Spam Filtering

CS109L - Spring 2015

May 12, 2015

1 Causal Networks

Unwanted email (commonly called *spam*) is now a major problem. Many people use spam filters to automatically discard email that looks like spam. Some of these filters use a probabilistic model to determine how likely a piece of email is to be spam, and then discard it if the probability of it being spam is sufficiently high (e.g. greater than 0.9, or whatever level the user thinks is appropriate). Here, we will consider a simple probabilistic model for email that incorporates some of the common characteristics of spam. A real spam filter would need a much more complex model, but the basic characteristics are essentially the same. Your task will be to write an `R` program to categorize emails as being either spam or *ham* (i.e. not-spam) by examining certain characteristics of each message.

The model we will use is based on a *causal network* (also called a *belief network*) which shows how each random variable depends on its parent random variable(s). Here is an example:

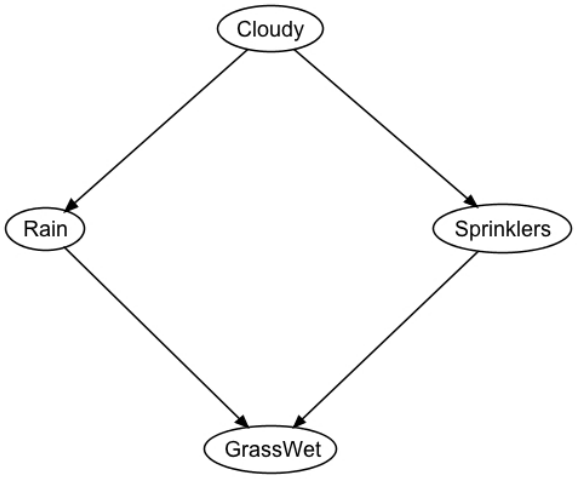

Figure 1: Weather Causal Network

The top node represents the **Cloudy** random variable, which is given the binary value 0 or 1 based on whether it was cloudy yesterday (0 for *false* and 1 for *true*). The two nodes below this (**Rain** and **Sprinklers**) are also binary random variables indicating whether it rained yesterday and if the sprinklers went off, respectively. The bottom row contains the binary random variable **GrassWet** indicating whether the grass is wet this

morning. You can see that **GrassWet** clearly depends on the values of **Rain** and **Sprinklers**, while each of these variables in turn depends on the value of **Cloudy**.[1]

Assume that we are living in the present, so the value of **GrassWet** is the only thing we can observe as we walk outside in the morning. A complete probabilistic model would need to specify the joint probability mass function for all possible combinations of values for these four random variables.[2] In this case, there are $2^4 = 16$ such possible combinations, but you can imagine in the case of a slightly larger causal network that the number of combinations increases exponentially, so we would rather not specify the probability of each combination of values separately. Instead, we use the fact that we can always write the joint probability for a set of random variables (e.g. W, X, Y, Z) as the following product, once we have chosen some order for them:

$$\begin{aligned} &P(W = w, X = x, Y = y, Z = z) \\ &\quad = P(Z = z | W = w, X = x, Y = y) \cdot P(Y = y | W = w, X = x) \cdot P(X = x | W = w) \cdot P(W = w) \end{aligned}$$

If we can furthermore simplify some of the factors above, we may be able to specify the model using many fewer numbers.

When the model is specified using a causal network, we order the variables so that the arrows go forward (top to bottom in this case) and then use conditional probabilities that are conditional only on the parents of a variable (node). A variable X is a parent of variable Y if there is an arrow from X to Y. For instance, in the network above we can do the following simplification:

$$P(\mathbf{GrassWet} | \mathbf{Rain}, \mathbf{Sprinklers}, \mathbf{Cloudy}) = P(\mathbf{GrassWet} | \mathbf{Rain}, \mathbf{Sprinklers})$$

When determining the probability of that it was cloudy yesterday conditional on observing wet grass, we would have the following:

$$P(\mathbf{Cloudy} | \mathbf{GrassWet} = 1) = \frac{P(\mathbf{GrassWet} = \mathbf{1} | \mathbf{Cloudy}) P(\mathbf{Cloudy})}{P(\mathbf{GrassWet} = \mathbf{1})}$$

Predicting the value of **Cloudy** then reduces to finding

$$\arg\max_y \; P(\mathbf{Cloudy} = y | X)$$

where X is a the set of *observations* (also known as *features*) that we have (i.e. everything we can see by walking outside this morning). Equivalently, we can assign **Cloudy** the value

$$\arg\max_y \; P(X, \mathbf{Cloudy} = y) = \arg\max_y \; P(X | \mathbf{Cloudy} = y) \cdot P(\mathbf{Cloudy} = y)$$

such that we maximize the joint probability of observations (X) and response (**Cloudy**) rather than the conditional probability of response given observation.[3]

2 Simple Spam Filter

To construct a spam filter, you will be using the `spam` dataset from the `kernlab` package. This dataset contains email messages classified as spam or ham. The 57 columns of the dataset mainly encode the

[1] Clearly **Rain** will depend on **Cloudy**. The other dependency is less obvious, so we will say that the sprinklers have photosensors that tell them to go off only when it is not cloudy.

[2] So the model would specify $P(\mathbf{Cloudy}, \mathbf{Rain}, \mathbf{Sprinklers}, \mathbf{GrassWet})$ for any set of values assigned to the four random variables.

[3] These two methods are "equivalent" in terms of their final prediction of y.

number of times that certain words or characters occur in each message reported. There are 4601 rows (emails) contained in the dataset, of which some fraction will be used for training your spam filter; the remaining fraction will be used to test the performance of your filter after training. The first 48 columns are continuous real variables representing the percentage of words (or numbers) per email that "match" the name given to the column (so the `$address` column gives the frequency of the word *address* and the `$num415` column gives the frequency of the number *415*).[4] The next 6 columns encode variables representing the percentage of single characters in the email matching the name of the column (so the `$charHash` column gives the frequency of the character #). The next 3 columns represent information about the average length of uninterrupted sequences of capital letters, the length of the longest uninterrupted sequence of capital letters, and the total number of capital letters in the email, respectively. The last column (`$type`) is a binary factor denoting whether each email in question was considered spam or not (with labels `nonspam` and `spam`).

For our spam filter, we will only use the first 48 columns as features for determining whether a message is spam. Consider the following causal network:

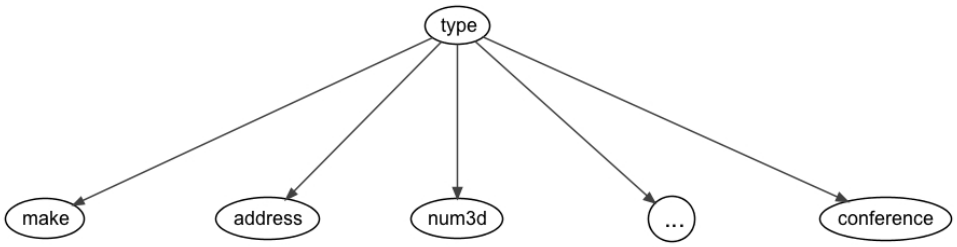

Figure 2: Spam Causal Network

This network models the **type** of message (spam or ham) as the parent of each of the 48 token frequency variables from the data set (**make**, **address**, etc.). By modeling our data this way, we are saying value of each of the 48 variables for a given message depends only on the type of the message.

Let $X = \{X_1, X_2, ..., X_{48}\}$ be the set of observable features in an email which we can use to classify the message (i.e. the token frequencies, which we can find by counting). Then, as shown previously, we classify the message as

$$\arg\max_y \ P(X, \textbf{type} = y) = \arg\max_y \ P(X|\textbf{type} = y) \cdot P(\textbf{type} = y)$$

To simplify the above expression, we will make the assumption of *conditional independence* between X and **type** such that $P(X|\textbf{type}) = \prod_{i=1}^{48} P(X_i|\textbf{type})$.[5]

As an additional simplification, we will re-encode the `spam` data frame to make each of the 48 columns a binary variable (factor) representing only whether or not the token exists in each message or not.[6] Then, to classify a message, we can find $P(X_i = yes|\textbf{type})$ for each X_i by calculating the mean number of values in column X_i whose value is *yes* from the sub- data frame containing only rows of the specified **type**. And, as we know, $P(X_i = no|\textbf{type})$ is just $1 - P(X_i = yes|\textbf{type})$.

[4] You don't need to worry about the naming scheme as long as you understand that these columns contain token frequencies.

[5] The *naive Bayes* assumption of conditional independence, as it is called, is not always valid. However, it is often a reasonable approximation of the actual probability, and it greatly simplifies our calculations. Later in CS109 you'll talk more extensively about conditional independence and the naive Bayes assumption.

[6] So the value will be *yes* when the token frequency is > 0, and *no* otherwise.

2.1 Instructions

There is surprisingly little code that you need to write to complete the spam filter, and I've already written some for you. Naturally, the first step will be to reformat the original `spam` data frame so that it only contains the first 48 columns as well as the last. Once you've done this, you should factor each of the first 48 columns into binary variables with labels *no* and *yes*, as previously described.

Since we're going to use the `spam` dataset for both training and testing, we need to pick some fraction of rows to use for the training, leaving the remaining rows for testing. Suppose for now that we use 1000 rows for training. If the rows of the data frame were listed in a random order, we could just take the first 1000 rows for training and use the rest of the rows for testing. You'll notice, however, that the first 1800 or so rows of `spam` are all marked as type *spam*, so you will need to randomly pick a subset of the data frame to use for training (so as to train our filter with both spam and ham messages). The `sample(...)` function should be helpful when choosing a random subset of rows.

To help you with the tasks outlined above, I've given you the following starter code with decomposed functions to complete:

```
RestructureSpamDataset <- function(spam) {
  ## Restures the 'spam' data frame according to the specifications in the
  ## handout (and summarized below).
  ##
  ## Args:
  ##   spam - the 'spam' data frame from the 'kernlab' package
  ## Returns:
  ##   A data frame with the first 48 columns of spam factored into
  ##   binary random variables (representing whether each word exists
  ##   or not) as well as the last column of 'spam'.
}

ChooseTrainingAndTesting <- function(dataset, k) {
  ## Chooses a random subset of the dataset to use for training, leaving
  ## the rest for testing.
  ##
  ## Args:
  ##   dataset - a data frame to split into training and testing data frames
  ##   k - the number of rows to include in the training data set
  ## Returns:
  ##   A list with elements $training and $testing, each a data frame with
  ##   a disjoint subset of rows taken randomly from 'dataset'.
}
```

The remaining functions to complete are shown below, with explanation following.

```
GivenNonSpam <- function(training) {
  ## Computes conditional probabilities of the form P(Xi | type = "nonspam").
  ##
  ## Args:
  ##   training - data frame with training data.
  ## Returns:
  ##   A matrix with 2 rows and 48 columns, one per observed variable
```

```
  ##   from the training data frame. The first row gives the conditional
  ##   probabilities P(Xi = "no" | type = "nonspam") for each of the 48
  ##   observed variables. The second row gives conditional probabilities
  ##   P(Xi = "yes" | type = "nonspam"). Note that all columns should sum
  ##   to one.
  ## Details:
  ##   This function uses Laplace smoothing for conditional probabilities.
  ##   You can implement Laplace smoothing by imagining, for each of
  ##   the 48 columns, that we see two additional datapoints ("no" and "yes").

  ## Choose the subset of the data frame which contains only "nonspam"
  ## rows, and keep only the first 48 columns.
  subset <- ## TODO

  ## Compute, for each of the 48 columns in the subset above, the probability
  ## of the value being "no". Remember to use Laplace smoothing.
  pNo <- ## TODO

  return(rbind(no = pNo, yes = 1 - pNo))
}

GivenSpam <- function(training) {
  ## Computes conditional probabilities of the form P(Xi | type = "spam").
  ## Details the same as above.

  ## Choose the subset of the data frame which contains only "nonspam"
  ## rows, and keep only the first 48 columns.
  subset <- ## TODO

  ## Compute, for each of the 48 columns in the subset above, the probability
  ## of the value being "no". Remember to use Laplace smoothing.
  pNo <- ## TODO

  return(rbind(no = pNo, yes = 1 - pNo))
}

Predict <- function(row, p.nonspam, p.spam, nonspam.probs, spam.probs) {
  ## Predicts whether the row (with 48 observed variables) is "spam" or
  ## "nonspam" by choosing argmax P(row | type) * P(type).
  ##
  ## Args:
  ##   row - vector with 48 values (one per observed variable)
  ##   p.nonspam - P(type = "nonspam")
  ##   p.spam - P(type = "spam")
  ##   nonspam.probs - conditional probabilities of the form P(Xi | type = "nonspam")
  ##   spam.probs - conditional probabilities of the form P(Xi | type = "spam")
  ## Returns:
  ##   "spam" or "nonspam"
}
```

The first two functions listed above compute (using the subset of rows selected for training) conditional prob-

abilities of the form $P(X_i|\textbf{type})$. In the case of `GivenSpam()`, we are computing the conditional probabilities $P(X_i = no|\textbf{type} = spam)$ and $P(X_i = yes|\textbf{type} = spam)$, which can be found by looking at the subset of rows from the training data of type *spam* and seeing what fraction have the value *no* and *yes*, respectively. When computing these fractions, you should imagine that there are two additional data points per column (one *no* and one *yes*) such that non-zero probability is assigned to every conditional observation.[7]

The final function to complete is `Predict()`, which given a set of observations (represented as a factor with 48 values) predicts the type of message as

$$\arg\max_y \ P(X, \textbf{type} = y) = \arg\max_y \ P(X|\textbf{type} = y) \cdot P(\textbf{type} = y) = \arg\max_y \ \prod_{i=1}^{48} P(X_i|\textbf{type} = y) \cdot P(\textbf{type} = y)$$

For your computer, finding a product of many small values (i.e. probabilities) is problematic, and you're likely to find after multiplying all the terms above together that the product is 0 for any value of y. To solve this problem, it is common practice to exploit the monotonicity of the *log* function and find the *log* of the product above.[8] So our prediction is

$$\arg\max_y \ \log \prod_{i=1}^{48} P(X_i|\textbf{type} = y) \cdot P(\textbf{type} = y) = \arg\max_y \ \sum_{i=1}^{48} \log P(X_i|\textbf{type} = y) + \log P(\textbf{type} = y)$$

To complete the `Predict()` function, use *log* probabilities and compute the sum above for $y = spam$ and $y = nonpam$, returning the value y which gives the larger sum.

Finally, use the function `RunTrainingAndTesting()` (which I've provided) to see your spam filter in action. When does including additional rows in training cease to increase the performance of your filter? You may be surprised to find how few rows are actually required in training to get reasonably good performance.

Now pat yourself on the back. Writing a spam filter is no easy feat!

[7] This method is known as *Laplace smoothing*, and is a way of making provisions for rare events that are never seen in training data (like seeing the word *Viagra* in a non-spam message). Many probabilistic models "break" when you assign zero probability to rare events, so we use smoothing to reshape probability density. In our example, although we are unlikely to see *Viagra* in a non-spam message during training, it is not an impossible event. Smoothing assigns this event a small probability.

[8] For now, you can take it on faith that this mathematically will not change the prediction for y. You'll see a proof of this later in the quarter.