

Intermediate Machine Learning

Based on the Kaggle Tutorial "[Intermediate Machine Learning](#)" by Alexis Cook.

1. Introduction

Goals:

- tackle data types often found in real-world datasets (missing values, categorical variables),
- design pipelines to improve the quality of your machine learning code,
- use advanced techniques for model validation (cross-validation),
- build state-of-the-art models that are widely used to win Kaggle competitions (XGBoost), and
- avoid common and important data science mistakes (leakage).

Competition submission steps reiterated:

1. save version with the **Save and Run all** version enabled.
2. once that is done, click on the number right next to the **Save Version** button.
3. This pulls up a list of versions on the right of the screen. Click on the ellipsis (...) to the right of the most recent version, and select Open in Viewer.
4. This brings you into view mode of the same page. You will need to scroll down to get back to these instructions.
5. Click on the Output tab on the right of the screen. Then, click on the file you would like to submit, and click on the blue Submit button to submit your results to the leaderboard.

2. Missing Values

There are many reasons why there would be missing values, some examples are: - The size of the 3rd bedroom for a 2 bedroom house - One survey respondent declining to share his income - etc.

Two ways to deal with this:

Method	notes
Drop columns with missing values	you will be ignoring lots of potentially important information.
Imputation	Fill in the missing blank with the average for that row. Maybe not exactly <i>the</i> best value for that spot, but this causes the model to be a lot more accurate.
Imputation Extention	add a column that notes if a given column had missing values that were imputed, might be more accurate than your average imputation (pun?).
IMPORTANT!	There is no universal rule on what is better in any case, and you certainly can't guess it without reading a huge part of the data manually. So just try 'em all and make an educated guess.

Method	notes
For more examples and techniques:	Have a look at this article

Tips to choose:

- A lot of missing data (think more than 50%) in a given column -- drop it.
- If not, try both the imputation methods and pick the better one.

Extention to imputation

- Some times, the imputed values could be systematically above or below their actual values, or really just be correctly and uniquely wrong (think accuracy v. precision)
- So a good idea would be to add another coloumn along the lines of "bed 3 missing" with a true or false value to let the model know that the value was artifically generated and is therefore different from the original model *someway*.
 - This may or may not help depending on your situation.
 - So be sure to try using it and then evaluate if it turns out to be better than plain old imputation.

Bed	Bath		Bed	Bath	Bed_was_missing
1.0	1.0		1.0	1.0	FALSE
2.0	1.0		2.0	1.0	FALSE
3.0	2.0		3.0	2.0	FALSE
NaN	2.0	→	2.0	2.0	TRUE

Good starting point to optimize models:

- make an accuracy function such as this one.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error

# Function for comparing different approaches
def score_dataset(X_train, X_valid, y_train, y_valid):
    model = RandomForestRegressor(n_estimators=10, random_state=0)
    model.fit(X_train, y_train)
    preds = model.predict(X_valid)
    return mean_absolute_error(y_valid, preds)
```

Method 1: Drop columns

- Be sure to drop the same columns from the training and validation data sets.

```
# Get names of columns with missing values
cols_with_missing = [col for col in X_train.columns
                      if X_train[col].isnull().any()]

# Drop columns in training and validation data
reduced_X_train = X_train.drop(cols_with_missing, axis=1)
reduced_X_valid = X_valid.drop(cols_with_missing, axis=1)
```

Method 2: Imputation

- use `sklearn.impute.SimpleImputer` class.
- Many different methods have been researched for imputation, but once put into modern machine learning models, it doesn't make much of a difference to using average values.

```
from sklearn.impute import SimpleImputer

# Imputation
my_imputer = SimpleImputer()
#### MUST fit transform training data
imputed_X_train = pd.DataFrame(my_imputer.fit_transform(X_train))
##### MUST only "regular" transform validation data
imputed_X_valid = pd.DataFrame(my_imputer.transform(X_valid))

# Imputation removed column names; put them back
imputed_X_train.columns = X_train.columns
imputed_X_valid.columns = X_valid.columns
```

QQ: I noticed that you called `fit_transform()` for the training dataset and `transform()` for the validation dataset. What gives?

Check out [this answer from stack exchange](#).

To center the data (make it have zero mean and unit standard error), you subtract the mean and then divide the result by the standard deviation:

$$x' = x - \mu\sigma$$

You do that on the training set of data. But then you have to apply the same transformation to your testing set (e.g. in cross-validation), or to newly obtained examples before forecast. But you have to use the exact same two parameters μ and σ (values) that you used for centering the training set.

Hence, every sklearn's transform's `fit()` just calculates the parameters (e.g. μ and σ in case of `StandardScaler`) and saves them as an internal object's state. Afterwards, you can call its `transform()` method to apply the transformation to any particular set of examples.

`fit_transform()` joins these two steps and is used for the initial fitting of parameters on the training set x , while also returning the transformed x' .

Internally, the transformer object just calls `first fit()` and then `transform()` on the same data.

Whhhhaaaaaattttt?

Given that there are so few missing values in the dataset, we'd expect imputation to perform better than dropping columns entirely. However, we see that dropping columns performs slightly better! While this can probably partially be attributed to noise in the dataset, another potential explanation is that the imputation method is not a great match to this dataset. That is, maybe instead of filling in the mean value, it makes more sense to set every missing value to a value of 0, to fill in the most frequently encountered value, or to use some other method. For instance, consider the `GarageYrBlt` column (which indicates the year that the garage was built). It's likely that in some cases, a missing value could indicate a house that does not have a garage. Does it make more sense to fill in the median value along each column in this case? Or could we get better results by filling in the minimum value along each column? It's not quite clear what's best in this case, but perhaps we can rule out some options immediately - for instance, setting missing values in this column to 0 is likely to yield horrible results! TLDR:

- There is no universal rule on what is better in which type of dataset, so you just gotta try 'em all.

Method 3: Extension to Imputation

```
# Make copy to avoid changing original data (when imputing)
X_train_plus = X_train.copy()
X_valid_plus = X_valid.copy()

# Make new columns indicating what will be imputed
for col in cols_with_missing:
    X_train_plus[col + '_was_missing'] = X_train_plus[col].isnull()
    X_valid_plus[col + '_was_missing'] = X_valid_plus[col].isnull()

# Imputation
my_imputer = SimpleImputer()
imputed_X_train_plus = pd.DataFrame(my_imputer.fit_transform(X_train_plus))
imputed_X_valid_plus = pd.DataFrame(my_imputer.transform(X_valid_plus))

# Imputation removed column names; put them back
imputed_X_train_plus.columns = X_train_plus.columns
imputed_X_valid_plus.columns = X_valid_plus.columns

print("MAE from Approach 3 (An Extension to Imputation):")
print(score_dataset(imputed_X_train_plus, imputed_X_valid_plus, y_train, y_valid))
```

REGARDLESS OF THE METHOD USED FOR REMOVING MISSING VALUES:

- To transform the actual dataset on which you will use your model to predict the answers for, use the following tool.

- Do not try to do this yourself manually... like how would you even do that? The final dataset will have different details missing from different columns than the dataset you used for training if you used the drop columns method for example.

```
# NOTE: this only works for imputation... I will add a method for dropping
the columns later.
# Use the following method to transform the final dataset so that it alligns
with the training dataset.
final_X_test = pd.DataFrame(final_imputer.transform(X_test))
```

3. Categorical Variables

- some variables only have a limited number of possible values
 - such as the brands of the cars owned by a set of people--it really only can be ['ford', 'tesla', 'honda', ... , 'and so on'].
- these variables are called **Categorical Variables**.

There are 3 main approaches to deal with categorical variables:

Sno | Method | Usefulness | description --- | ---- | ---- 1 | Drop Categorical Variables | Easiest [and worst] way | just remove them off the dataset. 2 | Label Encoding | Usually the 2nd best, but maybe better | Assign each value to an unique integer. 3 | One-Hot Encoding | Usuaully the best, use only with < 15 different possibilities | Create a true/false column for each possibility to denote if the item belongs to that class or not.

Method 1: Drop Categorical Variables

- Use this method IFF those variables have no useful information.

```
##### Example code to do this
# Get list of categorical variables
s = (X_train.dtypes == 'object') # get a dictionary with each column name followed
by a boolean that says if the data
                                # type that each column stores is "object" or
not.
object_cols = list(s[s].index) #

print("Categorical variables:")
print(object_cols)

# drop 'em
drop_X_train = X_train.select_dtypes(exclude=['object'])
drop_X_valid = X_valid.select_dtypes(exclude=['object'])

print("MAE from Approach 1 (Drop categorical variables):")
print(score_dataset(drop_X_train, drop_X_valid, y_train, y_valid))
```

Method 2: Label Encoding

Breakfast		Breakfast
Every day		3
Never		0
Rarely		1
Most days		2
Never		0



- Assumes an ordering/relative worth to each of these categories.
 - In this case, **Never (0) < Rarely (1) < Most Days (2) < Every Day (3)**
- If your categories can't really be ranked so clearly, they are called **ordinal variables**.
 - This method works well enough for these variables in tree based models
 - so be aware of this when looking for biases in data when using other types of models.
- Error Source:** The validation dataset might have unique values not seen before in the training dataset, so the model will crash.
 - Easiest way to solve it: Remove those problematic columns.

```
# All categorical columns
object_cols = [col for col in X_train.columns if X_train[col].dtype
== "object"]

# Columns that can be safely label encoded
good_label_cols = [col for col in object_cols if
                    set(X_valid[col]).issubset(set(X_train[col]))]

# Problematic columns that will be dropped from the dataset
bad_label_cols = list(set(object_cols)-set(good_label_cols))
```

Implementation:

- Scikit-learn has a **LabelEncoder** class that can be used to get label encodings. We loop over the categorical variables and apply the label encoder separately to each column.
- In this approach, we assign random unique integers for each possibility of the categorical variables for a column, which is a common (and easier) approach.
- To get a better model, you should manually assign better-informed numerical variables for each of these classes.


```
# Get a list of columns that store categorical variables
s = (X_train.dtypes == 'object')
object_cols = list(s[s].index)

# import the LabelEncoder class
from sklearn.preprocessing import LabelEncoder
# make a copy of the training and validation datasets
label_X_train = X_train.copy()
label_X_valid = X_valid.copy()
# Create a LabelEncoder object
encoder = LabelEncoder()
# Apply the label encoder to each column with categorical data
for col in object_cols:
    label_X_train[col] = label_encoder.fit_transform(X_train[col]) # fit and
transform for training dataset
    label_X_valid[col] = label_encoder.transform(X_valid[col]) # plain old
transform for validation dataset

#### At this point, feel free to train the model and see how well it works
```

Method 3: One-Hot encoding

Color			
Red			
Red			
Yellow			
Green			
Yellow			



Red	Yellow	Green
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0

- Create new columns to indicate if the given entry belongs to a given class in the form of true/false values.
- Use only if the number of possible values for the given categorical variable is less than 15 elements.
 - does not work well if the number of possibilities are greater than that 😞
- Works better than **label encoding** for nominal variables--i.e. variables whose possibilities do not have a clear ordering as per human logic.
- *Cardinality of a categorical variable*: The number of unique values that exist in the respective column.
- To save storage space, we only use **one-hot** encoding when the number of possible values for a given column is small
 - If you have a column with 100 unique values, and have a total of 10,000 rows, then you will add **99*10,000** extra data entries!!!
 - Drop or label-encode other columns with a large number of possibilities.

Implementation:

- Use the class **OneHotEncoder** from **scikit-learn**.
- A lot of parameters can be used to customize the behaviour of the class, a few examples are:

- `handle_unknown = 'ignore'` --> If the validation dataset contains classes not seen before in the training dataset, ignore that data entry.
- `sparse = False` --> return the encoded columns as a numpy array instead of a sparse matrix.
- supply only the categorical columns that we want to be one-hot encoded.
 - For instance, to encode the training data, we supply `X_train[object_cols]`.
 - where `object_cols` is a list of all columns with categorical variables.

```
from sklearn.preprocessing import OneHotEncoder

# Apply one-hot encoder to each column with categorical data, store them in their
own columns
OH_encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
OH_cols_train = pd.DataFrame(OH_encoder.fit_transform(X_train[object_cols]))
OH_cols_valid = pd.DataFrame(OH_encoder.transform(X_valid[object_cols]))

# One-hot encoding removed index; put it back
OH_cols_train.index = X_train.index
OH_cols_valid.index = X_valid.index

# Remove categorical columns (will replace with one-hot encoding)
num_X_train = X_train.drop(object_cols, axis=1)
num_X_valid = X_valid.drop(object_cols, axis=1)

# Add one-hot encoded columns to numerical features
OH_X_train = pd.concat([num_X_train, OH_cols_train], axis=1)
OH_X_valid = pd.concat([num_X_valid, OH_cols_valid], axis=1)

print("MAE from Approach 3 (One-Hot Encoding):")
print(score_dataset(OH_X_train, OH_X_valid, y_train, y_valid))
```

4. Pipelines

QQ	Answer
What?	A simple way to keep code organized - bundle preprocessing and modeling into one single step.
Why?	Cleaner code, Fewer bugs, Easier to deploy to production, More options for model validation (example: Cross-Validation)

Three Steps to create a pipeline

Step 1: Define Preprocessing Steps

- Bundle all the preprocessing steps using a `sklearn.compose.ColumnTransfer`

```
# Import the ColumnTransfer class
from sklearn.compose import ColumnTransfer
# Import the Pipeline class because that is just how you combine multiple steps...
from sklearn.pipeline import Pipeline
```



```
# Import the libraries needed for the actual preprocessing
from sklearn.impute import SimpleImputer # imputer to fill in missing values using
the average of each column
from sklearn.preprocessing import OneHotEncoder # encoder to convert Categorical
variables into numbers

# Preprocessing for numerical data -- Replace missing values with column averages
numerical_transformer = SimpleImputer(strategy='constant')

# ----- pipeline syntax example -----
# Preprocessing for categorical data -- Replace missing values with the most
frequently occurring value, and then OneHotEncode it.
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Bundle preprocessing for numerical and categorical data
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

```

Step 2: Define the Model

- Nothing special here

```
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators=100, random_state=0)

```

Step 3: Create and Evaluate the Pipeline

- Just like we did for preprocessing, we will now create a pipeline for the whole thing
- Notice:
 - Only a single line of code needed to do *everything* - imputation, OneHotEncoding, model training, etc..
 - Now, just directly pass in the unprocessed dataset to the pipeline line in the `.predict(data_set)` command, and it takes care of all the preprocessing for you!
- Pipeline syntax:
 - `Pipeline(steps=list(tuples))`, where `tuples = 'label', method`.

```
from sklearn.metrics import mean_absolute_error

# Bundle preprocessing and modeling code in a pipeline
my_pipeline = Pipeline(steps=[('preprocessor', preprocessor),

```

```

        ('model', model)
    ])

# Preprocessing of training data, fit model
my_pipeline.fit(X_train, y_train)

# Preprocessing of validation data, get predictions
preds = my_pipeline.predict(X_valid)

# Evaluate the model
score = mean_absolute_error(y_valid, preds)
print('MAE:', score)

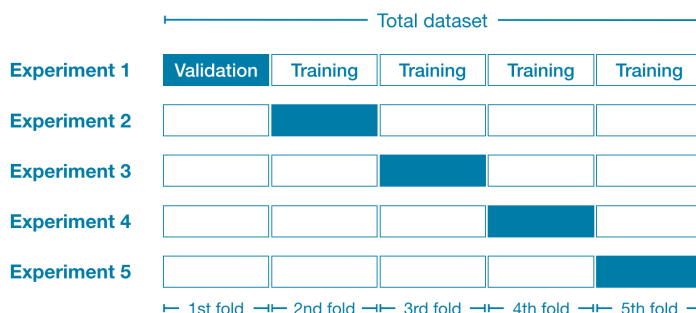
```

5. Cross-Validation

- Usually, we have to split the dataset into training and validation to be able to find the best training parameters and pre-processing methods.
- BUT: more the data our model is exposed to, the better.
- With a 80-20 training-validation split, **WHICH** 80% the model is accurate on could make a huge difference
- in cross-validation, you train and test the model multiple times, just changing which portion of data is hidden away during training.

Welcome cross-validation:

- In cross-validation, we run our modeling process on different subsets of the data to get multiple measures of model quality.
- For example, we could begin by dividing the data into 5 pieces, each 20% of the full dataset. In this case, we say that we have broken the data into 5 "folds".



Run one *experiment* per fold:

- ex 1: first fold = validation, others training
- ex 2: second fold = validation, others training ...
- keep going on until all data has been used to test and improve the model.

When to use cross-validation:

Note: with cross validation you are basically creating MULTIPLE models (one for each fold as validation set) so it is MANY times slower.

- Small datasets: extra computation doesn't take much time, MUST use cross-validation
- Large datasets: enough data exists that which 20% is hidden doesn't make much of a difference -- don't use cross-validation.

No exact definition on what's big and what's small -- intuition.

- Alternatively, you can run cross-validation and see if the scores for each experiment seem close. If each experiment yields the same results, a single validation set is probably sufficient

Example:

- While it's possible to do cross-validation without pipelines, it is quite difficult! Using a pipeline will make the code remarkably straightforward.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer

my_pipeline = Pipeline(steps=[('preprocessor', SimpleImputer()),
                              ('model', RandomForestRegressor(n_estimators=50,
                                                              random_state=0))
                              ])

# sci-kit learn's inbuilt tool that helps us obtain the cross-validation mae
# scores.
from sklearn.model_selection import cross_val_score

# Multiply by -1 since sklearn calculates *negative* MAE
scores = -1 * cross_val_score(my_pipeline, X, y,
                              cv=5,
                              scoring='neg_mean_absolute_error')

print("MAE scores:\n", scores)
```

```
MAE scores:
[301628.7893587  303164.4782723  287298.331666   236061.84754543
 260383.45111427]
```

Also note:

- Using cross-validation yields a much better measure of model quality, with the added benefit of cleaning up our code: note that we no longer need to keep track of separate training and validation sets. So, especially for small datasets, it's a good improvement!

6. XGBoost

- In this tutorial, you will learn how to build and optimize models with gradient boosting.

- This method dominates many Kaggle competitions and achieves state-of-the-art results on a variety of datasets.
- Have been using random forest method, which achieves better performance than a single decision tree simply by averaging the predictions of many decision trees.

Random Forest method is an **ensemble method**

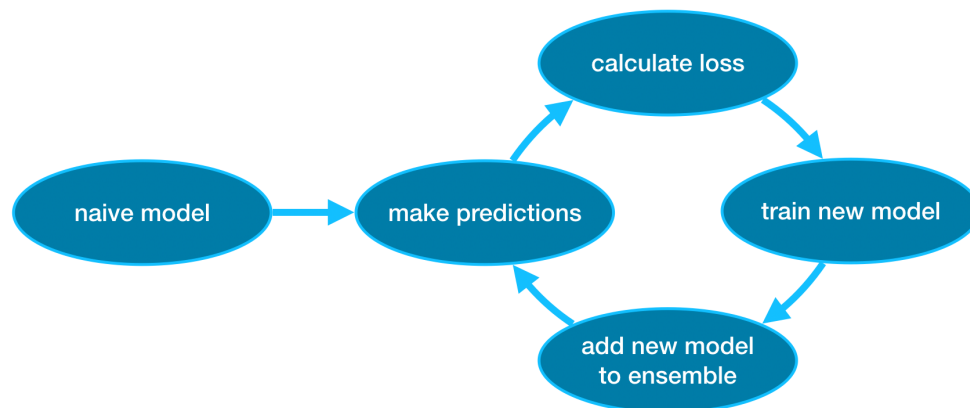
- ensemble methods combine the predictions of several different models (trees in the case of a random forest) to give a final prediction.
- Gradient boosting is another example of this method.

What is Gradient Boosting?

- Gradient boosting is a method that goes through cycles to iteratively add models into an ensemble.
- It begins by initializing the ensemble with a single model, whose predictions can be pretty naive. (Even if its predictions are wildly inaccurate, subsequent additions to the ensemble will address those errors.)

In each iteration:

1. Use the current ensemble to make predictions for each testing row. To make a prediction, add up all the predictions from all models in the ensemble.
2. Use these predictions to calculate the loss function (ex: mean squared error). Goal: minimize this loss function--hence the name "*Gradient Boosting*."
3. Create a new model with such parameters that adding this new model to the ensemble will reduce the loss.
4. Add the new model to the ensemble.
5. repeat!



Usage:

XGBoost stands for *extreme gradient boosting*, which is an implementation of gradient boosting with **several additional features focused on performance and speed**. (Scikit-learn has another version of gradient boosting, but XGBoost has some technical advantages.)

In the next code cell, we import the scikit-learn API for XGBoost (`xgboost.XGBRegressor`) and also introduce ourselves to the many parameters that XGBoost opens up for tuning.

Parameters: !!! (Quite important)

```
from xgboost import XGBRegressor
my_model = XGBRegressor(n_estimators=1000, learning_rate=0.05, n_jobs=4)
my_model.fit(X_train, y_train,
             early_stopping_rounds=5,
             eval_set=[(X_valid, y_valid)],
             verbose=False)
```

S.no	Parameter name	what does it alter?	How does it help?	Where to use it?
1	<code>n_estimators</code>	No. of times to repeat modeling cycle	Too low -- underfitting, Too high -- overfitting. Typical values range from 100-1000.	model defn.
2	<code>early_stopping_rounds=a</code>	If the quality of model decreases for 'a' straight iterations, stop training now regardless of <code>n_estimators</code> parameter	good way to find the ideal <code>n_estimators</code> value. Needs eval data to know when to stop--pass in X_valid and y_valid as paramters to <code>eval_set</code> . Set <code>n_estimators</code> high and use an appropriate <code>a</code> value.	model fitting.
3	<code>learning_rate</code>	How much each additional model contributes to the model.	low learning rate ==> overfitting not a problem. default value = 0.1. Good Practice: Low <code>learning_rate</code> , high <code>n_estimators</code> .	model defn
4	<code>n_jobs</code>	number of threads used from the processor used during training	makes training faster for large dataset by enabling parallelization.	model defn.

7. Data Leakage

- 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.
- 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.
- 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.

Two types of data leakage exists:

- Target

- Train-test contamination

Target Leakage

- using data that **won't be available in real life before the model is used for predictions** to train the data.
- Think timeline ==> will I know the correct value of field **a** in a record before my model is called upon to predict the value of another variable **b** for that record.
- Is target leakage happens, you will still get very good predictions in training and testing data (even with crossvalidation!) but absolutely terrible predictions in production.
- just because a field is in the dataset doesn't mean it is relevant to the predictions or even if we know its value before the model has to make a prediction.
- solution: drop such columns, there is literally nothing else you can do.
 - "To prevent this type of data leakage, any variable updated (or created) after the target value is realized (in real life, not training data) should be excluded." **Example:**
- patient data has a bunch of data, among which are
 - is_diseased
 - is_under_treatment
 - where **is_diseased** is what you want to predict using your model.
- The true value of **is_under_treatment** can only be determined AFTER we know if someone **is_diseased**
- just because someone is not under treatment yet doesn't mean they are not diseased--so it is just stupid to use if they are being treated as a factor that is used to predict if the person is diseased in the first place.

Train-Test Contamination

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

- You can corrupt this process in subtle ways if the validation data affects the preprocessing behavior. This is sometimes called **train-test contamination**.
- caused by doing *any* kind of **fitting**--be it for preprocessing or training the model--BEFORE the 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.
- this is why you should always use pipelines to do EVERYTHING really--all preprocessing and fitting.

Example:

```
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: 0.980294

An accuracy of 98% is a tad bit too high to be true --> lets inspect more...

The definitions of what is stored in a given variable can be quite hazy: Take this list of variables from the database:

variable	definition
card	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

- And take **expenditures** for example: Is the expenditure using the card which you are trying to predict if it will be issued, or is it from other sources.
 - if it is from the card in question, then target leakage Klaxons must go off
 - as this **expenditure** variable can be known ONLY IF AND AFTER the card is issued.
 - else, no worries.
- be sure to disregard any other variable whose value depends on the leaked variable!
 - so, in this case, the variable **share** defined as the ratio of expenditure to income should also be removed as its value depends on **expenditure**.
- The definition is unclear for **majorcards** and **active** too (are they from before or after the card was issued?) -- so nuke them too, just in case.

Think.exe: Example situations for determining if/not there is a data-leakage.

- This is tricky, and it depends on details of how data is collected (which is common when thinking about leakage). Would you at the beginning of the month decide how much leather will be used that month? If so, this is ok. But if that is determined during the month, you would not have access to it when you make the prediction. If you have a guess at the beginning of the month, and it is subsequently changed during the month, the actual amount used during the month cannot be used as a feature (because it causes leakage).
- You have a new idea. You could use the amount of leather Nike ordered (rather than the amount they actually used) leading up to a given month as a predictor in your shoelace model.
 - This could be fine, but it depends on whether they order shoelaces first or leather first. If they order shoelaces first, you won't know how much leather they've ordered when you predict their shoelace needs. If they order leather first, then you'll have that number available when you place your shoelace order, and you should be ok.
- These features should be available at the moment you want to make a prediction, and they're unlikely to be changed in the training data after the prediction target is determined.
 - But, the way he describes accuracy could be misleading if you aren't careful. If the price moves gradually, today's price will be an accurate predictor of tomorrow's price, but it may not tell you whether it's a good time to invest.
- This poses a risk of both target leakage and train-test contamination (though you may be able to avoid both if you are careful).
 - You have target leakage if a given patient's outcome contributes to the infection rate for his surgeon, which is then plugged back into the prediction model for whether that patient becomes infected.
 - You also have a train-test contamination problem if you calculate this using all surgeries a surgeon performed, including those from the test-set. This would happen because the surgeon-risk feature accounts for data in the test set. Test sets exist to estimate how the model will do when seeing new data. So this contamination defeats the purpose of the test set.
- We don't know the rules for when this is updated. If the field is updated in the raw data after a home was sold, and the home's sale is used to calculate the average, this constitutes a case of target leakage. At an extreme, if only one home is sold in the neighborhood, and it is the home we are trying to predict, then the average will be exactly equal to the value we are trying to predict. In general, for neighborhoods with few sales, the model will perform very well on the training data. But when you apply the model, the home you are predicting won't have been sold yet, so this feature won't work the same as it did in the training data

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.

Commands Bitbucket

```
##### Drop columns with missing values.
# Run this once you have found the names of all the columns with missing values.
reduced_X_train = X_train.drop(cols_with_missing, axis=1)

##### Use a SimpleImputer to replace missing values with their averages in
the column.
# Import the library
from sklearn.impute import SimpleImputer
# Create the imputer object.
my_imputer = SimpleImputer()
# call it on both the training and testing datasets.
imputed_X_train = pd.DataFrame(my_imputer.fit_transform(X_train))
imputed_X_valid = pd.DataFrame(my_imputer.transform(X_valid))

# Imputation removed column names; put them back
imputed_X_train.columns = X_train.columns
imputed_X_valid.columns = X_valid.columns

##### To implement the extension to imputation method, do some along the line
of:
# Make new columns indicating what will be imputed
for col in cols_with_missing:
    X_train_plus[col + '_was_missing'] = X_train_plus[col].isnull()
    X_valid_plus[col + '_was_missing'] = X_valid_plus[col].isnull()

##### Get a count of missing values by column
# Number of missing values in each column of training data
missing_val_count_by_column = (X_train.isnull().sum())
### side note: WOW, python can do this?? the array access works in mysterious ways
in python--at least to my c++ mind.
print(missing_val_count_by_column[missing_val_count_by_column > 0])

##### Use the following method to transform the final dataset so that it
aligns with the training dataset.
final_X_test = pd.DataFrame(final_imputer.transform(X_test))

##### Drop Categorical variables
drop_X_train = X_train.select_dtypes(exclude=['object'])
drop_X_valid = X_valid.select_dtypes(exclude=['object'])

##### Get number of unique entries in each column with categorical data
object_nunique = list(map(lambda col: X_train[col].nunique(), object_cols))
d = dict(zip(object_cols, object_nunique))

# Print number of unique entries by column, in ascending order
sorted(d.items(), key=lambda x: x[1])

##### Subtract two lists
bad_label_cols = list(set(object_cols)-set(good_label_cols))
```

