frac{1}{2}$, the same point where the posterior vulnerability makes a turn.

2.5.5 Generalizing over any prior distribution

But what happens on other prior distributions? `get_distribution()` creates a very specfic form of distributions, but there are many many more. Infintely many. How can we be sure about our channels behaviour on other priors?

Here comes handy the theorem that states that

> For any channel C, the maximum multiplicative Bayes leakage over all priors is always realized on a uniform prior θ.

The channel's multiplicative leakage on a uniform prior (which in our case is generated by `get_distribution(1/3)`) is equal to 2. So according to this theorem we can be sure that there is no prior which makes our channel's posterior vulnerability greater than 2 times its prior vulnerability.

Having established the concepts of secrets, channels and vulnerability, a natural question would arise: *How can we compare two different channels in respect to how much information do they leak?* Here comes into play the concept of refinement.

2.6 Refinement

Consider a scenario where we are using an app that provides us with restaurant suggestions near our location. But for privacy reasons we are not be very comfortable with sharing our actual location, so before reporting it we use some mechanism in order to add some noise to it.

For the sake of the following examples, consider the following locations

$$\mathtt{L1},\ \mathtt{L2},\ \mathtt{L3},\ \mathtt{L4}$$

which are not too far apart from each other. So for example, we might be at L3 but choose to report L1 and still get reasonable suggestions because we are still close to it and at the same time we are not giving away our location 100%. But in what way should we decide wether to report our actual location or not and how vulnerable does our privacy become?

[2]:
```
def get_pi(p, p_pos, n):
    return np.array([p if i == p_pos else (1-p)/(n-1) for i in␣
 ↪range(n)])

def get_C(p, n):
    return np.array([get_pi(p, i, n) for i in range(n)])
```

[3]:
```
# Number of possible locations
n = 4
```

2.6.1 Producing the actual location with probability $p = 0.7$

Consider the following channel matrix C_1. It produces the actual location with probability of 0.7 and distributes the remaining 0.3 to the other possible outcomes.

Table 2.2: C_1 **matrix**

C_1	L1	L2	L3	L4
L1	**0.7**	0.1	0.1	0.1
L2	0.1	**0.7**	0.1	0.1
L3	0.1	0.1	**0.7**	0.1
L4	0.1	0.1	0.1	**0.7**

[4]:
```
C1 = get_C(0.7, n)
print("C1:\n", C1)
```

```
C1:
 [[0.7 0.1 0.1 0.1]
 [0.1 0.7 0.1 0.1]
 [0.1 0.1 0.7 0.1]
 [0.1 0.1 0.1 0.7]]
```

Given a uniform prior, we compute its Posterior Bayes Vulnerability and Multiplicative Bayes Capacity.

```
[5]: pi = probab.uniform(len(C1[0]))
```

```
[6]: print("Posterior Bayes Vulnerability:", measure.bayes_vuln.
      ↪posterior(pi, C1))
     print("Multiplicative Bayes Capacity:", measure.bayes_vuln.
      ↪mult_capacity(C1))
```

```
Posterior Bayes Vulnerability: 0.7
Multiplicative Bayes Capacity: 2.8
```

Here the posterior vulnerability corresponds to the probability of guessing the actual location by observing the location C_1 outputs. Notice that the channel's posterior vulnerability is equal 0.7 which is equal to the p parameter.

2.6.2 Producing the actual location with probability $p = 0.6$

Let's epxeriment with a smaller p of let's say 0.6. Channel matrix C_2 represents this case.

Table 2.3: C_2 **matrix**

C_2	L1	L2	L3	L4
L1	**0.6**	0.133	0.133	0.133
L2	0.133	**0.6**	0.133	0.133
L3	0.133	0.133	**0.6**	0.133
L4	0.133	0.133	0.133	**0.6**

```
[7]: C2 = get_C(0.6, n)
     # C2[1] = np.array(get_pi(1, 1, n))
     print("C2:\n", C2)
```

```
C2:
 [[0.6        0.13333333 0.13333333 0.13333333]
 [0.13333333 0.6        0.13333333 0.13333333]
 [0.13333333 0.13333333 0.6        0.13333333]
 [0.13333333 0.13333333 0.13333333 0.6       ]]
```

Let's also compute its Posterior Bayes Vulnerability and Multiplicative Bayes Capacity again, under a uniform prior.

[8]:
```
print("Posterior Bayes Vulnerability:", measure.bayes_vuln.
 ↪posterior(pi, C2))
print("Multiplicative Bayes Capacity:", measure.bayes_vuln.
 ↪mult_capacity(C2))
```

```
Posterior Bayes Vulnerability: 0.6
Multiplicative Bayes Capacity: 2.4
```

Notice that again its posterior vulnerability is equal to the p parammeter which is the probability of the actual location appearing on the output.

2.6.3 Comparing C_1 and C_2

The vulnerability of C_2 is less than that of C_1, so **under that specific prior** we are sure that C_2 leaks in general less information about our true location than C_1. But what happens under different priors? Maybe a specific adversary has a different prior knowledge about what our true location might be **before** observing the channel's output. That would correspond to a different prior distribution.

Someone might say that C_2 seems like leaking less information in general. C_2 also has a smaller multiplicative capacity. But can we be sure?

Let's see what happens under a specific family of prior distritubtions generated by `get_pi(p)`.

[9]:
```
ps = np.linspace(0, 1, 100)
plt.plot(ps, [measure.bayes_vuln.posterior(get_pi(p, 0, n), C1) for␣
 ↪p in ps], label="C1")
plt.plot(ps, [measure.bayes_vuln.posterior(get_pi(p, 0, n), C2) for␣
 ↪p in ps], label="C2")
plt.xlabel('p')
plt.ylabel('Posterior Bayes vulnerability')
plt.legend()
None
```

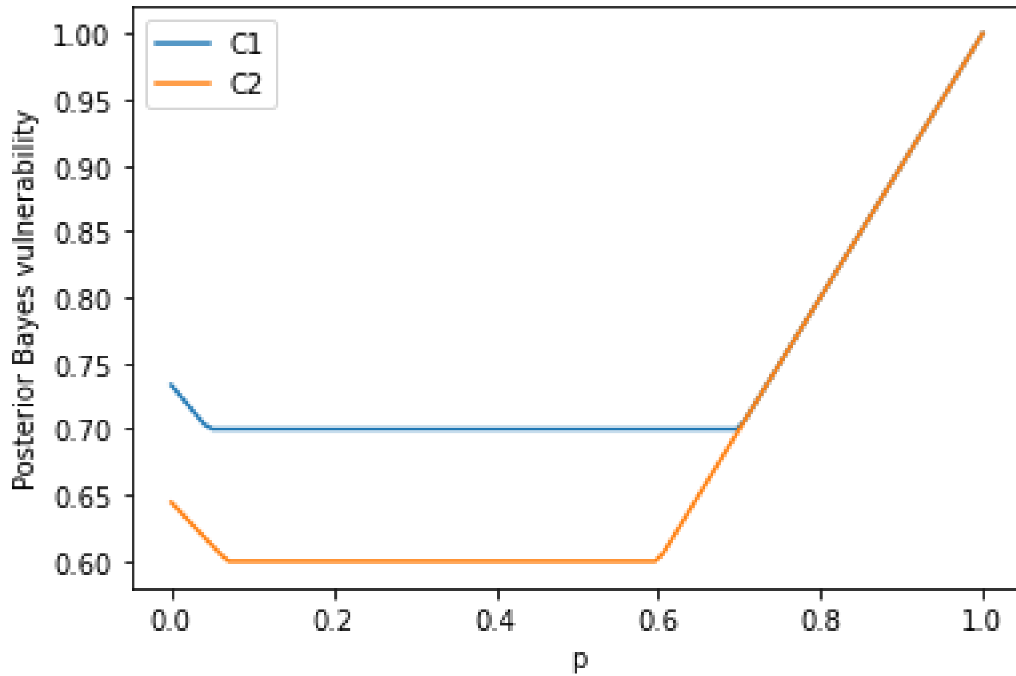

Figure 2.5: Posterior Bayes Vulnerability

Looks like that the posterior vulnerability of C_2 is always smaller than that of C_1. But if we want to be sure that this happens for every possible prior distribution (or even for every possible gain function), then we have to check for refinement.

If C_1 is refined by C_2, then C_2 is always more secure than C_1. That is, it always has a smaller vulnerability than C_1.

[10]:
```
refined = refinement.refined_by(C1, C2)
print("Is C1 refined by C2?", refined)
```

```
Is C1 refined by C2? True
```

Looks like our intuition was right!

Refinement also means that C_2 is a post processing of C_1. That means that there exists a channel R such that takes every output of C_1, processes it furtherly, produces an output and in total, their combined behaviour (C_1 and R) is exactly like that of C_2.

[11]:
```
R = channel.factorize(C2, C1)
print("Is C1.R == C2? ", np.allclose(C1.dot(R), C2))
print("\nR:\n", R)
print("\nC1.R:\n", C1.dot(R))
print("\nC2:\n", C2)
```

```
Is C1.R == C2?  True

R:
 [[0.83333333 0.05555556 0.05555556 0.05555556]
```

```
 [0.05555556 0.83333333 0.05555556 0.05555556]
 [0.05555556 0.05555556 0.83333333 0.05555556]
 [0.05555556 0.05555556 0.05555556 0.83333333]]

C1.R:
 [[0.6        0.13333333 0.13333333 0.13333333]
 [0.13333333 0.6        0.13333333 0.13333333]
 [0.13333333 0.13333333 0.6        0.13333333]
 [0.13333333 0.13333333 0.13333333 0.6       ]]

C2:
 [[0.6        0.13333333 0.13333333 0.13333333]
 [0.13333333 0.6        0.13333333 0.13333333]
 [0.13333333 0.13333333 0.6        0.13333333]
 [0.13333333 0.13333333 0.13333333 0.6       ]]
```

2.6.4 Different p for some rows

What if we don't use the same p for all the rows, but for some combinations of bits we choose a different distribution? Maybe there occured a weird bug in our mechanism and when the input is L2, then the output is always L2. Channel matrix C_3 represents this case.

Table 2.4: C_3 matrix

C_3	L1	L2	L3	L4
L1	**0.7**	0.1	0.1	0.1
L2	0	**1**	0	0
L3	0.1	0.1	**0.7**	0.1
L4	0.1	0.1	0.1	**0.7**

[12]:
```
C3 = get_C(0.6, n)
C3[1] = np.array(get_pi(1, 1, n))
print("C3:\n", C3)
```

```
C3:
 [[0.6        0.13333333 0.13333333 0.13333333]
 [0.         1.         0.         0.        ]
 [0.13333333 0.13333333 0.6        0.13333333]
 [0.13333333 0.13333333 0.13333333 0.6       ]]
```

Now how does C_3 compare to C_1? For 3 out of 4 inputs, C_3 behaves just like C_1, but for 01 C_3 always tells the truth. How does that affect the total vulnerability of C_3?

[13]:
```
print("Posterior Bayes Vulnerability:", measure.bayes_vuln.
 ↪posterior(pi, C3))
print("Multiplicative Bayes Capacity:", measure.bayes_vuln.
 ↪mult_capacity(C3))
```

```
Posterior Bayes Vulnerability: 0.7
Multiplicative Bayes Capacity: 2.8
```

Looks like C_3 has the same vulnerability **under a uniform prior**. Let's experiment with different priors and observe the vulnerabilities of the two channels.

[14]:
```
ps = np.linspace(0, 1, 100)
plt.plot(ps, [measure.bayes_vuln.posterior(get_pi(p, 0, n), C1) for
    p in ps], label="C1")
plt.plot(ps, [measure.bayes_vuln.posterior(get_pi(p, 0, n), C3) for
    p in ps], label="C3")
plt.xlabel('p')
plt.ylabel('Posterior Bayes vulnerability')
plt.legend()
None
```

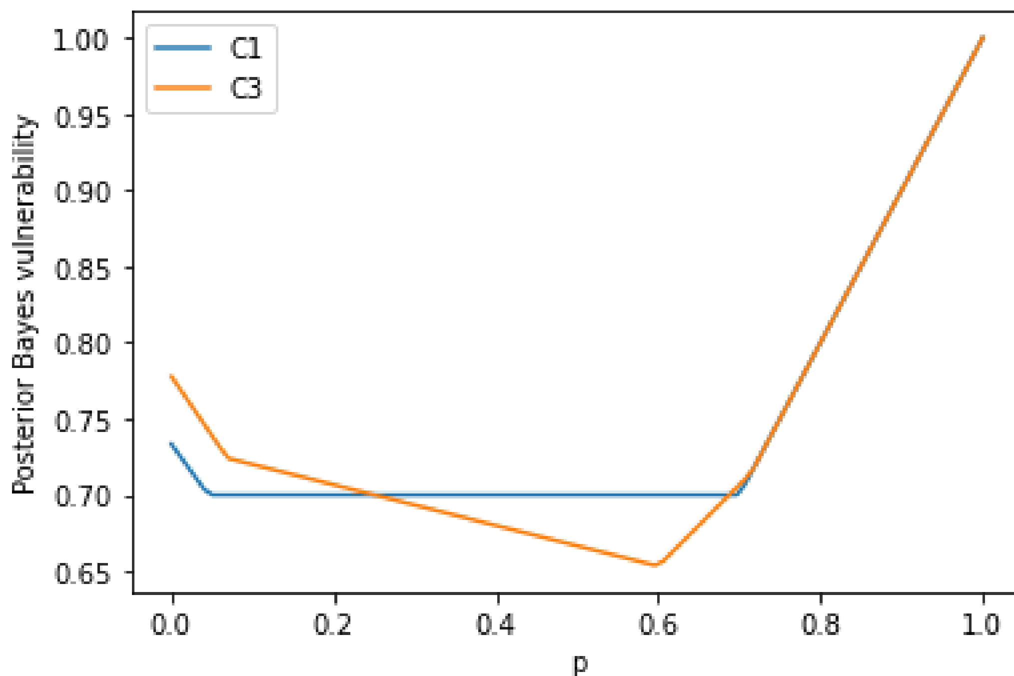

Figure 2.6: Posterior Bayes Vulnerability

Looks a bit more interesting than the previous plot! Here we see that for some priors, C_1 has the lowest vulnerability between the two, but for other priors C_3 has the lowest. This means that an adversary with a different knowledge about the prior distribution might prefer one channel over the other.

What we can derive from that is that C_1 can't be a refinement of C_3 because if it were, then it should always have a lower vulnerability than that of C_3.

For the same reason C_3 can't be a refinement of C_1.

This can also be verified by checking that there is no matrix R such that $C_3R = C_1$ or $C_1R = C_3$. This is what the `refined_by` function checks for.

[15]:
```
refined = refinement.refined_by(C1, C3)
print("Is C1 refined by C3?", refined)

refined = refinement.refined_by(C3, C1)
print("Is C3 refined by C1?", refined)
```

```
Is C1 refined by C3? False
Is C3 refined by C1? False
```

Notice also that C_1 and C_3 have the same capacity of 2.8. But that doesn't say much when it comes to comparing two channels. That's because capacity talks about worst case leakage happening on specific priors. But as seen in the plot above, for other priors one channel might be more vulnerable than the other. So if we want comparison under any prior, then we must check for refinement.

Absence of refinement also means that even under the same prior distribution π there might be an adversary that prefers C_3 and another one that prefers C_1.

For example, consider the following gain functions modeling two different adversaries.

[16]:
```
G1 = get_G1()
print(G1)
```

```
[[  1   0   0   0]
 [  0 100   0   0]
 [  0   0   1   0]
 [  0   0   0   1]]
```

[17]:
```
G2 = get_G2()
print(G2)
```

```
[[100   0   0   0]
 [  0   1   0   0]
 [  0   0 100   0]
 [  0   0   0 100]]
```

[18]:
```
print("Posterior g1 vulnerability of C1:", measure.g_vuln.
  ↪posterior(G1, pi, C1))
print("Posterior g1 vulnerability of C3:", measure.g_vuln.
  ↪posterior(G1, pi, C3))
print("Posterior g2 vulnerability of C1:", measure.g_vuln.
  ↪posterior(G2, pi, C1))
print("Posterior g2 vulnerability of C3:", measure.g_vuln.
  ↪posterior(G2, pi, C3))
```

```
Posterior g1 vulnerability of C1: 25.0
Posterior g1 vulnerability of C3: 25.45
Posterior g2 vulnerability of C1: 55.0
Posterior g2 vulnerability of C3: 48.33333333333333
```

Notice how adversary g_1 prefers C_3 because it has the highest vulnerability between the two whereas the best choice for g_2 is C_1.

3. CASE STUDIES

3.1 Voting systems

In the following section, we take concepts presented in previous sections of this work and apply them to the real world scenario of elections and voting systems.

3.1.1 Modeling elections as channels

We are going to start with a simple example to set the scene. Consider the smallest election ever with just 3 voters and 2 candidates. How can we model that using *QIF* terminology?

[2]:
```
num_voters = 3
num_candidates = 2
```

We know that a channel takes as input something secret and outputs something public. So what is secret in an election? The votes. But we have 3 persons voting. And each person has 2 possible options for his vote. So in total there are the following 2^3 scenarios (c_1 and c_2 below stand for *candidate 1* and *candidate 2* respectively):

```
X = {
    c1 c1 c1,
    c1 c1 c2,
    c1 c2 c1,
    c1 c2 c2,
    c2 c1 c1,
    c2 c1 c2,
    c2 c2 c1,
    c2 c2 c2,
    }
```

And that is how we can model the votes as our channel's input X.

[3]:
```
num_combinations = num_candidates ** num_voters
```

But what would the channel's output be? It depends. It could be just the name of the winning candidate; that is, the candidate which received the most votes. Or it could be for each candidate, the votes they received. Or maybe something else.

For the first scenario, where only the winning candidate is announced, the possible values for output Y are the candidates themselves, i.e.:

```
Y = { c1, c2 }
```

We model this scenario using channel matrix W.

[4]:
```
W = get_W(num_voters, num_candidates)
print(W)
```

Table 3.1: W **matrix**

W	c_1	c_2
$c_1c_1c_1$	1	0
$c_1c_1c_2$	1	0
$c_1c_2c_1$	1	0
$c_1c_2c_2$	0	1
$c_2c_1c_1$	1	0
$c_2c_1c_2$	0	1
$c_2c_2c_1$	0	1
$c_2c_2c_2$	0	1

```
[[1 0]
 [1 0]
 [1 0]
 [0 1]
 [1 0]
 [0 1]
 [0 1]
 [0 1]]
```

For the second scenario, where the votes for each candidate are announced, the possible values for output Y are:

```
Y = { (3,0), (2,1), (1,2), (0,3) }
```

We model this scenario using channel matrix C.

Table 3.2: C **matrix**

C	$(3,0)$	$(2,1)$	$(1,2)$	$(0,3)$
$c_1c_1c_1$	1	0	0	0
$c_1c_1c_2$	0	1	0	0
$c_1c_2c_1$	0	1	0	0
$c_1c_2c_2$	0	0	1	0
$c_2c_1c_1$	0	1	0	0
$c_2c_1c_2$	0	0	1	0
$c_2c_2c_1$	0	0	1	0
$c_2c_2c_2$	0	0	0	1

[5]:
```
C = get_C(num_voters, num_candidates)
print(C)
```

```
[[1 0 0 0]
 [0 1 0 0]
 [0 1 0 0]
```

```
 [0 0 1 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 1 0]
 [0 0 0 1]]
```

We also assume no prior knowledge about which voter votes for which candidate, so we use a uniform prior.

[6]:
```
pi = probab.uniform(num_combinations)
print(pi)
```

```
[0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125]
```

3.1.2 Computing the vulnerability of W

If we take a look at the hyper distribution of W, we see that each result happens with the same probability and upon observing it, the possible voting combinations have also happen with the same probability.

[7]:
```
from print_hyper import print_hyper
print_hyper(W, pi)
```

```
-------------------
|    0.50    0.50 |
-------------------
|    0.25    0.00 |
|    0.25    0.00 |
|    0.25    0.00 |
|    0.00    0.25 |
|    0.25    0.00 |
|    0.00    0.25 |
|    0.00    0.25 |
|    0.00    0.25 |
-------------------
```

So it is natural to expect that W's vulnerability is 0.25 that is, two times the channel's prior vulnerability (which was equal to 0.125).

[8]:
```
print("Prior bayes vulnerability:", measure.bayes_vuln.prior(pi))
print("Posterior bayes vulnerability of W:", measure.bayes_vuln.
 ↪posterior(pi, W))
print("Multiplicative leakage of W:", measure.bayes_vuln.
 ↪mult_leakage(pi, W))
```

```
Prior bayes vulnerability: 0.125
Posterior bayes vulnerability of W: 0.25
Multiplicative leakage of W: 2.0
```

3.1.3 Computing the vulnerability of C

If we do the same thing with C, we see that here the different values of y are not equally likely and the resulting posterior vulnerability is 0.5 that is, 4 times its prior vulnerability.

```
[9]: from print_hyper import print_hyper
     print_hyper(C, pi)
```

```
---------------------------------------
|    0.12    0.38    0.38    0.12 |
---------------------------------------
|    1.00    0.00    0.00    0.00 |
|    0.00    0.33    0.00    0.00 |
|    0.00    0.33    0.00    0.00 |
|    0.00    0.00    0.33    0.00 |
|    0.00    0.33    0.00    0.00 |
|    0.00    0.00    0.33    0.00 |
|    0.00    0.00    0.33    0.00 |
|    0.00    0.00    0.00    1.00 |
---------------------------------------
```

```
[10]: print("Prior bayes vulnerability:", measure.bayes_vuln.prior(pi))
      print("Posterior bayes vulnerability of C:", measure.bayes_vuln.
       ↪posterior(pi, C))
      print("Multiplicative leakage of C:", measure.bayes_vuln.
       ↪mult_leakage(pi, C))
```

```
Prior bayes vulnerability: 0.125
Posterior bayes vulnerability of C: 0.5
Multiplicative leakage of C: 4.0
```

But as someone might expect, these numbers get very small really fast as we consider more voters. That happens because the possible values for X, i.e.the voting combinations, grow exponentially to the number of voters. For example for an election of 10 voters and 2 candidates we have:

```
[11]: num_voters = 10
      num_candidates = 2
      num_combinations = num_candidates ** num_voters
```

```
[12]: print("Prior bayes vulnerability:", measure.bayes_vuln.prior(probab.
       ↪uniform(num_combinations)))
      print()
      print("W Posterior bayes vulnerability:", measure.bayes_vuln.
       ↪posterior(probab.uniform(num_combinations), get_W(num_voters,␣
       ↪num_candidates)))
```

```
print("W Multiplicative leakage:", measure.bayes_vuln.
 ↪mult_leakage(probab.uniform(num_combinations), get_W(num_voters,␣
 ↪num_candidates)))
print()
print("C Posterior bayes vulnerability:", measure.bayes_vuln.
 ↪posterior(probab.uniform(num_combinations), get_C(num_voters,␣
 ↪num_candidates)))
print("C Multiplicative leakage:", measure.bayes_vuln.
 ↪mult_leakage(probab.uniform(num_combinations), get_C(num_voters,␣
 ↪num_candidates)))
```

```
Prior bayes vulnerability: 0.0009765625

W Posterior bayes vulnerability: 0.001953125
W Multiplicative leakage: 2.0

C Posterior bayes vulnerability: 0.0107421875
C Multiplicative leakage: 11.0
```

Note that W's multiplicative leakage is again 2, but for C it has jumped up to 11. C still leaks more than W.

3.1.4 Comparing the two channels

Let's see the bigger picture using a graph with the number of voters ranging from 1 to 12.

[13]:
```
c = 2 # number of candidates
vs = [v for v in range(1, 12)]
```

[14]:
```
plt.plot(vs, [measure.bayes_vuln.posterior(probab.uniform(c ** v),␣
 ↪get_W(v, c)) for v in vs], label="W")
plt.plot(vs, [measure.bayes_vuln.posterior(probab.uniform(c ** v),␣
 ↪get_C(v, c)) for v in vs], label="C")
plt.xlabel('number of voters')
plt.ylabel('Posterior bayes vulnerability of W and C')
plt.legend()
None
```

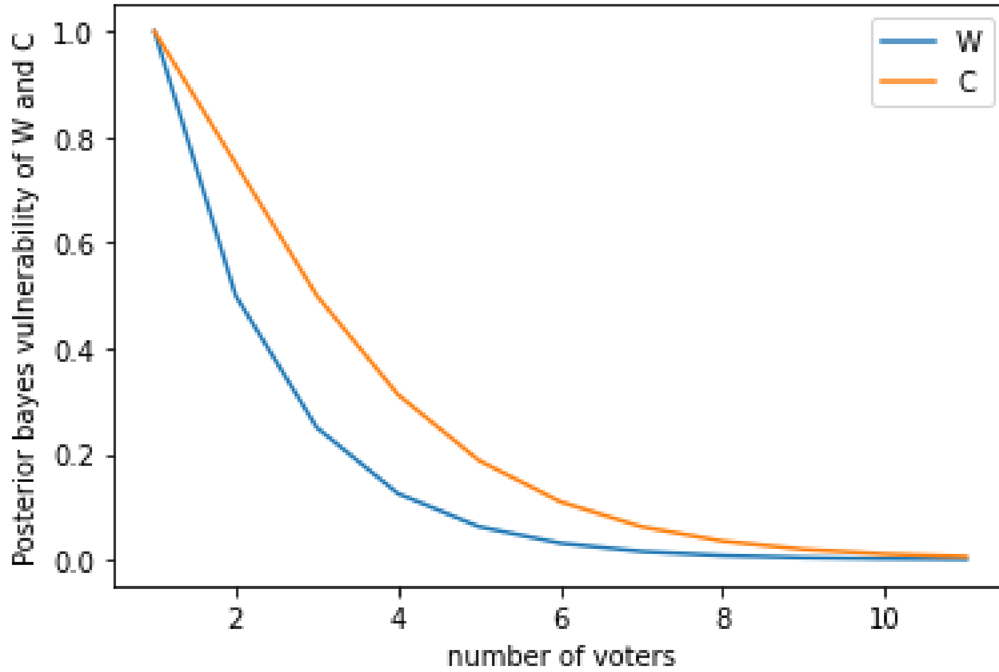

Figure 3.1: Posterior Bayes Vulnerability of W and C

Here we see that C exposes always more than W does. And that would not be a surprise to someone if they have observed that C is a post processing of W or in other words, C is a refinement of W.

That can also be realized more intuitevly if you think of the process of announcing the winner of the election. First the votes for each candidate are counted (which is what W does), then they are compared to see who has the most, and finally the winner's name is annnounced. So the process of comparing the votes for each candidate and deciding who has the most is the post processing of W.

This can also be done using a post process matrix R.

[15]:
```
plt.plot(vs, [measure.bayes_vuln.mult_leakage(probab.uniform(c **␣
 ↪v), get_W(v, c)) for v in vs], label="W")
plt.plot(vs, [measure.bayes_vuln.mult_leakage(probab.uniform(c **␣
 ↪v), get_C(v, c)) for v in vs], label="C")
plt.xlabel('number of voters')
plt.ylabel('Multiplicative bayes leakage of W and C')
plt.legend()
None
```

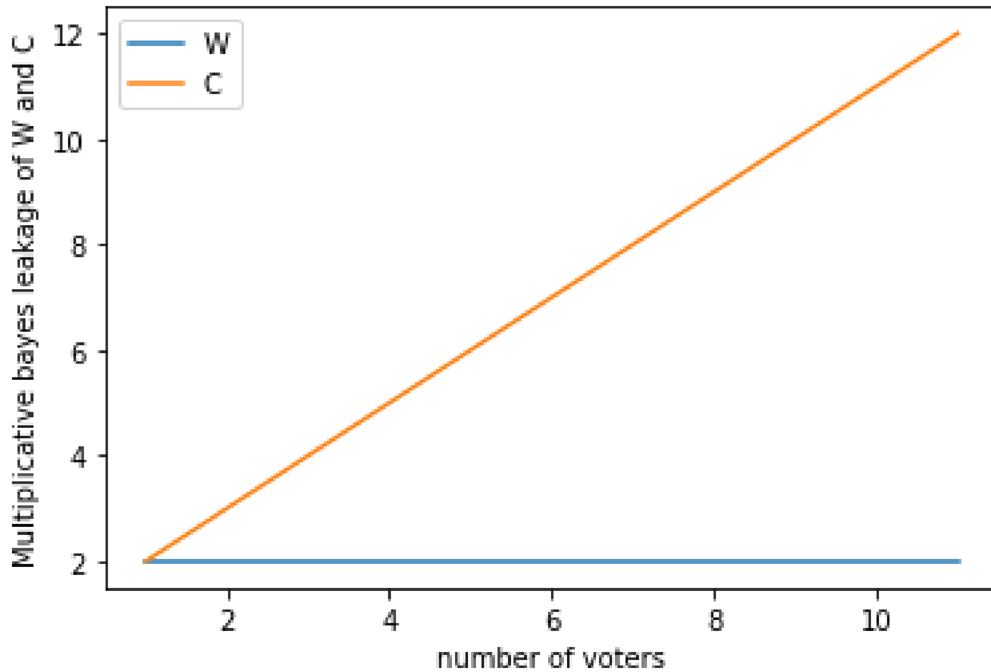

Figure 3.2: Multiplicative Bayes Leakage of W and C

Another interesting fact comes from observing each channel's multiplicative leakage. While W's leakage stays 2 no matter the number of voters, C' leakage increases linearly. This comes from the fact that as the number of voters increases, the number of possible outcomes also increases ending up with a more detailed partition of Y. Thus we can make a better guess once we observe that specific result.

But in general both posterior vulnerabilities drop down to 0 quite fast. So from around 6 voters or more we could argue that our secret is quite safe. **But safe against who?** Until now we used bayes vulnerability which corresponds to an adversary trying to correctly guess *the whole voting combination* that occured. But that is usually not the case.

3.1.5 Different Adversarial Models

We are now going to take into account two different, more realistic adversaries. We are going to do that through g vulnerability.

3.1.5.1 Adversary 1

Suppose that an adversary will benefit from guessing correctly how some voter voted, but their benefit does not depend on which voter they choose or who the voter voted for: any victim, and any vote will be a gain for them. We define for that a gain function g_1 which allows an adversary to choose from action set $Candidates \times Voters$, where (c, v) is their action of guessing that candidate c was selected by voter v.

g_1 can be defined as:

$$g_1((c,v),z) = \begin{cases} 1 & \texttt{if } c = z(v) \\ 0 & \texttt{if } c \neq z(v) \end{cases}$$

where $z(v)$ is the candidate voted for by voter v in voting pattern z.

If we construct G_1 with the possible actions as rows and the possible values of X as columns we get:

Table 3.3: g_1 function defined with a matrix

G_1	$c_1c_1c_1$	$c_1c_1c_2$	$c_1c_2c_1$	$c_1c_2c_2$	$c_2c_1c_1$	$c_2c_1c_2$	$c_2c_2c_1$	$c_2c_2c_2$
(c_1, e_1)	1	1	1	1	0	0	0	0
(c_1, e_2)	1	1	0	0	1	1	0	0
(c_1, e_3)	1	0	1	0	1	0	1	0
(c_2, e_1)	0	0	0	0	1	1	1	1
(c_2, e_2)	0	0	1	1	0	0	1	1
(c_2, e_3)	0	1	0	1	0	1	0	1

[2]:
```
num_voters = 3
num_candidates = 2
num_combinations = num_candidates ** num_voters
```

[3]:
```
G1 = get_G1(num_voters, num_candidates)
print(G1)
```

```
[[1 1 1 1 0 0 0 0]
 [1 1 0 0 1 1 0 0]
 [1 0 1 0 1 0 1 0]
 [0 0 0 0 1 1 1 1]
 [0 0 1 1 0 0 1 1]
 [0 1 0 1 0 1 0 1]]
```

The posterior vulnerability under g_1 gives us the probability that an adversary correctly guesses what some voter voted for. For channel matrix W (as defined in the previous part) it is equal to:

[4]:
```
pi = probab.uniform(num_combinations)
```

[5]:
```
W = get_W(num_voters, num_candidates)
```

[6]:
```
print("Prior g vulnerability:", measure.g_vuln.prior(G1, pi))
print("Posterior g vulnerability of W:", measure.g_vuln.
 ↪posterior(G1, pi, W))
print("Multiplicative g leakage of W:", measure.g_vuln.
 ↪mult_leakage(G1, pi, W))
```

```
Prior g vulnerability: 0.5
Posterior g vulnerability of W: 0.75
Multiplicative g leakage of W: 1.5
```

If we do the same with channel matrix C, we observe the exact same results.

```
[7]: C = get_C(num_voters, num_candidates)
```

```
[8]: print("Prior g vulnerability:", measure.g_vuln.prior(G1, pi))
     print("Posterior g vulnerability of C:", measure.g_vuln.
      ↪posterior(G1, pi, C))
     print("Multiplicative g leakage of C:", measure.g_vuln.
      ↪mult_leakage(G1, pi, C))
```

```
Prior g vulnerability: 0.5
Posterior g vulnerability of C: 0.75
Multiplicative g leakage of C: 1.5
```

As it turns out there is a theorem adressing the more general case of any number of voters which states that under the uniform prior π we have that:

$$\mathcal{L}_{g_1}^{\times}(\pi, W) = \mathcal{L}_{g_1}^{\times}(\pi, C)$$

We can verify that results by observing the following plots.

```
[9]: c = 3 # number of candidates
     vs = [v for v in range(1, 10)]
```

```
[10]: plt.plot(vs, [measure.g_vuln.mult_leakage(get_G1(v, c), probab.
       ↪uniform(c ** v), get_W(v, c)) for v in vs])
      plt.xlabel('number of voters')
      plt.ylabel('Mupliplicative leakage of W')
      None
```

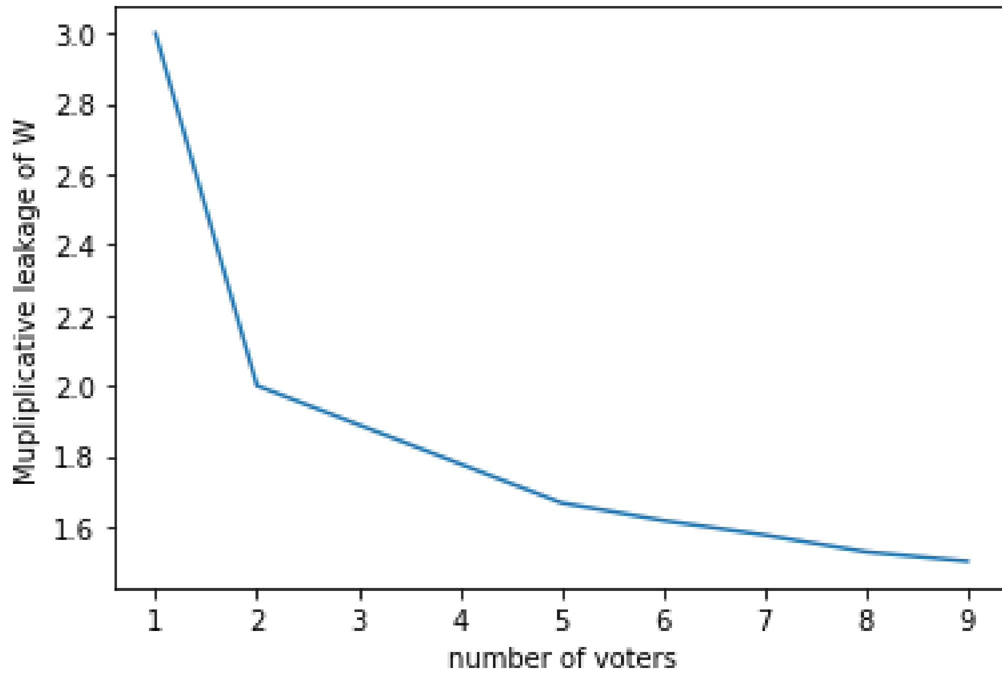

Figure 3.3: Multiplicative Leakage of W

[11]:
```
plt.plot(vs, [measure.g_vuln.mult_leakage(get_G1(v, c), probab.
  ↪uniform(c ** v), get_C(v, c)) for v in vs])
plt.xlabel('number of voters')
plt.ylabel('Mupliplicative leakage of C')
None
```

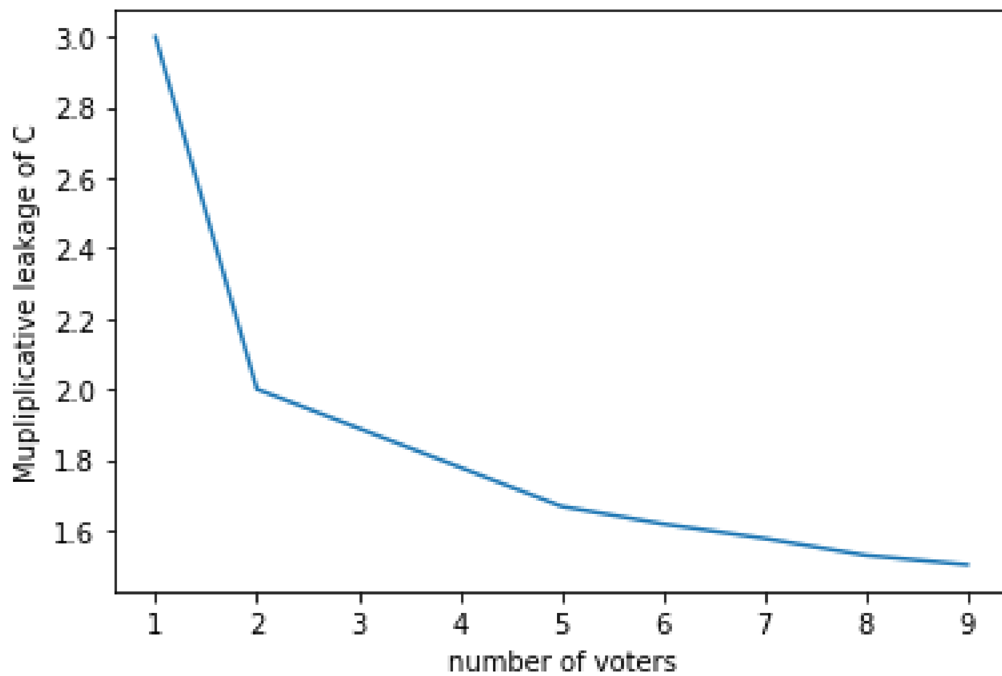

Figure 3.4: Multiplicative Leakage of C

Intuitevly this can be understood by the following thought process.

> *For any announcement of tallies for an election whose voting pattern is z, most voters voted for the candidate with the majority, and so the adversary's optimum guessing strategy is to pick $(maj(z), v)$ for any voter v, which is exactly the same guessing strategy if only the winner is announced. Hence since the optimal guessing strategies are the same for both W and C, the leakage with respect to g_1 must also be the same.*

3.1.5.2 Adversary 2

Another adversary might benefit if they can find a voter/candidate pair such that the voter did not vote for that candidate. For this adversary we define g_0 as follows:

$$g_0((c, v), z) = \begin{cases} 1 & \texttt{if } c \neq z(v) \\ 0 & \texttt{if } c = z(v) \end{cases}$$

[12]:
```
G0 = get_G0(num_voters, num_candidates)
print(G0)
```

```
[[0 0 0 0 1 1 1 1]
 [0 0 1 1 0 0 1 1]
 [0 1 0 1 0 1 0 1]
 [1 1 1 1 0 0 0 0]
 [1 1 0 0 1 1 0 0]
 [1 0 1 0 1 0 1 0]]
```

Notice that G_0 is the complement of G_1, meaning that it results from switching the 1s with 0s and vice versa.

[13]:
```
print("Prior g vulnerability:", measure.g_vuln.prior(G0, pi))
print("Posterior g vulnerability of W:", measure.g_vuln.
 ↪posterior(G0, pi, W))
print("Multiplicative g leakage of W:", measure.g_vuln.
 ↪mult_leakage(G0, pi, W))
```

```
Prior g vulnerability: 0.5
Posterior g vulnerability of W: 0.75
Multiplicative g leakage of W: 1.5
```

[14]:
```
print("Prior g vulnerability:", measure.g_vuln.prior(G0, pi))
print("Posterior g vulnerability of C:", measure.g_vuln.
 ↪posterior(G0, pi, C))
print("Multiplicative g leakage of C:", measure.g_vuln.
 ↪mult_leakage(G0, pi, C))
```

```
Prior g vulnerability: 0.5
Posterior g vulnerability of C: 0.75
Multiplicative g leakage of C: 1.5
```

Again, the vulnerability and leakage of C and W look the same. But as it turns out, for 3 candidates or more, this type of adversary has more to gain from elections where they announce the votes for each candidate (i.e. channel matrix C). Translated into QIF, it basically means that

$$\mathcal{L}^{\times}_{g_0}(\pi, W) < \mathcal{L}^{\times}_{g_0}(\pi, C)$$

for $|Candidates| \geq 3$.

The following plot vefiries that. Remember that the posterior vulnerability under g_0 gives us the probability that an adversary correctly guesses what some voter **did not** vote for.

```
[15]: c = 3 # number of candidates
      vs = [v for v in range(1, 10)]
```

```
[16]: plt.plot(vs, [measure.g_vuln.mult_leakage(get_G0(v, c), probab.
       ↪uniform(c ** v), get_W(v, c)) for v in vs], label="W")
      plt.plot(vs, [measure.g_vuln.mult_leakage(get_G0(v, c), probab.
       ↪uniform(c ** v), get_C(v, c)) for v in vs], label="C")
      plt.xlabel('number of voters')
      plt.ylabel('Muptiplicative leakages of W and C')
      plt.legend()
      None
```

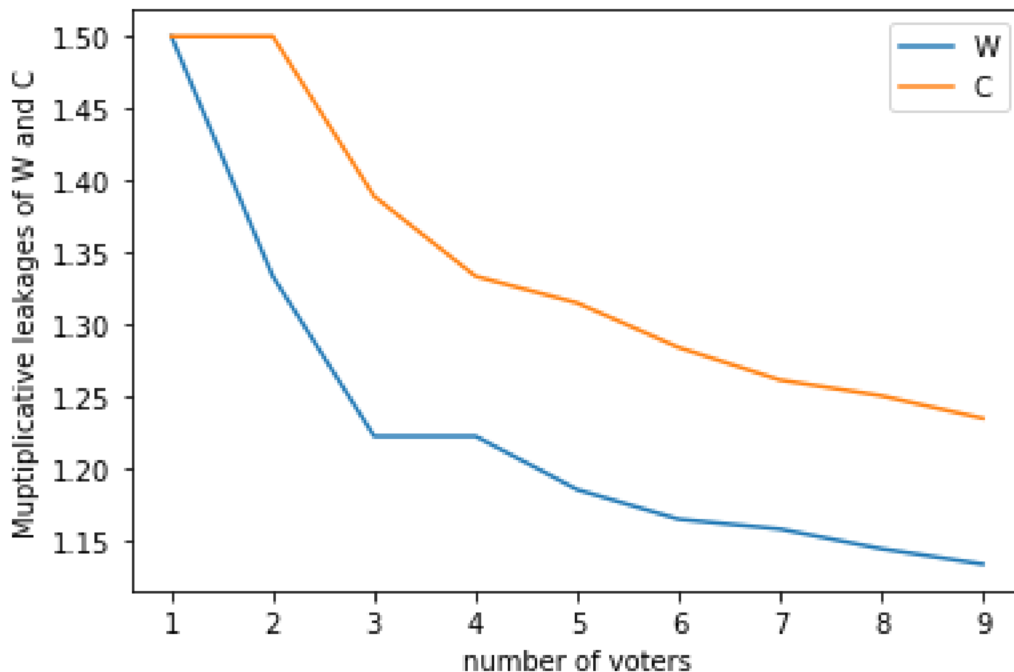

Figure 3.5: Multiplicative Leakages of W and C

This result also has a more intuitive explanation.

When tallies are released, the adversary can increase their gain by guessing that the candidate who received the least number of votes is most likely not someone the voter selected. That information is not available in W*.*

3.2 Differential Privacy

Differential Privacy and *Quantitative Information Flow* can be seen as having essentially the same goal, namely to control the leakage of sensitive information. In this section, we are going to try to explore similarities and differences bewteen the two approaches.

3.2.1 An example scenario

Assume we are interested in the eye color of a certain population $I = \{Alice, Bob, Charlie\}$. Let the possible values for each person in I be defined by the set $V = \{a, b, g\}$, where a stands for absent (i.e. the person is not in this specific database), b stands for *black* and g for *green*. Each dataset is a tuple $x_0x_1x_2 \in V^2$ where x_0 represents the eye color of *Alice*, x_1 of *Bob* and x_2 of *Charlie*.

The possible values of X are

```
X = {
    aaa, aab, aag,
    aba, abb, abg,
    aga, agb, agg,
    baa, bab, bag,
         ...
    gga, ggb, ggg
    }
```

Consider the following counting query.

```
SELECT COUNT(*)
FROM X
WHERE eye_color = 'b';
```

Its possible output values are

```
Y = {0, 1, 2, 3}
```

We can model this query as a deterministic channel f as below.

Table 3.4: f function defined with a matrix

f	0	1	2	3
aaa	1	0	0	0
aab	0	1	0	0
aag	1	0	0	0
aba	0	1	0	0
...	...	...	...	...
ggg	1	0	0	0

Now insted of reporting the true answer y, we process it further by passing it through a noise channel H and report a slightly different answer z.

Here are going to use the following mechanism for H.

Table 3.5: H matrix

H	0	1	2	3
0	$\frac{3}{4}$	$\frac{1}{6}$	$\frac{1}{18}$	$\frac{1}{36}$
1	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{12}$
2	$\frac{1}{12}$	$\frac{1}{6}$	$\frac{1}{2}$	$\frac{1}{4}$
3	$\frac{1}{36}$	$\frac{1}{18}$	$\frac{1}{6}$	$\frac{3}{4}$

What H does is basically add noise to the true answer of f and it does that by using the (truncated) geometric mechanism with parameter $a = \frac{1}{3}$. One thing to notice here is that the true answer of f has the highest probability within its row.

So the whole channel, from X to Z, i.e. from the database to the fuzzy query answer, can be depicted as below.

Table 3.6: C matrix

C	0	1	2	3
aaa	$\frac{3}{4}$	$\frac{1}{6}$	$\frac{1}{18}$	$\frac{1}{36}$
aab	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{12}$
aag	$\frac{3}{4}$	$\frac{1}{6}$	$\frac{1}{18}$	$\frac{1}{36}$
aba	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{12}$
...	...	...	...	...
ggg	$\frac{3}{4}$	$\frac{1}{6}$	$\frac{1}{18}$	$\frac{1}{36}$

[2]:
```
num_persons = 3
values = ['a', 'b', 'g']
num_values = len(values)
query_value = 'b'
```

[3]:
```
C = get_C(num_persons, values, query_value)
print(C)
```

```
[[0.75       0.16666667 0.05555556 0.02777778]
 [0.25       0.5        0.16666667 0.08333333]
 [0.75       0.16666667 0.05555556 0.02777778]
 [0.25       0.5        0.16666667 0.08333333]
 [0.08333333 0.16666667 0.5        0.25      ]
 [0.25       0.5        0.16666667 0.08333333]
 [0.75       0.16666667 0.05555556 0.02777778]
 [0.25       0.5        0.16666667 0.08333333]
 [0.75       0.16666667 0.05555556 0.02777778]
 [0.25       0.5        0.16666667 0.08333333]
 [0.08333333 0.16666667 0.5        0.25      ]
```

```
 [0.25       0.5        0.16666667 0.08333333]
 [0.08333333 0.16666667 0.5        0.25      ]
 [0.02777778 0.05555556 0.16666667 0.75      ]
 [0.08333333 0.16666667 0.5        0.25      ]
 [0.25       0.5        0.16666667 0.08333333]
 [0.08333333 0.16666667 0.5        0.25      ]
 [0.25       0.5        0.16666667 0.08333333]
 [0.75       0.16666667 0.05555556 0.02777778]
 [0.25       0.5        0.16666667 0.08333333]
 [0.75       0.16666667 0.05555556 0.02777778]
 [0.25       0.5        0.16666667 0.08333333]
 [0.08333333 0.16666667 0.5        0.25      ]
 [0.25       0.5        0.16666667 0.08333333]
 [0.75       0.16666667 0.05555556 0.02777778]
 [0.25       0.5        0.16666667 0.08333333]
 [0.75       0.16666667 0.05555556 0.02777778]]
```

Notice that Z can also occur from the matrix multiplication of f and H, that is

$$f \cdot H = Z$$

The following image graphically depicts the whole setting (ingore the notions of leakage and utility for now).

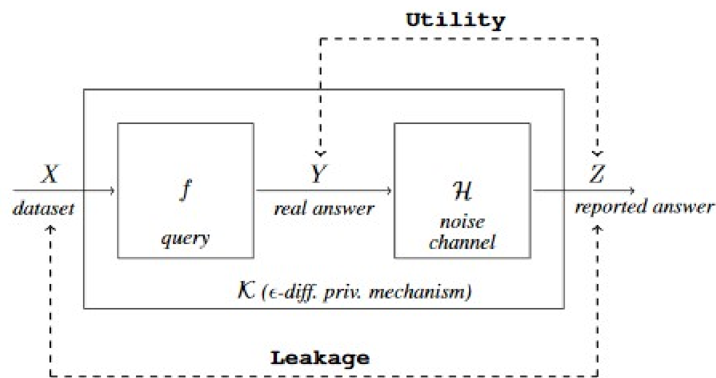

Figure 3.6: Leakage and utility for oblivious mechanisms

3.2.2 Assesing information leakage through *QIF*

To measure the leakage with *QIF* we must first define the prior distribution π over X. If we don't have any particular knowledge about it we use the uniform distribution.

```
[4]: pi = probab.uniform(num_persons ** num_values)
     print(pi)
```

```
[0.03703704 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704
 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704
 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704
```

```
 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704
 0.03703704 0.03703704 0.03703704]
```

Next, we compute the posterior distributions which depend both on C and π.

[5]:
```
from print_hyper import print_hyper
print_hyper(C, pi)
```

```
---------------------------------------
|    0.35    0.31    0.21    0.13 |
---------------------------------------
|    0.08    0.02    0.01    0.01 |
|    0.03    0.06    0.03    0.02 |
|    0.08    0.02    0.01    0.01 |
|    0.03    0.06    0.03    0.02 |
|    0.01    0.02    0.09    0.07 |
|    0.03    0.06    0.03    0.02 |
|    0.08    0.02    0.01    0.01 |
|    0.03    0.06    0.03    0.02 |
|    0.08    0.02    0.01    0.01 |
|    0.03    0.06    0.03    0.02 |
|    0.01    0.02    0.09    0.07 |
|    0.03    0.06    0.03    0.02 |
|    0.01    0.02    0.09    0.07 |
|    0.00    0.01    0.03    0.22 |
|    0.01    0.02    0.09    0.07 |
|    0.03    0.06    0.03    0.02 |
|    0.01    0.02    0.09    0.07 |
|    0.03    0.06    0.03    0.02 |
|    0.08    0.02    0.01    0.01 |
|    0.03    0.06    0.03    0.02 |
|    0.08    0.02    0.01    0.01 |
|    0.03    0.06    0.03    0.02 |
|    0.01    0.02    0.09    0.07 |
|    0.03    0.06    0.03    0.02 |
|    0.08    0.02    0.01    0.01 |
|    0.03    0.06    0.03    0.02 |
|    0.08    0.02    0.01    0.01 |
---------------------------------------
```

For each column of the matrix above, i.e. each possible outcome z, **QIF** models the threat as the highest probability within that column, i.e. the probability of sucess of the best possible guess for which database x we are dealing with. So we pick the maximum probability for each column and then **we weigh** each one with its respective outer probability, i.e. the probability of each z occuring. And the result is the vulnerability of C.

[6]:
```
print("QIF posterior vulnerability:", measure.bayes_vuln.
 ↪posterior(pi, C))
```

```
QIF posterior vulnerability: 0.09259259259259259
```

3.2.3 Assesing information leakage through *Differential Privacy*

Differential privacy works a bit differently.

First of all, it uses the notion of *adjacent* or *neighbor* databases which is used to indicate two databases that differ in the presence of, or in the value associated with, exactly one individual. We use $x_1 \sim x_2$ to indicate that x_1 and x_2 are adjacent. For example $bbg \sim bag$ and $aba \sim bba$ but $bba \nsim bgb$.

Now, for each column of C, i.e. each possible outcome z, ϵ**-differential privacy** is satisfied if there exists an $\epsilon \geq 0$ such that

$$\frac{C_{x,z}}{C_{x',z}} \leq e^{\epsilon}$$

for all x, x' in X such that $x \sim x'$.

From each column of C, we keep the biggest ϵ so that the above inequality holds for all columns. This way, ϵ represents the level of privacy of the whole channel.

[7]:
```
# The following function overestimates the real value of epsilon,
# but provides an upper bound for it.
print("Differential Privacy epsilon:", get_worst_epsilon(C))
```

```
Differential Privacy epsilon: 3.295836866004329
```

Let's verify that indeed that is the worst-case ϵ by observing the ϵ values for each column of C.

[8]:
```
for i in range(num_values+1):
    print("epsilon for column", i, "=", get_worst_epsilon(C, i))
```

```
epsilon for column 0 = 3.295836866004329
epsilon for column 1 = 2.1972245773362196
epsilon for column 2 = 2.1972245773362196
epsilon for column 3 = 3.295836866004329
```

Another way to think about it is that for each column of C, differential privacy measures the threat as the biggest difference between two adjacent x

3.2.4 Comparing the two approaches

First of all let's consider the basic ideas behind each approach.

QIF vulnerablity measures the probability that an adversary has of correctly guessing the secret x (i.e. the whole database in our case) upon observing the channel's output z.

ϵ**-differential privacy**'s basic idea on the other hand, is that that the presence or absence of any individual in a database, or changing the data of any individual,

should not significantly affect the probability of obtaining any specific answer for a certain query.

So clearly, they don't have the same goal or the same adversary in mind.

Going a little further we see that *QIF* vulnerability is sensitive to the prior distribution of X whereas ϵ-differential privacy is not.

For example consider the uniform and point distrubtions below.

[9]:
```
pi1 = probab.uniform(num_persons ** num_values)
print("pi1\n", pi1, "\n")
pi2 = probab.point(num_persons ** num_values)
print("pi2\n", pi2)
```

```
pi1
 [0.03703704 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704
 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704
 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704
 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704 0.03703704
 0.03703704 0.03703704 0.03703704]


pi2
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 ↪ 0.
 0. 0. 0.]
```

If we measure the information leakage thrgouh *QIF* for both cases we get:

[10]:
```
print("QIF posterior vulnerability for pi1:", measure.bayes_vuln.
 ↪posterior(pi1, C))
print("QIF posterior vulnerability for pi2:", measure.bayes_vuln.
 ↪posterior(pi2, C))
```

```
QIF posterior vulnerability for pi1: 0.09259259259259259
QIF posterior vulnerability for pi2: 1.0
```

But differential privacy does not consider the prior distribution of X, so we get:

[11]:
```
print("Differential Privacy epsilon for pi1:", get_worst_epsilon(C))
print("Differential Privacy epsilon for pi2:", get_worst_epsilon(C))
```

```
Differential Privacy epsilon for pi1: 3.295836866004329
Differential Privacy epsilon for pi2: 3.295836866004329
```

Which is also obvious from the fact that the `get_worst_epsilon` function does not take a `pi` parameter.

Another difference between the two is that *QIF* vulnerability is defined as the result of averagind the contribution of all the columns to the vulnerability, while differential privacy represents the worst case (i.e.the maximum ϵ for all z).

Hence there could be a column with a very high ϵ value which does not contribute very much to the average (typically because the corresponding output has very low probability of happening). In that case, *QIF* vulnerability could be very small, and still ϵ-differential privacy would have a really big ϵ value.

4. CONCLUSIONS

With this thesis, we have provided individuals interested in learning about *QIF* with an interactive way of getting in touch with the subject and exploring different scenarios where its theory can be applied. Since there are not many additional sources online for studying QIF, there is still room for additional tutorials or articles about it, making it more accessible to the interested reader.

ABBREVIATIONS - ACRONYMS

QIF	Quantitative Information Flow

BIBLIOGRAPHY

[1] Mário S. Alvim, Konstantinos Chatzikokolakis, Annabelle McIver, Carroll Morgan, Catuscia Palamidessi, and Geoffrey S. Smith. *The Science of Quantitative Information Flow*. Springer International Publishing, 2020.