In this tutorial, you will learn what **data leakage** is and how to prevent it. If you don't know how to prevent it, leakage will come up frequently, and it will ruin your models in subtle and dangerous ways. So, this is one of the most important concepts for practicing data scientists.

Introduction

Data leakage (or **leakage**) happens when your training data contains information about the target, but similar data will not be available when the model is used for prediction. This leads to high performance on the training set (and possibly even the validation data), but the model will perform poorly in production.

In other words, leakage causes a model to look accurate until you start making decisions with the model, and then the model becomes very inaccurate.

There are two main types of leakage: **target leakage** and **train-test contamination.**

Target leakage

Target leakage occurs when your predictors include data that will not be available at the time you make predictions. It is important to think about target leakage in terms of the *timing or chronological order* that data becomes available, not merely whether a feature helps make good predictions.

An example will be helpful. Imagine you want to predict who will get sick with pneumonia. The top few rows of your raw data look like this:

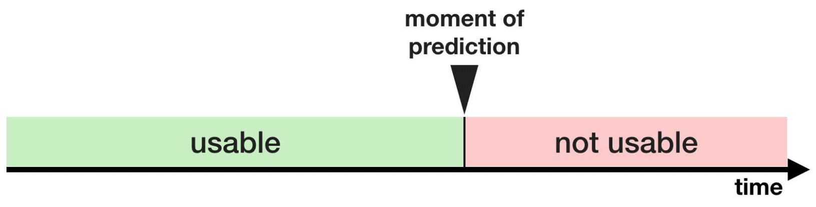
got_pneumonia	age	weight	male	took_antibiotic_medicine	...
False	65	100	False	False	...
False	72	130	True	False	...
True	58	100	False	True	...

People take antibiotic medicines *after* getting pneumonia in order to recover. The raw data shows a strong relationship between those columns, but `took_antibiotic_medicine` is frequently changed *after* the value for `got_pneumonia` is determined. This is target leakage.

The model would see that anyone who has a value of `False` for `took_antibiotic_medicine` didn't have pneumonia. Since validation data comes from the same source as training data, the pattern will repeat itself in validation, and the model will have great validation (or cross-validation) scores.

But the model will be very inaccurate when subsequently deployed in the real world, because even patients who will get pneumonia won't have received antibiotics yet when we need to make predictions about their future health.

To prevent this type of data leakage, any variable updated (or created) after the target value is realized should be excluded.



Train-Test Contamination

A different type of leak occurs when you aren't careful to distinguish training data from validation data.

Recall that validation is meant to be a measure of how the model does on data that it hasn't considered before. You can corrupt this process in subtle ways if the validation data affects the preprocessing behavior. This is sometimes called **train-test contamination**.

For example, imagine you run preprocessing (like fitting an imputer for missing values) before calling `train_test_split()`. The end result? Your model may get good validation scores, giving you great confidence in it, but perform poorly when you deploy it to make decisions.

After all, you incorporated data from the validation or test data into how you make predictions, so the may do well on that particular data even if it can't generalize to new data. This problem becomes even more subtle (and more dangerous) when you do more complex feature engineering.

If your validation is based on a simple train-test split, exclude the validation data from any type of *fitting*, including the fitting of preprocessing steps. This is easier if you use scikit-learn pipelines. When using cross-validation, it's even more critical that you do your preprocessing inside the pipeline!

Example

In this example, you will learn one way to detect and remove target leakage.

We will use a dataset about credit card applications and skip the basic data set-up code. The end result is that information about each credit card application is stored in a DataFrame `X`. We'll use it to predict which applications were accepted in a Series `y`.

Hide

In [1]:

```
import pandas as pd

# Read the data
data = pd.read_csv('../input/aer-credit-card-data/AER_credit_card_data.csv',
                   true_values = ['yes'], false_values = ['no'])

# Select target
y = data.card

# Select predictors
X = data.drop(['card'], axis=1)

print("Number of rows in the dataset:", X.shape[0])
X.head()
```

```
Number of rows in the dataset: 1319
```

Out[1]:

	reports	age	income	share	expenditure	owner	selfemp	dependents	months	ma
0	0	37.66667	4.5200	0.033270	124.983300	True	False	3	54	1
1	0	33.25000	2.4200	0.005217	9.854167	False	False	3	34	1
2	0	33.66667	4.5000	0.004156	15.000000	True	False	4	58	1
3	0	30.50000	2.5400	0.065214	137.869200	False	False	0	25	1
4	0	32.16667	9.7867	0.067051	546.503300	True	False	2	64	1

Since this is a small dataset, we will use cross-validation to ensure accurate measures of model quality.

In [2]:

```
from sklearn.pipeline import make_pipeline
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

# Since there is no preprocessing, we don't need a pipeline (used anyway as best
 practice!)
my_pipeline = make_pipeline(RandomForestClassifier(n_estimators=100))
cv_scores = cross_val_score(my_pipeline, X, y,
                            cv=5,
                            scoring='accuracy')

print("Cross-validation accuracy: %f" % cv_scores.mean())
```

```
Cross-validation accuracy: 0.979525
```

With experience, you'll find that it's very rare to find models that are accurate 98% of the time. It happens, but it's uncommon enough that we should inspect the data more closely for target leakage.

Here is a summary of the data, which you can also find under the data tab:

- `card` : 1 if credit card application accepted, 0 if not
- `reports` : Number of major derogatory reports
- `age` : Age n years plus twelfths of a year
- `income` : Yearly income (divided by 10,000)
- `share` : Ratio of monthly credit card expenditure to yearly income
- `expenditure` : Average monthly credit card expenditure
- `owner` : 1 if owns home, 0 if rents
- `selfempl` : 1 if self-employed, 0 if not
- `dependents` : 1 + number of dependents
- `months` : Months living at current address
- `majorcards` : Number of major credit cards held
- `active` : Number of active credit accounts

A few variables look suspicious. For example, does `expenditure` mean expenditure on this card or on cards used before appying?

At this point, basic data comparisons can be very helpful:

In [3]:

```
expenditures_cardholders = X.expenditure[y]
expenditures_noncardholders = X.expenditure[~y]

print('Fraction of those who did not receive a card and had no expenditures: %.2f
' \
      %((expenditures_noncardholders == 0).mean()))
print('Fraction of those who received a card and had no expenditures: %.2f' \
      %(( expenditures_cardholders == 0).mean()))
```

```
Fraction of those who did not receive a card and had no expenditures: 1.00
Fraction of those who received a card and had no expenditures: 0.02
```

As shown above, everyone who did not receive a card had no expenditures, while only 2% of those who received a card had no expenditures. It's not surprising that our model appeared to have a high accuracy. But this also seems to be a case of target leakage, where expenditures probably means *expenditures on the card they applied for*.

Since `share` is partially determined by `expenditure`, it should be excluded too. The variables `active` and `majorcards` are a little less clear, but from the description, they sound concerning. In most situations, it's better to be safe than sorry if you can't track down the people who created the data to find out more.

We would run a model without target leakage as follows:

In [4]:

```
# Drop leaky predictors from dataset
potential_leaks = ['expenditure', 'share', 'active', 'majorcards']
X2 = X.drop(potential_leaks, axis=1)

# Evaluate the model with leaky predictors removed
cv_scores = cross_val_score(my_pipeline, X2, y,
                            cv=5,
                            scoring='accuracy')

print("Cross-val accuracy: %f" % cv_scores.mean())
```

```
Cross-val accuracy: 0.830924
```

This accuracy is quite a bit lower, which might be disappointing. However, we can expect it to be right about 80% of the time when used on new applications, whereas the leaky model would likely do much worse than that (in spite of its higher apparent score in cross-validation).

Conclusion

Data leakage can be multi-million dollar mistake in many data science applications. Careful separation of training and validation data can prevent train-test contamination, and pipelines can help implement this separation. Likewise, a combination of caution, common sense, and data exploration can help identify target leakage.

What's next?

This may still seem abstract. Try thinking through the examples in **this exercise** to develop your skill identifying target leakage and train-test contamination!