Finding Informative Features

36-462/662: Data Mining, Spring 2020

Lecture 10, 13 February 2020

Contents

Everything we have learned how to do so far — similarity searching, nearest-neighbor and prototype classification, multidimensional scaling — relies on our having a vector of **features** or **attributes** for each object in data set. (The dimensionality of vector space equals the number of features.) The success of our procedures depends on our choosing good features, but I've said very little about how to do this. In part this is because designing good representations inevitably depends on domain knowledge. However, once we've picked a set of features, they're not all necessarily equally useful, and there are some tools for quantifying that.

The basic idea, remember, is that the features are the aspects of the data which show up in our representation. However, they're not what we *really* care about, which is rather something we don't, or can't, directly represent, for instance the **class** of the object (is it a story about art or about music? a

picture of a flower or a tiger?). We use the observable features to make a guess (formally, an **inference**) about the unobservable thing, like the class. Good features are ones which let us make better guesses — ones which reduce our **uncertainty** about the unobserved class.

Good features are therefore **informative**, **discriminative** or **uncertainty-reducing**. This means that they need to *differ* across the different classes, at least statistically. The frequency of occurrences of the word "the" in an English document isn't a useful feature, because it occurs about as often in all kinds of text. This means that looking at that count leaves us exactly as uncertain about which class of document we've seen as we were before. Similarly, the word "cystine" is going to be equally *rare* whether the topic is art or music, so it's also uninformative. On the other hand, the word "rhythm" is going to be more common in stories about music than in ones about art, so counting its occurrences *is* going to reduce our uncertainty. The important thing is that the distribution of the feature *differ* across the classes.

1 Entropy and Information

Information theory is one way of trying to make precise these ideas about uncertainty, discrimination, and reduction in uncertainty. (Information theory has many other uses, and is at once one of the great intellectual achievements of the twentieth century and a key technology of the world around us. But we'll just look at this aspect.) X is some feature of the data in our representation, and x is a particular value of the feature. How uncertain are we about X? Well, one way to measure this is the **entropy** of X:

$$H[X] = -\sum_{x} \Pr(X = x) \log_2 \Pr(X = x) \tag{1}$$

The entropy, in bits, equals the average number of yes-or-no questions we'd have to ask to figure out the value of X. (This is also the number of bits of computer memory needed to store the value of X.) If there are n possible values for X, and they are all equally likely, then our uncertainty is maximal, and $H[X] = \log_2 n$, the maximum possible value. If X can take only one value, we have no uncertainty, and $H[X] = 0$.

Similarly, our uncertainty about the class C, in the absence of any other information, is just the entropy of C:

$$H[C] = -\sum_{c} \Pr(C = c) \log_2 \Pr(C = c) \tag{2}$$

Now suppose we observe the value of the feature X. This will, in general, change our distribution for C, since we can use Bayes's Rule:

$$\Pr(C = c|X = x) = \frac{\Pr(C = c, X = x)}{\Pr(X = x)} = \frac{\Pr(X = x|C = c)}{\Pr(X = x)} \Pr(C = c) \tag{3}$$

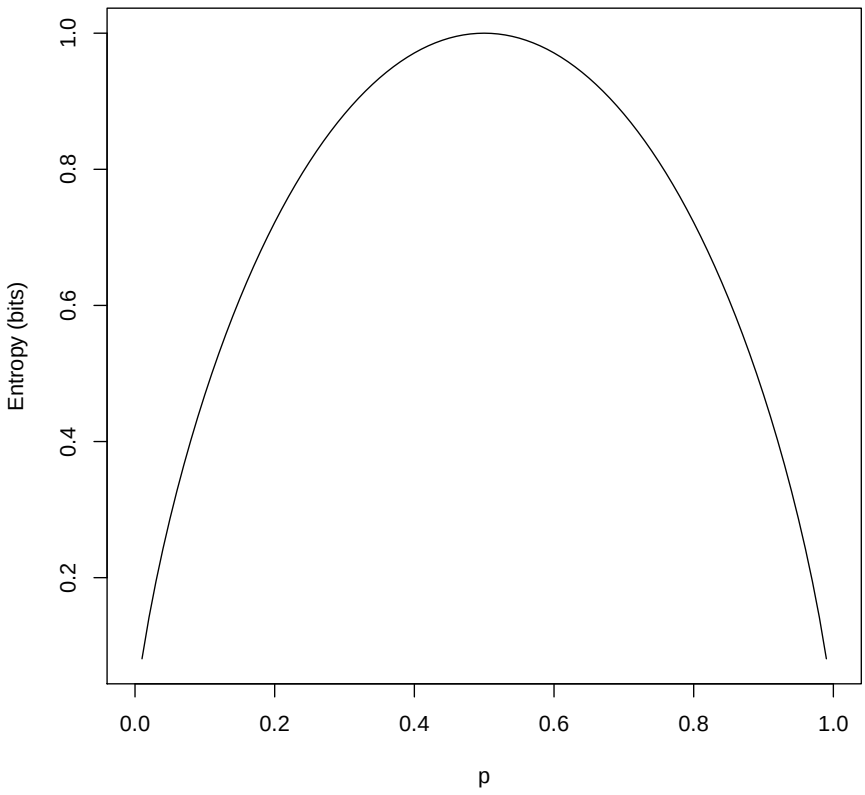


Figure 1: Entropy of a binary variable as a function of the probability of (either) class value. Note that it is symmetric around $p = 1/2$, where it is maximal.

$\Pr(X = x)$ tells us the frequency of the value x is over the whole population. $\Pr(X = x|C = c)$ tells us the frequency of that value is when the class is c. If the two frequencies are not equal, we should change our estimate of the class, making it larger if that feature is more common in c, and making it smaller if that feature is rarer. Generally, our uncertainty about C is going to change, and be given by the **conditional entropy**:

$$H[C|X = x] = -\sum_{c} \Pr(C = c|X = x) \log_2 \Pr(C = c|X = x) \tag{4}$$

The difference in entropies, $H[C] - H[C|X = x]$, is how much our uncertainty about C has changed, conditional on seeing $X = x$. This change in uncertainty

is **realized information**:

$$I[C; X = x] = H[C] - H[C|X = x] \tag{5}$$

Notice that the realized information can be negative. For a simple example, suppose that C is "it will rain today", and that it normally rains only one day out of seven. Then $H[C] = 0.59$ bits. If however we look up and see clouds ($X =$ cloudy), and we know it rains on half of the cloudy days, $H[C|X = \text{cloudy}] = 1$ bit, so our uncertainty has increased by 0.41 bits.

We can also look at the *expected* information a feature gives us about the class:

$$I[C; X] = H[C] - H[C|X] = H[C] - \sum_x \Pr(X = x) H[C|X = x] \tag{6}$$

The expected information is never negative. In fact, it's not hard to show that the only way it can even be zero is if X and C are **statistically independent** — if the distribution of X is the same for all classes c,

$$\Pr(X|C = c) = \Pr(X) \tag{7}$$

It's also called the **mutual information**, because it turns out that $H[C] - H[C|X] = H[X] - H[X|C]$. (You might want to try to prove this to yourself, using Bayes's rule and the definitions.)

1.1 Example: How Much Do Words Tell Us About Topics?

Let's look at this in a particular concrete data set. This is a collection of news articles from the *New York Times*, which have been extensively annotated with meta-data about things like date, authorship, keywords, and (importantly) their subject matter (Sandhaus, 2008)[1]. I selected a random sample of the stories about art, and another random sample of the stories about music. For each story, I took the text and converted it into a vector which counted the number of occurrences of each word[2]. The result is a feature vector for each news story, called the **bag-of-words** vector for that story. I've saved it all in a data frame called `nyt.frame` at `http://www.stat.cmu.edu/~cshalizi/dm/20/lectures/10/nyt.frame.csv`. It will be convenient to add the labels themselves as an extra column in the data frame:

```
nyt.frame <- read.csv("http://www.stat.cmu.edu/~cshalizi/dm/20/lectures/10/nyt.frame.csv")
dim(nyt.frame)
## [1]  102 4431
class.labels <- c(rep("art", 57), rep("music", 45))
nyt.frame <- data.frame(class.labels = as.factor(class.labels), nyt.frame)
dim(nyt.frame)
## [1]  102 4432
```

[1] We will work with this data set more extensively when we look at text mining and information retrieval.

[2] There was a certain amount of pre-processing having to do with things like punctuation, capitalization, and so forth. We'll go over those details later, when we do some text mining.

(Remember that `factor` is R's data type for categorical variables.)

C will be the class label, so its two possible values are "art" and "music". For our feature X, we will use whether or not a document contains the word "paint", i.e., whether the "paint" component of the bag-of-words vector is positive or not; $X = 1$ means the word is present, $X = 0$ that it's absent.[3] We can do the counting by hand, and get

	x	
c	"paint"	not "paint"
art	12	45
music	0	45

Let's calculate some entropies. We don't want to do this by hand, so let's write a function, `entropy`, to do so (Example 1).

```
# Calculate the entropy of a vector of counts or proportions Inputs: Vector of
# numbers Output: Entropy (in bits)
entropy <- function(p) {
    # Assumes: p is a numeric vector
    if (sum(p) == 0) {
        return(0)  # Case shows up when calculating conditional
        # entropies
    }
    p <- p/sum(p)  # Normalize so it sums to 1
    p <- p[p > 0]  # Discard zero entries (because 0 log 0 = 0)
    H <- -sum(p * log(p, base = 2))
    return(H)
}
```

Code Example 1: The `entropy` function.

Notice that we can either give this `entropy` function a vector of probabilities, or a vector of counts, which it will normalize to probabilities

```
entropy(c(0.5, 0.5))
## [1] 1
entropy(c(1, 1))
## [1] 1
entropy(c(45, 45))
## [1] 1
```

There are 57 art stories and 45 music stories, so:

```
entropy(c(57, 45))
## [1] 0.9899928
```

[3] X is thus an **indicator variable**.

In other words, $H[C] = 0.99$. Of course in general we don't want to put in the numbers like that; this is where the `class.labels` column of the data frame is handy:

```
table(nyt.frame[, "class.labels"])
##
##   art music
##    57    45
entropy(table(nyt.frame[, "class.labels"]))
## [1] 0.9899928
```

From the 2×2 table above, we can calculate that

- $H[C|X = \text{“paint”}] = 0$
- $H[C|X = \text{not “paint”}] = 1$
- $\Pr(X = \text{“paint”}) = 12/102 = 0.12$
- $I[C; X] = H[C] - (\Pr(X = 1) H[C|X = 1] + \Pr(X = 0) H[C|X = 0]) = 0.11$

In words, when we see the word “paint”, we can be certain that the story is about art ($H[C|X = \text{“paint”}] = 0$ bits). On the other hand, when “paint” is absent we are as uncertain as if we flipped a fair coin ($H[C|X = \text{not “paint”}] = 1.0$ bits), which is actually a bit more uncertainty than we'd have if we didn't look at the words at all ($H[C] = 0.99$ bits). Since “paint” isn't that common a word ($\Pr(X = \text{“paint”}) = 0.12$), the *expected* reduction in uncertainty is small but non-zero ($I[C; X] = 0.11$).

If we want to repeat this calculation for another word, we don't want to do all these steps by hand. It's a mechanical task so we should be able to encapsulate it in more code (Code Example 2).

If this works, it should agree with what we calculated by hand above:

```
word.mutual.info(matrix(c(12, 0, 45, 45), nrow = 2))
## [1] 0.1076399
```

which is exactly what the manual calculation gave before rounding off to two significant figures. (With about a hundred examples, it's nonsense to calculate *anything* to one part in a million.)

Now we can calculate the information a word gives us about a category so long as we can get indicator counts. Doing this manually is tedious, so again, let's automate (Code Example 3).

Again, let's double-check this:

```
word.class.indicator.counts(nyt.frame, "paint")
##       [,1] [,2]
## art     12   45
## music    0   45
```

```
# Get the expected information a word's indicator gives about a document's class
# Inputs: array of indicator counts Calls: entropy() Outputs: mutual information
word.mutual.info <- function(counts) {
    # Assumes: counts is a numeric matrix get the marginal entropy of the classes
    # (rows) C
    marginal.entropy = entropy(rowSums(counts))
    # Get the probability of each value of X
    probs <- colSums(counts)/sum(counts)
    # Calculate the entropy of each column
    column.entropies = apply(counts, 2, entropy)
    conditional.entropy = sum(probs * column.entropies)
    mutual.information = marginal.entropy - conditional.entropy
    return(mutual.information)
}
```

Code Example 2: The `word.mutual.info` function. `apply(foo,2,bar)` applies the function `bar` to each column of the array `foo` and collects the results in a vector; changing the middle argument to 1 applies `bar` to the rows of `foo`. See `help(apply)`.

```
# Count how many documents in each class do or don't contain a word Presumes that
# the data frame contains a column, named 'class.labels', which has the classes
# labels; may be more than 2 classes Inputs: dataframe of word counts with class
# labels (BoW), word to check (word) Outputs: table of counts
word.class.indicator.counts <- function(BoW, word) {
    # What are the classes?
    classes <- levels(BoW[, "class.labels"])
    # Prepare a matrix to store the counts, 1 row per class, 2 cols (for
    # present/absent)
    counts <- matrix(0, nrow = length(classes), ncol = 2)
    # Name the rows to match the classes
    rownames(counts) = classes
    for (i in 1:length(classes)) {
        # Get a Boolean vector showing which rows belong to the class
        instance.rows = (BoW[, "class.labels"] == classes[i])
        # sum of a boolean vector is the number of TRUEs
        n.class = sum(instance.rows)  # Number of class instances
        present = sum(BoW[instance.rows, word] > 0)
        # present = Number of instances of class containing the word
        counts[i, 1] = present
        counts[i, 2] = n.class - present
    }
    return(counts)
}
```

Code Example 3: The `word.class.indicator.counts` function.

Putting the pieces together,

```
word.mutual.info(word.class.indicator.counts(nyt.frame, "paint"))
## [1] 0.1076399
```

2 Finding Informative Features

Here's one information-theoretic procedure for finding the important words.

1. Count how often each class $c = 1, 2 \ldots K$ appears.
2. For each word, make the $K \times 2$ table of classes by word indicators.
3. Compute the mutual information in each table.
4. Return the m most-informative words.

This ranks words by how informative it is to see them *at all* in the document. We could also look at how much information we get from the *number* of times they appear in the document — the table we build in step two would no longer necessarily by $K \times 2$, as the number of columns would depend on the number of different values for that word's feature.

The `info.bows` function (Code Example 4) does steps (1)–(3) of the ranking procedure.

This does *two* calculations for each word: how much the entropy of the class is reduced when the word is *present*, and how much the entropy is reduced *on average* by checking the word's indicator (the mutual information). I have *not* given code for the function for the first calculation, `word.realized.info`, but you can figure it out from what I have said.

Table 1 shows the ten words whose presence or absence in a document have the most information for the art/music classification task. Figure 2 plots this for all 4431 distinct words in the data.

Of course, nothing in this really hinges on our features being words; we could do the same thing for *any* set of features.

Calculating the expected information is actually very similar to performing a χ^2 test for independence. (Remember that mutual information is 0 if and only if the two variables are statistically independent.) In fact, if the sample size is large enough, the samples are IID, and the variables really are independent, then the sample mutual information has a χ^2 distribution (Kullback, 1968).[4]

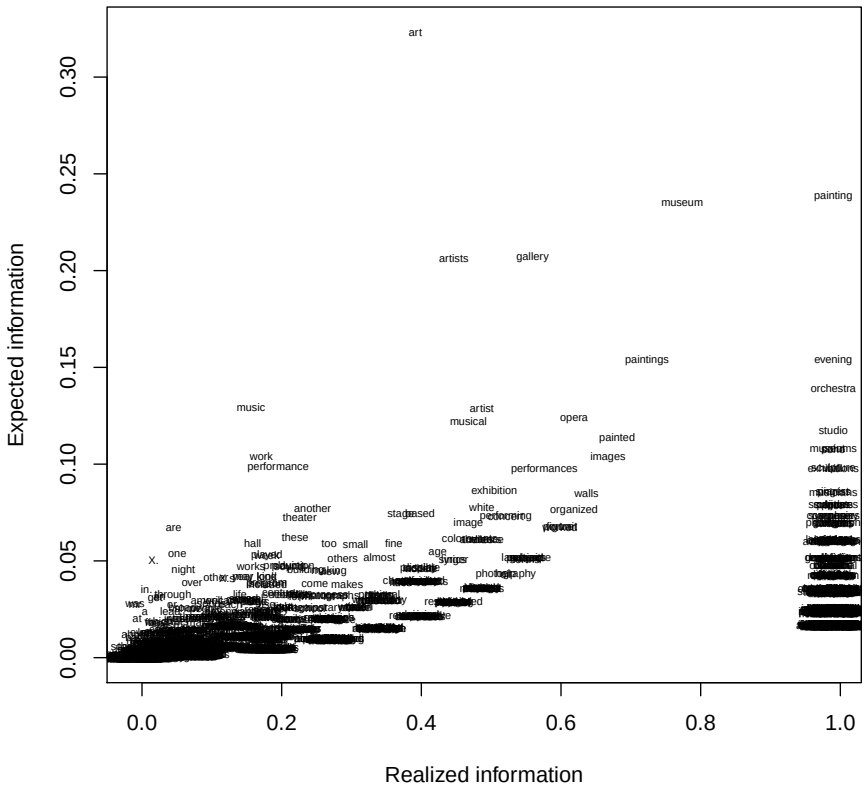
[4] In general, working out the bias, standard error, and sampling distribution of mutual information estimates is not easy. See, for instance, Victor (2000); Paninski (2003).

```
# Calculate realized and expected information of word indicators for classes
# Assumes: one column of the data is named 'class.labels' Inputs: data frame of
# word counts with class labels Calls: word.class.indicator.counts(),
# word.realized.info(), word.mutual.info() Output: two-column matrix giving the
# reduction in class entropy when a word is present, and the expected reduction
# from checking the word
info.bows <- function(BoW) {
    lexicon <- colnames(BoW)
    # One of these columns will be class.labels, that's not a lexical item
    lexicon <- setdiff(lexicon, "class.labels")
    vocab.size = length(lexicon)
    word.infos <- matrix(0, nrow = vocab.size, ncol = 2)
    # Name the rows so we know what we're talking about
    rownames(word.infos) = lexicon
    for (i in 1:vocab.size) {
        counts <- word.class.indicator.counts(BoW, lexicon[i])
        word.infos[i, 1] = word.realized.info(counts)
        word.infos[i, 2] = word.mutual.info(counts)
    }
    return(word.infos)
}
```

Code Example 4: The `info.bows` function

	I[C;X]		I[C;X=1]
art	0.32	abandoned	0.99
painting	0.24	abc	0.99
museum	0.23	abroad	0.99
gallery	0.21	abstractions	0.99
artists	0.21	academic	0.99
paintings	0.15	accents	0.99
evening	0.15	accept	0.99
orchestra	0.14	acclaimed	0.99
music	0.13	accounted	0.99
artist	0.13	achievement	0.99

Table 1: Most informative words for discriminating between art and music. Left: ranked by expected information, $I[C; X]$. Right: ranked by realized information when the word is present, $I[C; X = 1]$.




```
plot(0, xlim = range(info.matrix[, 1]), ylim = range(info.matrix[, 2]), xlab = "Realized information",
    ylab = "Expected information", type = "n")
text(info.matrix[, 1], info.matrix[, 2], rownames(info.matrix), cex = 0.5)
```

Figure 2: How much do we reduce our uncertainty about whether a *Times* story is about art or music by checking for various words? The horizontal axis ("realized information") shows the reduction in entropy when the word is present. The vertical axis ("expected information") shows the average reduction in entropy from checking for the word. Notice that the two values tend to rise together, but that the expected information tends to be smaller than the realized information (the scales on the two axes are different). The code beneath the figure shows how it was produced.

3 Feature Interactions and Feature Selection

We've talked about using features to predict variables which are not immediately represented, like future events, or the category to which an object belongs. We now have the machinery for saying how much entropy a single random variable has, and how much knowledge of one variable reduces the entropy of another — how much information they have about each other. This leads to a basic idea for how to select features: if you want to predict C and have features $X_1, X_2, \ldots X_p$, calculate $I[C; X_i]$ for $i = 1 : p$, and take the most informative feature.

Normally — and this is especially true when you have enough features that you *want* to do feature selection — you have many features, so we want a way to talk about using *multiple* features to predict C. You *could* just go down the list of informative features, but it's easy to suspect this isn't the right thing to do. Look at Table 1. "Painting" was the second most informative word, but "paintings" was number 6. Do you *really* think you'll learn much about the document from checking whether it has the word "paintings" if you already know whether it contains "painting"? Or looking for "music" after you've checked "orchestra"?

Our remaining subject, accordingly, is to extend the information theory we've seen to handle multiple features, and the idea that there can be **interactions** among features, that they can be more or less informative in different contexts.

3.1 Entropy for Multiple Variables

The **joint entropy** of two random variables X and Y is just the entropy of their joint distribution:

$$H[X, Y] \equiv -\sum_{x,y} \Pr(X = x, Y = y) \log_2 \Pr(X = x, Y = y) \tag{8}$$

This definition extends naturally to the joint entropy of an arbitrary number of variables.

A crucial property is that joint entropy is **sub-additive**:

$$H[X, Y] \leq H[X] + H[Y] \tag{9}$$

with equality if and only if X and Y are statistically independent. In terms of uncertainty, this says that you can't be more uncertain about the pair (X, Y) than you are about its components. In terms of coding, it says that the number of bits you need to encode the pair is no more than the number of bits you would need to encode each of its members. Again, this extends to any arbitrary number of variables.

Conditional entropy and mutual information can both be defined in terms of the joint entropy:

$$H[Y|X] = H[X, Y] - H[X] \tag{10}$$

$$I[X; Y] = H[X] + H[Y] - H[X, Y] \tag{11}$$

The last of these says that the mutual information is the difference between the joint entropy and the sum of the marginal entropies. This can be extended to any number of variables, giving what's called the **multi-information** or **higher-order mutual information**,

$$I[X;Y;Z] = H[X] + H[Y] + H[Z] - H[X,Y,Z] \tag{12}$$

(and so on for more than three variables). This is zero if and only if all the variables are statistically independent of each other.

3.2 Information in Multiple Variables

If all we want is to know how much information a set of variables $X_1, X_2, \ldots X_k$ have about a given outcome or target variable C, that is just

$$I[C; X_1, X_2, \ldots X_k] = H[C] + H[X_1, X_2, \ldots X_k] - H[C, X_1, X_2, \ldots X_k] = H[C] - H[C|X_1, X_2, \ldots X_k] \tag{13}$$

It should not be hard to convince yourself that adding an extra variable increases joint entropy, decreases conditional entropy, and increases information:

$$H[X,Y] \geq H[X] \tag{14}$$

$$H[C|X,Y] \leq H[C|X] \tag{15}$$

$$I[C;X,Y] \geq I[C;X] \tag{16}$$

and similarly with more predictor variables.

3.2.1 Example Calculation

Let's do an example. The single most informative word for our documents, we saw above, is "art", call its indicator X_1, followed by "painting" (say X_2). How informative is the *pair* of words?

To calculate this, we need a $2 \times 2 \times 2$ (class × "art" × "painting") contingency table, or alternately a 2×4 (class × ("art", "painting")) table. Because it's easier to get two dimensions down on the page than three, I'll use the latter right now, but we'll switch later on, and you should get used to alternating between the two perspectives.

	"art" yes		"art" no	
	"painting" yes	"painting" no	"painting" yes	"painting" no
art	22	25	2	8
music	0	8	0	37

I'll use the `word.mutual.info` function from above:

```
art.painting.indicators <- matrix(c(22, 25, 2, 8, 0, 8, 0, 37), byrow = TRUE, nrow = 2)
word.mutual.info(art.painting.indicators)
## [1] 0.4335985
```

so $I[C; X_1, X_2] = 0.43$ bits.

From the work above, we know that $I[C; X_1] = 0.32$ bits, and $I[C; X_2] = 0.24$ bits. So using both words gives us more information than either word alone. But we get less information than the sum of the individual informations.

3.3 Conditional Information and Interaction

If we have three variables, X, Y and C, we can ask how much information Y contains about C, after we condition on[5] X:

$$I[C;Y|X] \equiv H[C|X] - H[C|Y,X] \tag{17}$$

Notice that this is the average of

$$H[C|X=x] - H[C|Y,X=x] \equiv I[C;Y|X=x] \tag{18}$$

over possible values of x. For each x, we have an ordinary mutual information, which is non-negative, so the average is also non-negative. In fact, $I[C;Y|X] = 0$ if and only if C and Y are **conditionally independent** given X. This is written[6] $C \perp\!\!\!\perp Y|X$.

3.3.1 Interaction

While $I[C;Y|X]$ is non-negative, it can be bigger than, smaller than or equal to $I[C;Y]$. When it is not equal, we say that there is an **interaction** between X and Y — as far as their information about C. It is a positive interaction if $I[C;Y|X] > I[C;Y]$, and negative when the inequality goes the other way. If the interaction is negative, then we say that (some of) the information in Y about C is **redundant** given X.

You can begin to see how this connects to feature selection: it would seem natural to prefer variables containing *non*-redundant information about C. We can explicate this a little more with a touch more math.

3.3.2 The Chain Rule

The **chain rule** for joint entropy is that

$$H[X_1, X_2, \ldots X_k] = H[X_1] + \sum_{i=2}^{k} H[X_i|X_1, \ldots X_{i-1}] \tag{19}$$

To see this, notice that

$$H[X_i|X_1, \ldots X_{i-1}] = H[X_1, \ldots X_i] - H[X_1, \ldots X_{i-1}] \tag{20}$$

If we use this to expand the sum in Eq. 19, we see that every term in the sum is added and subtracted once and so cancels out, *except* for $H[X_1, X_2, \ldots X_k]$. This is an example of a **telescoping sum**, one which, as it were, folds up like a telescope.

[5] Some people call this "controlling for" X, but that's a misleading phrase, unless we make some causal assumptions.

[6] One way to do this in LaTeXis `\rotatebox{90}{\ensuremath{\models}}`. If you have trouble with this in your LaTeX setup, $\perp$ is often acceptable.

This implies a chain rule for mutual information:

$$I[C; X_1, X_2, \ldots X_k] = I[C; X_1] + \sum_{i=2}^{k} I[C; X_i | X_1, \ldots X_{i-1}] \tag{21}$$

To see *this*, thing about just the $k = 2$ case:

$$\begin{aligned} I[C; X_1] &= H[C] - H[C|X_1] & (22) \\ I[C; X_2|X_1] &= H[C|X_1] - H[C|X_2, X_1] & (23) \end{aligned}$$

Add the two lines and the $H[C|X_1]$ terms cancel, leaving

$$H[C] - H[C|X_1, X_2] = I[C; X_1, X_2] \tag{24}$$

Remember that a moment ago we said there was a positive interaction between X_1 and X_2 when

$$I[C; X_2|X_1] > I[C; X_2] \tag{25}$$

and a negative interaction if the inequality was reversed. This means that there is a positive interaction just when

$$I[C; X_1, X_2] > I[C; X_1] + I[C; X_2] \tag{26}$$

and we get more information about C from using both features than we would expect from using either of them on their own. Of course if there is a negative interaction, we get *less*,

$$I[C; X_1, X_2] < I[C; X_1] + I[C; X_2] \tag{27}$$

because some of what X_2 has to tell us about C is redundant given what X_1 says. (Or vice versa.)

4 Feature Selection with Mutual Information (Once More with Feeling)

Here is an improved procedure for selecting features from a set $X_1, X_2, \ldots X_p$ for predicting an outcome C.

1. Calculate $I[C; X_i]$ for all $i \in 1 : p$. Select the feature with the most information, call it $X_{(1)}$.
2. Given k selected features $X_{(1)}, X_{(2)}, \ldots X_{(k)}$, calculate $I[C; X_i | X_{(1)}, \ldots X_{(k)}]$ for all i not in the set of selected variables.
3. Set $X_{(k+1)}$ to be the variable with the highest conditional information and go to step 2.

Picking i to maximize $I[C; X_i | X_{(1)}, \ldots X_{(k)}]$ is the same as picking it to maximize $I[C; X_{(1)}, \ldots X_{(k)}, X_i]$. (Why?) So at each step, we are picking the variable with the most *non-redundant* information, given the variables we have already selected.

There are two things to notice about this algorithm.

1. It is **greedy**.
2. It doesn't know when to stop.

As to the first point: A **greedy** optimization algorithm is one which always takes the step which improves things the most *right away*, without concern about what complications it might create down the line. "Gallery" is not as good by itself as "art", but it could be that "gallery" is part of a better *combination* of features than any combination involving "art". A greedy algorithm closes itself off to such possibilities. What it gains in exchange for this is a more tractable search problem. We will see a lot of greedy algorithms.[7]

As to the second point: As I've written it, the algorithm will simply add all the features in a certain order. (There is always *some* variable, among the unselected features, which adds more information than the others.) In practice, we want to modify the last step so it checks for a **stopping condition** before going back to step 2. One possibility is to decide on a number of features q to use, and stop once $k = q$. Another is to stop when $I[C; X_{(k)} | X_{(1)}, \ldots X_{(k)}]$ gets sufficiently small — for instance, smaller than we can reliably estimate given our finite data, or smaller than we think can be useful to us.[8]

4.1 Example for the *Times* Corpus

Before I can illustrate the new, better procedure, I need to actually come up with a way to calculate the mutual information values it needs — doing it by hand is infeasible, and the code above is only for a single feature. I'll actually go over some of the design process, so that you how I get to the code, rather than just the destination.

4.1.1 Code

I want to pick the variable which adds the *most* information, given the ones which are already chosen. So with a choice of $X_{(1)}, \ldots X_{(k)}$, I want to evaluate $I[C; X_j, X_{(1)}, \ldots X_{(k)}]$ for all not-yet-selected j. To figure out how to do this, I write some **pseudocode**:

[7] Of course, if you are willing to spend the computing time, it's easy to make a procedure less greedy: it can "look ahead a step" by considering adding *pairs* of features, for example, or it can try deleting a feature (other than the one it just added) and going from there, etc.

[8] Verleysen *et al.* (2009) discusses the issue of the stopping criterion in detail. Parts of the paper use methods more advanced than we've seen so far, but you should be able to follow it by the end of the course.

```
Given: a data-frame with p features and class labels
       a number of features q to pick
Desired: the q most informative features
until q features are selected
  calculate how much information each unselected variable adds given the others
  select the most informative variable
```

Both parts of the procedure need some expansion to turn into code, but calculating the information sounds harder, so think about that first.

```
Given: a data-frame with p features and class labels
       k already-selected features
       a feature to consider selecting
Desired: Information about the class in the selected features and the candidate
  Calculate marginal entropy of the class
  Calculate joint entropy of the features
  Calculate joint entropy of the class and the features
  Calculate mutual information from entropies
```

We know how to do the last step. We also know how to get the entropy of the class, since that's just a single variable. The tricky bit is that we don't have a way of calculating the joint entropy of more than one variable.

To find the joint entropy, we need the joint distribution. And it turns out that our friend the `table()` function will give it to us. Before, we've seen things like `table(document)`, giving us the counts of all the different values in the vector `document`. If we give `table` multiple arguments, however, and they're all the same length, we get a multi-dimensional array which counts the occurrences of *combinations* of values. For example,

```
ape = table(nyt.frame[, "art"] > 0, nyt.frame[, "painting"] > 0, nyt.frame[, "evening"] >
    0, dnn = c("art", "painting", "evening"))
```

creates a $2 \times 2 \times 2$ table, counting occurrences of the three words "art", "painting" and "evening".[9] Thus if I want to know how many stories contain "art" and "painting" but not "evening",

```
ape[2, 2, 1]
## [1] 22
```

I find that there are 22 of them. See Code Example 5

Now I need another bit of R trickery. The output of `table()` is an object of class `table`, which is a sub-type of the class `array`, which is a kind of **structure**, meaning that *inside* it's just a vector, with a fancy interface for picking out different components. I can force it back to being a vector:

```
as.vector(ape)
## [1] 34 32  2 22 11  1  0  0
```

[9] The `dnn` argument of `table` names the dimensions of the resulting contingency table.

```
ape = table(nyt.frame[, "art"] > 0, nyt.frame[, "painting"] > 0, nyt.frame[, "evening"] >
    0, dnn = c("art", "painting", "evening"))
ape
## , , evening = FALSE
##
##        painting
## art     FALSE TRUE
##   FALSE    34    2
##   TRUE     32   22
##
## , , evening = TRUE
##
##        painting
## art     FALSE TRUE
##   FALSE    11    0
##   TRUE      1    0
```

Code Example 5: Use of `table()` to create a multi-dimensional contingency table, and the organization of the result.

and this gives me the count of each of the eight possible combinations of values for the indicator variables. Now I can invoke the `entropy()` function I wrote earlier:

```
entropy(as.vector(ape))
## [1] 2.053455
```

So the joint entropy of the three features is just over 2 bits.

Now we just need to automate this computation of the joint entropy for an arbitrary set of features. It would be nice if we could just make a vector of column numbers, `v` say, and then say

```
table(nyt.frame[,v])
```

but unfortunately that gives a one-dimensional rather than a multi-dimensional table. (Why?) To make things work, we'll paste together a string which would give us the commands we'd want to issue, and then have R act as though we'd typed that ourselves (Code Example 6).

Let's try this out. Sticking in all the > 0 over and over is tiresome, so I'll just make another frame where this is done already:

```
nyt.indicators = data.frame(class.labels = nyt.frame[, 1], nyt.frame[, -1] > 0)
```

Check this on some easy cases, where we know the answers.

```
columns.to.table(nyt.indicators, c("class.labels"))
##  class.labels
##   art music
```

```
# Create a multi-dimensional table from given columns of a data-frame Inputs:
# frame, vector of column numbers or names Outputs: multidimensional contingency
# table
columns.to.table <- function(frame, colnums) {
    my.factors = c()
    for (i in colnums) {
        # Create commands to pick out individual columns, but don't evaluate them yet
        my.factors = c(my.factors, substitute(frame[, i], list(i = i)))
    }
    # paste those commands together
    col.string = paste(my.factors, collapse = ", ")
    # Name the dimensions of the table for comprehensibility
    if (is.numeric(colnums)) {
        # if we gave column numbers, get names from the frame
        table.names = colnames(frame)[colnums]
    } else {
        # if we gave column names, use them
        table.names = colnums
    }
    # Encase the column names in quotation marks to make sure they stay names and R
    # doesn't try to evaluate them
    table.string = paste("\"", table.names, "\"", collapse = ",")
    # paste them together
    table.string = paste("c(", table.string, ")", collapse = ",")
    # Assemble what we wish we could type at the command line
    expr = paste("table(", col.string, ", dnn=", table.string, ")", collapse = "")
    # execute it parse() takes a string and parses it but doesn't evaluate it eval()
    # actually substitutes in values and executes commands
    return(eval(parse(text = expr)))
}
```

Code Example 6: The `columns.to.table` function. The `table` command creates multi-dimensional contingency tables if given multiple arguments, so we need to somehow provide the appropriate objects to it. The trick here is to write out the R command we wish to execute as a string, and then get R to run it (with the `parse` and `eval` functions). There are other ways of doing this, but creating the command you want before you execute it is useful in other situations, too.

```
# Calculate the joint entropy of given columns in a data frame Inputs: frame,
# vector of column numbers or names Calls: columns.to.table(), entropy() Output:
# the joint entropy of the desired features, in bits
jt.entropy.columns = function(frame, colnums) {
    tabulations = columns.to.table(frame, colnums)
    H = entropy(as.vector(tabulations))
    return(H)
}
```

Code Example 7: Calculating the joint entropy of an arbitrary set of columns in a data frame.

```
##    57    45
columns.to.table(nyt.indicators, c("class.labels", "art"))
##                art
##  class.labels  FALSE TRUE
##          art      10   47
##          music    37    8
columns.to.table(nyt.indicators, c("art", "painting", "evening"))
## , ,  evening  = FALSE
##
##         painting
##  art    FALSE TRUE
##   FALSE    34    2
##   TRUE     32   22
##
## , ,  evening  = TRUE
##
##         painting
##  art    FALSE TRUE
##   FALSE    11    0
##   TRUE      1    0
```

So far this looks good. Now we can calculate the joint entropy (Code Example 7).

We can check this on our "art"/"painting"/"evening" example:

```
jt.entropy.columns(nyt.indicators, c("art", "painting", "evening"))
## [1] 2.053455
```

From the joint entropy, we can get the information selected columns have about the class feature.

The word "art" is # 244 in the list of column names for this frame, and "painting" is # 2770. So we can check this function against our earlier results like so:

```
info.in.multi.columns(nyt.indicators, 244)
## [1] 0.32327
```

```
# Compute the information in multiple features about the outcome Inputs: data
# frame, vector of feature numbers, number of target feature (optional,
# default=1) Calls: jt.entropy.columns Output: mutual information in bits
info.in.multi.columns = function(frame, feature.cols, target.col = 1) {
    H.target = jt.entropy.columns(frame, target.col)
    H.features = jt.entropy.columns(frame, feature.cols)
    H.joint = jt.entropy.columns(frame, c(target.col, feature.cols))
    return(H.target + H.features - H.joint)
}
```

Code Example 8: Finding the information a set of columns have about a given target. What happens if `target.col` is a vector rather than a single column number?

```
info.in.multi.columns(nyt.indicators, 2770)
## [1] 0.238395
```

The payoff, though, is this:

```
info.in.multi.columns(nyt.indicators, c(244, 2770))
## [1] 0.4335985
```

Code Examples 9 and 10 take us back up our chain of thought, until Code Example 11 is what we sought at the beginning. In itself, it does very little; this is as it should be.

Computational efficiency note The code above is not the most efficient possible implementation of the greedy search scheme. For instance, we end up computing $H[C]$, the entropy of the target variable, *many* times, though it never changes. It would be faster to compute it once, in the top-level function `best.q.columns`, and then pass it as an argument down to the `info.in.multi.columns` function.[10]

4.1.2 Results

Let's take the top seven words:

```
best.7 = best.q.columns(nyt.indicators, 7)
colnames(nyt.indicators)[best.7]
## [1] "art"      "youre"    "features" "music"    "gallery"  "heavy"    "second"
info.in.multi.columns(nyt.indicators, best.7)
## [1] 0.9629841
```

Table 2 shows how much information we can from each additional word along this path.

[10] Using global variables for tasks like this is just begging for trouble down the road when you change something.

```
# Information about target after adding a new column to existing set Inputs: new
# column, vector of old columns, data frame, target column (default 1) Calls:
# info.in.multi.columns() Output: new mutual information, in bits
info.in.extra.column <- function(new.col, old.cols, frame, target.col = 1) {
    mi = info.in.multi.columns(frame, c(old.cols, new.col), target.col = target.col)
    return(mi)
}
```

Code Example 9: `info.in.extra.column`

```
# Identify the best column to add to an existing set Inputs: data frame,
# currently-picked columns, target column (default 1) Calls:
# info.in.extra.column() Output: index of the best feature
best.next.column <- function(frame, old.cols, target.col = 1) {
    # Which columns might we add?
    possible.cols = setdiff(1:ncol(frame), c(old.cols, target.col))
    # How good are each of those columns?
    infos = sapply(possible.cols, info.in.extra.column, old.cols = old.cols, frame = frame,
        target.col = target.col)
    # which of these columns is biggest?
    best.possibility = which.max(infos)
    # what column of the original data frame is that?
    best.index = possible.cols[best.possibility]
    return(best.index)
}
```

Code Example 10: Picking the most-informative column to add, given the columns already selected. `sapply` is used to avoid an explicit iteration over the possibilities.

```
# Identify the best q columns for a given target variable Inputs: data frame, q,
# target column (default 1) Calls: best.next.column() Output: vector of column
# indices
best.q.columns <- function(frame, q, target.col = 1) {
    possible.cols = setdiff(1:ncol(frame), target.col)
    selected.cols = c()
    for (k in 1:q) {
        new.col = best.next.column(frame, selected.cols, target.col)
        selected.cols = c(selected.cols, new.col)
    }
    return(selected.cols)
}
```

Code Example 11: Function for greedy selection of features by their information about a target variable. Note how almost all of the work has been passed off to other functions.

word	cumulative information
“art”	0.32
“youre”	0.49
“features”	0.62
“music”	0.75
“gallery”	0.84
“heavy”	0.90
“second”	0.96

Table 2: The seven most informative words, as selected by the greedy search.

Some of these words were informative by themselves — “art”, “music”, “gallery” — but others were not.

5 Sufficient Statistics and the Information Bottleneck

For any random variable X and any function f, $f(X)$ is another random variable. How does the entropy of $f(X)$ relate to that of X? It's easy to believe that

$$H[X] \geq H[f(X)] \tag{28}$$

After all, we can't be more uncertain about the value of a function than about the input to the function. (Similarly, it can't be harder to encode the function's value.) This is in fact true, and the only way to get an equality here is if f is a one-to-one function, so it's just, in effect, changing the label on the random variable. It's also true that

$$I[C; X] \geq I[C; f(X)] \tag{29}$$

but now we can have equality even if f is not one-to-one. When this happens, we say that the function is **sufficient** for predicting C from X — it's **predictively sufficient** for short, or a **sufficient statistic**. To mark this, we'll write it as $\epsilon(X)$ rather than just $f(X)$.[11]

Intuitively, a sufficient statistic ϵ captures all the information X has about C; everything else about X is so much extraneous detail, so how could it be of any use to us? More formally, it turns out that optimal prediction of C only needs a sufficient statistic of X, not X itself, no matter how we define "optimal".[12]

All of this applies to functions of several variables as well, so if we have features $X_1, \ldots X_p$, what we'd *really* like to do is find a sufficient statistic $\epsilon(X_1, \ldots X_p)$, and then forget about the original features. Unfortunately, finding an *exactly* sufficient statistic is hard, except in special cases when you make a lot of hard-to-check assumptions about the joint distribution of C and the X_i. (One which has "all the advantages of theft over honest toil" is to *assume* that your favorite features are sufficient; this is a key part of what's called the **method of maximum entropy**.) There is however a tractable alternative for finding *approximately* sufficient statistics.

A sufficient statistic solves the optimization problem

$$\max_{f \in \mathcal{F}} I[C; f(X)] \tag{30}$$

where $\mathcal{F}$ contains all the functions of X. Let's modify the problem:

$$\max_{f \in \mathcal{F}} I[C; f(X)] - \beta H[f(X)] \tag{31}$$

[11] Of course, one-to-one functions are sufficient, but also trivial, so we'll ignore them in what follows.

[12] Even more formally: any loss function can be minimized by a decision rule which depends only on a sufficient statistic. (If you can follow that sentence, you most likely already know the result, but that's why this is a footnote.)

where β is some positive number which we set. Call the solution to this problem η_β. What happens here? Well, the objective function is indifferent between increasing $I[C; f(X)]$ by a bit, and lowering $H[f(X)]$ by $1/\beta$ bits. Said another way: it is willing to lose up to β bits of predictive information if doing so compresses the statistic by an extra bit. As $\beta \to 0$, it becomes unwilling to lose *any* predictive information, and we get back a sufficient statistic. As $\beta \to \infty$, we become indifferent to prediction and converge on η_∞, which is a constant function.

The random variable $\eta_\beta(X)$ is called a **bottleneck variable** for predicting C from X; it's approximately sufficient, with β indicating how big an approximation we're tolerating. The method is called the **information bottleneck**, and the reason it's more practical than trying to find a sufficient statistic is that there are algorithms which can solve the optimization in Eq. 31, at least if X and C are both discrete.[13] This is a very cool topic — see Tishby *et al.* (1999) — which we may revisit if time permits.

Further Reading

Information theory appeared almost fully formed in Shannon (1948), a classic paper which is remarkably readable. The best available textbook on information theory, covering its applications to coding, communications, prediction, gambling, the foundations of probability, etc., is Cover and Thomas (1991). Poundstone (2005) is a popular book about how information theory connects to gambling and the stock market; Poundstone (1984) explains how it connects to fundamental aspects of physical science, as does Wiener (1954).

References

Cover, Thomas M. and Joy A. Thomas (1991). *Elements of Information Theory*. New York: Wiley.

Kullback, Solomon (1968). *Information Theory and Statistics*. New York: Dover Books, 2nd edn.

Paninski, Liam (2003). "Estimation of entropy and mutual information." *Neural Computation*, **15**: 1191–1254. URL `http://www.stat.columbia.edu/~liam/research/abstracts/info_est-nc-abs.html`.

Poundstone, William (1984). *The Recursive Universe: Cosmic Complexity and the Limits of Scientific Knowledge*. New York: William Morrow.

[13] To begin to see how this is possible, imagine that X takes on m discrete values. Then $f(X)$ can have at most m values, too. This means that the number of *possible* functions of X is finite, and in principle we could evaluate the objective function of Eq. 31 on each of them. For large m the number of functions is the m^{th} "Bell number", and these grow *very* rapidly indeed — the first ten are $1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975$ — so exhaustive search is out of the question, but cleverer algorithms exist.

— (2005). *Fortune's Formula: The Untold Story of the Scientific Betting Systems That Beat the Casinos and Wall Street*. New York: Hill and Wang.

Sandhaus, Evan (2008). "The New York Times Annotated Corpus." Electronic database. URL `http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2008T19`.

Shannon, Claude E. (1948). "A Mathematical Theory of Communication." *Bell System Technical Journal*, **27**: 379–423. Reprinted in **?**.

Tishby, Naftali, Fernando C. Pereira and William Bialek (1999). "The Information Bottleneck Method." In *Proceedings of the 37th Annual Allerton Conference on Communication, Control and Computing* (B. Hajek and R. S. Sreenivas, eds.), pp. 368–377. Urbana, Illinois: University of Illinois Press. URL `http://arxiv.org/abs/physics/0004057`.

Verleysen, Michel, Fabrice Rossi and Damien Fran cois (2009). "Advances in Feature Selection with Mutual Information." In *Similarity-Based Clustering* (Thomas Villmann and Michael Biehl and Barbara Hammer and Michel Verleysen, eds.), vol. 5400 of *Lecture Notes in Computer Science*, pp. 52–69. Berlin: Springer Verlag. URL `http://arxiv.org/abs/0909.0635`. doi:10.1007/978-3-642-01805-3_4.

Victor, Jonathan D. (2000). "Asymptotic Bias in Information Estimates and the Exponential (Bell) Polynomials." *Neural Computation*, **12**: 2797–2804. doi:10.1162/089976600300014728.

Wiener, Norbert (1954). *The Human Use of Human Beings: Cybernetics and Society*. Garden City, New York: Doubleday, 2nd edn. Republished London: Free Association Books, 1989; first edition Boston: Houghton Mifflin, 1950.

Exercises

These are for you to think about, rather than to hand in.

1. How would you reproduce Figure 1?
2. Looking at Figure 2, why does expected information tend to generally increase with realized information?
3. Why does expected information tend to be smaller than realized information?
4. Why are so many words vertically aligned at the right edge of the plot?
5. Write `word.realized.info`.
6. What code would you have to change to calculate the information the *number* of appearances of a word gives you about the class?
7. Read `help(order)`. How would you reproduce Table 1?
8. Prove Eqs. 10 and 11.
9. What will `info.in.multi.columns()` do if its `target.cols` argument is a vector of column numbers, rather than a single column number?
10. Prove Eqs. 28 and 29.
11. Why is "youre" so informative after checking "art"?