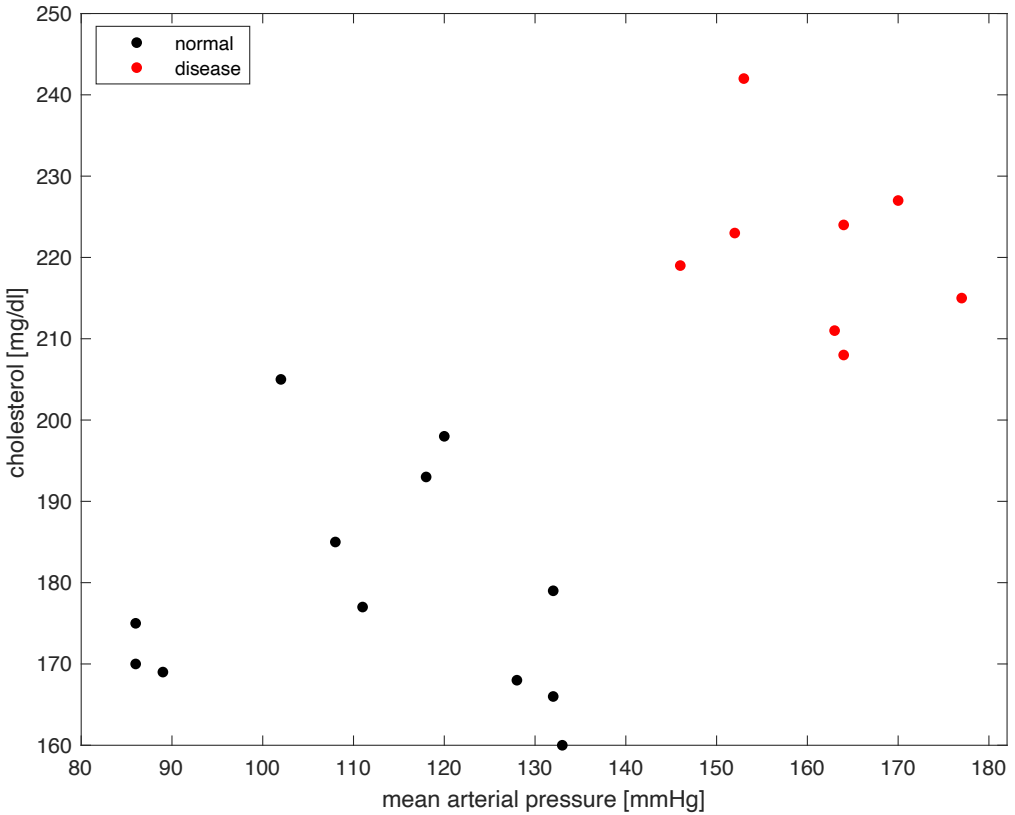
Fitting SVM models in Matlab

- `mdl = fitcsvm(X,y)`
  - fit a classifier using SVM
  - **X** is a matrix
    - columns are predictor variables
    - rows are observations
  - **y** is a response vector
    - +1/-1 for each row in X
    - can be any set of integers or strings
  - returns a *ClassifierSVM* object, which we stored in variable **mdl**
- `predict(mdl,newX)`
  - returns responses for matrix **newX** using the classifier **mdl**

Example: Heart Attack prediction from Blood Pressure and Cholesterol

ha_data = *20×3 table*

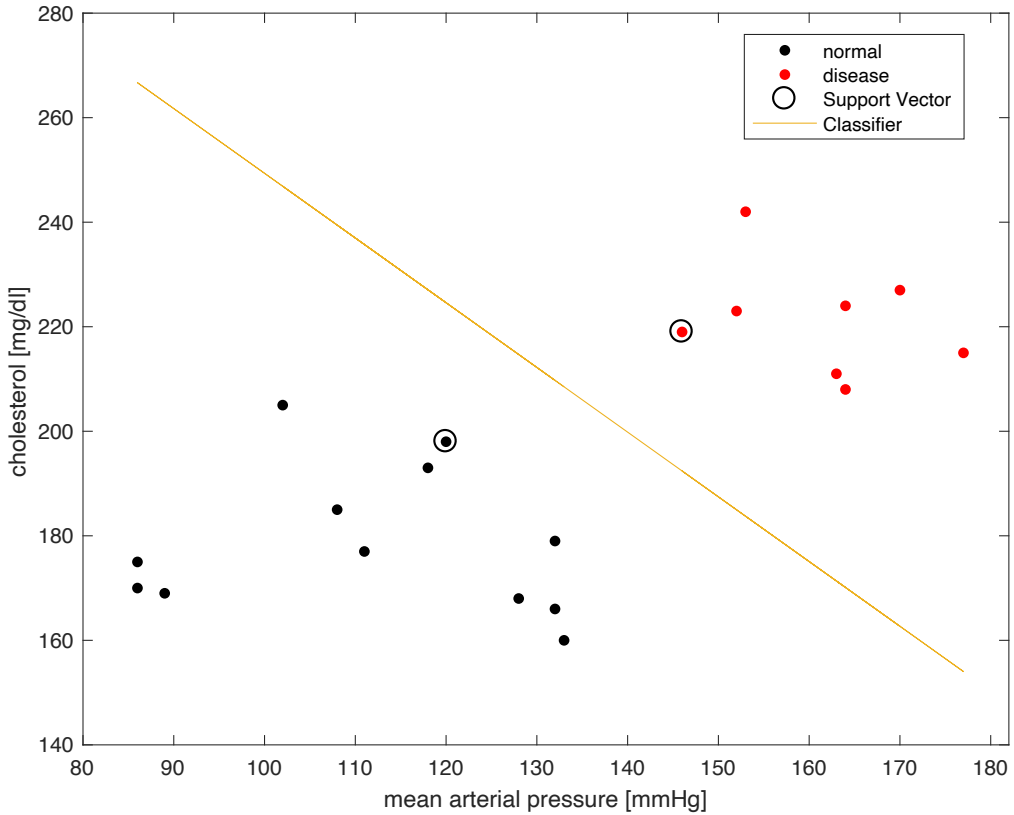
BloodPressure	Cholesterol	HeartAttack
133	160	-1
132	166	-1
128	168	-1
89	169	-1
86	170	-1
86	175	-1
111	177	-1
132	179	-1
108	185	-1
118	193	-1
120	198	-1
102	205	-1
164	208	1
163	211	1
177	215	1
146	219	1
152	223	1
164	224	1
170	227	1
153	242	1



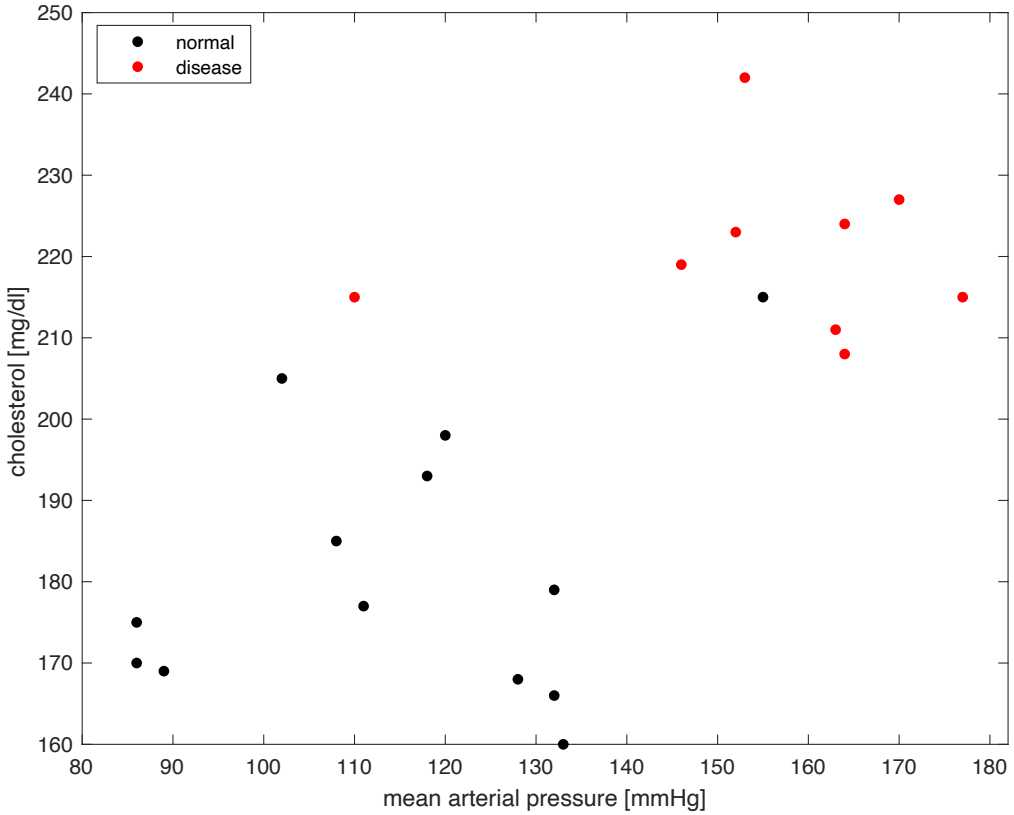
Example: Heart Attack prediction from Blood Pressure and Cholesterol

```
mdl = fitcsvm([ha_data.BloodPressure ha_data.Cholesterol], ha_data.HeartAttack)
ha_data.predicted = predict(mdl, [ha_data.BloodPressure ha_data.Cholesterol])
```

BloodPressure	Cholesterol	HeartAttack	predicted
133	160	-1	-1
132	166	-1	-1
128	168	-1	-1
89	169	-1	-1
86	170	-1	-1
86	175	-1	-1
111	177	-1	-1
132	179	-1	-1
108	185	-1	-1
118	193	-1	-1
120	198	-1	-1
102	205	-1	-1
164	208	1	1
163	211	1	1
177	215	1	1
146	219	1	1
152	223	1	1
164	224	1	1
170	227	1	1
153	242	1	1



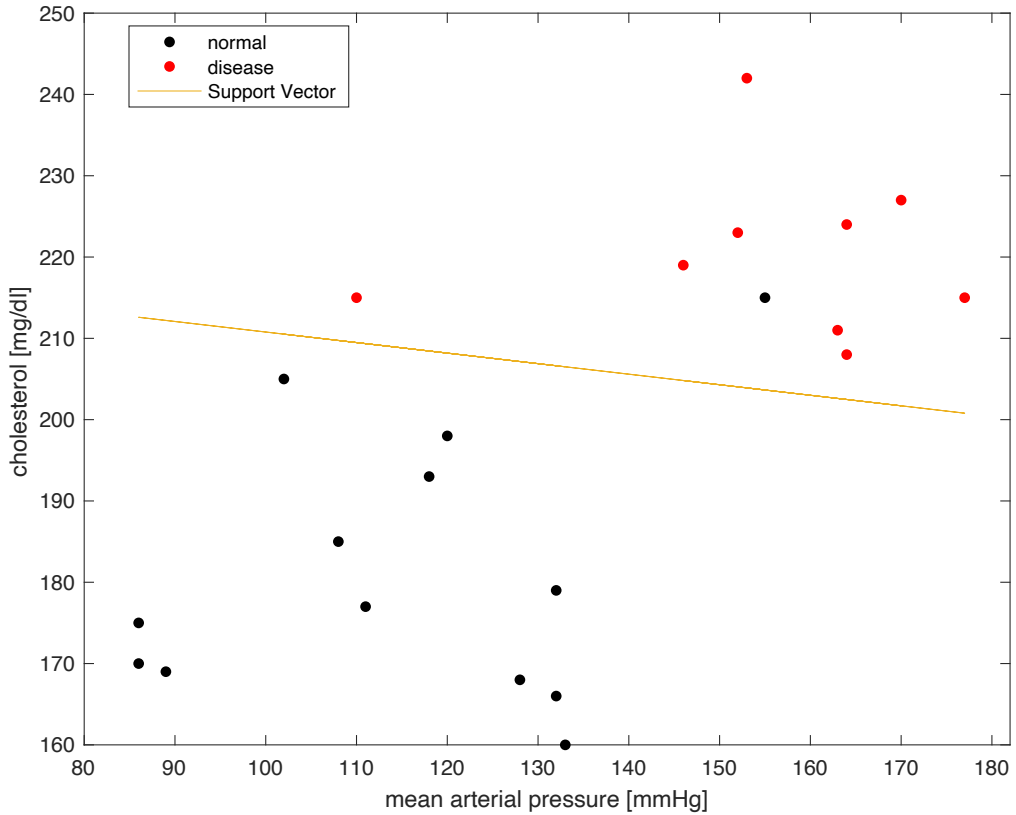
What if we cannot perfectly classify the data?



What if we cannot perfectly classify the data?

```
mdl = fitcsvm([ha_data.BloodPressure ha_data.Cholesterol], ha_data.HeartAttack)
ha_data.predicted = predict(mdl, [ha_data.BloodPressure ha_data.Cholesterol])
```

BloodPressure	Cholesterol	HeartAttack	predicted
133	160	-1	-1
132	166	-1	-1
128	168	-1	-1
89	169	-1	-1
86	170	-1	-1
86	175	-1	-1
111	177	-1	-1
132	179	-1	-1
108	185	-1	-1
118	193	-1	-1
120	198	-1	-1
102	205	-1	-1
164	208	1	1
163	211	1	1
177	215	1	1
146	219	1	1
152	223	1	1
164	224	1	1
170	227	1	1
153	242	1	1
155	215	-1	1
110	215	1	1



Fundamental Theorem of Modeling*

- **Data used for training cannot be used for validation.**

- Why not? To avoid overfitting.
- Imagine we create a model that predicts a person's ***height*** from their ***name***.
- We train our model using the names and heights of people in our class.
- Everyone in our class has a different name, so the mapping is 1-to-1. If we tested our model with anyone in our class, it would predict their height perfectly!
- But clearly this is a horrible model; there are many other people with our same name but different height. We only think our model is perfect because we tested on data we trained with.

*this is not actually a theorem.

What are our options?

1. Don’t validate your model.
   - Not a scientifically valid approach.

2. Train with only a subset of your data; leave the rest for validation.
   - Your model would be underpowered.
   - Fit is sensitive to which points you left out.

3. Collect new data to validate the trained model.
   - Can be expensive and/or infeasible.
   - Also, wouldn’t you want to train with these data as well?

Best solution: ***Cross Validation***

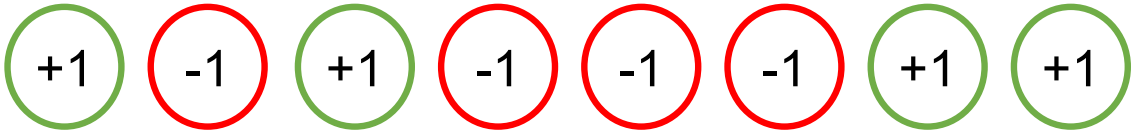
- We split our data into two groups: **training** and **testing**
- Train and test the model using the respective sets.
- Repeat this process several times.

- Advantages of Cross Validation
  - All points are used for both training and testing (at separate times).
  - Overfit models will perform poorly, making them easy to identify.
  - Good models will perform consistently across all testing sets.

- The “final” model is training using the entire dataset.

Example: training an SVM Classifier

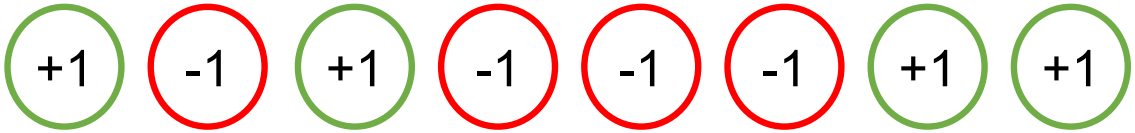
- *n* data points



- Method 1: Leave-One-Out (L1O) Cross Validation
  1. Remove the first data point.
  2. Train on the remaining *n*-1 points.
  3. Test the removed point.
  4. Repeat using point 2 – *n*.
  5. Final accuracy: (# correct) / *n*

Method 2: *k*-fold Cross Validation

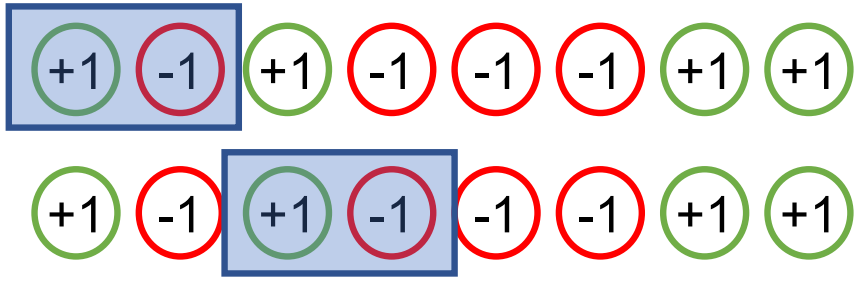
- *n* data points

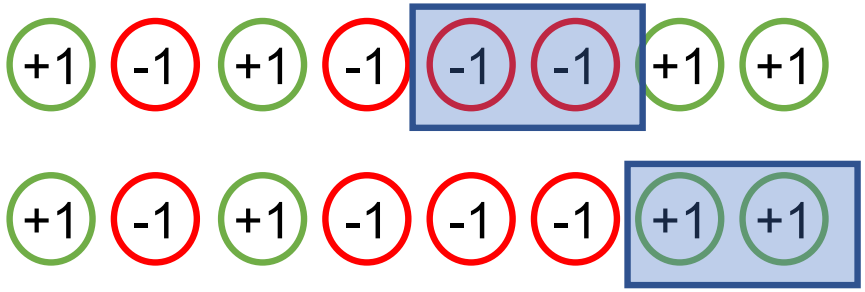


- Split the points into *k* evenly sized groups.
- For each group:
  - Remove the group from the data.
  - Training on the remaining points.
  - Validate using the removed points.

= testing set

- Example: *k* = 4





Comparing L1O to *k*-fold Cross Validation

- L1O Advantages
  - Trained models are closest to the final model, since only one point is removed.
- L1O Disadvantages
  - If models take a long time to train, L1O can be infeasible.
- *k*-fold Advantages
  - Faster to train
  - More stringent (works well with n/k points removed).
  - Statistical power for each sub-model, since multiple points tested.
- *k*-fold Disadvantages
  - What value of *k* should we use?

Note that when $k=n$, the methods are identical!

Picking *k* for Cross Validation (XV)

- For large datasets, k=10 is commonly used.

- For biomedical applications, samples can be noisy.
- Each cycle uses n/k points for testing and $n(1\text{-}1/k)$ points for training. Thus, a k-fold XV has k-1 times more points used for training than testing. Try to keep $k > 3\text{-}4$.

k-fold Cross Validation in Matlab

- `mdl = fitcsvm(...)`
- `xval = crossval(mdl,'Kfold',5)`
  - default for Kfold is 10
- `kfoldLoss(xval)`
  - Gives the average misclassification rate ("loss") across all folds

```
mdl = fitcsvm([ha_data.BloodPressure ha_data.Cholesterol], ha_data.HeartAttack)
xval = crossval(mdl,'KFold',10);
kfoldLoss(xval)
        ans = 0.0909
```