Cross Validation

You're already well aware of the importance *of* splitting your data into training and testing sets to validate the accuracy of your models by checking how well it fits the data. This method of avoiding overfitting introduces three issues. The first is each time you run train / test split, unless you set the `random_state` parameter, you're going to get back different accuracy scores! A few of you have already experienced this with a couple of the automatically graded labs answers =).

```
>>> from **sklearn.cross_validation** import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=

>>>  Test how well your model can recall its training data:
>>> model.fit(X_train, y_train).score(X_train, y_train)
0.943262278808

>>>  Test how well your model can predict unseen data:
>>> model.fit(X_test, y_test).score(X_test, y_test)
0.894716422024
```

SciKit-Learn's `train_test_split()` method selects a random subset of your data to withhold as the testing validation set. Without a deterministic selection of training data and testing data, you might train using the best subset of data but test on outliers, or some permutation in-between. The second issue introduced is by withholding data from training, you essentially lose some of your training data! Machine learning is only as accurate as the data its trained upon, so generally more data means better results. Neglecting to train your models on your hard collected data is like refusing to take your rightful change at the bank.

But the most important issue introduced is that with some of the more configurable estimators, such as SVC, you will probably end up running your model many times while tinkering with the various parameters, such as C and gamma. In fact, you were instructed to do just that in one of your labs, by writing nested for loops that iterated

over your desired parameter space. The problem with this is, you're going to stop tinkering once you get to a set of parameter that produce optimal results. By doing this, you've essentially *leaked* some information from your testing set into your training set—not good! Your model, armed with these secret details about your testing set, might still perform poorly in the real-world if it overfit that data.

The way to overcome this is by splitting your data once more. Now you have a training set, a testing set, and a validation set that you don't optimize your models against and only use for scoring. Such a process is too tedious. Having to deal with three sets of data means you must remember to do the same transformations to all sets, otherwise you'll get incorrect shape errors when fitting and training. This just introduces opportunities for silly mistakes. Part of being a great programmer is getting the machine to do all the tedious tasks in order to increase your efficiency as much as possible. The creators of SciKit-Learn totally got that, and have put together your new favorite method: `cross_val_score()`.

This method takes as input your model along with your **training** dataset and performs K-fold cross validations on it. In other words, your training data is first cut into a number of "K" sets. Then, "K" versions of your model are trained, each using an independent K-1 number of the "K" available sets. Each model is evaluated with the last set, it's out-of-bag set. If this sounds super familiar to you, it's because this is the same bootstrapping technique used in random forest.

```
# 10-Fold Cross Validation on your training data
>>> from sklearn import cross_validation as cval
>>> cval.cross_val_score(model, X_train, y_train, cv=10)
array([ 0.93513514,  0.99453552,  0.97237569,  0.98888889,  0.96089385,
        0.98882682,  0.99441341,  0.98876404,  0.97175141,  0.96590909])

>>> cval.cross_val_score(model, X_train, y_train, cv=10).mean()
0.97614938602520218
```

Cross validation allows you to use all the data you provide as both *training* and *testing*, so many resources online will recommend you don't even do the extra step of splitting your data into a training and testing set and just feed the lot directly into your cross validator. There are advantages and disadvantages of this. The main advantage is the overall simplicity of your process. The disadvantage is that it still is possible for some information to leak into your training dataset, as we discussed above with the SVC

example. This information leak might even occur prior to you fitting your model, for example it might be at the point of transforming your data using isomap or principle component analysis.

For these reasons, the SciKit-Learn documentation recommends you still keep a completely separate testing set to conduct scoring, after cross validating your models. **Make sure** you read through the documentation before going through the knowledge checks, particularly for the `cv` parameter and the different types of cross validator iterators. For example, it's absolutely critical that you know the difference between *KFold* and *StratifiedKFold* before you start using `cross_val_score()`, to avoid introducing a serious bug into your machine learning!

In the wild, the **best** process to use depending on how many samples you have at your disposal and the machine learning algorithms you are using, is either of the following:

1. Split your data into training, validation, and testing sets.
2. Setup a pipeline, and fit it with your **training** set
3. Access the accuracy of its output using your **validation** set
4. Fine tune this accuracy by adjusting the hyper-paramters of your pipeline
5. when you're comfortable with its accuracy, finally evaluate your pipeline with the **testing** set

OR

1. Split your data into training and testing sets.
2. Setup a pipeline with CV and fit / score it with your **training** set
3. Fine tune this accuracy by adjusting the hyper-paramters of your pipeline
4. When you're comfortable with its accuracy, finally evaluate your pipeline with the testing set