W4995 Applied Machine Learning

Introduction to Supervised Learning

01/23/17

Andreas Müller

Hey everybody.
Today, we'll be talking more in-depth about supervised learning, model evaluation and model selection.
You might have seen, I changed the syllabus a bit based on the feedback I got so far.
Also, I changed the due date of the homework, as I promised last week.
Oh and if you want, there's a jupyter notebook for todays lecture in the github repository.

Supervised Learning

$$(x_i, y_i) \propto p(x, y) \quad \text{i.i.d.}$$

$$x_i \in \mathbb{R}^n$$

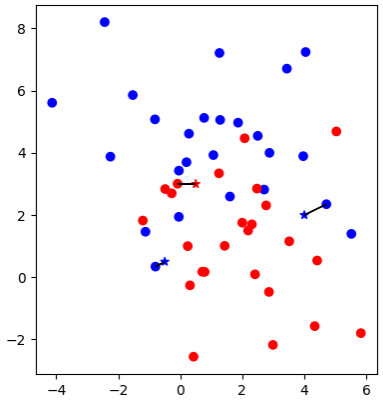
$$y_i \in \mathbb{R}$$

$$f(x_i) \approx y_i \qquad f(x) \approx y$$

But let's dive right back in into supervised learning.
As a reminder, in supervised learning, the dataset we learn form is input-output pairs (x_i, y_i), where x_i is some n_dimensional input, or feature vector, and y_i is the desired output we want to learn.
We assume these samples are drawn from some unknown joint distribution p(x, y).
You can think of this as there being some (not necessarily deterministic) process that goes from x_i to y_i, but that we don't know.
We try to find a function that approximates the output y_i for each known input x_i. But we also demand that for new inputs x , and unobserved y, f(x) is approximately y.

I want to go through to simple examples of supervised algorithms today, nearest neighbors and nearest centroids.

Nearest neighbors



$$f(x) = y_i, i = \text{argmin}_j ||x_j - x||$$

Let's say we have this two-class classification dataset here, with two features, one on the x axis and one on the y axis.

And we have three new points as marked by the stars here.

If I make a prediction using a one nearest neighbor classifier, what will it predict?

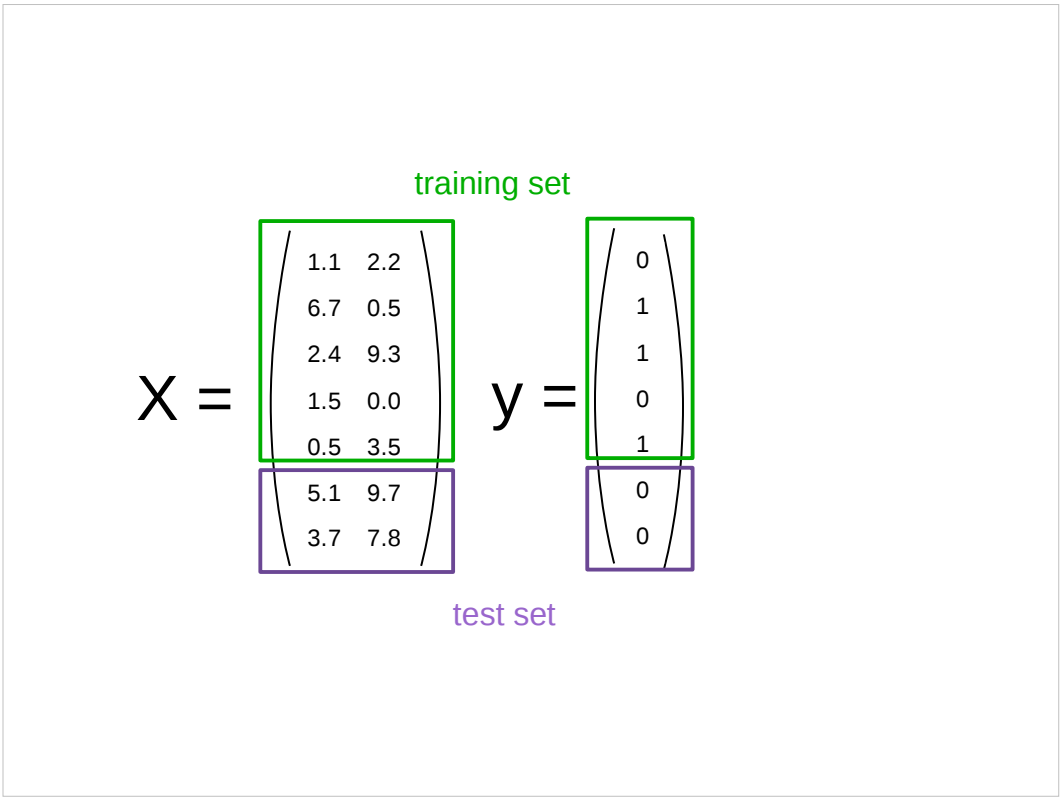
It will predict the label of the closest data point in the training set.

That is basically the simplest machine learning algorithm I can come up with.

Here's the formula:

the prediction for a new x is the y_i so that x_i is the closest point in the training set.

Ok, so now how do we find out whether this model is any good?



We split the data into a training and a test set.

So we take some part of the data set, let's say 75%
and the corresponding output, and train the model,
and then apply the model on the remaining 25% to
compute the accuracy

How do we do this in scikit-learn?

With train_test_split.

Implementing KNN in scikit-learn

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)

from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("accuracy: {:.2f}".format(knn.score(X_test, y_test)))
```

```
accuracy: 0.77
```

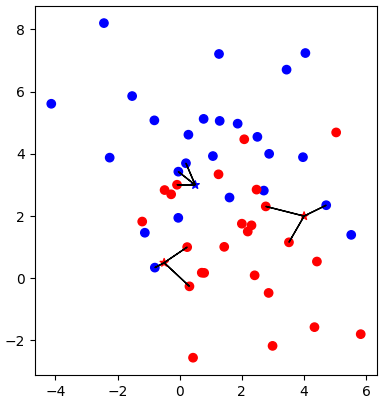
Ok so we import train_test_split form model selection, which does a random split into 75%/25%.
We provide it with the data X, which are our two features, and the labels y.

As you might already know, all the models in scikit-learn are implemented in python classes, with a single object used to build and store the model.

We start by importing our model, the KneighborsClassifier, and instantiate it with n_neighbors=1.
Then we can call the fit method to build the model, here knn.fit(X_train, y_train)
All models in scikit-learn have a fit-method, and all the supervised ones take the data X and the outcomes y.
Then, we can use knn.score to make predictions on the test data, and compare them against the true labels y_test.

For classification models, the score method will always compute accuracy. Who here has not seen this before?

Nearest neighbors



So this was the predictions as made by one-nearest neighbor.

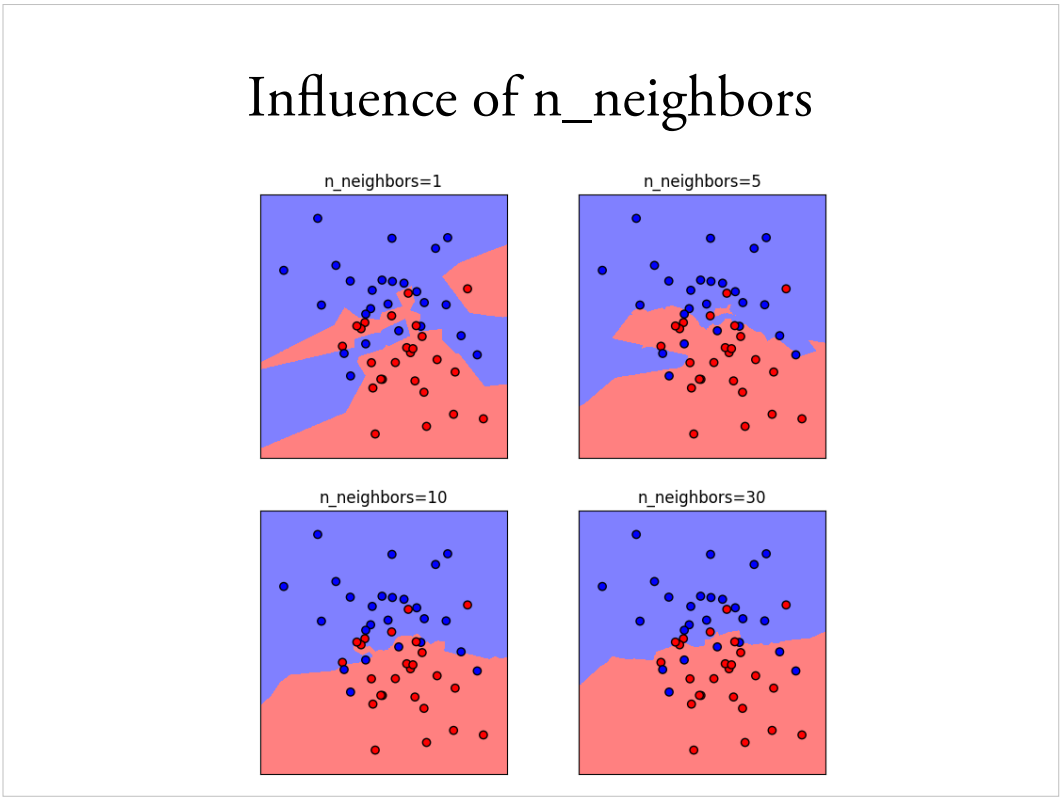
But we can also consider more neighbors, for example three. Here is the three nearest neighbors for each of the points and the corresponding labels.

We can then make a prediction by considering the majority among these three neighbors.

And as you can see, in this case all the points changed their labels! (I was actually quite surprised when I saw that, I just picked some points at random).

Clearly the number of neighbors that we consider matters a lot. But what is the right number?

The is a problem you'll encounter a lot in machine learning, the problem of tuning parameters of the model, also called hyper-parameters, which can not be learned directly from the data.



Here's an overview of how the classification changes if we consider different numbers of neighbors.

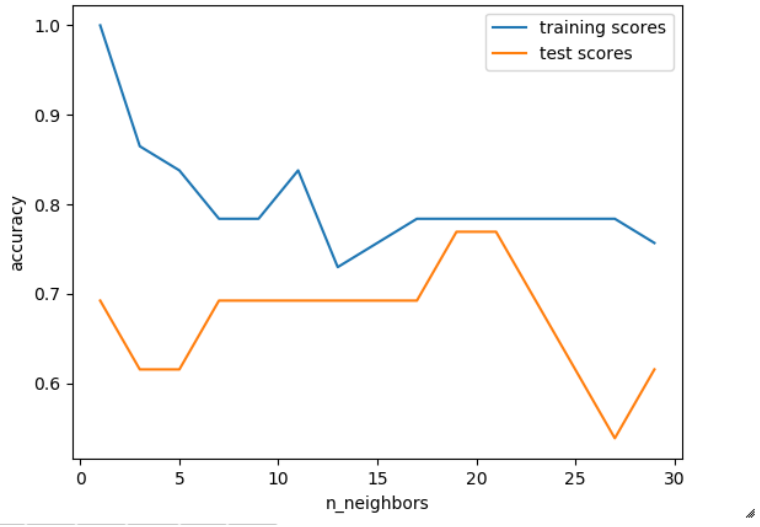
You can see as red and blue circles the training data. And the background is colored according to which class a datapoint would be assigned to for each location.

For one neighbor, you can see that each point in the training set has a little area around it that would be classified according to it's label. This means all the training points would be classified correctly, but it leads to a very complex shape of the decision boundary.

If we increase the number of neighbors, the boundary between red and blue simplifies, and with 40 neighbors we mostly end up with a line.

This also means that now many of the training data points would be labeled incorrectly.

Model Complexity

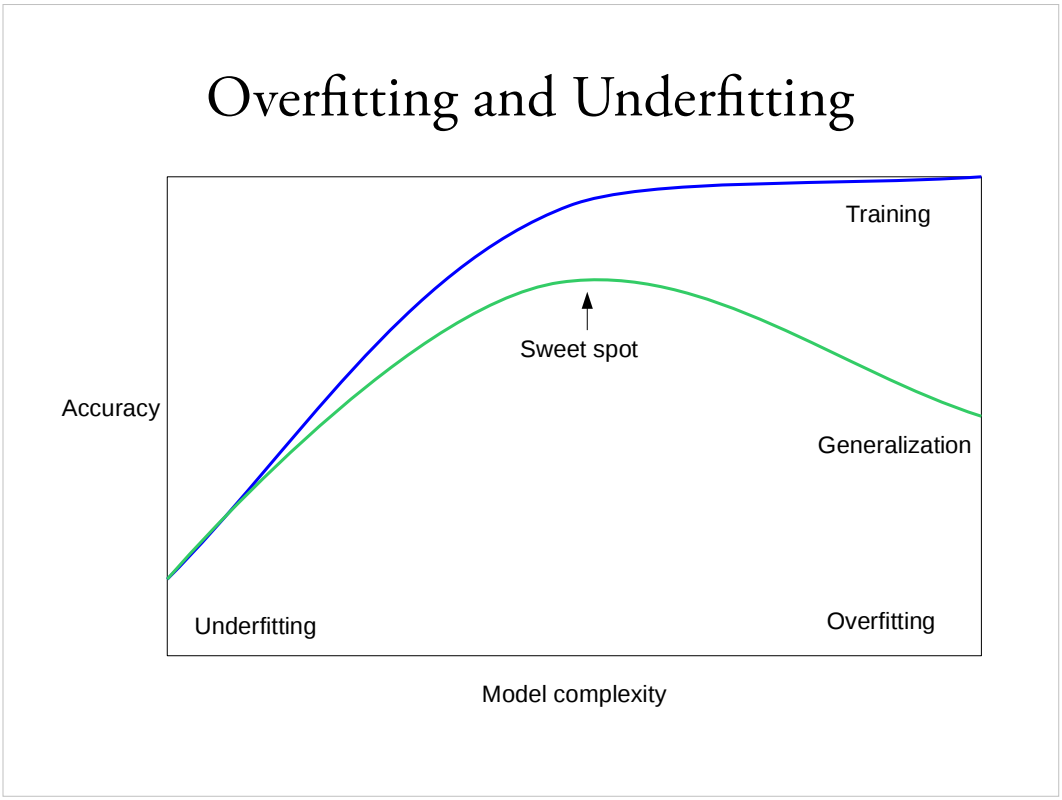


We can look at this in more detail by comparing training and test set scores for the different numbers of neighbors.

Here, I did a random 75%/25% split again. This is a very noisy plot as the dataset is very small and I only did a random split, but you can see a trend here.

You can see that for a single neighbor, the training score is 1 so perfect accuracy, but the test score is only 70%. If we increase the number of neighbors we consider, the training score goes down, but the test score goes up, with an optimum at 19 and 21, but then both go down again.

This is a very typical behavior, that I sketched in a schematic for you.



This chart has accuracy on the y axis, and an abstract concept of model complexity on the x axis.

If we make our machine learning models more complex, we will get better training set accuracy, as the model will be able to capture more of the variations in the data.

But if we look at the generalization performance, we get a different story. If the model complexity is too low, the model will not be able to capture the main trends, and a more complex model means better generalization.

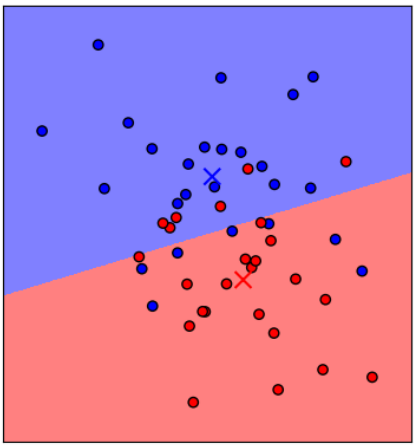
However, if we make the model too complex, generalization performance drops again, because we basically learn to memorize the dataset.

If we use too simple a model, this is often called underfitting, while if we use to complex a model, this is called overfitting. And somewhere in the middle is a sweet spot.

Most models have some way to tune model complexity, and we'll see many of them in the next couple of weeks.

So going back to nearest neighbors, what parameters correspond to high model complexity and what to low model complexity? high n_neighbors = low complexity!

Nearest Centroid



$$f(x) = argmin_{i \in Y} ||\bar{x}_i - x||$$

Before we go to more techniques to find the right hyper parameters, I want to introduce another simple model, nearest centroid.

The nearest centroid model simply computes the centroid, or the mean of each class, and classifies each point by which centroid is the closest.

You can write that down as a formula such as here, where x_i bar is the centroid, and I runs over the classes.

You can see that this yields a linear decision boundary between the two classes.

Nearest Centroids with scikit-learn

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)

from sklearn.neighbors import NearestCentroid
nc = NearestCentroid()
nc.fit(X_train, y_train)
print("accuracy: {:.2f}".format(nc.score(X_test, y_test)))
```

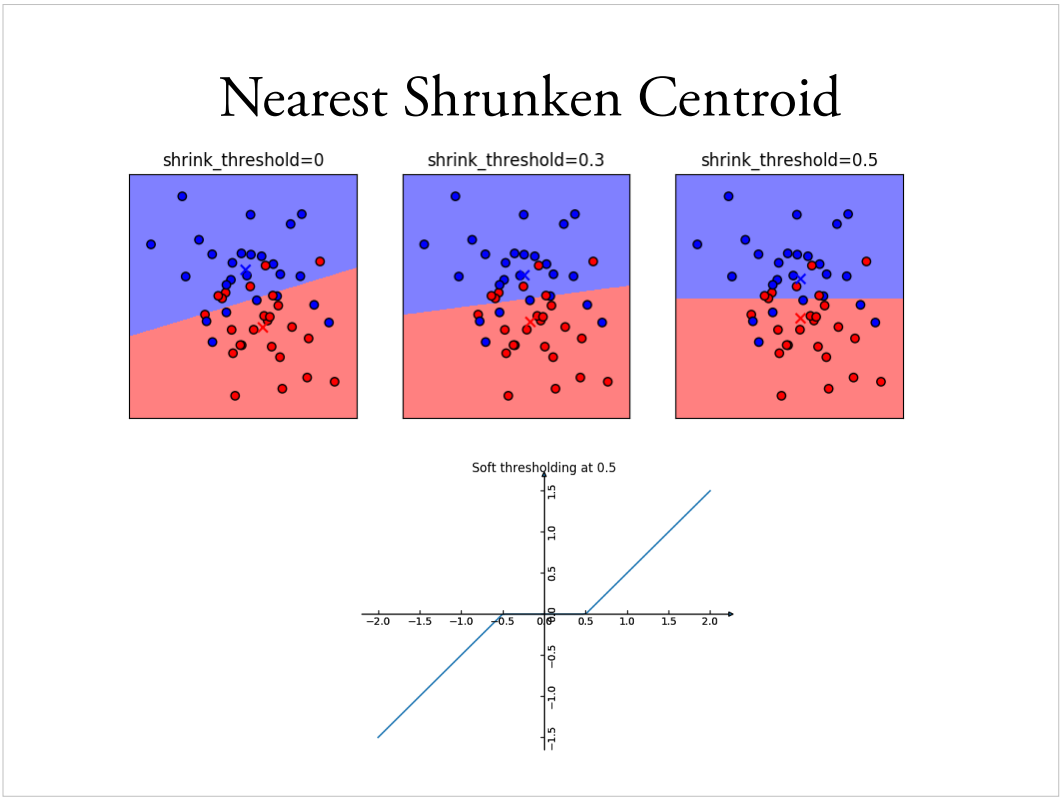
```
accuracy: 0.77
```

Again, we can easily build this model with scikit-learn and evaluate it using a test set.

We use the train_test_split function to split the data into 75% training and 25% test data,

import the nearest centroid model, fit it on the training set, and compute the accuracy on the test set.

Questions so far?



There's a variant of the nearest centroid called the nearest shrunken centroid, which allows you to limit model complexity.

In nearest shrunken centroid, you pick a positive shrinking threshold, and then you apply the soft-threshold function here to each of the centroids.

The soft-thresholding function shrinks each component of a vector towards zero by the threshold, and if that would cross zero, to set this component to zero.

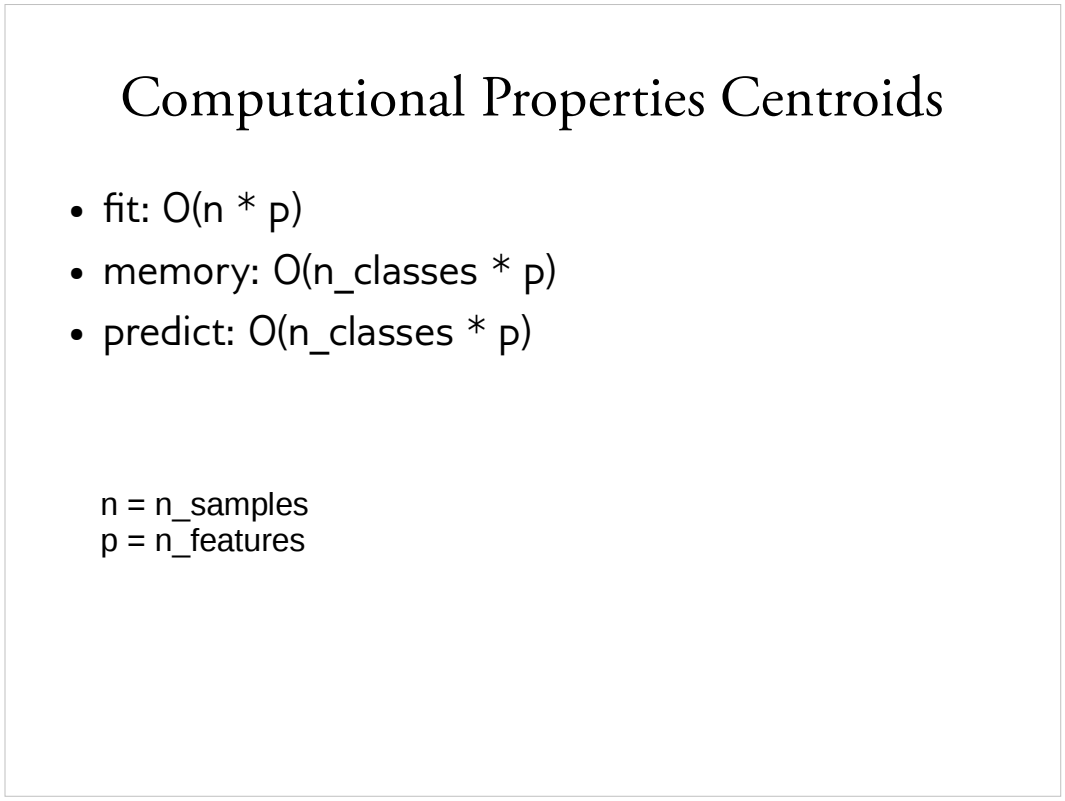
Actually the mean of the data is subtracted beforehand so that this makes sense ;)

What that means is that if the centroids have an entry that is close to zero, this direction gets ignored.

Here you can see different settings of the shrinking threshold. No shrinking, a threshold of .3 and a threshold of .5.

Because the centers are less than .5 apart from the data mean, the last panel has a perfectly horizontal line.

When do you think NC works better than KNN? High dimensions!



I want to compare some computational properties of the nearest shrunken centroid and k nearest neighbors.

What do you think are the important computational properties of machine learning models?

There are basically three: time to build the model, memory to store the model, and time to make a prediction.

What are those?

Fit time is simple, it's computing the centroids which is n * p. Basically we need to look at each feature exactly once.

Memory is storing the centroids, which is number of classes times number of features, and predict time is computing the distance to each of them, which is the same. When is this slow? Basically never - in high dim with only few important, maybe trees could be better, but we could also shrink those dimensions away.

Computational Properties Neighbors

- fit: no time
- memory: O(n * p)
- predict: O(n * p)

Kd-tree

- fit: O(n log n)
- memory: O(n * p)
- predict:
  O(k * log(n))
  FOR FIXED p!

n = n_samples
p = n_features

Ok, now for the k nearest neighbors. How would you implement this? (brute force or neighbors structure)
When not using a structure: fit no time, memory is the whole dataset, and prediction requires comparing against all data points.
If we create a kd-tree or ball tree?
Fit takes O(n log n), memory is the same. Prediction might be faster in low dimensions.

What's bad about this?
memory scales with data, prediction speed scales with data :-/

Can have high complexity models though.
When would you use one vs the other? Large datasets , high dimensions: use nearest centroid maybe.

Parametric and non-parametric models

- Parametric model:

  Number of “parameters” (degrees of freedom) independent of data.
- Non-parametric model:

  Degrees of freedom increase with more data.

These two models are examples of two general families of machine learning models, parametric models and non-parametric models.

Anyone know what they are?

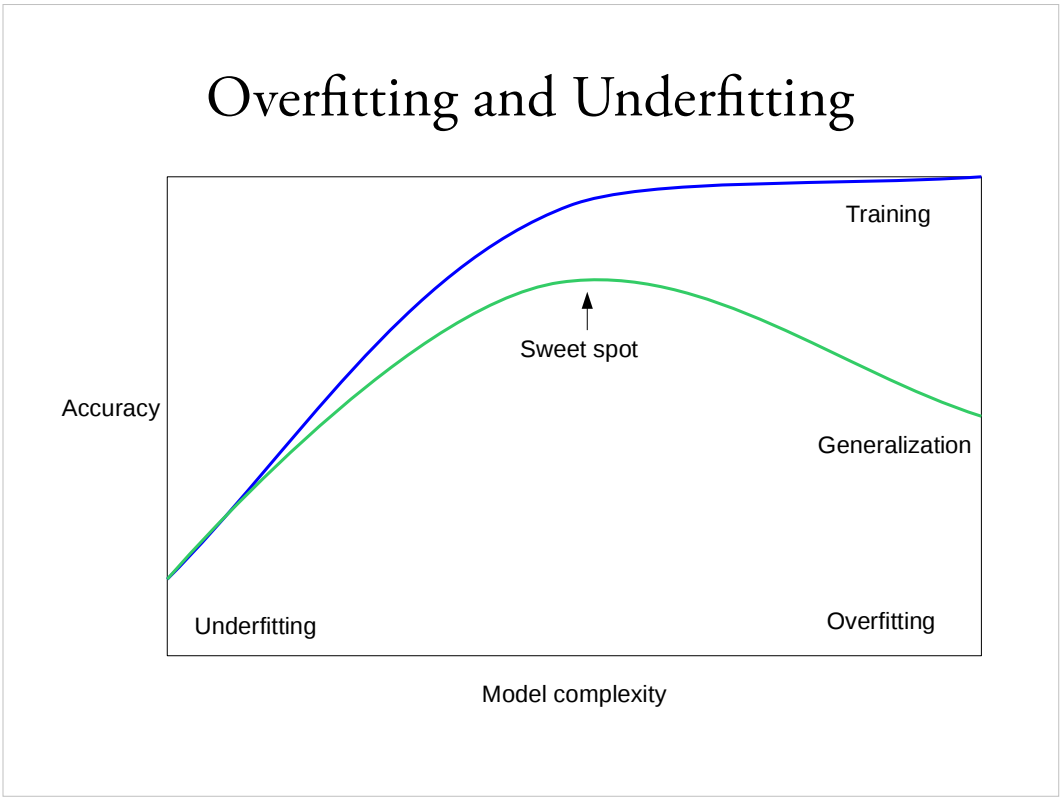
Parametric models have a fixed set of parameters that are learned from the data, such as the centroids.

There are always n_classes * n_features many numbers, no matter how much data you have.

Non-parametric models scale their complexity with data, such as nearest neighbors. More data here means more complex decision boundaries, and more numbers to store.

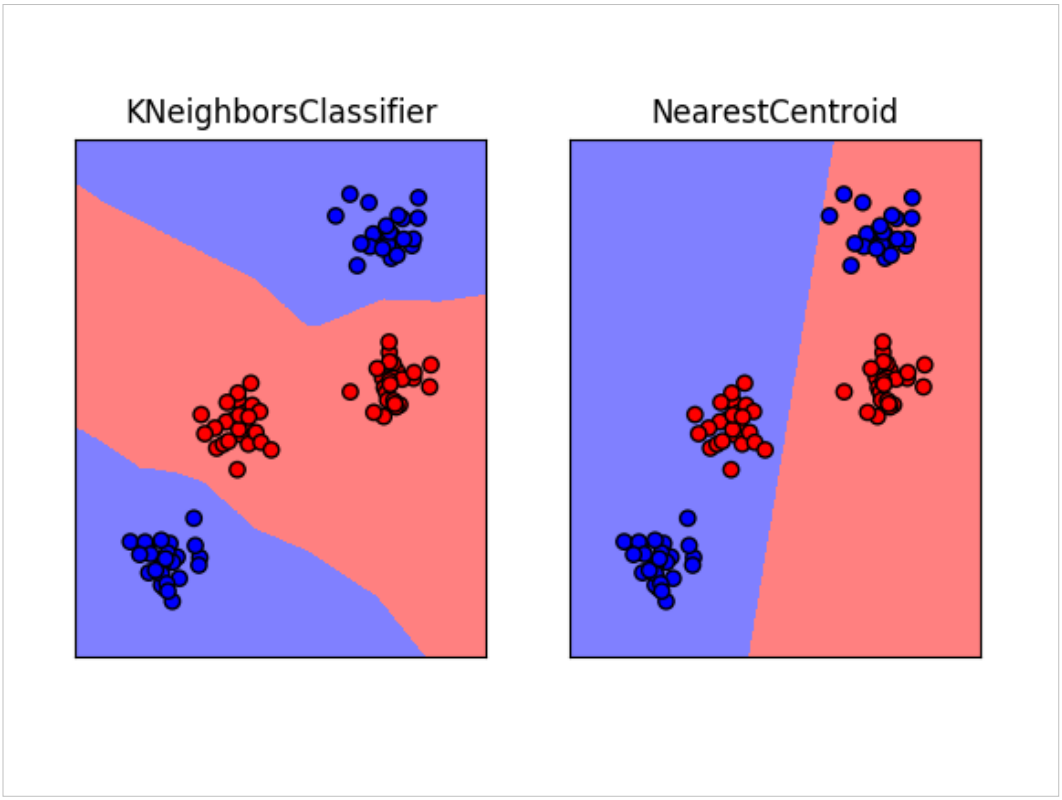
Examples?

Trees and forests and SVMs non-parametric, linear models parametric, neural nets parametric (arguably, since they have sooo many parameters).



Coming back to this chart, you usually find that non-parametric models are more likely to overfit, as they can increase their capacity to fit any dataset, while in some cases, a parametric model might not be able to fit some data.

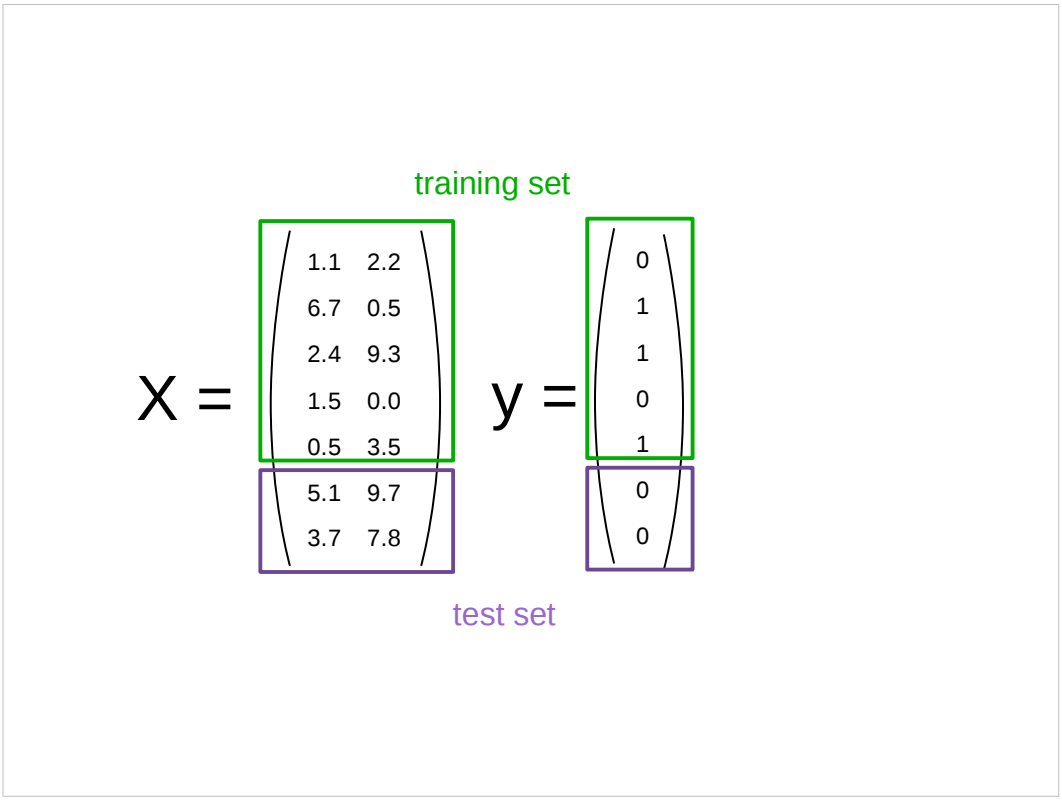
Again, neural networks are somewhat their own thing, as usually people choose networks that could learn any possible data.



Here is a very simple example where a nearest neighbor algorithm might work, while nearest centroid fails because it is limited to linear decision boundaries.

We'll talk more about linear decision boundaries and their strength and weaknesses on Wednesday.

Now, I want to talk more about how to adjust the parameters in these models - or really, in any model.



So far we've work with a split of the data into a training and a test set, build the model on the training set, and evaluated on the test set.

So now, lets say we want to adjust the parameter n_neigbhbors in k neighbors algorithm, how could we do this?

[split, try out different values of k, choose the best on test set]

What's the problem with that?

- good for choosing k, overly optimistic for getting accuracy!

Overfitting the test set

```
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import scale

data = load_breast_cancer()
X, y = data.data, data.target
X = scale(X)

X_trainval, X_test, y_trainval, y_test = train_test_split(X, y)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval)
```

```
knn = KNeighborsClassifier(n_neighbors=5).fit(X_train, y_train)

print("Validation: {:.3f}".format(knn.score(X_val, y_val)))
print("Test: {:.3f}".format(knn.score(X_test, y_test)))
```

```
Validation: 0.963
Test: 0.972
```

So I'll try to illustrate this concept of overfitting the test set. Basically, the idea is that if you try out too many things on the test set, you will learn about noise in the test set, and this knowledge will not generalize to new data.

So here I give you an example with the breast cancer dataset that's build into scikit-learn.

I split the data twice, now I have a training, a validation and a test set.

I build a nearest neighbors model on the training set, and apply it to the test set and the validation set.

The results are not the same, but they are pretty close. That they are different is a consequence of the small dataset and the noisy data.

Overfitting the test set

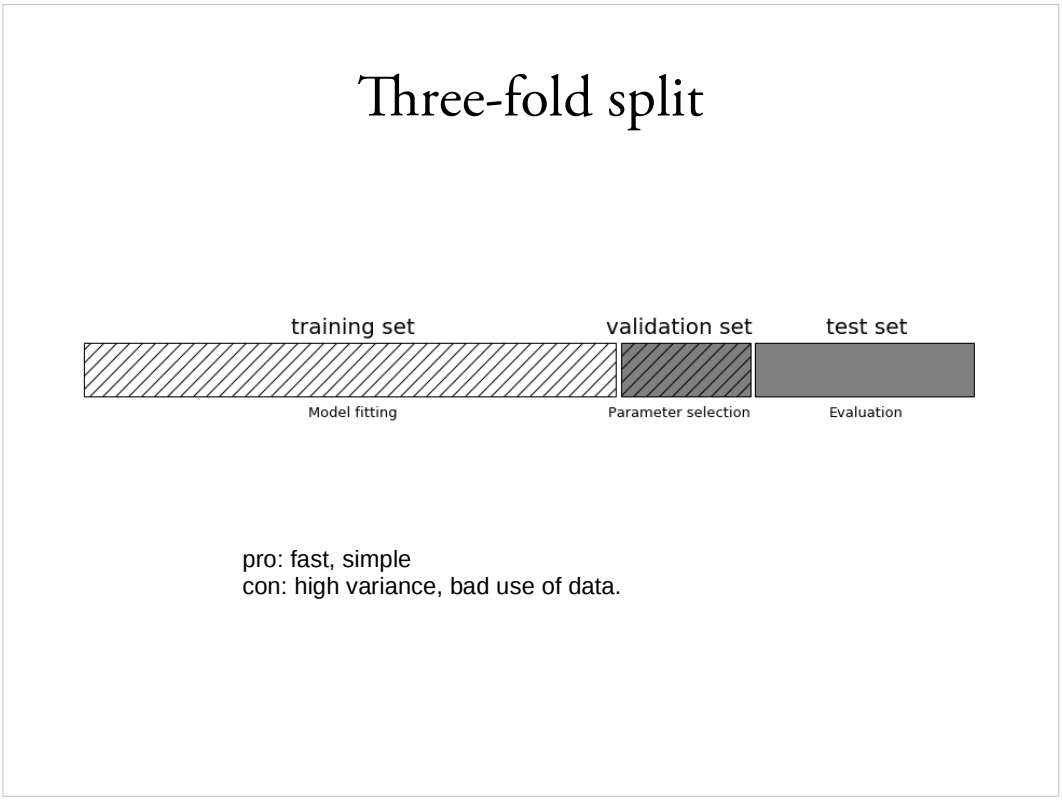
```
val = []
test = []

for i in range(1000):
    rng = np.random.RandomState(i)
    noise = rng.normal(scale=.1, size=X_train.shape)
    knn = KNeighborsClassifier(n_neighbors=5).fit(X_train + noise, y_train)
    val.append(knn.score(X_val, y_val))
    test.append(knn.score(X_test, y_test))

print("Validation: {:.3f}".format(np.max(val)))
print("Test: {:.3f}".format(test[np.argmax(val)]))
```

```
Validation: 0.991
Test: 0.958
```

So now let me propose a silly way to tune my classifier.
I add random noise to my training set for fitting.
And I repeat that 1000 times, and I check which of these thousand runs has the best performance on the validation set. That particular run has 99.1% percent accuracy, much better than before.
If I test the same set on the test set, I get a much lower accuracy then before: the noise I added was good for the validation set, but not the test set.
If I try this often enough, I can find noise that will give me 100% on the validation set, but that doesn't mean it will be good for any new data.
By selecting the best among many you introduce a bias, and the validation accuracy is not a good measure of generalization performance any more.



The simplest way to combat this overfitting to the test set is by using a three-fold split of the data, into a training, a validation and a test set.

We use the training set for model building, the validation set for parameter selection and the test set for a final evaluation of the model.

So how many models should you try out on the test set? Only one! Ideally use use the test-set exactly once, otherwise you make a multiple hypothesis testing error!

What are downsides of this? We lose a lot of data for evaluation, and the results depend on the particular sampling.

```
val_scores = []
neighbors = np.arange(1, 15, 2)
for i in neighbors:
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train, y_train)
    val_scores.append(knn.score(X_val, y_val))
print("best validation score: {:.3f}".format(np.max(val_scores)))
best_n_neighbors = neighbors[np.argmax(val_scores)]
print("best n_neighbors: {}".format(best_n_neighbors))

knn = KNeighborsClassifier(n_neighbors=best_n_neighbors)
knn.fit(X_trainval, y_trainval)
print("test-set score: {:.3f}".format(knn.score(X_test, y_test)))
```

```
best validation score: 0.972
best n_neighbors: 3
test-set score: 0.965
```

Here is an implementation of the three-fold split for selecting the number of neighbors.

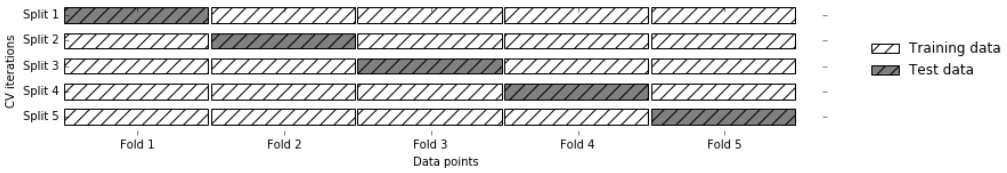
For each number of neighbors that we want to try, we build a model on the training set, and evaluate it on the validation set.

We then pick the best validation set score, here that's 97.2%, achieved when using three neighbors.

We then retrain the model with this parameter, and evaluate on the test set.

Do you have any idea how to make this more robust?

Cross-validation



Pro: more stable, more data
con: slower

The answer is of course cross-validation. In cross-validation, you split your data into multiple folds, usually 5 or 10, and built multiple models.

You start by using fold1 as the test data, and the remaining ones as the training data. You build your model on the training data, and evaluate it on the test fold.

For each of the splits of the data, you get a model evaluation and a score. In the end, you can aggregate the scores, for example by taking the mean.

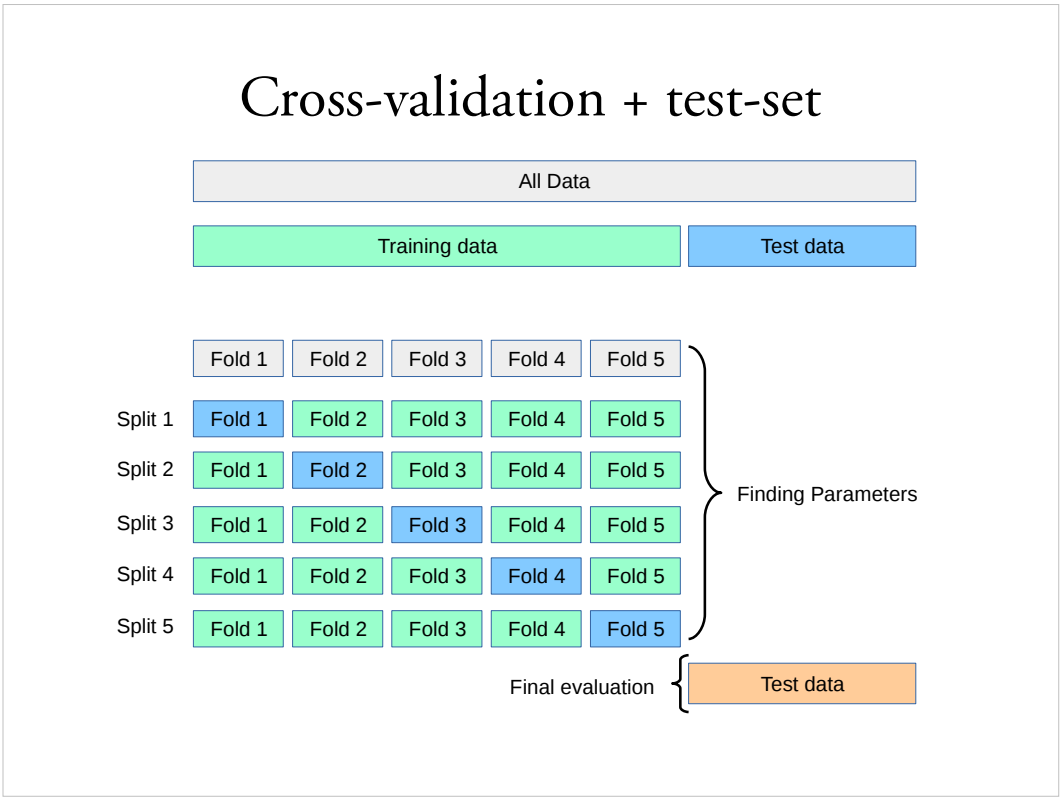
What are the pros and cons of this?

Each data point is in the test-set exactly once!

Takes 5 or 10 times longer!

Better data use (larger training sets).

Does that solve all problems? No, it replaces only one of the splits, usually the inner one!



Here is how the workflow looks like when we are using five-fold cross-validation together with a test-set split for adjusting parameters.

We start out by splitting of the test data, and then we perform cross-validation on the training set.

Once we found the right setting of the parameters, we retrain on the whole training set and evaluate on the test set.

```
from sklearn.model_selection import cross_val_score

X_train, X_test, y_train, y_test = train_test_split(X, y)

cross_val_scores = []

for i in neighbors:
    knn = KNeighborsClassifier(n_neighbors=i)
    scores = cross_val_score(knn, X_trainval, y_trainval, cv=10)
    cross_val_scores.append(np.mean(scores))

print("best cross-validation score: {:.3f}".format(np.max(cross_val_scores)))
best_n_neighbors = neighbors[np.argmax(cross_val_scores)]
print("best n_neighbors: {}".format(best_n_neighbors))

knn = KNeighborsClassifier(n_neighbors=best_n_neighbors)
knn.fit(X_train, y_train)
print("test-set score: {:.3f}".format(knn.score(X_test, y_test)))
```

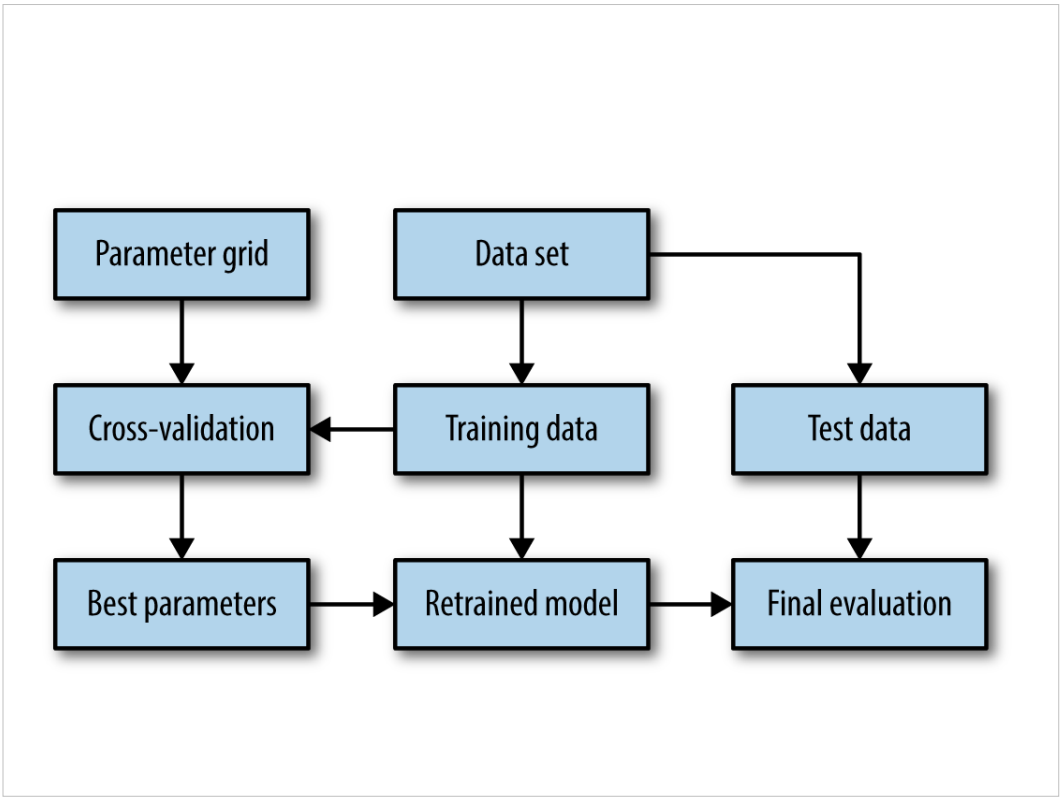
```
best cross-validation score: 0.972
best n_neighbors: 3
test-set score: 0.972
```

Here is an implementation of this for k nearest neighbors.

We split the data, then we iterate over all parameters and for each of them we do cross-validation.

We had seven different values of n_neighbors, and we are running 10 fold cross-validation. How many models to we train in total?

10 * 7 + 1 = 71 (the one is the final model)



Here is a conceptual overview of this way of tuning parameters, we start of with the dataset and a candidate set of parameters we want to try, labeled parameter grid, for example the number of neighbors.

We split the dataset in to training and test set. We use cross-validation and the parameter grid to find the best parameters.

We use the best parameters and the training set to build a model with the best parameters,

and finally evaluate it on the test set.

Nested Cross-validation

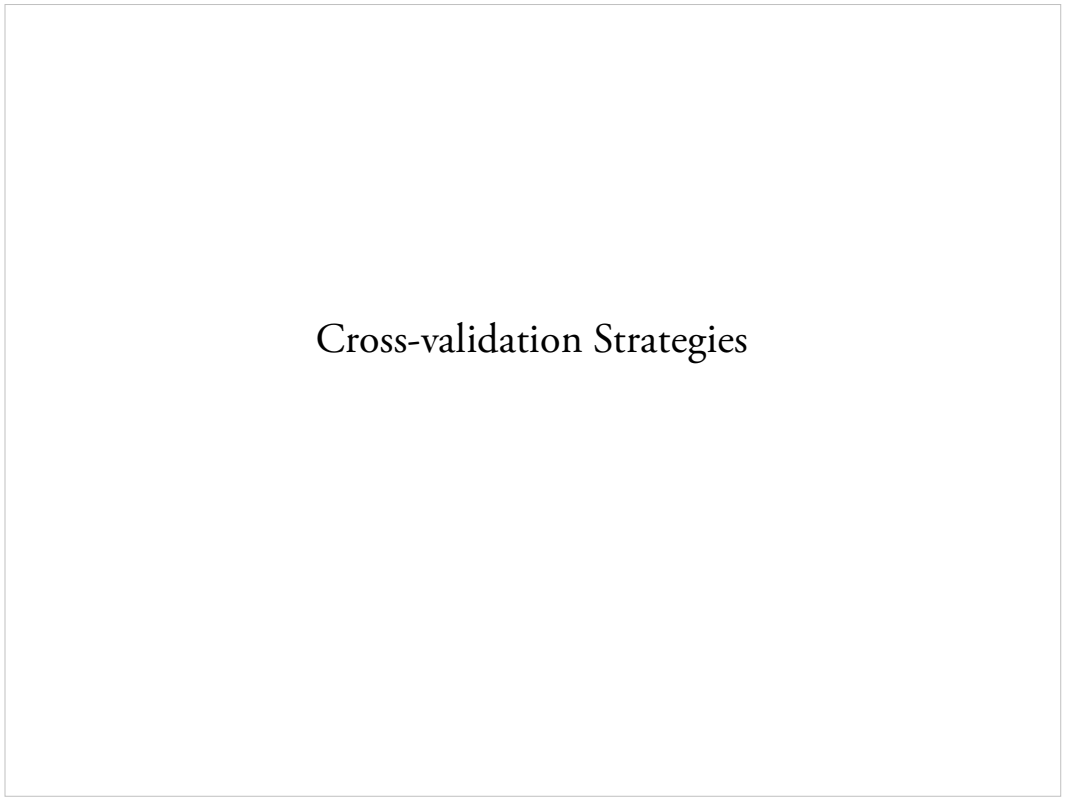
- Replace outer split by CV loop
- Doesn't yield single model
  (inner loop might have different best parameter settings)
- Takes a long time, not that useful in practice

We could additionally replace the outer split of the data by cross-validation. That would yield what's known as nested cross-validation.

This is sometimes interesting when comparing different models, but it will not actually yield one final model. It will yield one model for each loop of the outer fold, which might have different settings of the parameters.

Also, this takes a really long time to train, by an additional factor of 5 or 10, so this is not used very commonly in practice.

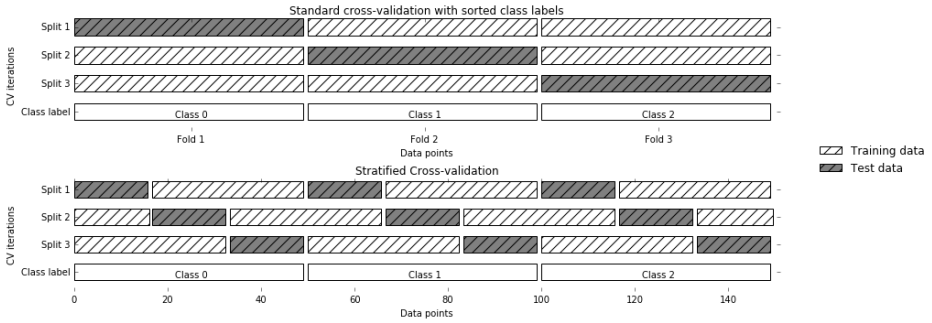
But let's dive into the cross-validation a bit more.



So I mentioned k-fold cross validation, where k is usually 5 or ten, but there are many other strategies.

One of the most commonly ones is stratified k-fold cross-validation.

StratifiedKFold



Stratified:
Ensure relative class frequencies in each fold reflect relative class frequencies on the whole dataset.

The idea behind stratified k-fold cross-validation is that you want the test set to be as representative of the dataset as possible.

StratifiedKFold preserves the class frequencies in each fold to be the same as of the overall dataset.

Here is and example of a dataset with three classes that are ordered. If you apply standard three-fold to this, the first third of the data would be in the first fold, the second in the second fold and the third in the third fold. Because this data is sorted, that would be particularly bad. If you use stratified cross-validation it would make sure that each fold has exactly 1/3 of the data from each class.

This is also helpful if your data is very imbalanced. If some of the classes are very rare, it could otherwise happen that a class is not present at all in a particular fold.

Defaults in scikit-learn

- Three-fold is default number of folds
- For classification cross-validation is stratified
- train_test_split has stratify option:
  `train_test_split(X, y, stratify=y)`
- No shuffle by default!

Before we go to the other strategies, I wanted to point out the default behavior in scikit-learn.
By default, all cross-validation strategies are three-fold.
If you do cross-validation for classification, it will be stratified by default.
Because of how the interface is done, that's not true for train_test_split and if you want a stratified train_test_split, which is always a good idea, you should use stratify=y
Another thing that's important to keep in mind is that by default scikit-learn doesn't shuffle! So if you run cross-validation twice with the default parameters, it will yield exactly the same results.

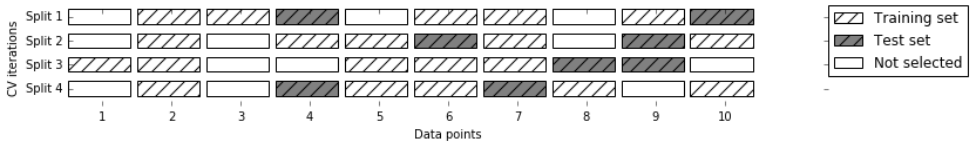
Repeated KFold and LeaveOneOut

- LeaveOneOut : KFold(n_folds=n_samples)
  High variance, takes a long time
- Better: repeated KFold.
  Apply KFold or StratifiedKFold multiple times with shuffled data. Reduces variance!

If you want even better estimates of the generalization performance, you could try to increase the number of folds, with the extreme of creating one fold per sample. That's called "LeaveOneOut cross-validation". However, because the test-set is so small every time, and the training sets all have very large overlap, this method has very high variance.

A better way to get a robust estimate is to run 5-fold or 10-fold cross-validation multiple times, while shuffling the dataset.

ShuffleSplit / StratifiedShuffleSplit

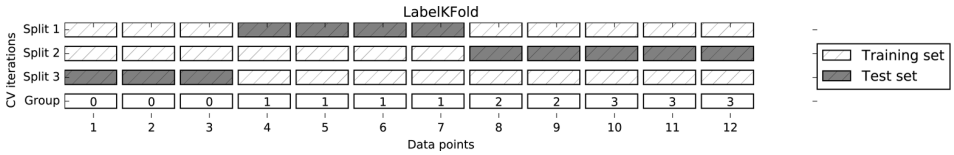


Another interesting variant is shuffle split and stratified shuffle split. In shuffle split, we repeatedly sample disjoint training and test sets randomly.

You only have to specify the number of iterations, the training set size and the test set size. This also allows you to run many iterations with reasonably large test-sets.

It’s also great if you have a very large training set and you want to subsample it to get quicker results.

GroupKFold



A somewhat more complicated approach is group k-fold.

This is actually for data that doesn't fulfill our IID assumption and has correlations between samples.

The idea is that there are several groups in the data that each contain highly correlated samples.

You could think about patient data where you have multiple samples for each patient, then the groups would be which patient a measurement was taken from.

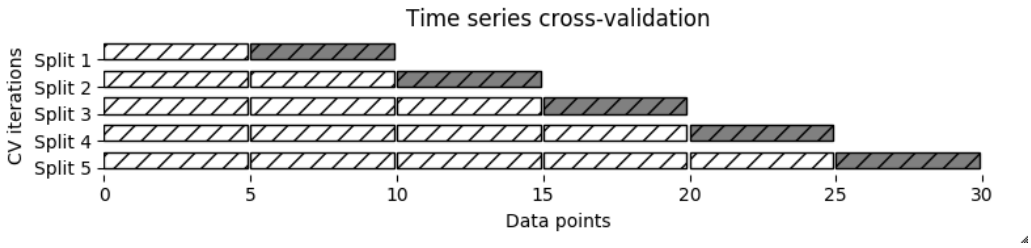
If you want to know how well your model generalizes to new patients, you need to ensure that the measurements from each patient are either all in the training set, or all in the test set.

And that's what GroupKFold does.

In this example, there are four groups, and we want three folds. The data is divided such that each group is contained in exactly one fold.

There are several other cross-validation methods in scikit-learn that use these groups.

TimeSeriesSplit



Using Cross-Validation Generators

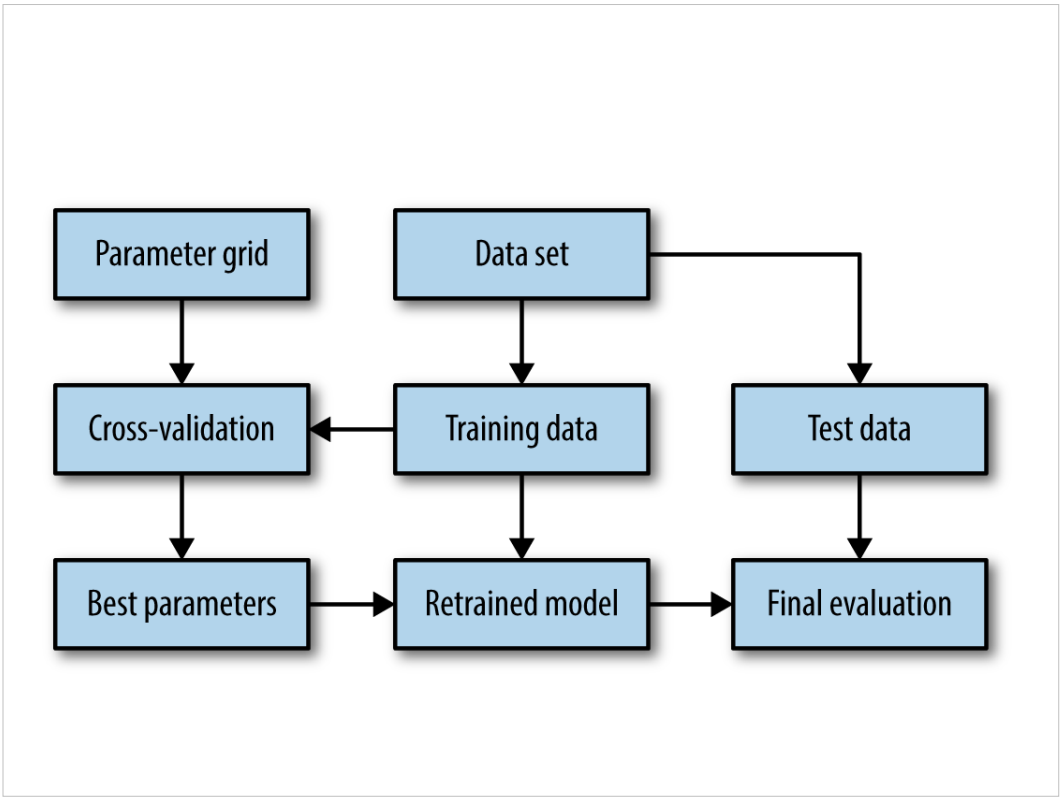
```
from sklearn.model_selection import KFold, StratifiedKFold, ShuffleSplit
kfold = KFold(n_splits=5)
skfold = StratifiedKFold(n_splits=5, shuffle=True)
ss = ShuffleSplit(n_splits=20, train_size=.4, test_size=.3)

print("KFold:\n{}".format(
      cross_val_score(KNeighborsClassifier(), X, y, cv=kfold)))

print("StratifiedKFold:\n{}".format(
      cross_val_score(KNeighborsClassifier(), X, y, cv=skfold)))

print("ShuffleSplit:\n{}".format(
      cross_val_score(KNeighborsClassifier(), X, y, cv=ss)))
```

```
KFold:
[ 0.92982456  0.95614035  0.96491228  0.98245614  0.96460177]
StratifiedKFold:
[ 0.95652174  0.9826087   0.95575221  0.99115044  0.95575221]
ShuffleSplit:
[ 0.98245614  0.95321637  0.97076023  0.95321637  0.97660819  0.96491228
  0.97076023  0.92982456  0.94736842  0.97660819  0.97660819  0.95321637
  0.95906433  0.97076023  0.94736842  0.9122807   0.95906433  0.93567251
  0.95906433  0.94152047]
```


Parameter grid
Data set
Cross-validation
Training data
Test data
Best parameters
Retrained model
Final evaluation

Grid-Search

```
from sklearn.model_selection import GridSearchCV

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y)


param_grid = {'n_neighbors':  np.arange(1, 15, 2)}
grid = GridSearchCV(KNeighborsClassifier(), param_grid=param_grid, cv=10)
grid.fit(X_train, y_train)
print("best mean cross-validation score: {:.3f}".format(grid.best_score_))
print("best parameters: {}".format(grid.best_params_))

print("test-set score: {:.3f}".format(grid.score(X_test, y_test)))
```

```
best mean cross-validation score: 0.967
best parameters: {'n_neighbors': 7}
test-set score: 0.951
```

GridSearchCV results

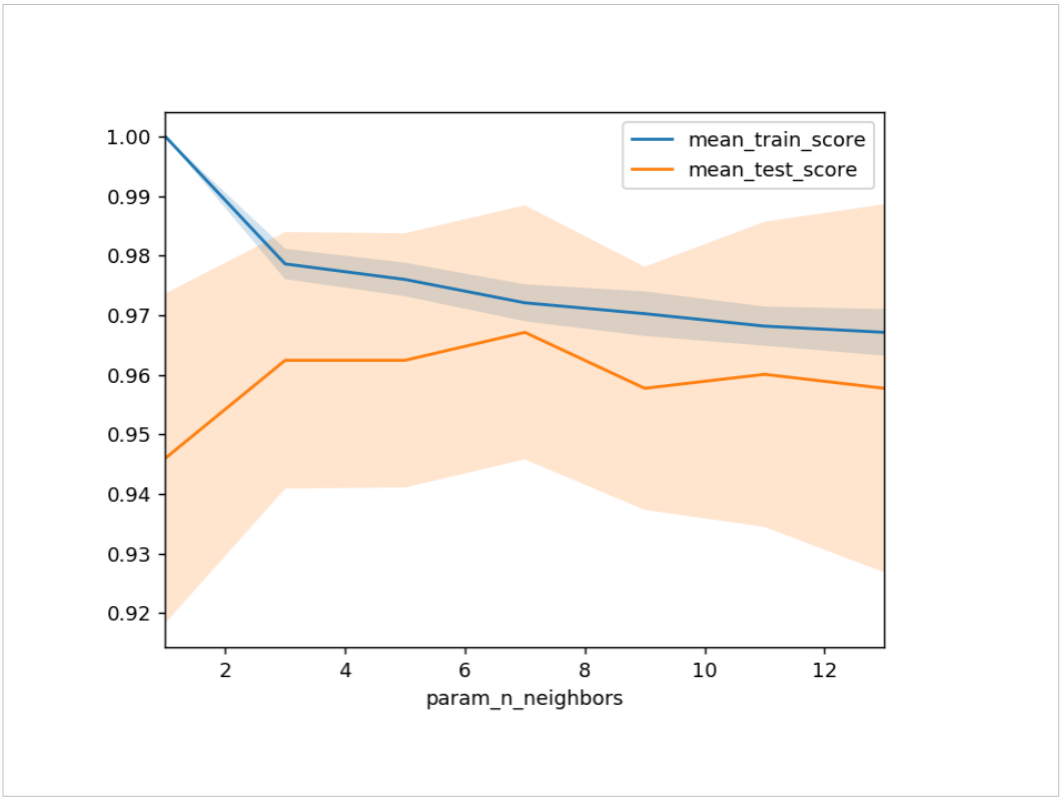
```
import pandas as pd
results = pd.DataFrame(grid.cv_results_)
```

```
results.columns
```

```
Index(['mean_fit_time', 'mean_score_time', 'mean_test_score',
       'mean_train_score', 'param_n_neighbors', 'params', 'rank_test_score',
       'split0_test_score', 'split0_train_score', 'split1_test_score',
       'split1_train_score', 'split2_test_score', 'split2_train_score',
       'split3_test_score', 'split3_train_score', 'split4_test_score',
       'split4_train_score', 'split5_test_score', 'split5_train_score',
       'split6_test_score', 'split6_train_score', 'split7_test_score',
       'split7_train_score', 'split8_test_score', 'split8_train_score',
       'split9_test_score', 'split9_train_score', 'std_fit_time',
       'std_score_time', 'std_test_score', 'std_train_score'],
      dtype='object')
```

```
results.params
```

```
0     {'n_neighbors': 1}
1     {'n_neighbors': 3}
2     {'n_neighbors': 5}
3     {'n_neighbors': 7}
4     {'n_neighbors': 9}
5    {'n_neighbors': 11}
6    {'n_neighbors': 13}
Name: params, dtype: object
```


mean_train_score
mean_test_score
1.00
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
2
4
6
8
10
12
param_n_neighbors