Introduction to Data Science
Lecture 8
Machine Learning
Part 2

EMSE 6992 Fall 2018

Benjamin Harvey

Outline

- Supervised Learning
  - Classification/Regression
  - Bias/Variance tradeoff
- K-NN
- Naïve Bayes
- Training Issues
  - Measuring model quality
  - Over-fitting
  - Cross-validation

Machine Learning

- **Supervised:** We are given input/output samples (X, y) which we relate with a function y = f(X). We would like to "learn" f, and evaluate it on new data. Types:
  - **Classification:** y is discrete (class labels).
  - **Regression:** y is continuous, e.g. linear regression.

- **Unsupervised:** Given only samples X of the data, we compute a function f such that y = f(X) is "simpler".
  - **Clustering:** y is discrete
  - Y is continuous: **Matrix factorization, Kalman filtering, unsupervised neural networks**.

Machine Learning

- **Supervised:**
  - Is this image a cat, dog, car, house?
  - How would this user score that restaurant?
  - Is this email spam?
  - Is this blob a supernova?
- **Unsupervised:**
  - Cluster some hand-written digit data into 10 classes.
  - What are the top 20 topics in Twitter right now?
  - Find and cluster distinct accents of people at GWU.

Techniques

- **Supervised Learning:**
  - kNN (k Nearest Neighbors)
  - Naïve Bayes
  - Linear + Logistic Regression
  - Support Vector Machines
  - Random Forests
  - Neural Networks
- **Unsupervised Learning:**
  - Clustering
  - Topic Models
  - HMMs (Hidden Markov Models)
  - Neural Networks

Outline

- Supervised Learning
  - Classification/Regression
  - Bias/Variance tradeoff
- K-NN
- Naïve Bayes
- Training Issues
  - Measuring model quality
  - Over-fitting
  - Cross-validation

Predicting from Samples

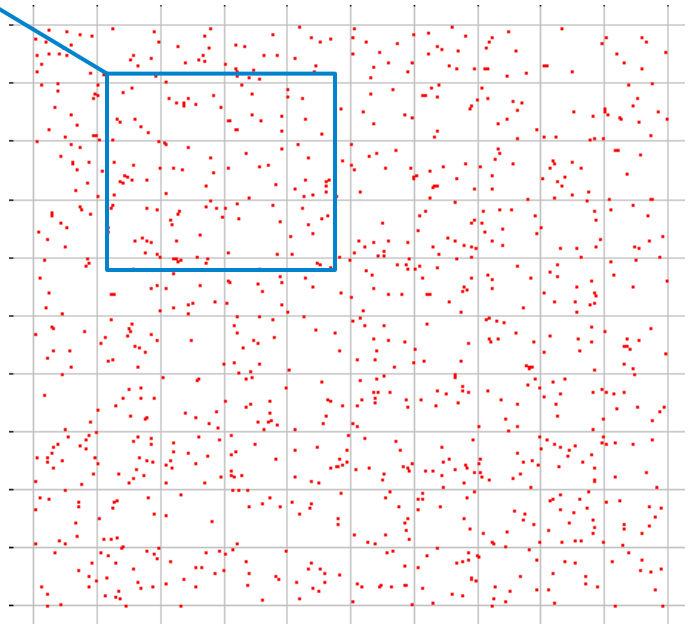
- Most datasets are **samples** from an **infinite population**.
- We are most interested in **models of the population**, but we have access only to a **sample** of it.

For datasets consisting of (X,y)

- features X + label y

a model is a prediction y = f(X)

We train on a training sample D
and we denote the model as $f_D(X)$



Bias and Variance

Our data-generated model $f_D(X)$ is a **statistical estimate** of the true function $f(X)$.

Because of this, its subject to bias and variance:

Bias: if we train models $f_D(X)$ on many training sets D, bias is the expected difference between their predictions and the true y's.

i.e. $$Bias = \mathrm{E}[f_D(X) - y]$$

E[] is taken over points X and datasets D

Variance: if we train models $f_D(X)$ on many training sets D, variance is the variance of the estimates:

$$Variance = \mathrm{E}\left[\left(f_D(X) - f(X)\right)^2\right]$$

Where $f(X) = \mathrm{E}[f_D(X)]$ is the average prediction on X.

Bias and Variance Tradeoff

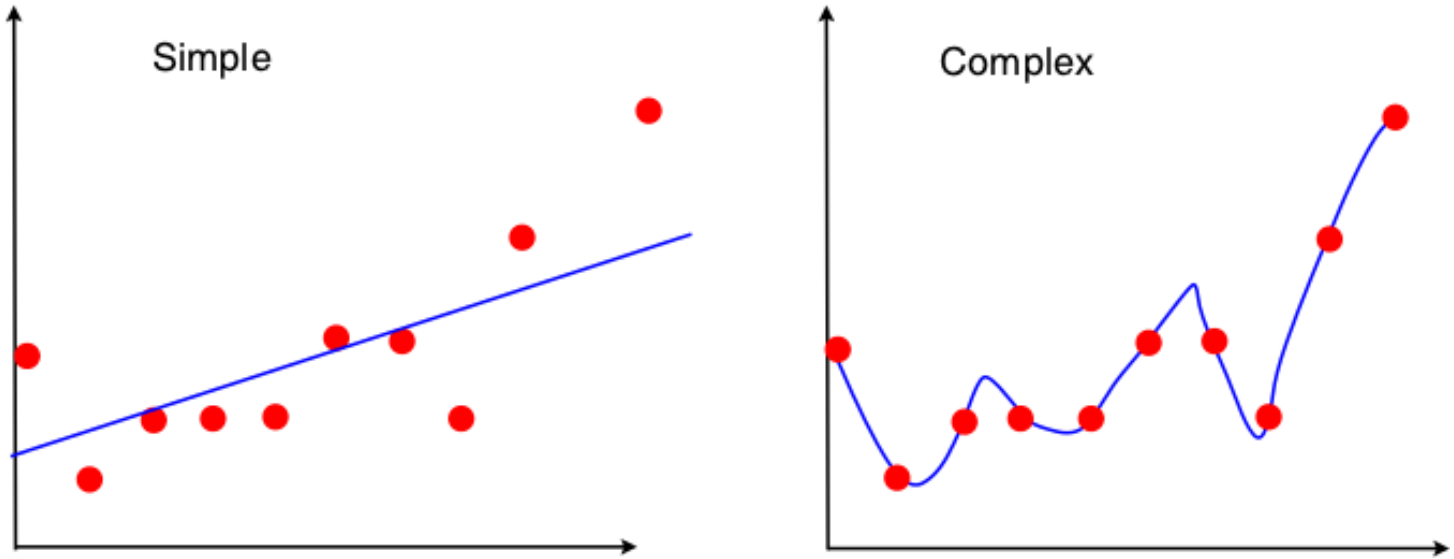
There is usually a bias-variance tradeoff caused by model complexity.

Complex models (many parameters) usually have lower bias, but higher variance.

Simple models (few parameters) have higher bias, but lower variance.

Bias and Variance Tradeoff

e.g. a linear model can only fit a straight line. A high-degree polynomial can fit a complex curve. But the polynomial can fit the individual sample, rather than the population. Its shape can vary from sample to sample, so it has high variance.



Bias and Variance Tradeoff

The total expected error is

$$Bias^2 + Variance$$

Because of the bias-variance trade-off, we want to **balance** these two contributions.

If $Variance$ strongly dominates, it means there is too much variation between models. This is called **over-fitting**.

If $Bias$ strongly dominates, then the models are not fitting the data well enough. This is called **under-fitting**.

Outline

- Supervised Learning
  - Classification/Regression
  - Bias/Variance tradeoff
- K-NN
- Naïve Bayes
- Training Issues
  - Measuring model quality
  - Over-fitting
  - Cross-validation

k-Nearest Neighbors

Given a query item:
Find k closest matches
in a labeled dataset ↓

k-Nearest Neighbors

Given a query item:
Find k closest matches

Return the most
Frequent label

k = 3 votes for “cat”

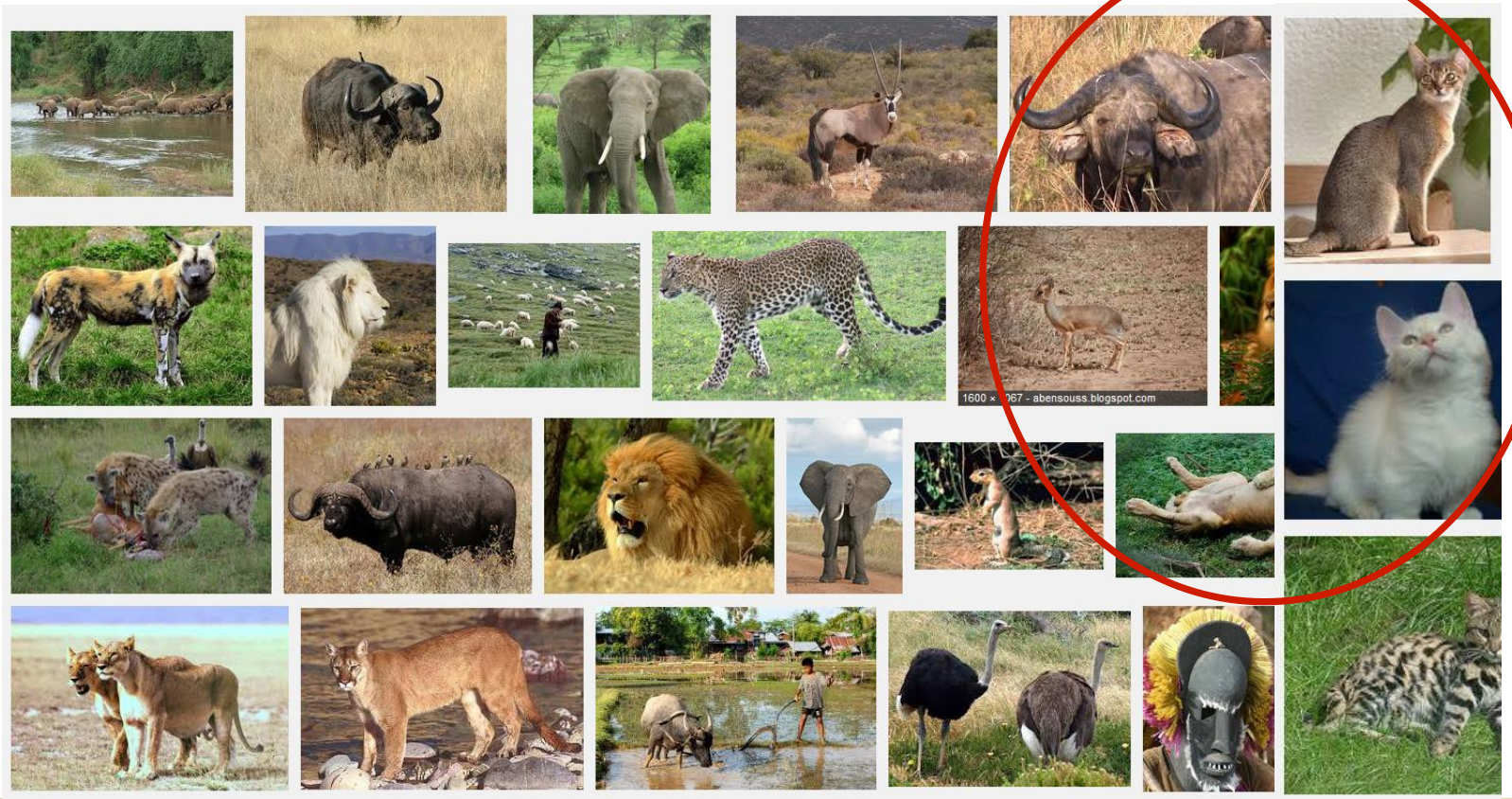
k-Nearest Neighbors

2 votes for cat,

1 each for Buffalo,

Deer, Lion

Cat wins…



k-NN issues

The Data is the Model

- No training needed.
- Accuracy generally improves with more data.
- Matching is simple and fairly fast if data fits in memory.
- Usually need data in memory, but can be run off disk.

Minimal Configuration:

- Only parameter is k (number of neighbors)
- But two other choices are important:
  - Weighting of neighbors (e.g. inverse distance)
  - Similarity metric

k-NN Flavors

Classification:

- Model is y = f(X), y is from a discrete set (labels).
- Given X, compute y = majority vote of the k nearest neighbors.
- Can also use a weighted vote* of the neighbors.

Regression:

- Model is y = f(X), y is a real value.
- Given X, compute y = average value of the k nearest neighbors.
- Can also use a weighted average* of the neighbors.

* Weight function is usually the inverse distance.

K-NN distance measures

- **Euclidean Distance:** Simplest, fast to compute

$$d(x, y) = \|x - y\|$$

- **Cosine Distance:** Good for documents, images, etc.

$$d(x, y) = 1 - \frac{x \cdot y}{\|x\|\|y\|}$$

- **Jaccard Distance:** For set data:

$$d(X, Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|}$$

- **Hamming Distance:** For string data:

$$d(x, y) = \sum_{i=1}^{n} (x_i \neq y_i)$$

K-NN metrics

- **Manhattan Distance:** Coordinate-wise distance

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

- **Edit Distance:** for strings, especially genetic data.

- **Mahalanobis Distance:** Normalized by the sample covariance matrix – unaffected by coordinate transformations.

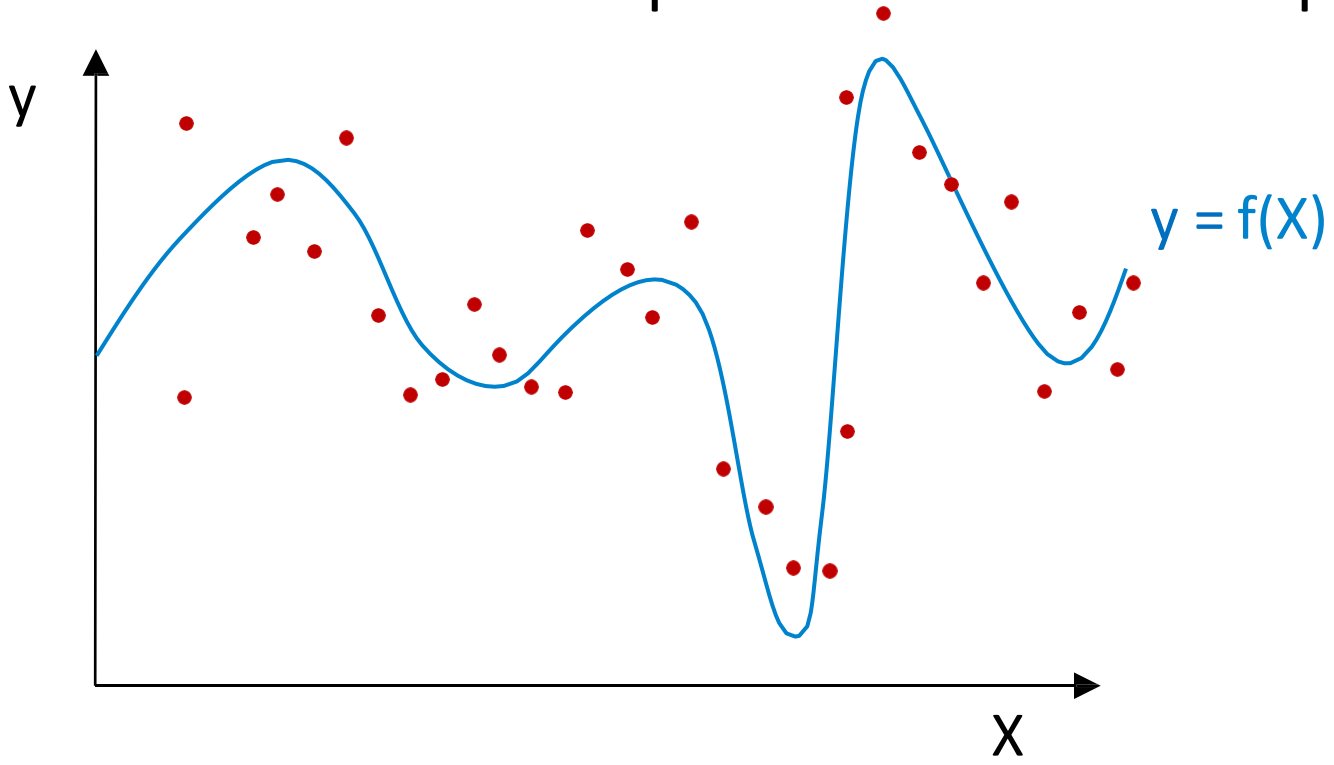
Choosing k

We have a bias/variance tradeoff:

- Small k → ?
- Large k → ?

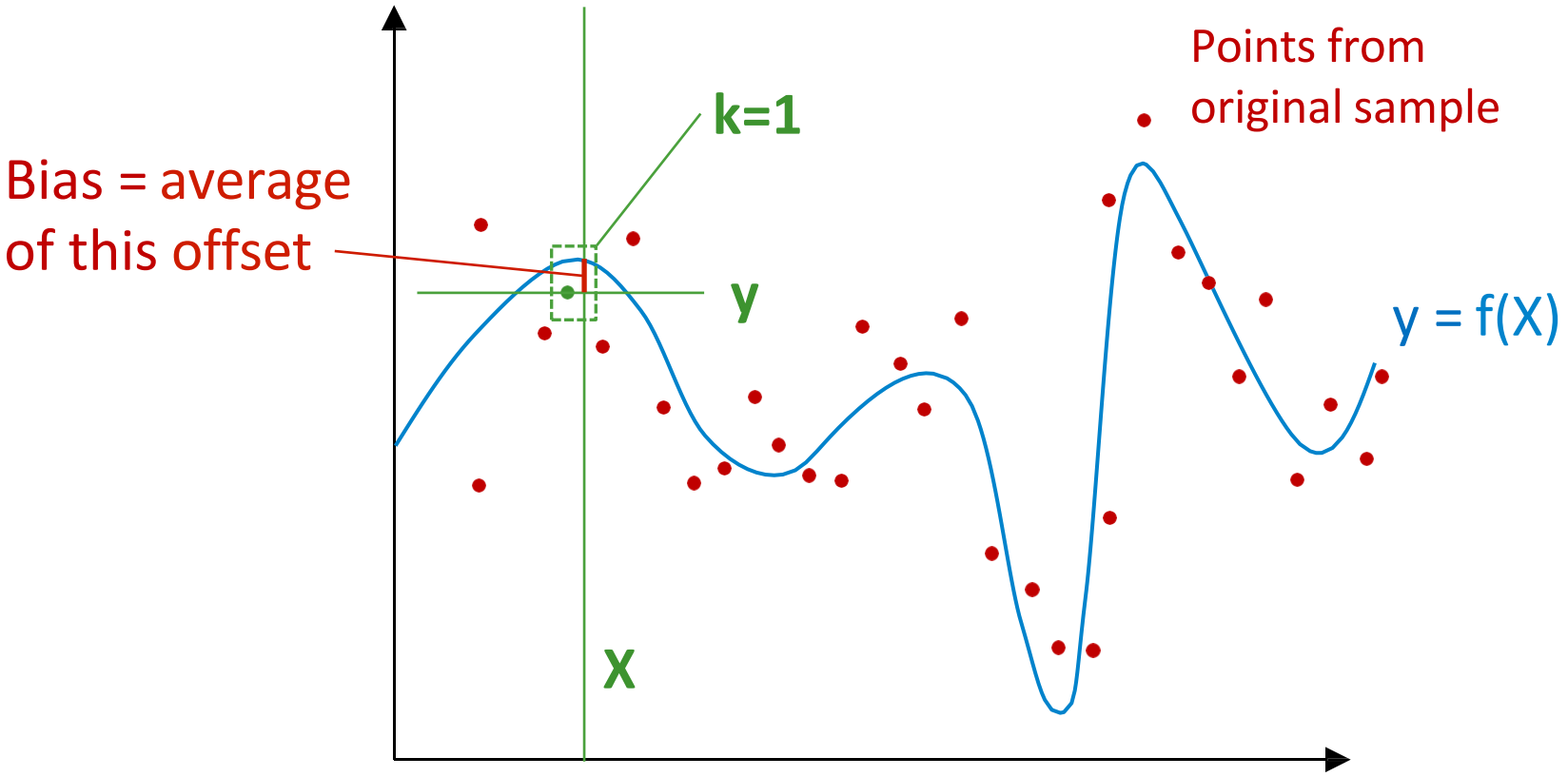
Choosing k

- Small k → low bias, high variance
- Large k → high bias, low variance
- Assume the real data follows the blue curve, with some mean-zero additive noise. Red points are a data sample.



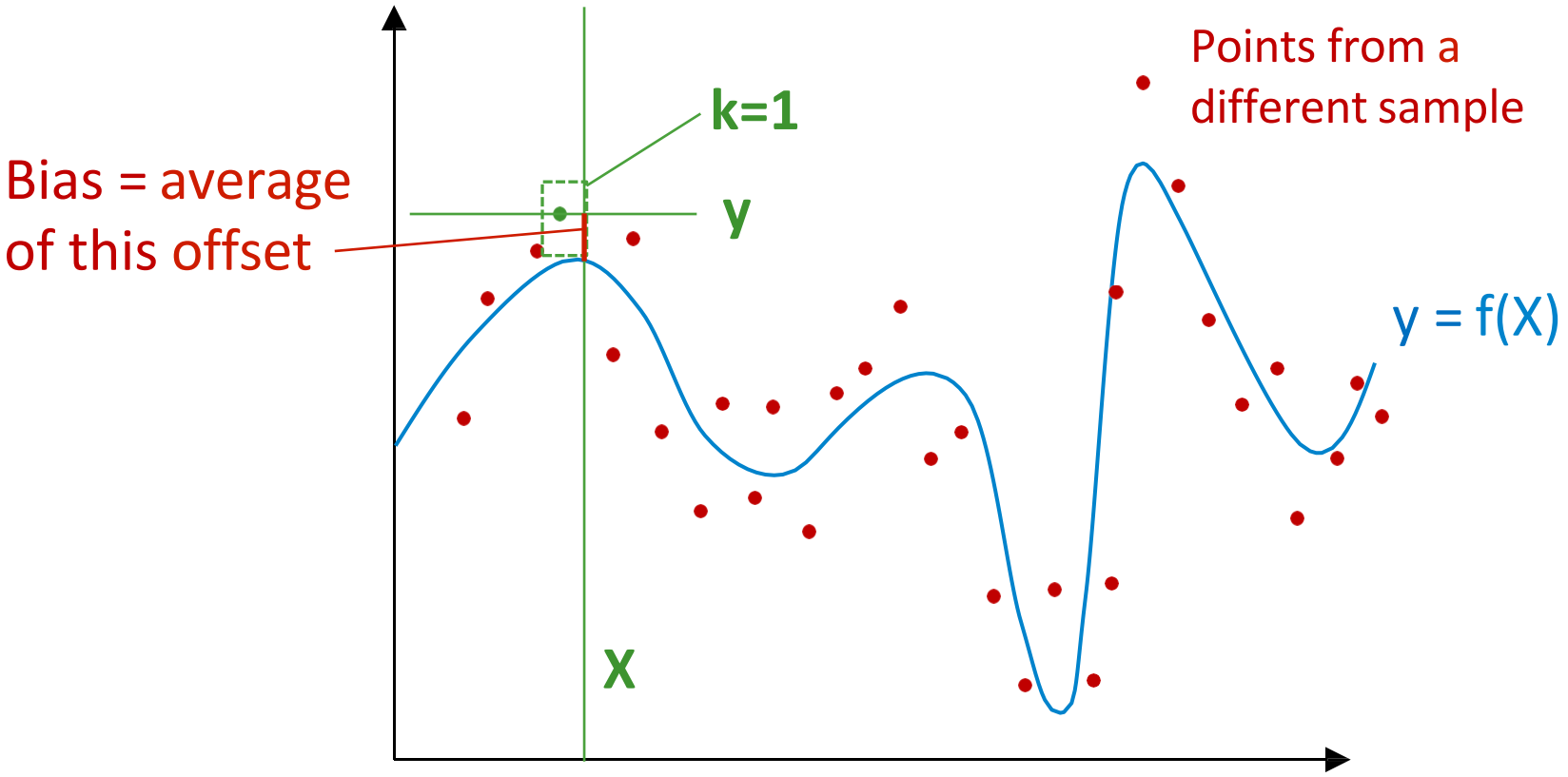
Choosing k

- **Small k** → low bias, high variance
- Large k → high bias, low variance



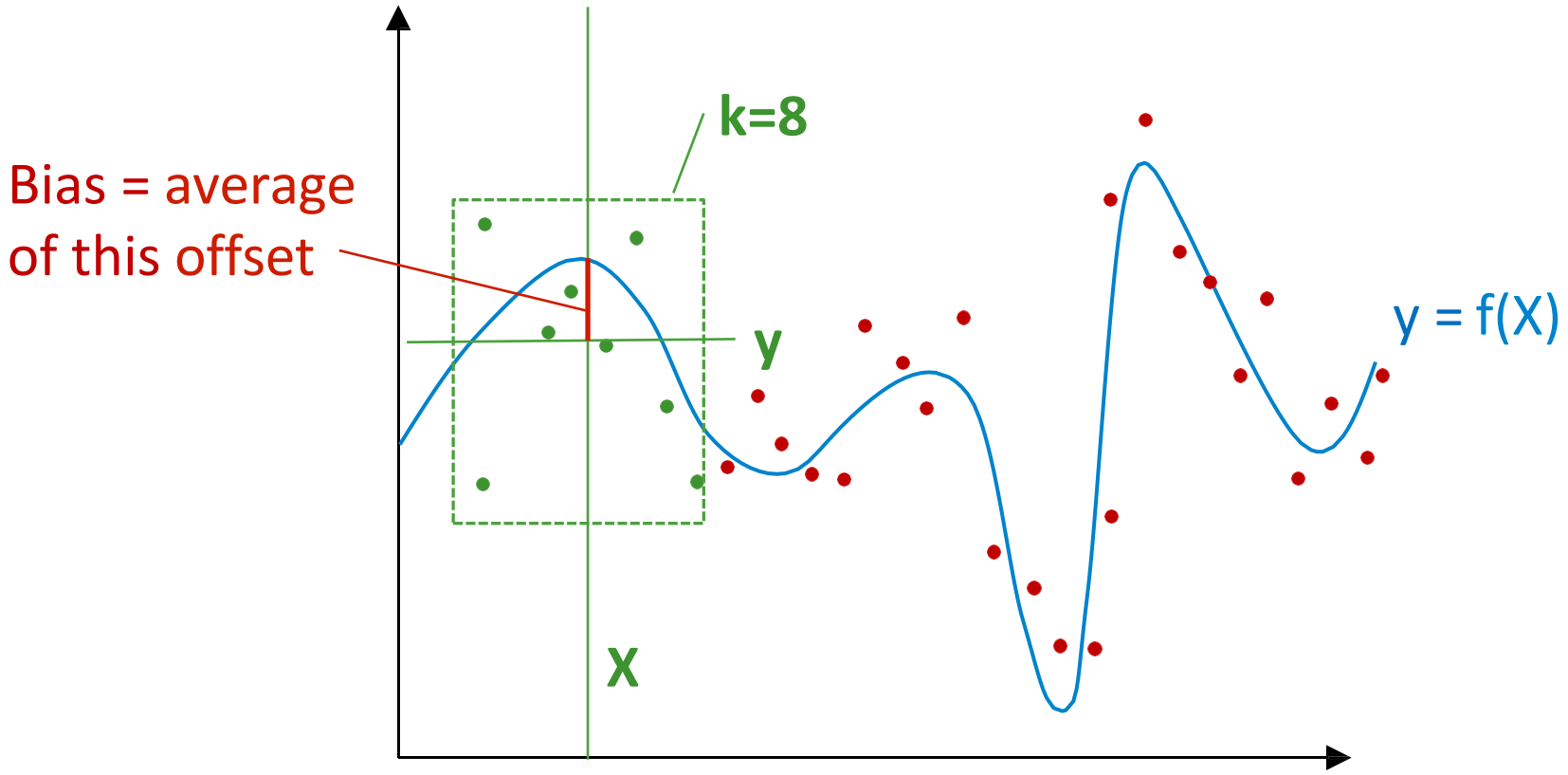
Choosing k

- **Small k** → low bias, high variance
- Large k → high bias, low variance



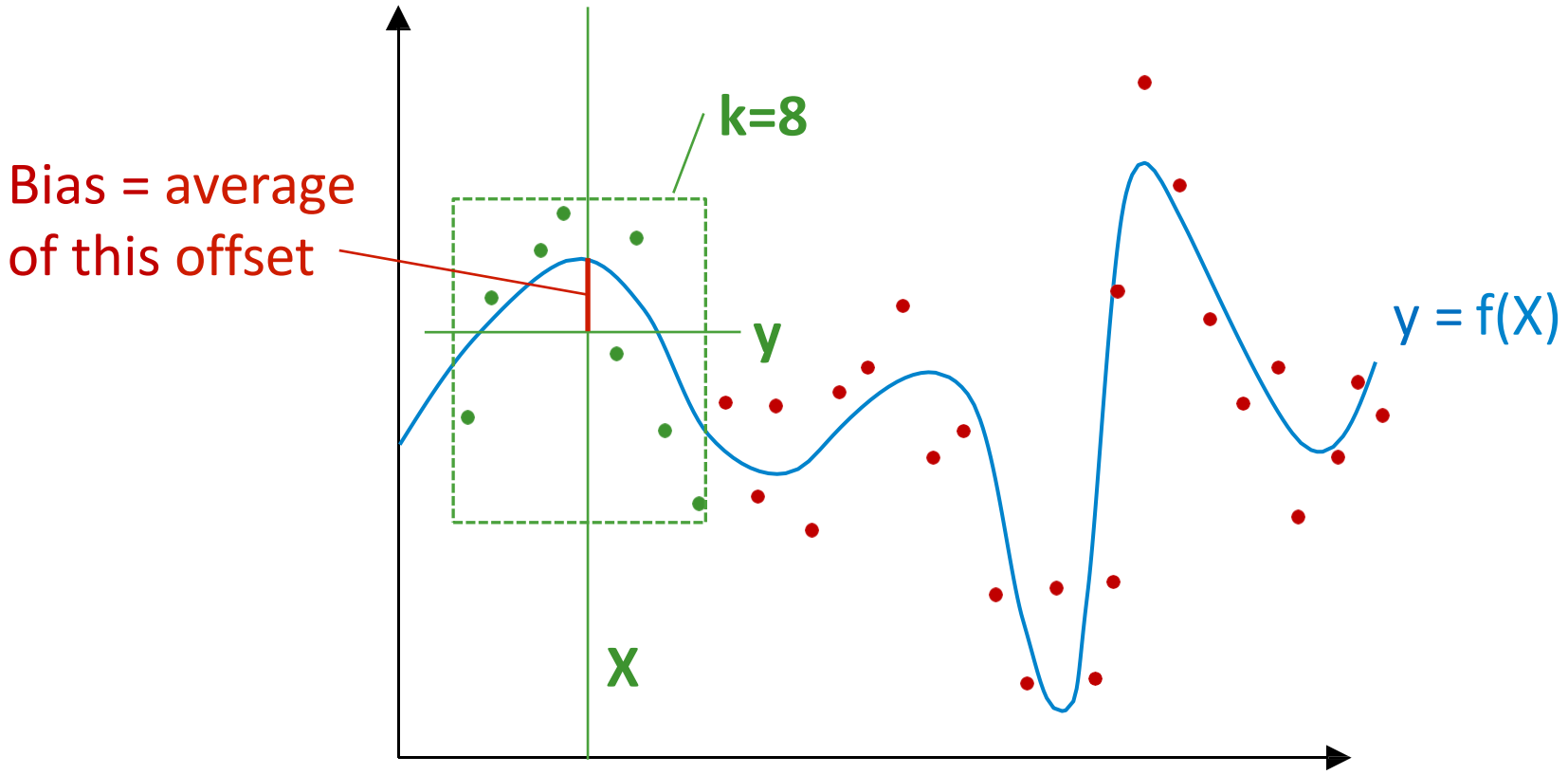
Choosing k

- Small k → low bias, high variance
- **Large k** → high bias, low variance



Choosing k

- Small k → low bias, high variance
- **Large k** → high bias, low variance



Choosing k in practice

Use cross-validation! Break data into train,validation and test subsets, e.g. 60-20-20 % random split.

Predict: For each point in the validation set, predict using the k-Nearest neighbors from the training set. Measure the error rate (classification) or squared error (regression).

Tune: try different values of k, and use the one that gives minimum error on the validation set.

Evaluate: test on the test set to measure performance.

kNN and the curse of dimensionality

The curse of dimensionality refers to phenomena that occur in high dimensions (100s to millions) that do not occur in low-dimensional (e.g. 3-dimensional) space.

In particular data in high dimensions are much sparser (less dense) than data in low dimensions.

For kNN, that means there are less points that are very close in feature space (very similar), to the point we want to predict.

kNN and the curse of dimensionality

Example: Consider a collection of uniformly random points in the unit cube. In one dimension, the average squared Euclidean distance between any two points is:

$$\int_0^1 \int_0^1 (x-y)^2 dx\, dy = \frac{1}{6}$$

In N dimensions, we add up the squared differences for all N coordinates (because the coordinates are independent in a uniform random cube), giving:

$$d^2 = \mathrm{E}[\|x-y\|^2] = \frac{N}{6}$$

So the euclidean distance scales as $\sqrt{N}$

kNN and the curse of dimensionality

From this perspective, its surprising that kNN works at all in high dimensions.

Luckily real data are not like random points in a high-dimensional cube. Instead they live in **dense clusters** and near **much lower-dimensional surfaces**.

Finally, points can be very "similar" even if their euclidean distance is large. E.g. documents with the same few dominant words (with tf-idf weighing) are likely to be on the same topic.

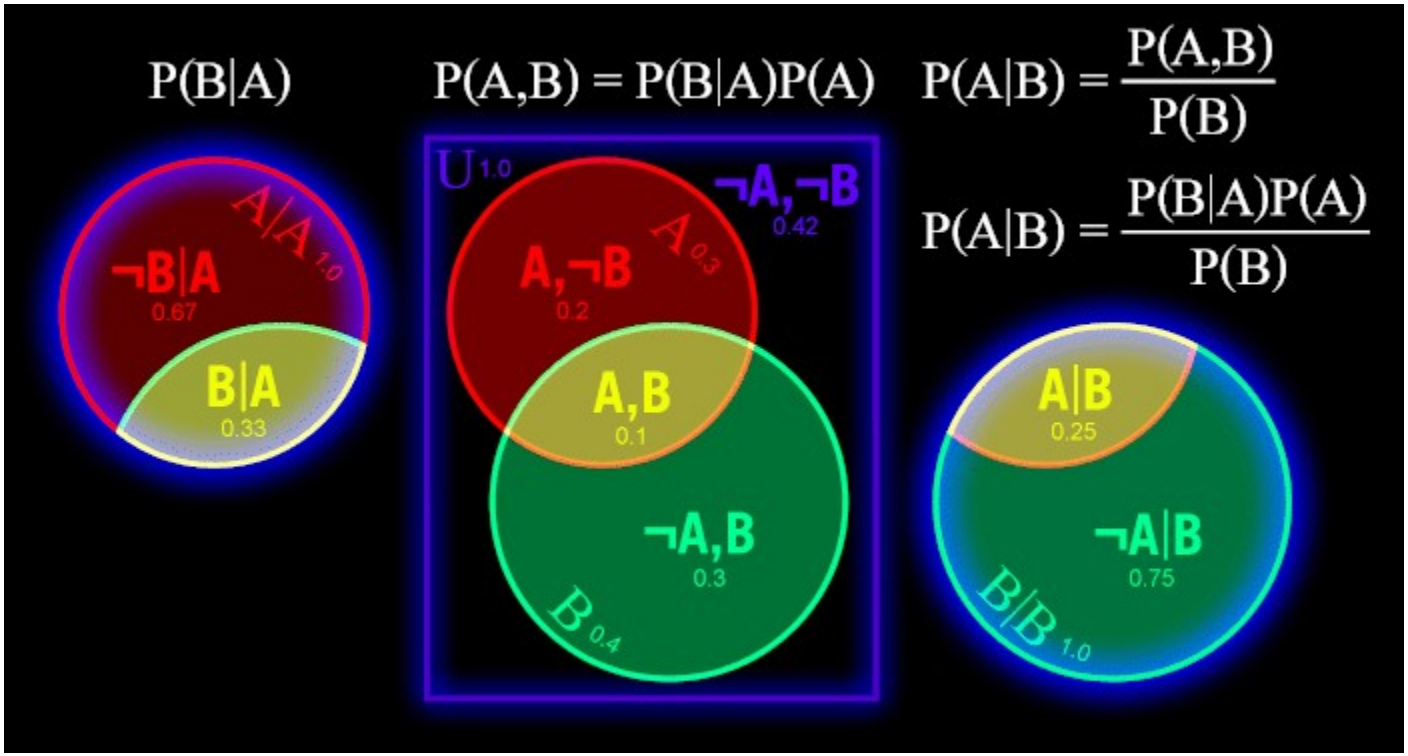
Outline

- Supervised Learning
  - Classification/Regression
  - Bias/Variance tradeoff
- K-NN
- Naïve Bayes
- Training Issues
  - Measuring model quality
  - Over-fitting
  - Cross-validation

Bayes' Theorem

- P(A|B) = probability of A given that B is true.
- $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$
- In practice we are most interested in dealing with events e and data D
- e = "I have a cold"
- D = "runny nose," "watery eyes," "coughing"
- $P(e|D) = \frac{P(D|e)P(e)}{P(D)}$
- So Bayes' theorem is "diagnostic".

Bayes' Theorem



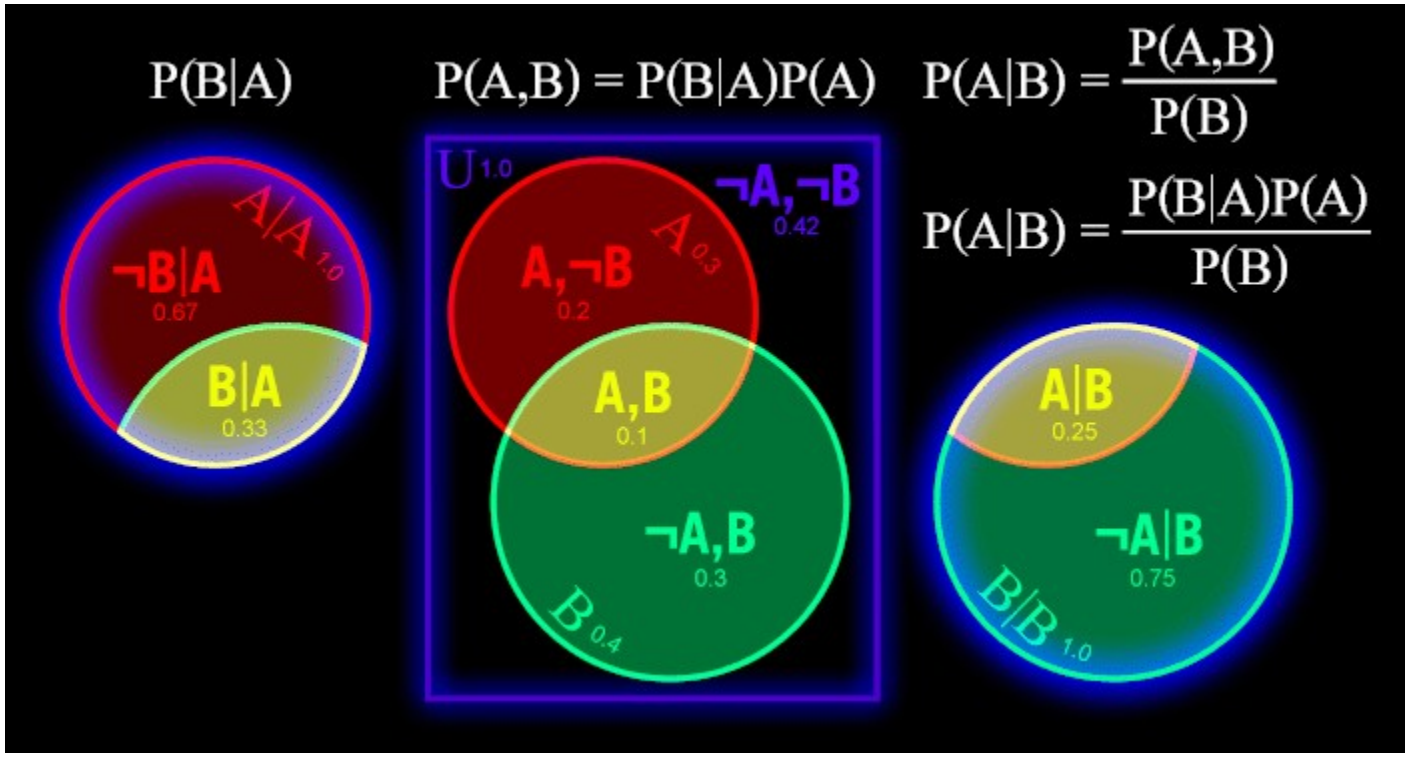
Assumes probability of event is proportional to area of set.

P(A) = area of A.

P(A,B) = area of A intersect B.

P(A|B) = P(A,B) / P(B)

Bayes' Theorem



The theorem follows by writing:

$$P(A|B)P(B) = P(A,B) = P(B|A)P(A)$$

From which we get:

$$P(A|B) = P(B|A)\,P(A) / P(B)$$

Bayes' Terminology

D = Data, e = some event

$$P(e|D) = \frac{P(D|e)P(e)}{P(D)}$$

P(e) is called the **prior probability** of e. Its what we know (or think we know) about e with no other evidence.

P(D|e) is the **conditional probability** of D given that e happened, or just the likelihood of D. This can often be measured or computed precisely – it follows from your model assumptions.

P(e|D) is the **posterior probability** of e given D. It's the answer we want, or the way we choose a best answer.

You can see that the posterior is heavily colored by the prior, so Bayes' has a GIGO liability. e.g. its not used to test hypotheses

Naïve Bayes Classifier

Let's assume we have an instance (e.g. a document d) with a set of features $(X_1, \ldots, X_k)$ and a set of classes $\{c_j\}$ to which the document might belong.

We want to find the **most likely class** that the document belongs to, given its features.

The joint probability of the class and features is:

$$\Pr(X_1, \ldots, X_k, c_j)$$

Naïve Bayes Classifier

Key Assumption: (Naïve) the features are **generated independently** given c_j. Then the joint probability factors:

$$\Pr(X, c_j) = \Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j) = \Pr(c_j) \sum_{i=1}^{k} \Pr(X_i \mid c_j)$$

We would like to figure out the **most likely class for** (i.e. to classify) the document, which is the c_j which maximizes:

$$\Pr(c_j \mid X_1, \ldots, X_k)$$

Naïve Bayes Classifier

Now from Bayes we know that:

$$\Pr(c_j \mid X_1, \ldots, X_k) = \Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j) / \Pr(X_1, \ldots, X_k)$$

But to choose the best c_j, we can ignore $\Pr(X_1, \ldots, X_k)$ since it's the same for every class. So we just have to maximize:

$$\Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j)$$

So finally we pick the category c_j that maximizes:

$$\Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j) = \Pr(c_j) \prod_{i=1}^{k} \Pr(X_i \mid c_j)$$

Naïve Bayes Classifier

Now from Bayes we know that:

$$\Pr(c_j \mid X_1, \ldots, X_k) = \Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j) / \Pr(X_1, \ldots, X_k)$$

(A B B A A B)

But to choose the best c_j, we can ignore $\Pr(X_1, \ldots, X_k)$ since it's the same for every class. So we just have to maximize:

$$\Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j)$$

So finally we pick the category c_j that maximizes:

$$\Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j) = \Pr(c_j) \prod_{i=1}^{k} \Pr(X_i \mid c_j)$$

Data for Naïve Bayes

In order to find the best class, we need two pieces of data:

- $\Pr(c_j)$ the prior probability for the class c_j.
- $\Pr(X_i \mid c_j)$ the conditional probability of the feature X_i given the class c_j.

Data for Naïve Bayes

For these two data, we only need to record counts:

- $\Pr(c_j) = \frac{N_d(c_j)}{N_d}$

- $\Pr(X_i|c_j) = \frac{N_w(X_i, c_j)}{N_w(c_j)}$

Where $N_d(c_j)$ is the number of documents in class c_j, N_d is the total number of documents.

$N_w(X_i, c_j)$ is the number of times X_i occurs in a document in c_j, and and $N_w(c_j)$ is the total number of features in all docs in c_j.

“Training” Naïve Bayes

So there is no need to train a Naïve Bayes classifier, only to accumulate the N_w and N_d counts.

But count data is only an approximation to those probabilities however, and a count of zero is problematic (why)?

Naïve Bayes and Smoothing

But count data is only an approximation to those probabilities however, and a count of zero is problematic: It forces the classifier not to choose classes with zero counts.

Instead of direct count ratios, we can use **Laplace Smoothing**:

$$p = \frac{N_1 + \alpha}{N_2 + \beta}$$

With constants α and β. These values push p toward a **prior** of α/β.

These constants can either be set based on prior knowledge, or learned during a training phase.

Practical Naïve Bayes

We want the category c_j that maximizes:

$$\Pr(X_1, \ldots, X_k | c_j) \Pr(c_j) = \Pr(c_j) \sum_{i=1}^{k} \Pr(X_i | c_j)$$

The log of this expression has the same maximum:

$$\log \Pr(c_j) + \sum_{i=1}^{k} \log \Pr(X_i | c_j)$$

This formula is much better numerically (avoids floating point underflow) and involves simple vector operations.

Complexity of evaluating this expression is proportional to k, the number of features.

Practical Naïve Bayes

Find the class c_j that maximizes:

$$\log \Pr(c_j) + \sum_{i=1}^{k} \log \Pr(X_i | c_j)$$

Often we have sparse data (e.g. documents) where most X_i are zero.

In that case, we can evaluate the NB formula in time proportional to s, where s is the number of non-zeros in each document. (good HW problem).

Good, Bad and Ugly of NB Classifiers

- **Simple and fast.** Depend only on term frequency data for the classes. One pass over the training dataset, no iteration.
- **Compact model.** Size is proportional to numfeats x numclasses.
- **Very well-behaved numerically.** Term weight depends only on frequency of that term. Decoupled from other terms.
- **Can work very well with sparse data**, where combinations of dependent terms are rare.
- **Subject to error and bias** when term probabilities are not independent (e.g. URL prefixes).
- **Can't model patterns** in the data.
- **Typically not as accurate** as other methods for the above reasons.

Summary

- Supervised Learning
  - Classification/Regression
  - Bias/Variance tradeoff
- K-NN
- Naïve Bayes
- **Training Issues**
  - Measuring model quality
  - Over-fitting
  - Cross-validation

Model Quality

Almost every model **optimizes some quality criterion**:

- For linear regression it was the **Residual Sum-of-Squares**
- For k-Means it is the **"Inertia"** – the mean squared distance from each sample to its cluster center.
- …

The quality criterion is chosen often because of its good properties:

- **Convexity:** so that there is a unique, best solution
- **Closed form** for the optimum (linear regression) or at least for the gradient (for SGD).
- An algorithm that **provably converges**.

Model Quality

There are typically other criteria used to measure the quality of models. e.g. for clustering models:

- Silhouette score
- Inter-cluster similarity (e.g. mutual information)
- Intra-cluster entropy

For regression models:

- Stability of the model (sensitivity to small changes)
- Compactness (sparseness or many zero coefficients)

Evaluating Clusterings: Silhouette

The silhouette score is

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

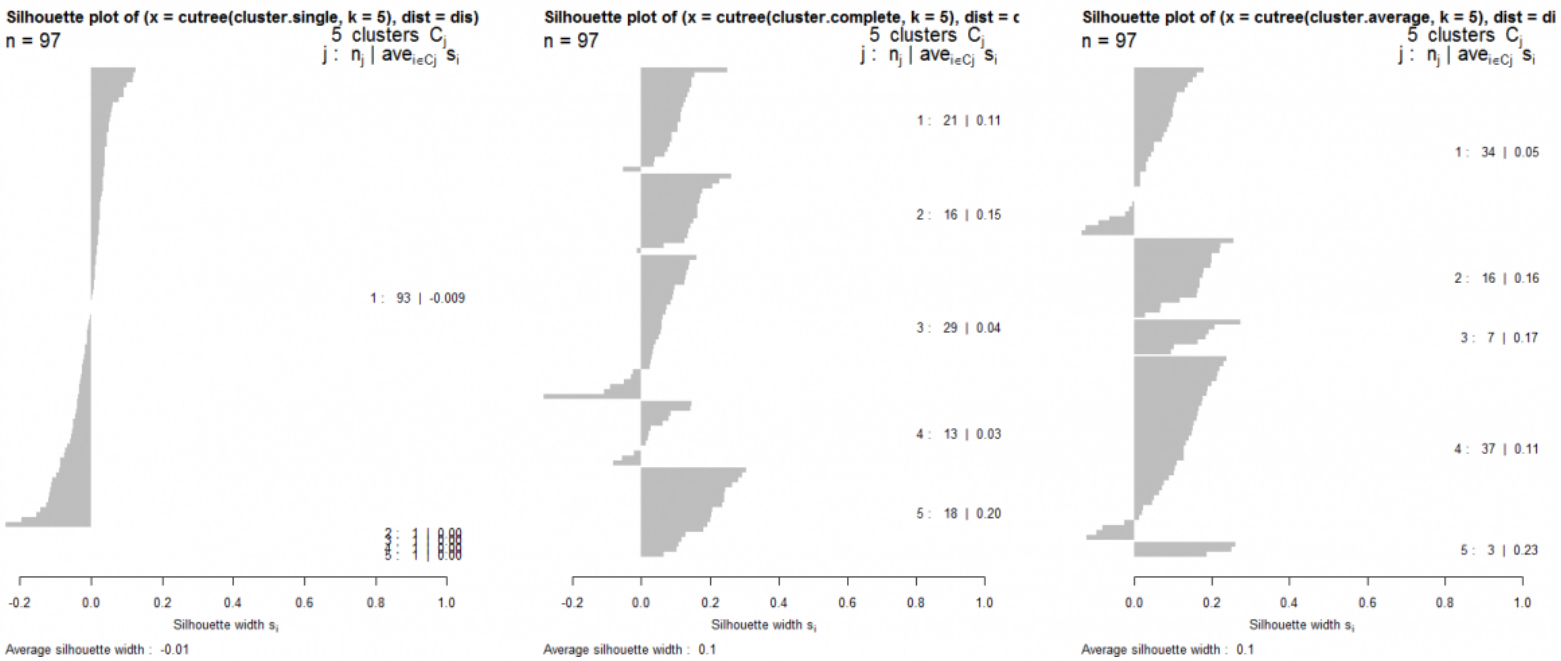
where a(i) is the mean distance from sample i to its own cluster, b(i) the mean distance from i to the second-closest cluster.

- Perhaps surprisingly, silhouette scores can be, and often are, negative.

Evaluating Clusterings: Silhouette

Silhouette plot: horizontal bars with cluster score.

Sort (vertically) first by cluster, then by score.



Regularization with Secondary Criteria

While secondary criteria can be measured after the model is built, its too late then to affect the model.

Using secondary criteria **during** the optimization process is called **"regularization".**

Examples:

- **L1 regularization** adds a term to the measure being optimized which is the **sum of absolute value of model coefficients**.
- **L2 regularization** adds a term to the measure being optimized which is the **sum of squares of model coefficients**.

Regularization with Secondary Criteria

L1 regularization in particular is very widely used. It has the following impacts:

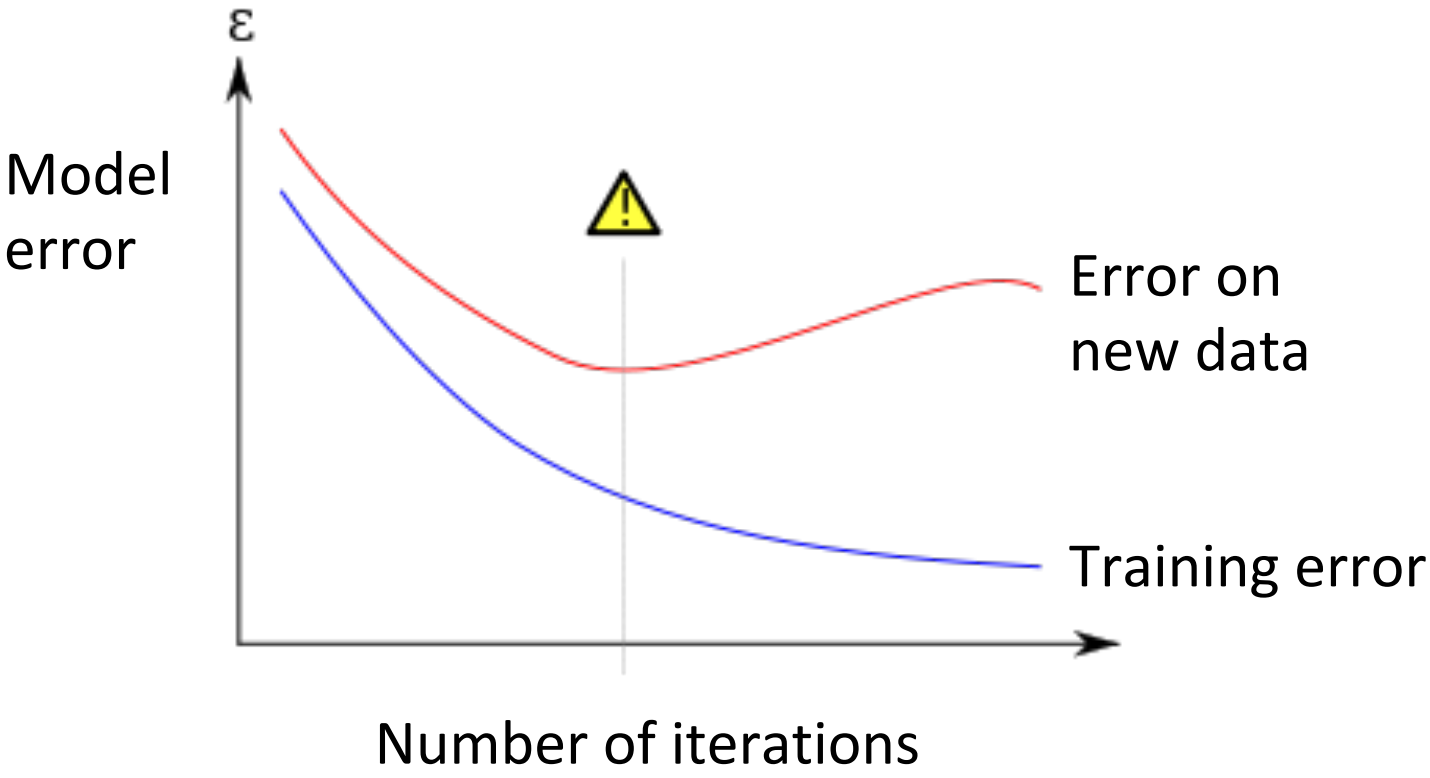
- Yields a **convex optimization** problem in many cases, so there is a unique solution.
- The solution is usually **stable** to small input changes.
- The solution is **quite sparse** (many zero coefficients) and requires less disk and memory to run.
- L1 regularization on factorization models tends to **decrease the correlation** between model factors.

Over-fitting

- Your model should ideally fit an **infinite sample** of the type of data you're interested in.
- In reality, you only have a **finite set** to train on. A good model for this subset is a good model for the infinite set, up to a point.
- Beyond that point, the model quality (measured on new data) starts to **decrease**.
- Beyond that point, the model is **over-fitting** the data.

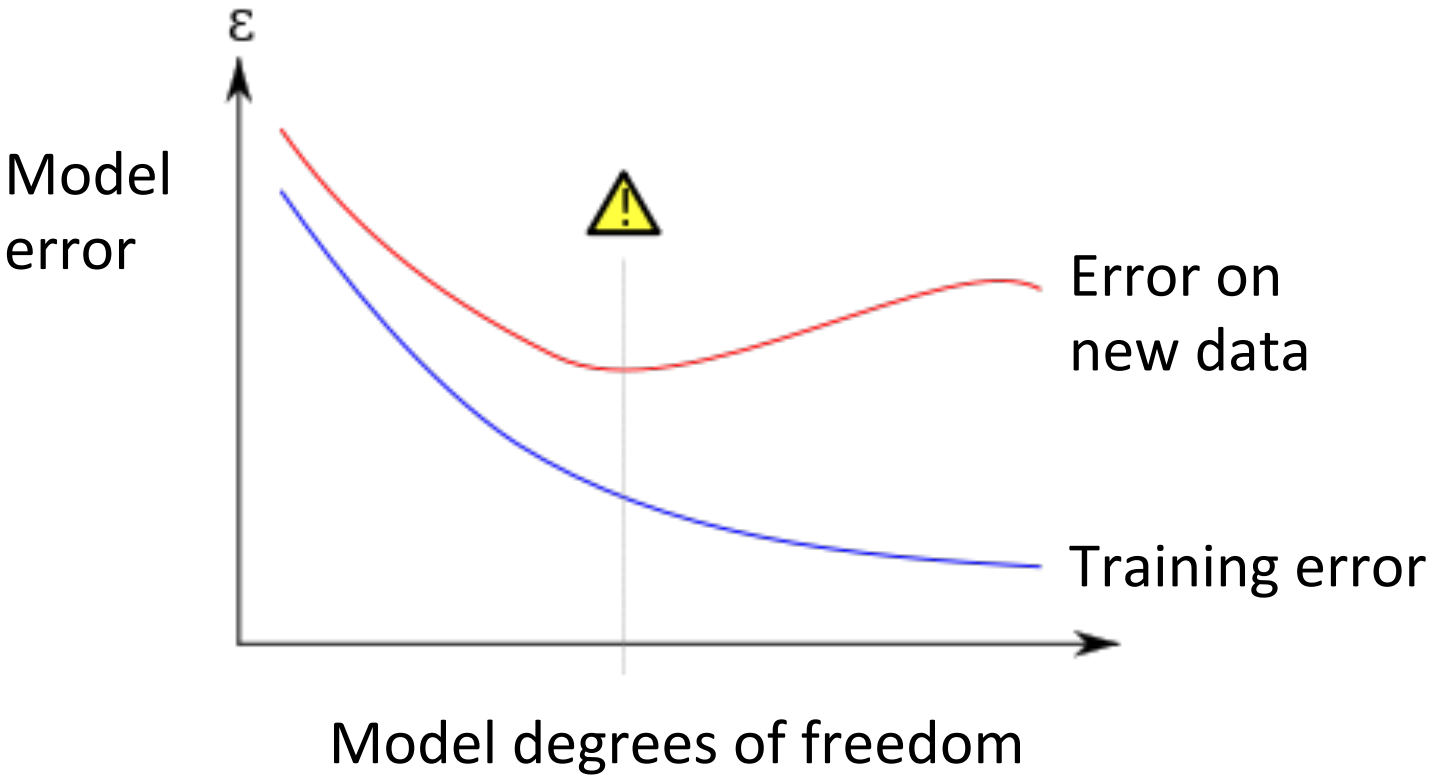
Over-fitting

Over-fitting during training



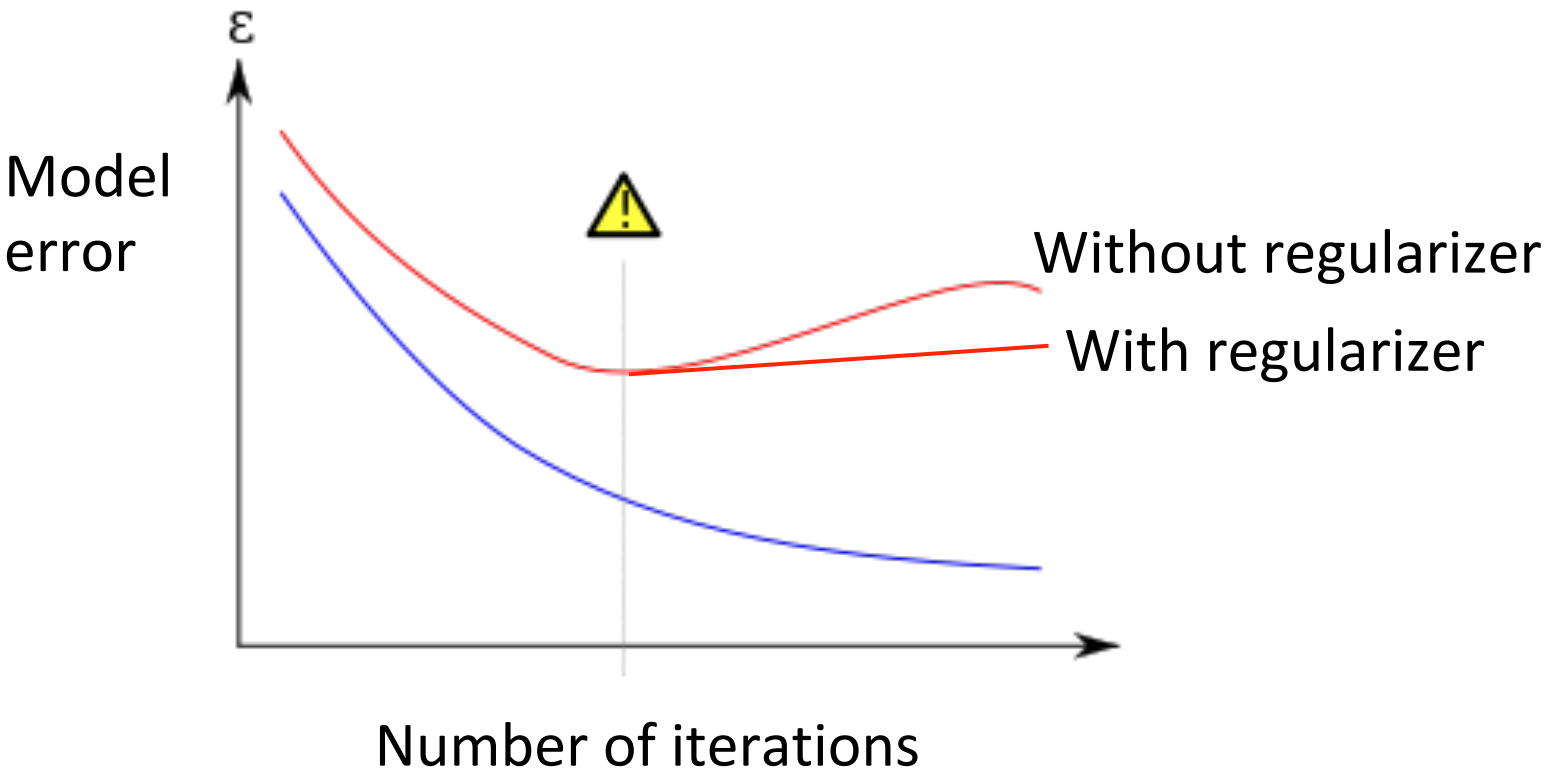
Over-fitting

Another kind of over-fitting



Regularization and Over-fitting

Adding a regularizer:



Cross-Validation

- Cross-validation involves **partitioning** your data into distinct **training** and **test** subsets.

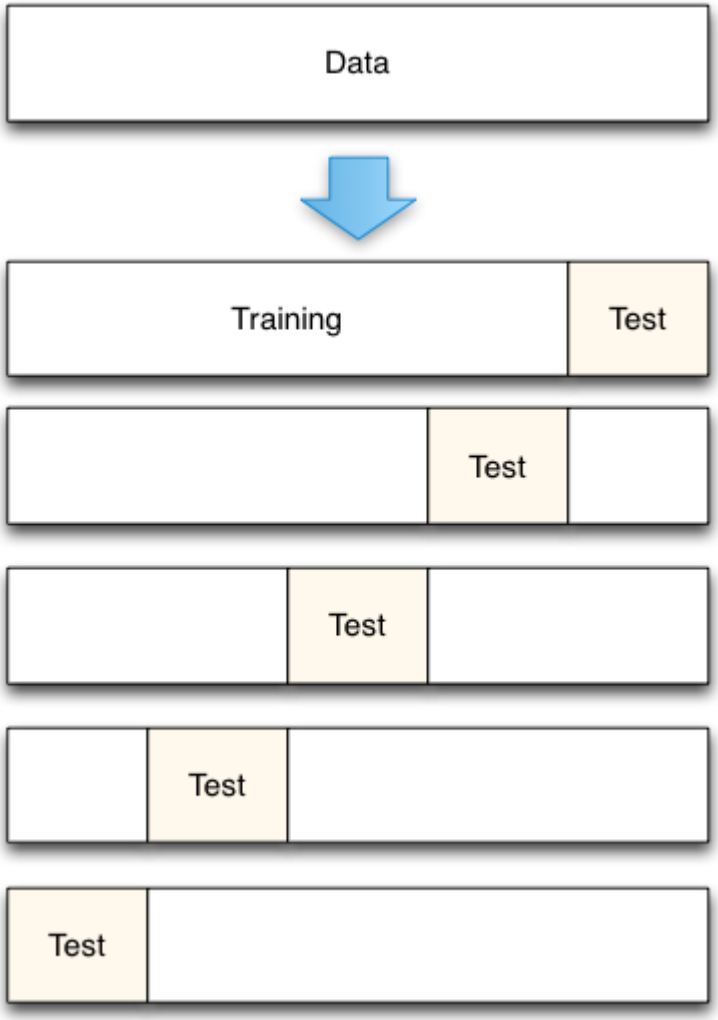
- The test set **should never** be used to **train** the model.

- The test set is then used to **evaluate** the model after training.

K-fold Cross-Validation

- To get more accurate estimates of performance you can do this k times.
- Break the data into k equal-sized subsets A_i
- For each i in 1,…,k do:
  - Train a model on all the other folds A_1,…, A_{i-1}, A_{i+1},…, A_k
  - Test the model on A_i
- Compute the **average performance** of the k runs

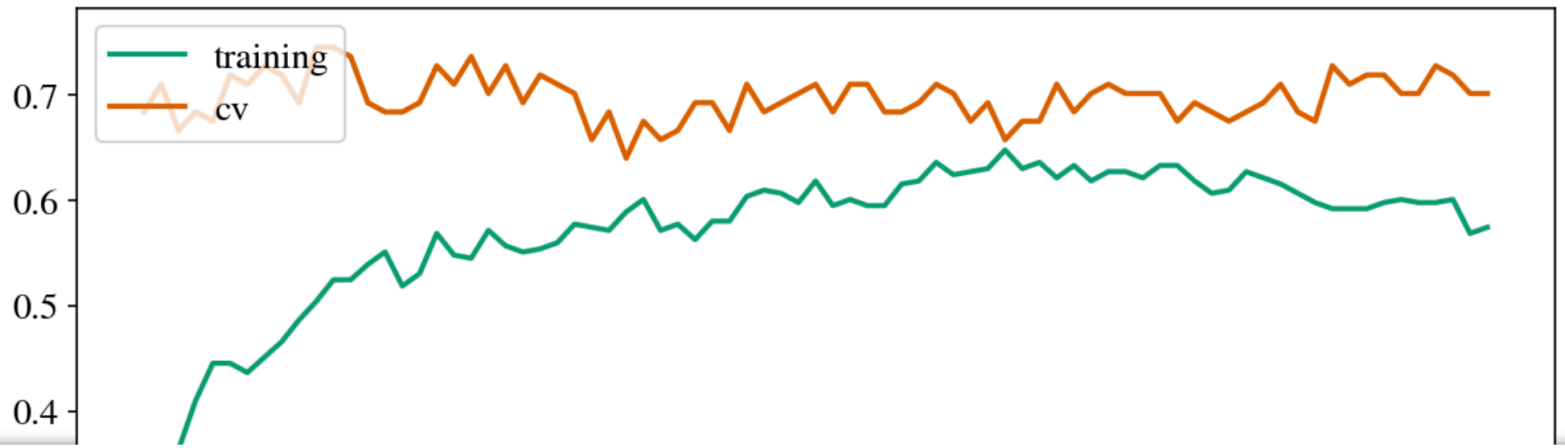
5-fold Cross-Validation



Homework Examples

In [68]:
```
#Instruction#11: Develop an a KNN classifier
Xcv,Xte,ycv,yte=train_test_split(Xtest, ytest, train_size=0.5)
print (ytrain.shape, ycv.shape, yte.shape)
trscores=[]
cvscores=[]
ns=np.arange(1,80,1)
ones=np.ones(len(ns))
for n in ns:
    clf = KNeighborsClassifier(n).fit(Xtrain, ytrain)
    trscores.append(clf.score(Xtrain, ytrain))
    cvscores.append(clf.score(Xcv, ycv))
plt.plot(ns, ones-trscores, label="training")
plt.plot(ns, ones-cvscores, label="cv")
plt.legend(loc='upper left');
print (clf.score(Xte, yte))
```

```
(341,) (114,) (114,)
0.333333333333
```



Homework Examples

In [69]:
```
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, train_size=0.8)
Xtr=np.concatenate((Xtrain, Xtest))
```

In [72]:
```
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import classification_report
parameters = {"n_neighbors": np.arange(1,80,1)}
clf = KNeighborsClassifier()
gs = GridSearchCV(clf, param_grid=parameters, cv=10)
gs.fit(Xtrain, ytrain)
#print gs.grid_scores_
print (gs.best_params_, gs.best_score_)
y_true, y_pred = ytest, gs.predict(Xtest)
print(classification_report(y_true, y_pred))
```

```
{'n_neighbors': 49} 0.34065934065934067
             precision    recall  f1-score   support

          1       0.29      0.60      0.39        35
          2       0.20      0.02      0.04        45
          3       0.28      0.29      0.29        34

avg / total       0.25      0.28      0.22       114
```

In [73]:
```
points_plot(Xtr, Xtrain, Xtest, ytrain, ytest, gs)
```

```
SCORE
```

Out[73]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x113508ef0>
```