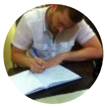
How to avoid letting a model cheat on you and break your heart



Colin Fraser Follow
Aug 24, 2016 · 11 min read

There are a lot of ways to screw up when you're trying to do data science. Some of them are very technical. *Leakage* is not one of those very technical screw-ups. In fact, you hardly have to know anything about mathematics or statistics or computer science at all to understand the basic premise of leakage. However, it is one of the sneakiest, most sinister, and widespread mistakes a data scientist can make. The *Handbook of Statistical Analysis and Data Mining Applications* calls leakage one of the top ten data mining mistakes. I might put it at number one.

The most fascinating thing about leakage—in a sort of can't-take-my-eyes-off-the-car-crash kind of a way—is the betrayal. The broken promises. The heartbreak. Your model performs way better than you expect it to in the test set. It's almost too good to be true. A part of you knows that no one model should have all that power, and yet, the numbers are right in front of you, showing you that you can predict the stock market with 90% accuracy or perfectly predict what product online customers will buy next. You know that you should think about this but the temptation to simply pat yourself on the back for building the world's greatest data mining algorithm is too great. Your eyes turn into dollar signs. You'll be famous. You're the greatest data scientist of all time. You're a hero.

But when you release your model into the wild, something awful happens. Not only does the model perform worse than it did in your test runs—you thought that might happen—it falls flat on its face. It doesn't make a single accurate prediction. In fact, in the most insidious cases of leakage, the model actually performs worse than flipping a coin, and you'd have been better off not building it at all. The model that you trained and tested for hours, that made all these promises to you about giant AUCs and tiny RMSEs and whatever other gauntlet of evaluation criteria you threw at it, is spitting in your face. It lied to you. It played you. Why?

And what's frustrating is this: the culprit is usually an obvious little mistake that you made right at the very beginning of the project. You can spend five hundred hours building and tuning some fancy algorithm that leverages all of the state-of-the-art machine learning tools that we have available today, and it will all be for nothing if you screw up and allow data leakage into your training data from the start.

When is a data leaky?

Machine learning experts have struggled a little bit to come up with a single formal definition of leakage, and perhaps it is best understood first by example. This is a modified version of an example that is presented in *Leakage in Data Mining: Formulation, Detection, and Avoidance*, which has many more examples and ideas on this topic if you're interested. Suppose you're a data scientist working at some kind of online store, and you're trying to build a model to predict, based on past purchases, whether a customer is likely to buy jewelry. You gather a massive dataset containing purchase histories for every single customer who has ever shopped at the store, along with their total spend on each of the three product categories that the store sells. Maybe it looks something like this:

```
CUSTOMER JEWELRY  MOVIES ELECTRONICS
   11123 1003.23   24.99     1594.95
   11124    0.00   96.45        0.00
   11125   58.12    0.00        0.00
   11126    0.00  549.20      190.67
   11127    8.23 2400.10      523.45
     ...     ...     ...         ...
```

Using this dataset, your plan is to build a model which predicts whether a customer buys jewelry as a function of their spending on movies and electronics.

You've just committed leakage. You're already hooped. You've built a leaky dataset, and any model that you train on this data, no matter how many tens of hours you put into boosting and bagging and deep learning and validating and cross-validating and whatever else you can imagine, is doomed to betray you from the start. In fact, in many cases, *the better your model performs in testing, the worse it will in real life.* Such is the treachery of leakage.

Can you see why? It has to do with customer 11125.

Customer 11125 has 0 spend on movies and 0 spend on electronics. If the store only sells jewelry, movies, and electronics, then the only way that customer 11125 can possibly be in the dataset is if he bought jewelry. Due to the way that the data was collected and formatted, the condition of having 0 spend on movies and electronics *logically implies* the condition of having bought jewelry.

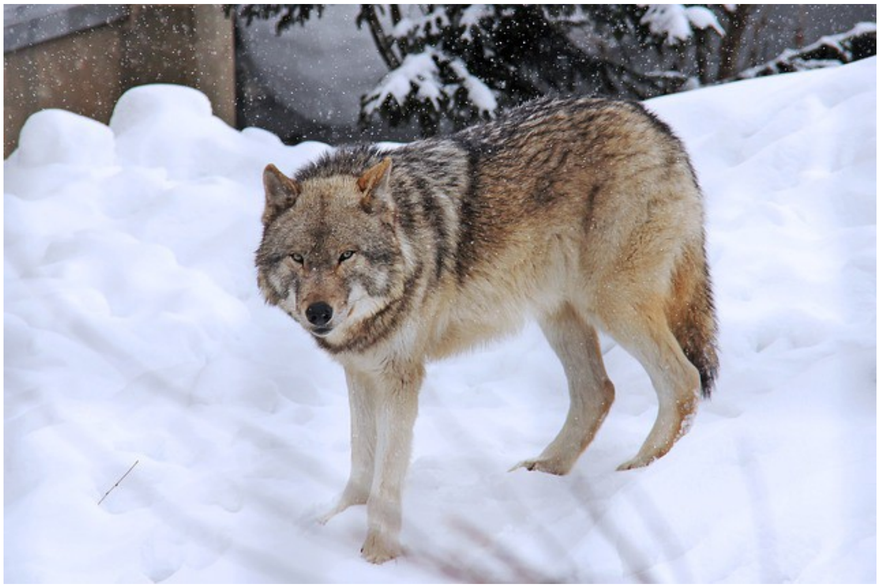
Any half decent model will learn this little tidbit and use it to make perfect predictions about the customers with 0 spend on movies and electronics. And if there are a lot of those customers in the data, then your model will make a whole lot of perfect predictions in testing. And that will look pretty good.

But it's a mirage. Again, the condition of having positive spend on jewelry is logically implied by the condition of having 0 spend on electronics and movies in this dataset. The model correctly identifies that customers who bought jewelry bought jewelry. This is an unhelpful identification.

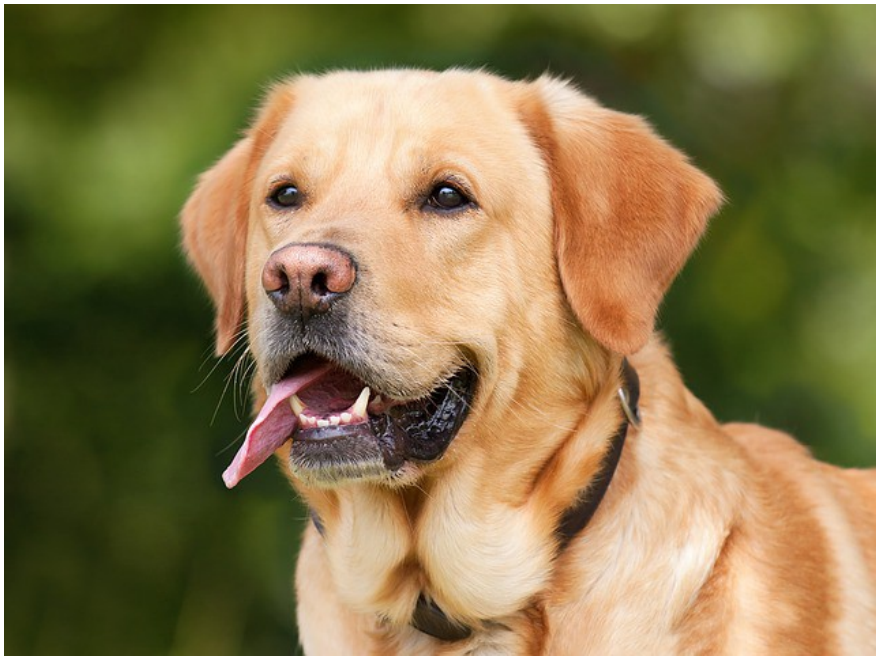
Information is in some sense *leaking* from the features—information that the model has no right to see. There's some kind of giveaway in the data about the outcomes that you're trying to predict, and the model uses that giveaway to cheat and make better predictions than would be possible in real life. You model has learned something, but the thing that it learned is a quirk of the *data* and not something about the *customers* that you're actually trying to model.

Freedom of Information

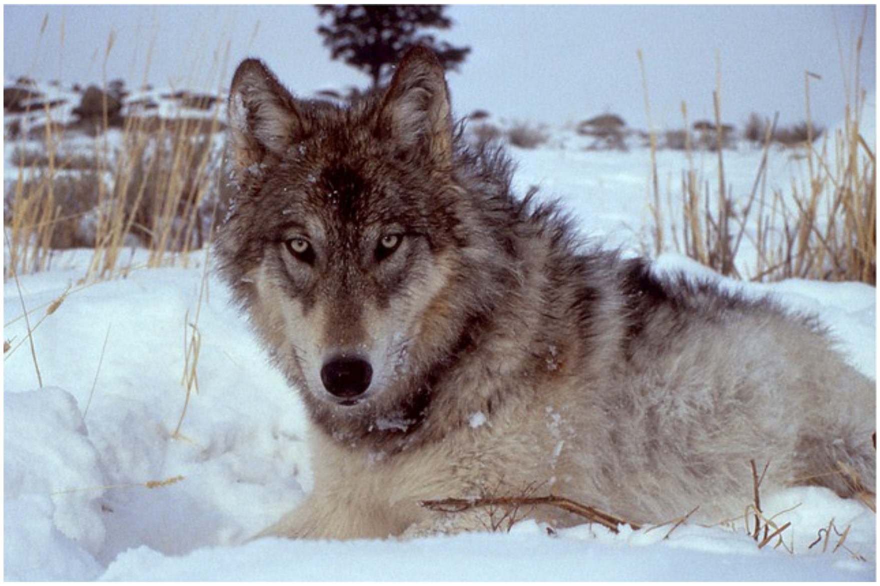
What information does your model have the right to see? That can be a tricky question to answer, but it is the key spotting and correcting leakage. Leakage is present when your features contain information that your model should not legitimately be allowed to use. But *legitimacy* is not always obvious, or even unambiguously defined. For instance, consider the case of a classifier that looks at an image and decides whether it contains a wolf or a dog. You train it on some images like these.



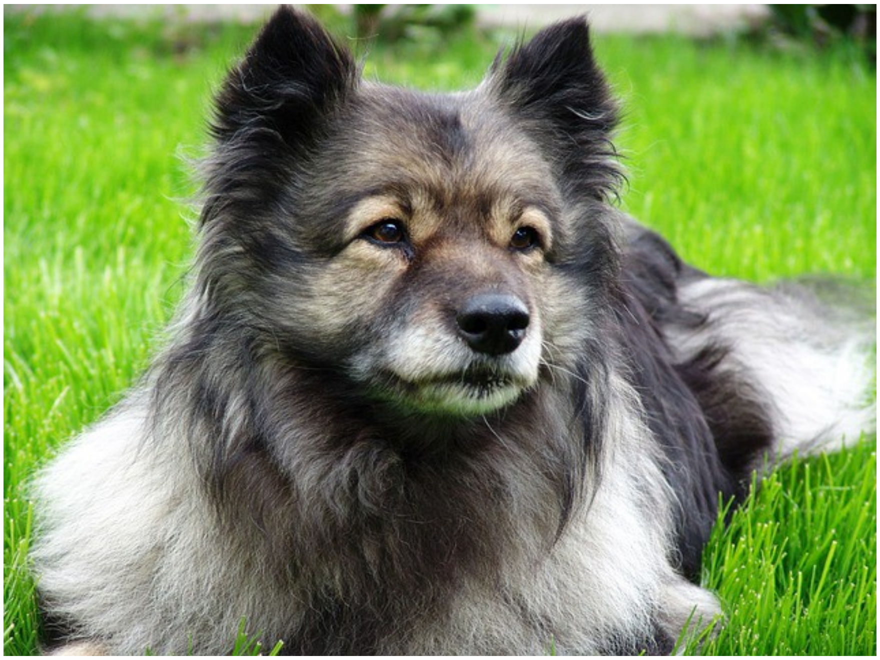
Wolf



Dog



Wolf



Dog

Your classifier seems to do quite well, identifying most wolves and most dogs in your test set. However, on further investigation (perhaps with a little help from the LIME technique described in the paper that I stole this example from), you determine that your model has simply decided that a “wolf” is a thing on a white background and a “dog” is a thing on a green background. The correct prediction is fully specified by the background color of the image. Is this leakage?

That's a little bit of a hard question. It is undoubtedly true that the color of the background of an image can help to predict the foreground, and humans performing this same classification task probably use the background as a feature in their own assessments. The background is a legitimate feature that rational observer would use to determine the content of an image.

Not a wolf

The answer to whether this is a leakage problem or not depends on how you expect your model to generalize. If you expect that the images that your model will be looking at have the property that all of the wolf images will have white backgrounds and all of the dog images won't, then run with the one you've got. It works just fine. On the other hand, if you expect your model to know that a Chihuahua on a blanket is not a wolf, then you do have a leakage problem at the data collection stage. Some quirk about your data collection process gave the background of the image too much predictive power, and your model is using this quirk of the data to cheat on its predictions.

There's no such thing as a time machine

The accidental introduction of time-travel into the data is probably the most common source of leakage in machine learning. Due to the way that data is often stored and aggregated, data that are generated at different times may be stored in the same place. A general rule might be: if X happened after Y, you shouldn't build a model that uses X to predict Y. Claudia Perlich calls this the *No Time Machine Condition* (NTMC). A model that determines whether it rains at 9:00 AM as a function of whether the road is wet at 9:05 AM might be very accurate, but it will also be very useless.

Violations of NTMC are usually harder to spot than using the wet road to predict rain. Imagine you're a data scientist at some kind of SaaS company and you're trying to build a model to predict customer churn (that is, to predict which customers will cancel their subscriptions). I have helpfully assembled a dataset for you, with an indicator variable marking whether each customer cancelled their subscription in 2015, along with some aggregated statistics for each customer for the year.

```
CUSTOMER HELP_DESK_CALLS_2015 DOWNTIME_MINS_2015 CHURNED ...
   45220                    4               1293   FALSE ...
   45221                    0                  0    TRUE ...
   45222                    1                 43    TRUE ...
   45223                    4               1001   FALSE ...
   45224                    3               6298   FALSE ...
     ...                  ...                ...     ... ...
```

CUSTOMER is the customer number, HELP_DESK_CALLS_2015 is the total number of times that the customer called into the help desk in 2015, DOWNTIME_MINS_2015 is the total number of minutes that the customer spent with their product not working properly in 2015, and CHURNED has a value of TRUE if and only if that customer churned at some point in 2015.

Can you spot the leakage?

We have violated NTMC. The variable CHURNED indicates whether a customer cancels their account at any point in the year. Our features, HELP_DESK_CALLS_2015 and DOWNTIME_MINS_2015 are yearly totals which are, by definition, only available at the end of the year. There's no way that we could possibly know the values of these variables before the churn event takes place, so our model has no business using them to predict churn. There is no such thing as a time machine.

“Well,” I might reply, with a crazed look in my eye that says *don’t make me redo this data collection*, “at least we have a model. Maybe there’s some kind of leakage problem with some of the variables, but it’s better than nothing.”

Is it?

Consider the relationship between DOWNTIME_MINS_2015 and CHURNED in the data. DOWNTIME_MINS_2015 is the aggregate total number of minutes that the product was down for the customer in 2015. Consider a customer who churns on January 1, 2015. Her DOWNTIME_MINS_2015 is probably quite low—maybe 0. After all, she’s not even a customer for a single whole day in 2015, so she doesn’t have a very long time to accumulate downtime minutes. If there are many customers like this who cancel early in the year, then the data might contain many examples of churners with small numbers for DOWNTIME_MINS_2015. The model can exploit this pattern in the data to identify customers with low downtime minutes as churners.

But this seems like it’s probably the exact opposite of reality! The data collection process results in a negative relationship between the variables DOWNTIME_MINS_2015 and CHURNED, but the real-life process that relates real-life customer churn to real-life downtime minutes could very well have a *positive* relationship between those two variables! This results in the perverse outcome where **the more likely your model says a customer is to churn, the less likely they really are to churn**! Your “predictive model” is not only making inaccurate predictions, it is making predictions which are systematically wrong. You’re anti-predicting! Your churn prediction model predicts non-churn! Data leakage has created a model which truly is worse than having no model at all, all the while performing outstandingly well in backtesting.

Such is the treachery of leakage.

What can you do?

I’m going to go out on a limb and claim that most sources of leakage are violations of NTMC, and modelers should be vigilant in ensuring that they are not accidentally introducing time travel into their models. Predictive modelling is usually (but not always) about using today’s data to predict what will happen tomorrow, but since training data is almost always collected after the fact, it is very easy to overlook the temporal sequence of the events that generate the data. This oversight can be huge.

In NTMC cases, the fix is straightforward, at least in principle: somehow or another, your dataset needs a time dimension. In the example of the online store, if the dataset had been split into two time periods, where purchases (including jewelry) in time period 1 are used to predict whether a customer purchases jewelry in time period 2, the leakage problem disappears. Moreover, it's usually the case that the more time information you can provide, the better. If the dataset in the churn prediction model could be rearranged into a panel format, where we have multiple observations across time for each customer, not only could the leakage issue be resolved, but a much wider array of potential features and models could be considered.

The single most important thing that you can do to prevent data leakage from seeping into your models is to understand the process that generates the data, and how that process relates to the process that you're actually trying to simulate in order to form predictions. The whole point of modelling is to learn things about some process out there in the real world, and in some sense to approximate that real-world process. But models don't see the real-world process. They see the data. Leakage happens when there is some systematic correlation between features and outcomes *in the data* that either does not exist or is not observable in the real world. In order to spot that deceptive correlation, it is crucial to understand the data that you are working with. How is it collected? How is it aggregated? What are the criteria for being included in this sample?

This level of understanding is not always easy to obtain in many modern data science contexts, where data is collected from disparate sources, possibly with the involvement of multiple people or teams, many of whom won't have the answers. But when a deep understanding of the data generation process is achieved, it usually makes leakage problems almost obvious, and the fix becomes apparent.