# Generalization in Machine Learning

July 21, 2017

Elise Jennings (ALCF)
ejennings@anl.gov

---

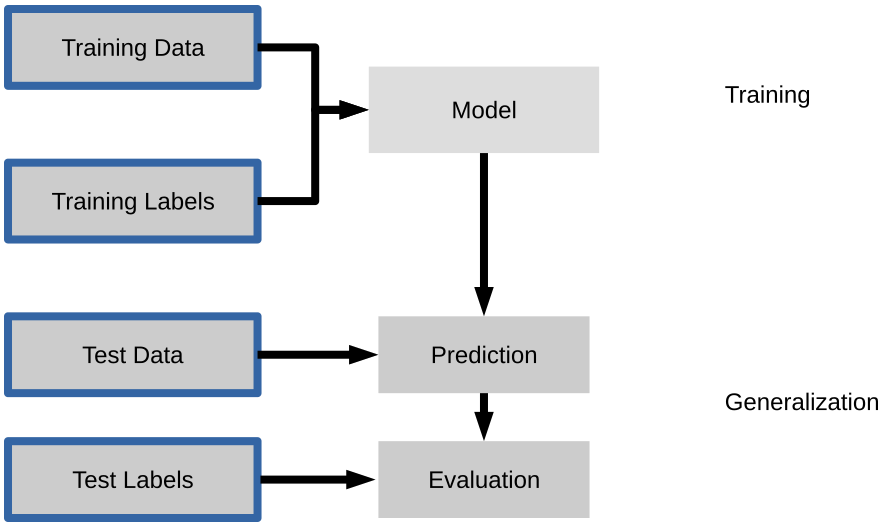
The data we collect is noisy and is only one realization from a probability distribution function (Bayesian view of the world). Ultimate test for trained ML model: After tuning the model using the training set, how well does the algorithm perform on unseen data? We shouldn't be too impressed if the ML model fits the training set to 99.9% accuracy!

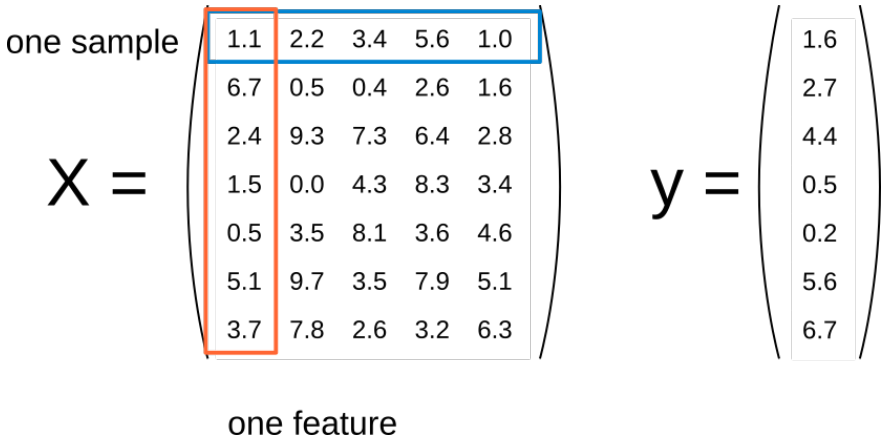
**Topics for this session**

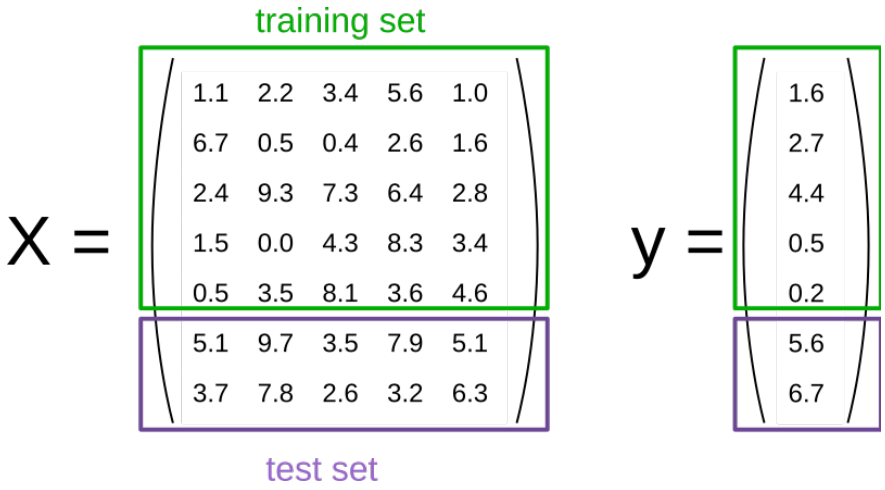
- Over-fitting
- Cross-validation
- Metrics

---

## 1 Recap: Supervised learning






## 2 Overfitting and Model Complexity

---

*Note: these notes were generated from a jupyter notebook and some code cells did not wrap around in the LaTeX conversion*

---

**Overfitting**

- Model fits the training data too well: could be fitting to noise.
- Result is poor performance of model

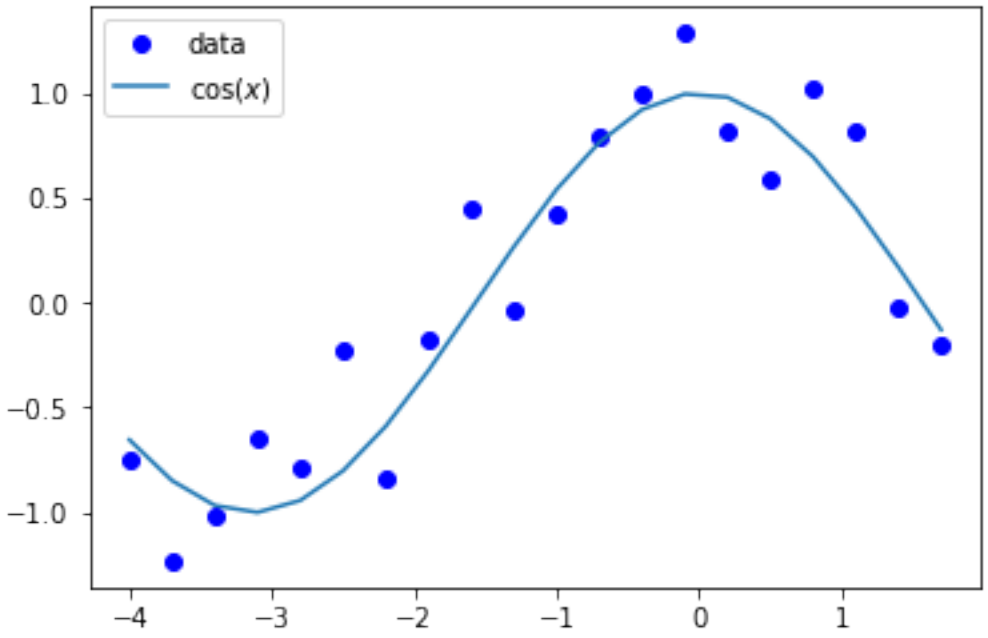
An example of overfitting:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        import math
```

```
In [124]: #generate some noisy data from a cosine function
          x=np.arange(-4,2,0.3)
          y_true = y = [np.cos(i) for i in x]
          y = np.array([np.cos(i)+np.random.normal(0,0.3) for i in x])
          x = x.reshape(len(x),1)
```

```
In [125]: plt.plot(x,y, marker="o",linestyle="None", color="blue",label="data")
          plt.plot(x,y_true,linestyle="-",label="$\cos(x)$")
          plt.legend(loc=2)
```

```
Out[125]: <matplotlib.legend.Legend at 0x11cb9c6d0>
```



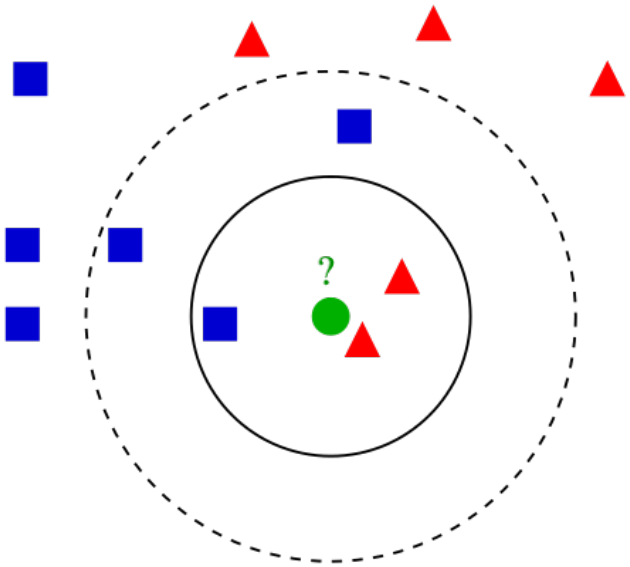

let's fit to the noisy data using k-nearest neighbor regression and Decision Trees

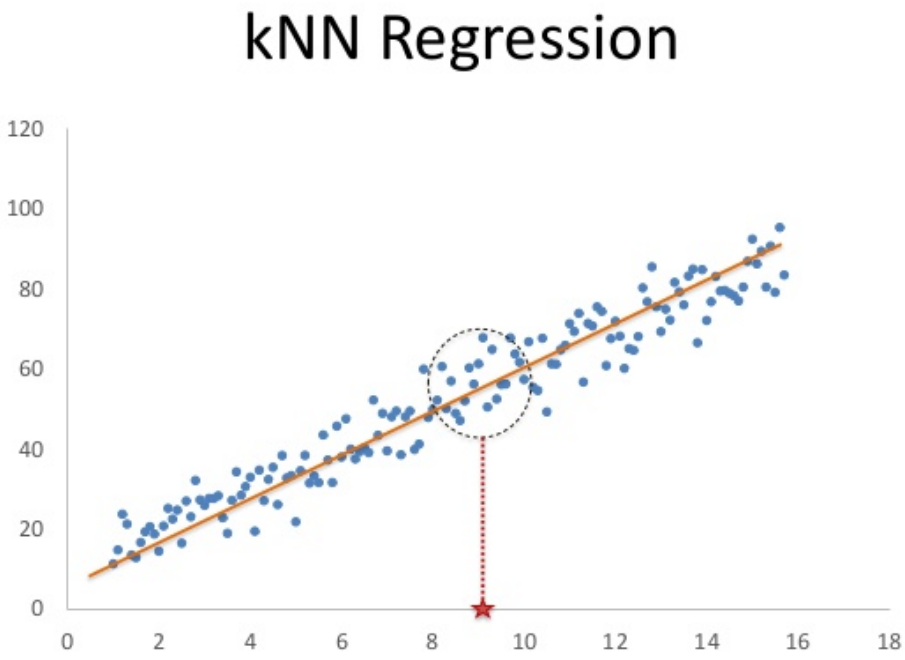
```
In [126]:
          from sklearn.svm import SVR
          from sklearn import tree
          from sklearn.neighbors import KNeighborsRegressor
```

**K-Nearest Neighbours algorithm**

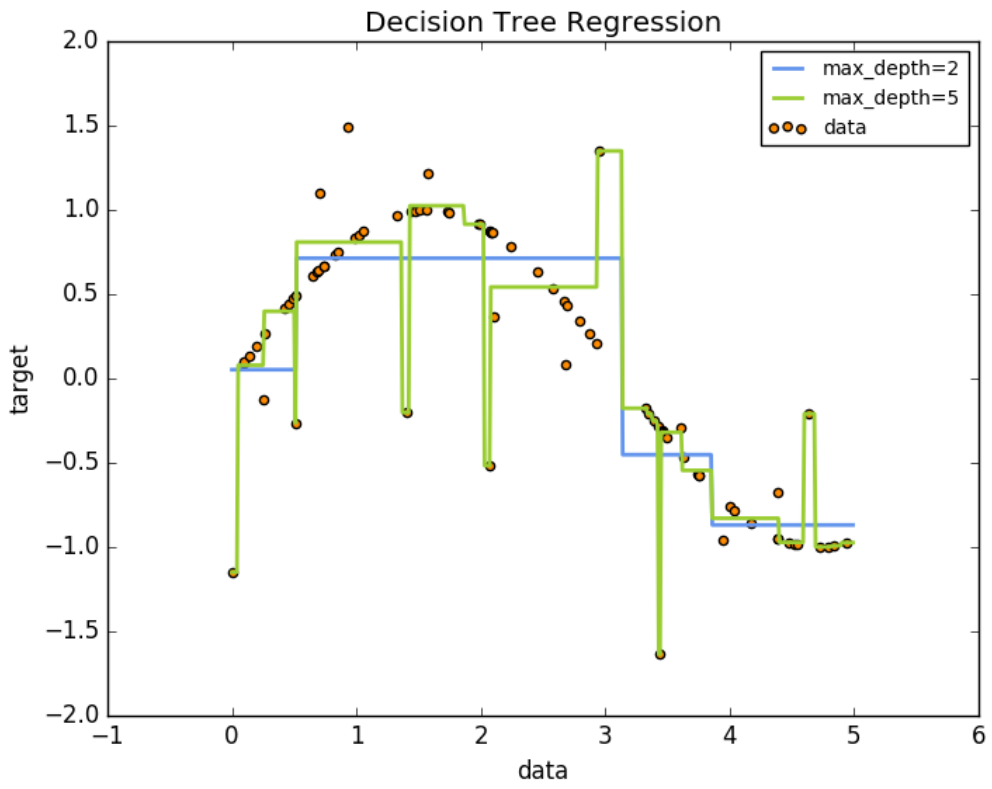
- information using K nearest neighbours
- average over neighbours
- too many neighbours -> oversmoothing
- too few neighbours -> fitting to individual points

**kNN Classification**



**Regression Trees**
Split on intervals of the data. Increasing depth of tree = decreasing intervals




In [127]:
```
#Parameters of the models

n_neighbors = 3 #KNN regressor
MaxDepth=3 #depth of the tree

tree_regression = tree.DecisionTreeRegressor(max_depth=MaxDepth)
kneighbor_regression = KNeighborsRegressor(n_neighbors=n_neighbors)
```

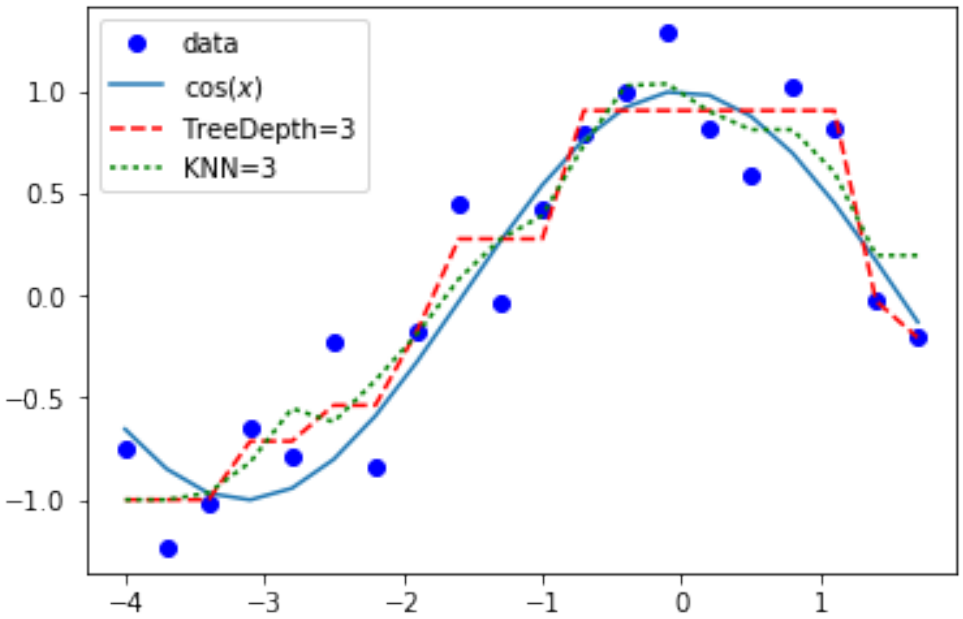
Fit to data and predict (Here we are just using the model to predict the training set again)

In [128]:
```
y_tree = tree_regression.fit(x,y).predict(x)
y_knn = kneighbor_regression.fit(x, y).predict(x)
```

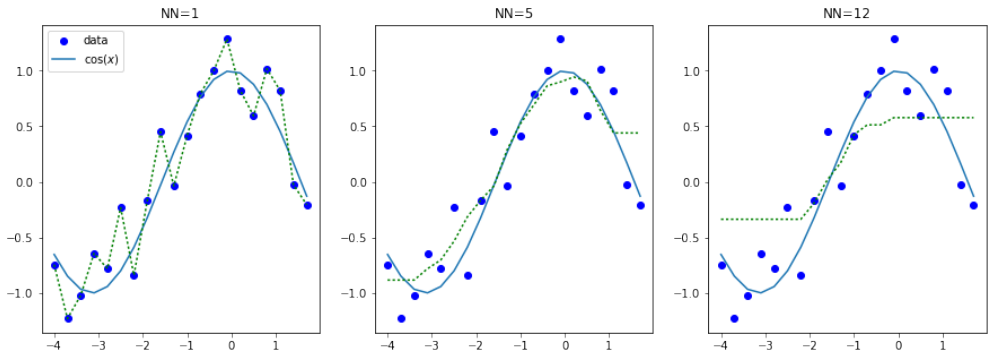
In [130]:
```
plt.plot(x,y, marker="o",linestyle="None", color="blue",label="data")
plt.plot(x,y_true,linestyle="-",label="$\cos(x)$")
```

```
plt.plot(x,y_tree,linestyle="--",color="red", label="TreeDepth="+str(MaxD
plt.plot(x,y_knn,linestyle=":",color="green", label="KNN="+str(n_neighbor
plt.legend()
```

```
Out[130]: <matplotlib.legend.Legend at 0x11c16a110>
```

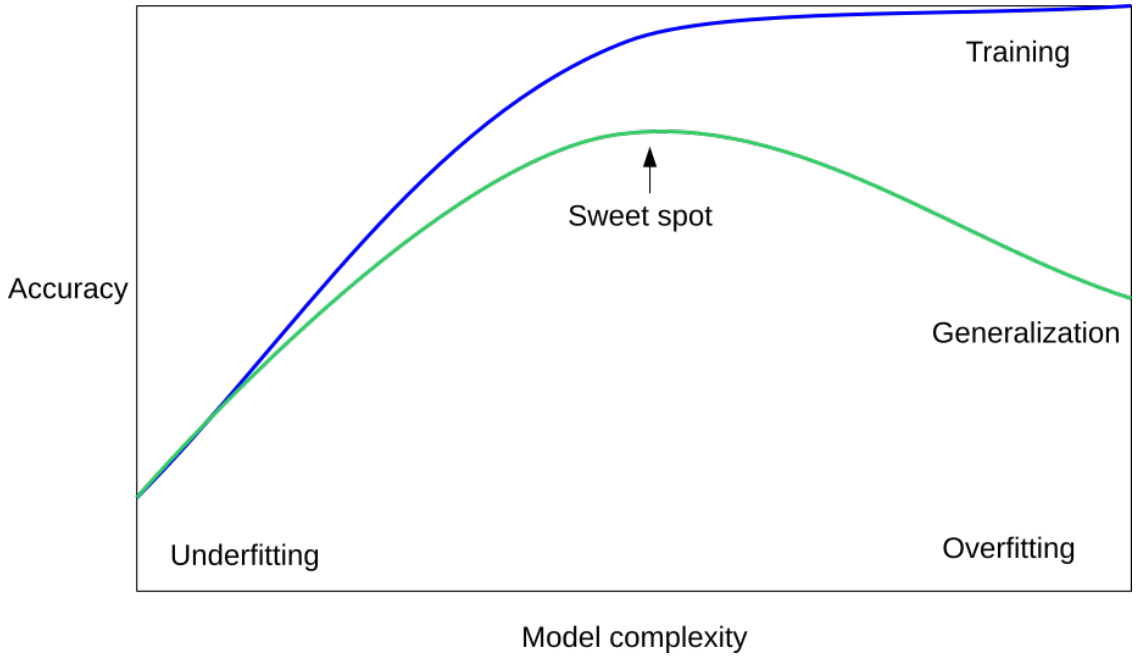



```
In [131]: plots=[131,132,133]
          nn=[1,5,12]
          fig, axes = plt.subplots(1, 3, figsize=(15, 5))
          for n, ax in zip(nn, axes):
              kneighbor_regression = KNeighborsRegressor(n_neighbors=n)
              y_knn = kneighbor_regression.fit(x, y).predict(x)

              ax.plot(x,y, marker="o",linestyle="None", color="blue",label="data")
              ax.plot(x,y_true,linestyle="-",label="$\cos(x)$")
              if n==1: ax.legend(loc=2)

              ax.plot(x,y_knn,linestyle=":",color="green", label="KNN="+str(n_neigh
              ax.set_title("NN="+str(n))
```




**Many ways we can accidentally overfit to training data**

- complicated model with simple data e.g. increasing degree of polynomial
- using training data over and over again
- fit model after model on same training data




**A very brief note on Regularization**
We can reduce overfitting by adding a penalty to the loss function
For example:
**L2 regularization**

$$\hat{w} = \arg\min_w (\mathbf{Y} - \mathbf{X}\mathbf{w})^{\mathbf{T}} (\mathbf{Y} - \mathbf{X}\mathbf{w}) + \lambda \mathbf{w}^{\mathbf{T}} \mathbf{w}$$

**L1 regularization**

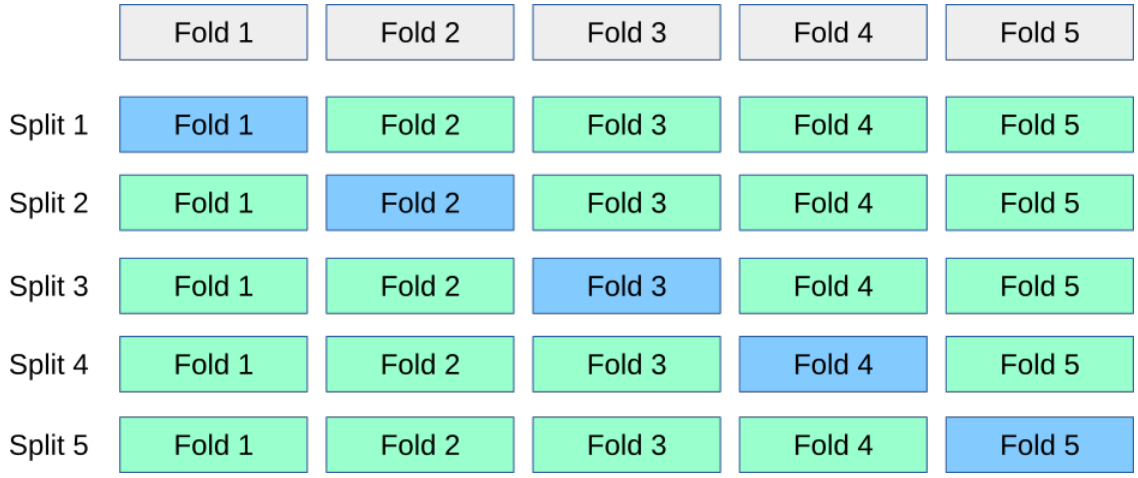
$$\hat{w} = \arg\min_w (\mathbf{Y} - \mathbf{X}\mathbf{w})^{\mathbf{T}} (\mathbf{Y} - \mathbf{X}\mathbf{w}) + \lambda \sum_{\mathbf{j}} |\mathbf{w_j}|$$

You can select different loss and penalty functions. E.g. in scikit learn: LinearSVC(penalty='l1', loss='squared_hinge')

---

# 3 Cross-Validation

Put aside part of the training set, called a validation set to evaluate the learning algorithm. Generate "folds" in the training data and a validation set in each fold. The variation in the accuracy of the model on different folds gives us an error on the trained model. Can vary - the number of folds - the metric (scoring function) - the methods of cross validation - the parameters of the model in each fold (hyperparameters) - feature selection (discard irrelevant features)



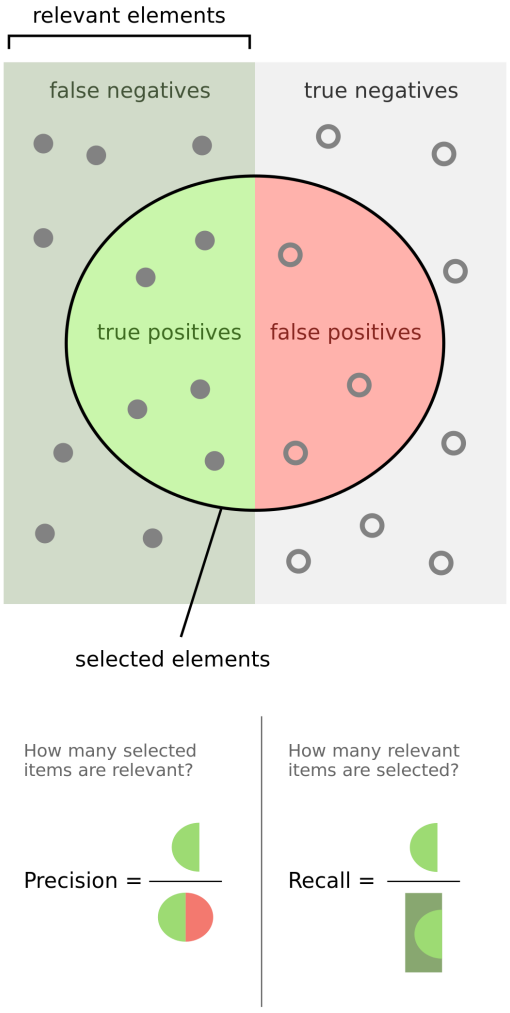

## 3.1 Example metrics

We need a way to score the accuracy of the trained model. How well did it predict the target function in the test set? Metrics can be different for regression and classification.

```
In [9]: from sklearn.metrics.scorer import SCORERS
        print(SCORERS.keys())
```

'f1', 'f1_weighted', 'f1_samples', 'neg_mean_squared_error', 'precision_weighted', 'recall_samples', 'recall_micro', 'adjusted_rand_score', 'recall_macro', 'mean_absolute_error', 'precision_macro', 'neg_log_loss', 'neg_mean_absolute_error', 'f1_macro', 'recall_weighted', 'accuracy', 'precision_samples', 'median_absolute_error', 'precision', 'log_loss', 'precision_micro', 'average_precision', 'roc_auc', 'r2', 'recall', 'mean_squared_error', 'f1_micro', 'neg_median_absolute_error'

A **confusion matrix** (contingency table) is a nice way to show the accuracy of a classifier.

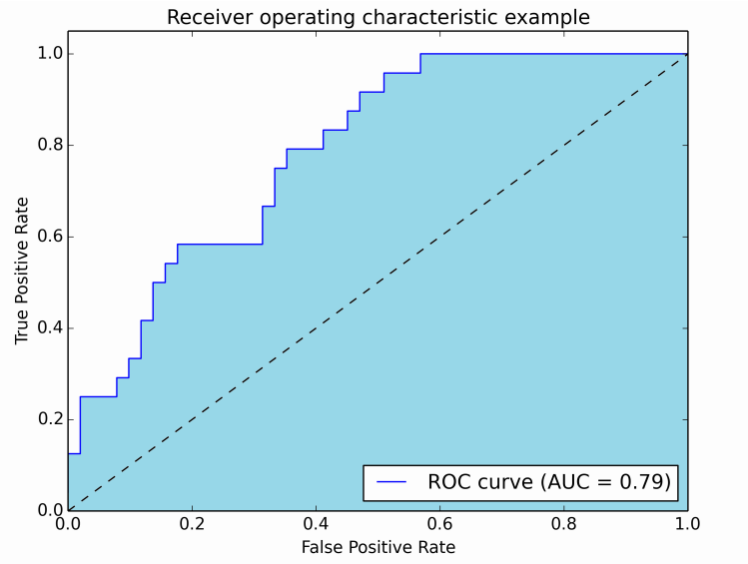


| | | Prediction | |
|---|---|---|---|
| | | Positive | Negative |
| Actual | Positive | TP | FN |
| | Negative | FP | TN |

### 3.2 AUROC = Area Under the Receiver Operating Characteristic curve

Plot of True positve rate $\frac{TP}{TP+FN}$ versus False positve rate $\frac{FP}{TP+TN}$. This illustrates the performance of a binary classifier system as its threshold is varied

ROC curve of a random predictor has an AUROC of 0.5.




### 3.3 The f1 metric (binary classification)

$f_1 = 2\frac{\text{precision*recall}}{\text{precision+recall}}$

### 3.4 The $R^2$ metric (regression)

$R^2 = 1 - \frac{u}{v}$
where
$u = \sum_i (y^i_{true} - y^i_{pred})^2$
and $v$ is the variance of the data
$v = \sum_i (y^i_{true} - \bar{y}_{true})^2$

```
In [132]: #Both KNN and decision trees above use R^2 metric
          print tree_regression.score(x,y)
          print kneighbor_regression.score(x,y)
```

```
0.933444136038
0.616395773724
```

### 3.5 Cross validation and scoring example using a scikit-learn dataset

```
In [133]: from sklearn.datasets import load_iris
          iris = load_iris()
          print iris.keys()
          #print iris['target_names'], iris['data'].shape, iris['target'], iris['fe
          #print iris['DESCR']
```

```
['target_names', 'data', 'target', 'DESCR', 'feature_names']
```

```
In [134]: X = iris.data
          y = iris.target
```

```
In [135]: #sklearn.model_selection.cross_val_score(estimator, X, y=None, groups=Nor
          #scoring=None, cv=None, n_jobs=1, verbose=0, fit_params=None, pre_dispato

          from sklearn.model_selection import cross_val_score #Evaluate a score by
          from sklearn.svm import LinearSVC
```

```
In [136]: cross_val_score(LinearSVC(), X, y, cv=5)

          #print LinearSVC.score.__doc__
```

```
Out[136]: array([ 1.        ,  1.        ,  0.93333333,  0.9       ,  1.        ])
```

```
In [137]: cross_val_score(LinearSVC(), X, y, cv=5, scoring="f1_weighted") # require
          #Calculate metrics
          #for each label, and find their unweighted mean. This does not take label
```

```
Out[137]: array([ 1.        ,  1.        ,  0.93333333,  0.89974937,  1.        ])
```

Let's go to a binary task for a moment

```
In [138]: y % 2
```

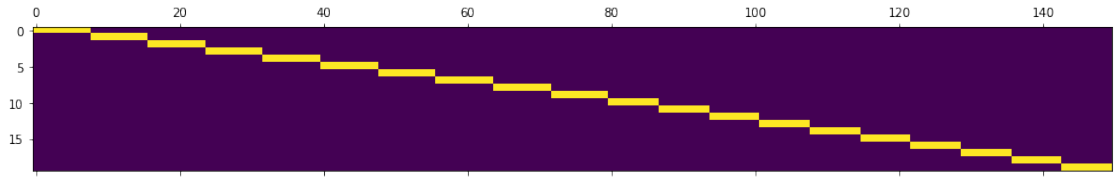
```
Out[138]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In [139]: cross_val_score(LinearSVC(), X, y % 2)

Out[139]: array([ 0.74509804,  0.76      ,  0.55102041])

In [140]: cross_val_score(LinearSVC(), X, y % 2, scoring="f1")

Out[140]: array([ 0.38095238,  0.625     ,  0.47619048])
```

Implementing your own scoring metric:

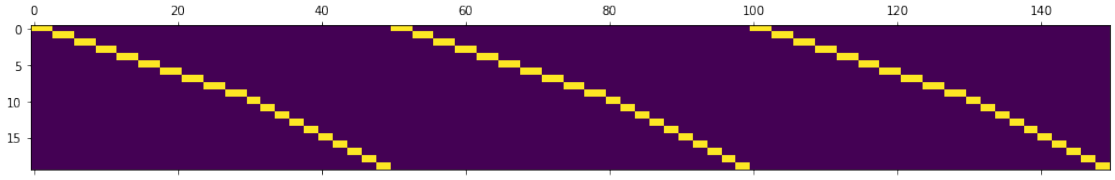
```
In [141]: def my_accuracy_scoring(est, X, y):
              return np.mean(est.predict(X) == y)

          cross_val_score(LinearSVC(), X, y%2, scoring=my_accuracy_scoring)

Out[141]: array([ 0.74509804,  0.76      ,  0.55102041])
```

### 3.6 There are different methods to create splits in cross-validation

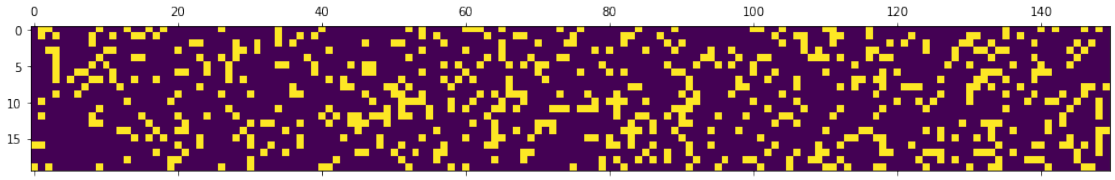
```
In [142]: from sklearn.model_selection import ShuffleSplit #random data omitted in

          shuffle_split = ShuffleSplit( 10, test_size=.4)
          cross_val_score(LinearSVC(), X, y, cv=shuffle_split)

Out[142]: array([ 0.9       ,  0.95      ,  1.        ,  0.95      ,  0.96666667,
                  0.95      ,  0.96666667,  0.96666667,  0.93333333,  0.95      ])

In [43]: from sklearn.cross_validation import StratifiedKFold, KFold, ShuffleSplit

         def plot_cv(cv, n_samples):
             masks = []
             for train, test in cv:
                 mask = np.zeros(n_samples, dtype=bool)
                 mask[test] = 1
                 masks.append(mask)
             plt.matshow(masks)

In [44]: plot_cv(KFold(len(X), n_folds=20), len(iris.target))
```



```
In [45]: plot_cv(StratifiedKFold(y, n_folds=20), len(y))
```

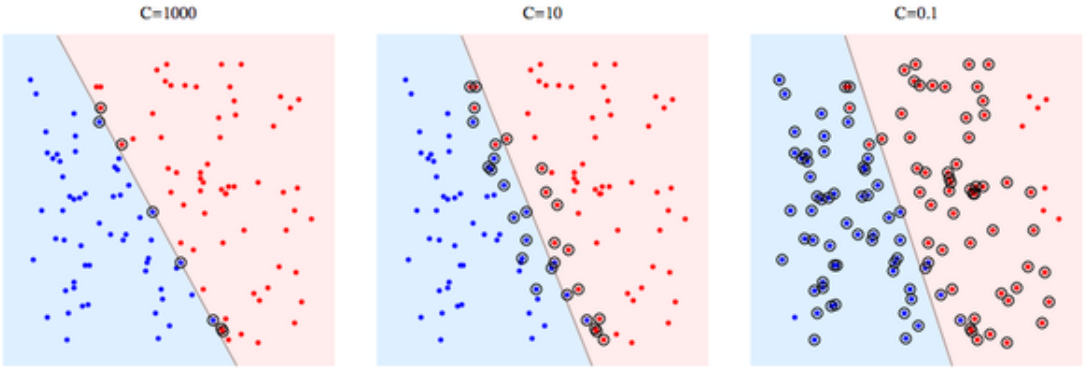



```
In [46]: plot_cv(ShuffleSplit( len(X),n_iter=20, test_size=.2), len(iris.target))

         #ShuffleSplit(
```




### 3.7 Hyperparameters of a model

We saw how varying values for k in a nearest neighbours algorithm affected the model accuracy.
How do we efficiently search for the best parameter values? (for a given metric)
This can be implemented during cross validation.
E.g. Linear SVC
Grid search over specified parameter values for an estimator like the "C" parameter.
We want a hyperplane that has the largest minimal margin **and** correctly classifies as many instances as possible. There is a tradeoff between the two.



```
In [143]: from sklearn.model_selection import GridSearchCV

          y = iris.target
          grid = GridSearchCV(LinearSVC(),{'C' : [0.001, 0.01,0.1,1.1, 100., 1000]}
          grid.fit(X, y)
          print(grid.best_params_)
          #print(grid.cv_results_)
```

```
{'C': 1.1}
```

Beware of parameter degeneracies in a model !

```
In [144]: grid = GridSearchCV(LinearSVC(penalty='l1',dual=False),{'C' : [0.001, 0.0
          grid.fit(X, y)
          print(grid.best_params_)
```

```
{'C': 100.0}
```

## 4 Validation Curves

Visual summary of the results of cross validation, hyperparameter search for training and test sets

```
In [49]: from sklearn.datasets import load_digits
         from sklearn.ensemble import RandomForestClassifier
         from sklearn.model_selection import validation_curve
         from sklearn.svm import LinearSVC
         from sklearn.neighbors import KNeighborsClassifier
```

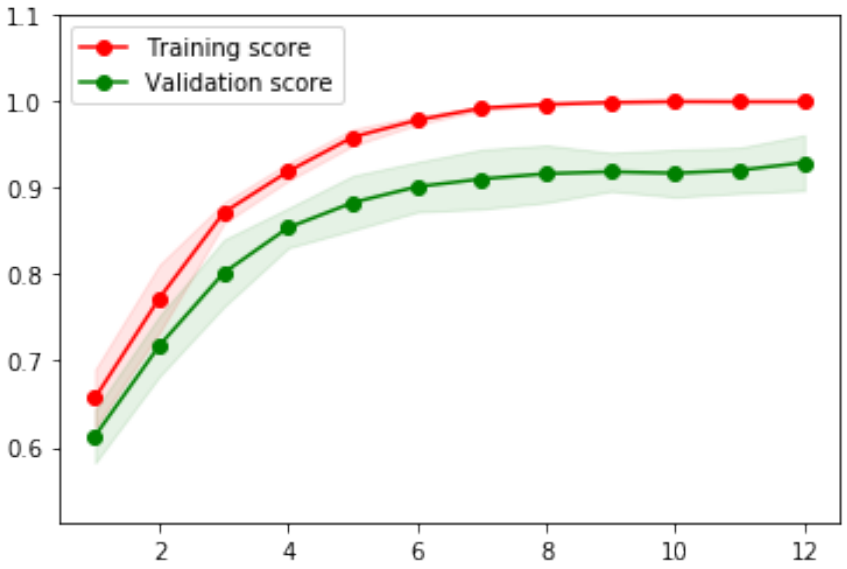
```
In [50]: digits = load_digits()
         X, y = digits.data, digits.target
```

```
In [51]: model = RandomForestClassifier(n_estimators=20)
         param_range = range(1, 13)
         training_scores, validation_scores = validation_curve(model, X, y,
                                                               param_name="max_dept
                                                               param_range=param_ra
```

```
In [104]: def plot_validation_curve(parameter_values, train_scores, validation_scor
              train_scores_mean = np.mean(train_scores, axis=1)
              train_scores_std = np.std(train_scores, axis=1)
              validation_scores_mean = np.mean(validation_scores, axis=1)
              validation_scores_std = np.std(validation_scores, axis=1)

              plt.fill_between(parameter_values, train_scores_mean - train_scores_s
```

```
                                train_scores_mean + train_scores_std, alpha=0.1,
                                color="r")
            plt.fill_between(parameter_values, validation_scores_mean - validatio
                                validation_scores_mean + validation_scores_std, alph
            plt.plot(parameter_values, train_scores_mean, 'o-', color="r",
                     label="Training score")
            plt.plot(parameter_values, validation_scores_mean, 'o-', color="g",
                     label="Validation score")
            plt.ylim(validation_scores_mean.min() - .1, train_scores_mean.max() +
            #plt.xscale('log')
            plt.legend(loc="best")
```

```
In [55]: plt.figure()
         plot_validation_curve(param_range, training_scores, validation_scores)
```




## 5 Exercise 1

Use KFold cross validation and StratifiedKFold cross validation (3 or 5 folds) for LinearSVC on the iris dataset. Why are the results so different? How could you get more similar results?

## 6 Exercise 2

Plot the validation curve on the digit dataset for: * a LinearSVC with a logarithmic range of regularization parameters `C` starting at $10^{-6}$. * KNeighborsClassifier with a linear range of neighbors `n_neighbors`.

What do you expect them to look like? How do they actually look like?