This tutorial is part of the Learn Machine Learning series. In this step, you will learn what data leakage is and how to prevent it.

What is Data Leakage

Data leakage is one of the most important issues for a data scientist to understand. If you don't know how to prevent it, leakage will come up frequently, and it will ruin your models in the most subtle and dangerous ways. Specifically, leakage causes a model to look accurate until you start making decisions with the model, and then the model becomes very inaccurate. This tutorial will show you what leakage is and how to avoid it.

There are two main types of leakage: Leaky Predictors and a Leaky Validation Strategies.

Leaky Predictors

This occurs when your predictors include data that will not be available at the time you make predictions.

For example, imagine you want to predict who will get sick with pneumonia. The top few rows of your raw data might 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. So 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. Validation data comes from the same source, so 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.

To prevent this type of data leakage, any variable updated (or created) after the target value is realized should be excluded. Because when we use this model to make new predictions, that data won't be available to the model.

Leaky Data Graphic

Leaky Validation Strategy

A much different type of leak occurs when you aren't careful distinguishing training data from validation data. For example, thi

s happens if you run preprocessing (like fitting the Imputer for missing values) before calling train_test_split. Validation is meant to be a measure of how the model does on data it hasn't considered before. You can corrupt this process in subtle ways if the validation data affects the preprocessing behavoir.. The end result? Your model will get very good validation scores, giving you great confidence in it, but perform poorly when you deploy it to make decisions.
Preventing Leaky Predictors

There is no single solution that universally prevents leaky predictors. It requires knowledge about your data, case-specific inspection and common sense.

However, leaky predictors frequently have high statistical correlations to the target. So two tactics to keep in mind:

    To screen for possible leaky predictors, look for columns that are statistically correlated to your target.
    If you build a model and find it extremely accurate, you likely have a leakage problem.

Preventing Leaky Validation Strategies

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 use pipelines and do your preprocessing inside the pipeline.
Example

We will use a small dataset about credit card applications, and we will build a model predicting which applications were accepted (stored in a variable called card). Here is a look at the data:

```
import pandas as pd

data = pd.read_csv('../input/AER_credit_card_data.csv',
                   true_values = ['yes'],
                   false_values = ['no'])
print(data.head())

   card  reports       age  income     share  expenditure  owner
  selfemp  \
0  True        0  37.66667  4.5200  0.033270   124.983300   True
   False
1  True        0  33.25000  2.4200  0.005217     9.854167  False
   False
2  True        0  33.66667  4.5000  0.004156    15.000000   True
   False
3  True        0  30.50000  2.5400  0.065214   137.869200  False
   False
4  True        0  32.16667  9.7867  0.067051   546.503300   True
   False
```

```
   dependents  months  majorcards  active
0           3      54           1      12
1           3      34           1      13
2           4      58           1       5
3           0      25           1       7
4           2      64           1       5
```

We can see with data.shape that this is a small dataset (1312 ro
ws), so we should use cross-validation to ensure accurate measur
es of model quality

```
data.shape
```

```
(1319, 12)
```

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

y = data.card
X = data.drop(['card'], axis=1)

# Since there was no preprocessing, we didn't need a pipeline he
re. Used anyway as best practice
modeling_pipeline = make_pipeline(RandomForestClassifier())
cv_scores = cross_val_score(modeling_pipeline, X, y, scoring='ac
curacy')
print("Cross-val accuracy: %f" %cv_scores.mean())
```

```
Cross-val accuracy: 0.979528
```

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 rare eno
ugh that we should inspect the data more closely to see if it is
 target leakage.

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

    card: Dummy variable, 1 if application for credit card accep
ted, 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 in
come
    expenditure: Average monthly credit card expenditure
    owner: 1 if owns their home, 0 if rent
    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 m ean expenditure on this card or on cards used before appying?

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

```
expenditures_cardholders = data.expenditure[data.card]
expenditures_noncardholders = data.expenditure[~data.card]

print('Fraction of those who received a card with no expenditure
s: %.2f' \
      %(( expenditures_cardholders == 0).mean()))
print('Fraction of those who received a card with no expenditure
s: %.2f' \
      %((expenditures_noncardholders == 0).mean()))
```

```
Fraction of those who received a card with no expenditures: 0.02
Fraction of those who received a card with no expenditures: 1.00
```

Everyone with card == False had no expenditures, while only 2% of those with card == True had no expenditures. It's not surprising that our model appeared to have a high accuracy. But this seems a data leak, 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, 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 leakage as follows:

```
potential_leaks = ['expenditure', 'share', 'active', 'majorcards
']
X2 = X.drop(potential_leaks, axis=1)
cv_scores = cross_val_score(modeling_pipeline, X2, y, scoring='a
ccuracy')
print("Cross-val accuracy: %f" %cv_scores.mean())
```

```
Cross-val accuracy: 0.806677
```

This accuracy is quite a bit lower, which on the one hand is 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 then that (even in spite of it's 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 is a first step, and pipelines can help implement this separation. Leaking predictors are a more frequent issue, and leaking predictors are harder to track down. A combination of caution, common sense and data exploration can help identify leaking predictors so you remove them from your model.

Exercise

Review the data in your ongoing project. Are there any predictor s that may cause leakage? As a hint, most datasets from Kaggle c ompetitions don't have these variables. Once you get past those carefully curated datasets, this becomes a common issue.

Click here to return the main page for Learning Machine Learning .

------
Leakage
Introduction

Leakage is one of the scariest things in machine learning (parti cularly competitions). Leakage makes your models look good, unti l you put them into production and realize that they're actually roundly terrible. To quote the Kaggle wiki entry on the subject :

> Data Leakage is the creation of unexpected additional inform ation in the training data, allowing a model or machine learning algorithm to make unrealistically good predictions.
>
> Leakage is a pervasive challenge in applied machine learning , causing models to over-represent their generalization error an d often rendering them useless in the real world. It can caused by human or mechanical error, and can be intentional or unintent ional in both cases.

Leakage is particularly bad because it invalidates or weakens cr oss validation scoring. The accuracy of cross validation as a pr ediction for how well our model will do in validation or on prod uction data is incredibly important; so much so that it's often said that "above all, trust your CV". If we undermine that, we u ndermine most of the tools and techniques in our toolbox!
Target leakage

The most obvious form of leakage is when a variable in a dataset is derived from the target variable in some way. For example, i f we are predicting annual_gdp, a column with GDP in 2016 dollar s, standardized_gdp, would be an example of a leak, because it's just the same data transformed a little bit. In order to build a real model and not a linear transform, we would need to remove this column from our model entirely. Again from the Kaggle wiki :

> One concrete example we've seen occurred in a prostrate canc er dataset. Hidden among hundreds of variables in the training d ata was a variable named PROSSURG. It turned out this represente d whether the patient had received prostate surgery, an incredib ly predictive but out-of-scope value.
>
> The resulting model was highly predictive of whether the pat ient had prostate cancer but was useless for making predictions on new patients.

With some practice working with and inspecting machine learning

features, this kind of "variable leak" is catchable, but it becomes tedious when the feature matrix has enough predictors in it. Domain knowledge helps a ton here.
Out-of-core leakage

Leakage is the number one problem in machine learning competitions because it can be weaponized by model-makers in a way that would never make sense in a production system. This is "out-of-core leakage". For an example of what this looks like, see this old Kaggle post explaining why one leak caused a competition identifying right whales to be reset. They're very challenging to catch because even experienced competition-runners (like the Kaggle team) can't match the time and depth competitors can bring to probing datasets for weaknesses.
Knowledge leakage

Which brings us to knowledge leakage, which is what I want to cover in more depth in this notebook. I'll actually just be going over the information presented in this fantastic blog post on the subject, so you should probably read that first.

To guard against overfitting, machine learning relies heavily on cross validation and related holdout and parameter search schemes. The effectiveness of the technique relies on our building a model on a training data, then testing it for fitness on training data that it's never seen before.

This is only an effective technique if we can prevent information about our test data from leaking into our training data. In theory this is easy: just don't use observations from the test data in the training data. However, there are things we can do during the pre-processing before we train a model that injects information about our test data into the training process! Doing this will increase our cross validation accuracy on the data we train on, but will worsen our accuracy in practice on validation or production data.

Let's demo how this can happen (NB: we're reimplementing the blog post code here; some things have changed in the library in the meanwhile however, so this code is a little different from that which originally ran).

We'll build a 100×10000

feature matrix: that is, 100 observations across 10000 synthetic features. This is a massively overdetermined feature matrix. Then we'll perform feature selection: we'll measure the correlation of each of the columns with the target column, and take the top two scorers as our model inputs. We'll train on those, and measure what our mean squared error (MSE) is (for more on model fit metrics click here).

```
import numpy as np
import pandas as pd
import scipy.stats as st
```

```
np.random.seed(0)
df = np.random.randint(0,10,size=[100,10000])
y = np.random.randint(0,2,size=100)
df = pd.DataFrame(df)
X = df.values

corr = np.abs(
    np.array([st.pearsonr(X[:, i], y)[0] for i in range(X.shape[
1])])
)
corrmax_indices = np.argpartition(np.abs(corr), -2)[-2:]

X_selected = X[:, corrmax_indices]

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

clf = LogisticRegression()
clf.fit(X_selected, y)
mse = cross_val_score(clf, X_selected, y, cv=10, scoring='neg_me
an_squared_error')

pd.Series(mse).abs().mean()

0.24989898989898984
```

Our mean squared error is pretty good, and we trust our CV, so we think this is a result reflective of practical performace. However, is it really? Can you spot the error?

It's subtle. The reason we picked a matrix with so many features is because it accentuates the error we've made with the procedure here. By measuring the correlation of the columns and taking the two highest scorers before doing cross validation, we actually injected incidental information about which variables are most highly correlated in both the train and test sets. Hence when we run the cross validation, we've "pre-selected" incidental correlation that we know beforehand performs well in the test set.

We picked a lot of variables to make this effect easily noticable (with 10000 variables, some of them are going to end up quite correlated with the target). We can see how strong of an effect this creates by doing this same variable selection after a train-test split:

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_test, y_train, y_test = train_test_split(X, y, test_s
ize=0.4)

corr = np.abs(
    np.array([st.pearsonr(X_train[:, i], y_train)[0] for i in ra
nge(df.shape[1])])
)
corrmax_indices = np.argpartition(np.abs(corr), -2)[-2:]
```

```
X_selected = X_train[:, corrmax_indices]

clf = LogisticRegression()
clf.fit(X_selected, y_train)
y_hat = clf.predict(X_test[:, corrmax_indices])
mean_squared_error(y_test, y_hat)
```

0.45000000000000001

It looks like knowledge leaking almost halved our mean squared error!

The correct approach to dealing with this problem is to think harder about how we will structure our pipeline. Best-fit variable selection like this should live inside of our cross validation; that is, it should only be done after we've already done train-test splitting. This will at least give us a more realistic index on performance:

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import StratifiedKFold

kf = StratifiedKFold(n_splits=10)

mse_results = []

for train_index, test_index in kf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    corr = np.abs(
        np.array([st.pearsonr(X_train[:, i], y_train)[0] for i in range(df.shape[1])])
    )
    corrmax_indices = np.argpartition(np.abs(corr[:-1]), -2)[-2:]

    X_train_selected = X_train[:, corrmax_indices]

    clf = LogisticRegression()
    clf.fit(X_train_selected, y_train)
    mse = mean_squared_error(clf.predict(X_test[:, corrmax_indices]), y_test)
    mse_results.append(mse)

mse = pd.Series(mse).mean()
mse
```

0.6666666666666666

Conclusion

Knowledge leakage is a difficult problem to address completely. The one thing I recommend doing to avoid this problem is being conscientious about using pipelines, like the one scikit-learn provides, to hande pre-processing and training as one contiguous u nit (the scikit-learn user guide in fact lists "safety" in this regard as one of the three reasons to use pipelining).

For small to moderately-sized datasets, I do not think that knowledge leakage is a huge problem. Pipelining over feature selection has its own problems (it introduces overfitting into cross validation?). The amount of error you introduce into your model via knowledge leaking is relatively small: maybe even a rounding error on your overall model accuracy.

However, it becomes a problem when there are lots of variables, especially when the feature matrix is overdetermined (more variables than observations). In these cases you do want to be careful about how you design your pre-processing.

When in doubt, I recommend running an exercise like the one I demonstrated here on your dataset. See how much of a difference knowledge leaking makes for a dataset shaped like yours!