Machine Learning

Created by **LIU Jinchao (J.C.)** at **Dec 1, 2018**

0. Definition

Supervised Learning

- *Supervised learning* considers input-output pairs $(\mathbf{x}, y)$
  - learn a mapping from input to output.
  - *classification*: output $y \in \pm 1$
  - *regression*: output $y \in \mathbb{R}$
- "Supervised" here means that the algorithm is learning the mapping that we want.

Unsupervised Learning

- Unsupervised learning only considers the input data $\mathbf{x}$.
  - There are no output values.
- **Goal:** Try to discover inherent properties in the data.
  - Clustering
  - Dimensionality Reduction
  - Manifold Embedding

Manifold Embedding

- Project high-dimensional vectors into 2- or 3-dimensional space for visualization.
  - Points in the low-dim space have similar pair-wise distances as in the high-dim space.
- **For example:** visualize a collection of hand-written digits (images).

1. Python Introduction

1.1 Identifiers and Variables

- Identifiers
  - same as in C
- Naming convention:
  - `ClassName` -- a class name
  - `varName` -- other identifier
  - `_privateVar` -- private identifier
  - `__veryPrivate` -- strongly private identifier
  - `__special__` -- language-defined special name
- Variables
  - no declaration needed
  - no need for declaring data type (automatic type)
  - need to assign to initialize
    - use of uninitialized variable raises exception
  - automatic garbage collection (reference counts)

1.2 List

In [ ]:
```
a = [1,2,3]
a.append(4)      # add item to end
a.pop()          # remove last item and return it
a.insert(0,42)   # insert 42 at index 0
del a[2]         # delete item 2
a.reverse()      # reverse the entries
a.sort()         # sort the entries
```

In [ ]:
```
myList = [1, 2, 2, 2, 4, 5, 5]
myList4 = [4*item for item in myList]   # multiply each item by 4
myList4
```

In [ ]:
```
# can also use conditional to select items
[4*item*4 for item in myList if item>2]
```

1.3 Tuple

In [ ]:
```
x = (1,2,'three')
y = 4,5,6            # parentheses not needed!
z = (1,)   # tuple with 1 element (the trailing comma is required)
```

1.4 String

In [ ]:
```
"he" + "llo"    # concatenation
"hello"*3       # repetition
len("hello")    # length

"112211".count("11")          # 2
"this.com".endswith(".com")   # True
"wxyz".startswith("wx")       # True
"abc".find("c")               # finds first: 2
",".join(['a', 'b', 'c'])     # join list: 'a,b,c'
"aba".replace("a", "d")       # replace all: "dbd"
"a,b,c".split(',')            # make list: ['a', 'b', 'c']
"  abc    ".strip()           # "abc",  also rstrip(), lstrip()
```

In [ ]:
```
# String formatting: automatically fill in type
"{} and {} and {}".format('string', 123, 1.6789)      #  'string and 123
 and 1.6789'

"{:d} and {:f} and {:0.2f}".format(False, 3, 1.234)   #  '0 and 3.000000
 and 1.23'
```

1.5 Dictionary

In [ ]:
```
mydict = {'name': 'john', 42: 'sales', ('hello', 'world'): 6734}
print(mydict['name'])           # get value for key 'name'
mydict['name'] = 'jon'          # change value for key 'name'
mydict[2] = 5                   # insert a new key-value pair
del mydict[2]                   # delete entry for key 2
mydict.keys()                   # iterator of all keys (no random acces
s)
mydict.values()                 # iterator of all values
'name' in mydict                # check the presence of a key
```

1.6 Operators

- Arithmetic: `+`, `-`, `*`, `/`, `%`, `**` (exponent), `//` (floor division)
- Assignment: `=`, `+=`, `-=`, `/=`, `%=`, `**=`, `//=`
- Equality: `==`, `!=`
- Compare: `>`, `>=`, `<`, `<=`
- Logical: `and`, `or`, `not`
- Membership: `in`, `not in`
- Identity: `is`, `is not`
  - checks reference to the same object

1.7 Set

In [ ]:
```
a=[1, 2, 2, 2, 4, 5, 5]
sA = set(a)
sB = {4, 5, 6, 7}
print(sA - sB)     # set difference
print (sA | sB)    # set union
print (sA & sB)    # set intersect
```

1.8 Loop

In [ ]:
```
x = ['a', 'b', 'c']
for i,n in enumerate(x):
    print(i, n)
```

In [ ]:
```
# `zip` creates pairs of items between the two lists
x = ['a', 'b', 'c']
y = ['A', 'B', 'C']
for i,j in zip(x,y):
    print(i,j)
```

In [ ]:
```
x = {'a':1, 'b':2, 'c':3}
for (key,val) in x.items():
    print(key, val)
```

- loop control (same as C)
  - `break`, `continue`
- else clause
  - runs after list is exhausted
  - does *not* run if loop break

1.9 Function

In [ ]:
```
? sum3        # ipython magic -- shows a help window about the function
```

1.10 Class

- Defining a class
  - `self` is a reference to the object instance (passed *implicitly*)
- There are *no* "private" members
  - everything is accessible
  - convention to indicate *private*:
    - `_variable` means private method or variable (but still accessible)
  - convention for *very private*:
    - `__variable` is not directly visible
    - actually it is renamed to `_classname__variable`

In [ ]:
```
class MyList:
    "class documentation string"
    num = 0                    # a class variable
    def  __init__(self, b):   # constructor
        self.x = [b]           # an instance variable
        MyList.num += 1        # modify class variable
    def appendx(self, b):      # a class method
        self.x.append(b)       # modify an instance variable
        self.app = 1           # create new instance variable
```

In [ ]:
```
print(c.__dict__)    # Dictionary with the namespace.
print(c.__doc__)     # Class documentation string
print(c.__module__)  # Module which defines the class
print(MyList.__name__)    # Class name
print(MyList.__bases__)   # tuple of base classes
```

`Inheritence`

- Multiple inheritence
  - `class ChildClass(Parent1, Parent2, ...)`
  - calling method in parent
    - `super(ChildClass, self).method(args)`

In [ ]:
```
class MyListAll(MyList):
    def __init__(self, a):   # overrides MyList
        self.allx = [a]
        MyList.__init__(self, a)   # call base class constructor
    def popx(self):
        return self.x.pop()
    def appendx(self, a):          # overrides MyList
        self.allx.append(a)
        MyList.appendx(self, a)    # "super" method call
```

1.11 Saving Objects with Pickle

In [ ]:
```
try:
    file = open('blah.pickle', 'r')
    blah = pickle.load(file)
    file.close()
except:                # catch everything
    print("No file!")
else:                  # executes if no exception occurred
    print("No exception!")
finally:
    print("Bye!")      # always executes
```

1.12 pandas

- pandas is a Python library for data wrangling and analysis.
- `Dataframe` is a table of entries (like an Excel spreadsheet).
  - each column does not need to be the same type
  - operations to modify and operate on the table

1.13 DataFrame

In [ ]:
```
df['Name']

df[df.Age > 30]    # select Age greater than 30
df.mean()
```

1.14 NumPy

- Library for multidimensional arrays and 2D matrices
- `ndarray` class for multidimensional arrays
  - elements are all the same type
  - aliased to `array`

1.15 Array

- When operating on arrays, data is sometimes copied and sometimes not.
- *No copy is made for simple assignment.*
  - **Be careful!**

In [ ]:
```
a = array([1,2,3,4])
b = a               # simple assignment (no copy made!)
b is a              # yes, b references the same object
b[1] = -2           # changing b also changes a

c = a.view()    # create a view of a
c is a          # not the same object
                # False

c.base is a     # but the data is owned by a
                # True

d = a.copy()        # create a complete copy of a (new data is created)
d is a              # not the same object
                    # False
d.base is a         # not sharing the same data
                    # False
```

1.16 Matplotlib

In [ ]:
```
# setup matplotlib
%matplotlib inline
# setup output image format (Chrome works best)
import IPython.core.display
IPython.core.display.set_matplotlib_formats("svg") # file format
import matplotlib.pyplot as plt
```

In [ ]:
```
x = linspace(0,2*pi,16)
y = sin(x)
plt.plot(x, y, 'bo-')
plt.grid(True)
plt.ylabel('y label')
plt.xlabel('x label')
plt.title('my title')
plt.show()
```

- plot string specifies three things (e.g., `'bo-'`)
  - colors:
    - **b**lue, **r**ed, **g**reen, **m**agenta, **c**yan, **y**ellow, blac**k**, **w**hite
  - markers:
    - "." point
    - "o" circle
    - "v" triangle down
    - "^" triangle up
    - "<" triangle left
    - ">" triangle right
    - "8" octagon
    - "s" square
    - "p" pentagon
    - "*" star
    - "h" hexagon1
    - "+" plus
    - "x" x
    - "d" thin_diamond
  - line styles:
    - '-' solid line
    - '--' dashed line
    - '-.' dash-dotted line
    - ':' dotted lione

1.17 Split

- use `model_selection` module
  - `train_test_split` - give the percentage for training and testing.
  - `StratifiedShuffleSplit` - also preserves the percentage of examples for each class.

In [ ]:
```
trainX, testX, trainY, testY = \
  model_selection.train_test_split(X, Y,
     train_size=0.5, test_size=0.5, random_state=4487)
```

1.18 Text Document Representation (Bag of Words)

- `Bag-of-Words (BoW) model`
  - Let $\mathcal{V} = \{w_1, w_2, \cdots w_V\}$ be a list of V words (called a **vocabulary**).
  - represent a text document as a vector $\mathbf{x} \in \mathbb{R}^V$.
    - each entry x_j represents the number of times word w_j appears in the document.
- Example:
  - Document: "This is a test document"
  - Vocabulary: $\mathcal{V} = \{\text{"this"}, \text{"test"}, \text{"spam"}, \text{"foo"}\}$
  - Vector representation: $\mathbf{x} = [1, 1, 0, 0]$
- NOTE:
  - `the order of the words is not used!`
  - rearranging words leads to the same representation!
- Example:
  - "this is spam" $\rightarrow \mathbf{x} = [1, 0, 1, 0]$
  - "is this spam" $\rightarrow \mathbf{x} = [1, 0, 1, 0]$
- This is why it is called "bag-of-words"

In [ ]:
```
# setup the document vectorizer to make BoW
# - use english stop words
# - only use the most frequent 100 words in the dataset
cntvect = feature_extraction.text.CountVectorizer(stop_words='english',
max_features=100)

# create the vocabulary, and return the document vectors
# NOTE: we only use the training data!
trainX = cntvect.fit_transform(traintext)

# calculate vectors for the test data
testX  = cntvect.transform(testtext)
```

1.19 Term-Frequency Inverse Document Frequency (TF-IDF)

- some words are common among many documents
  - common words are less informative because they appear in both classes.
- `inverse document frequency (IDF)` - measure rarity of each word
  - $IDF(j) = \log \frac{N}{N_j}$
    - N is the number of documents.
    - N_j is the number of documents with word j.
  - IDF is:
    - 0 when a word is common to all documents
    - large value when the word appears in few documents
- `TF-IDF vector` : downscale words that are common in many documents
  - multiply TF and IDF terms
  - $x_j = \frac{w_j}{|D|} \log \frac{N}{N_j}$

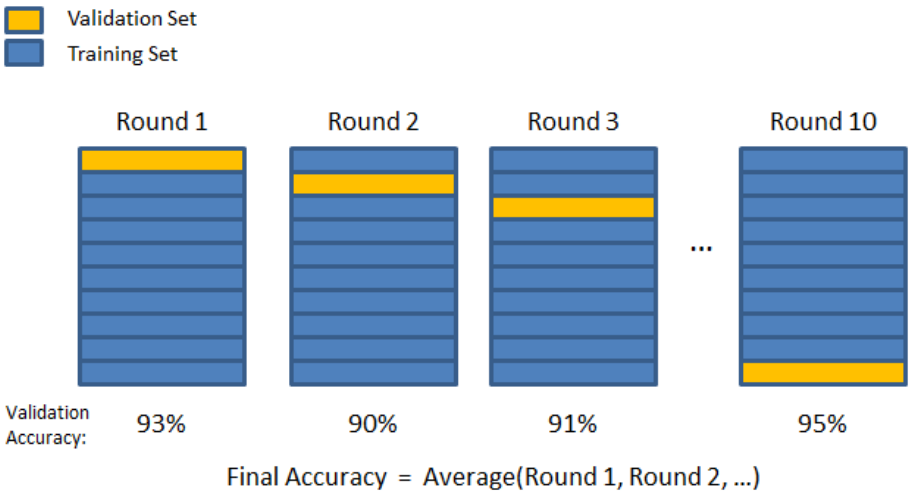
In [ ]:
```
# TF representation
tf_trans = feature_extraction.text.TfidfTransformer(norm='l1', use_idf=False)
Xtf = tf_trans.fit_transform(X)
```

In [ ]:
```
# TF-IDF representation
# (For TF, pass use_idf=False)
tf_trans = feature_extraction.text.TfidfTransformer(use_idf=True, norm='l1')
# 'l1' - entries sum to 1

# setup the TF-IDF representation, and transform the training set
trainXtf = tf_trans.fit_transform(trainX)

# transform the test set
testXtf = tf_trans.transform(testX)
```

- `Other text preprocessing`
  - *Stemming*
    - convert related words into a common root word
    - example: testing, tests --> "test"
    - see NLTK toolbox (http://www.nltk.org (http://www.nltk.org))
  - *Lemmatisation*
    - similar to stemming
    - groups inflections of word together (gone, going, went -> go)
    - see NLTK
  - Removing numbers and punctuation.
- **Other word models**
  - *N-grams*
    - similar to BoW except look at pairs of consecutive words (or N consecutive words in general)
  - *word vectors*
    - each word is a real vector, where direction indicates the "concept"
    - words about similar things point in the same direction
    - adding and subtracting word vectors yield new word vectors

1.20 Cross-validation

- Use *cross-validation* on the training set to select the best value of C.
- Run many experiments on the training set to see which parameters work on different versions of the data.
  - Split the data into batches of training and validation data.
  - Try a range of C values on each split.
  - Pick the value that works best over all splits.



- **Procedure**

  1. select a range of C values to try
  2. Repeat K times
     A. Split the training set into training data and validation data
     B. Learn a classifier for each value of C
     C. Record the accuracy on the validation data for each C
  3. Select the value of C that has the highest average accuracy over all K folds.
  4. Retrain the classifier using all data and the selected C.

- scikit-learn already has built-in `cross_validation` module (more later).
- for logistic regression, use *LogisticRegressionCV* class

1.21 Feature Pre-processing

- **Method 1:** scale each feature dimension so the mean is 0 and variance is 1.
  - $\tilde{x}_d = \frac{1}{s}(x_d - m)$
  - s is the standard deviation of feature values.
  - m is the mean of the feature values.
- **NOTE:** the parameters for scaling the features should be estimated from the training set!
  - same scaling is applied to the test set.

In [ ]:
```
# using the iris data
scaler = preprocessing.StandardScaler()  # make scaling object
trainXn = scaler.fit_transform(trainX)   # use training data to fit scal
ing parameters
testXn  = scaler.transform(testX)        # apply scaling to test data
```

- **Method 2:** scale features to a fixed range, -1 to 1.
  - $\tilde{x}_d = 2 * (x_d - min)/(max - min) - 1$
  - max and min are the maximum and minimum features values.

In [ ]:
```
# using the iris data
scaler = preprocessing.MinMaxScaler(feature_range=(-1,1))    # make scal
ing object
trainXn = scaler.fit_transform(trainX)   # use training data to fit scal
ing parameters
testXn  = scaler.transform(testX)        # apply scaling to test data
```

1.22 One-hot encoding

- encode a categorical variable as a vector of ones and zeros
  - if there are K categories, then the vector is K dimensions.
- Example:
  - x=cat → x=[1 0 0]
  - x=dog → x=[0 1 0]
  - x=horse → x=[0 0 1]

In [ ]:
```
# one-hot encoding example
X = [[0], [1], [0], [2], [2]]  # original categorical data {0,1,2}
ohe = preprocessing.OneHotEncoder(sparse=False)
ohe.fit(X)         # finds the number of categories in the training set:
 0-max(X)
ohe.transform(X)   # transform to one-hot-encoding
```

1.23 Binning

- encode a real value as a vector of ones and zeros
  - assign each feature value to a bin, and then use one-hot-encoding

In [ ]:
```
# example
X = [[-3], [0.5], [1.5]]  # the data
bins = [-2,-1,0,1,2]      # define the bins

# map from value to bin number
Xbins = digitize(X, bins=bins)

# map from bin number to 0-1 vector
ohe = preprocessing.OneHotEncoder(n_values=len(bins), sparse=False)
ohe.fit(Xbins)
ohe.transform(Xbins)
```

1.24 Data transformations - polynomials

- Represent interactions between features using polynomials
- Example:
  - 2nd-degree polynomial models pair-wise interactions
    - $[x_1, x_2] \rightarrow [x_1^2, x_1 x_2, x_2^2]$
  - Combine with other degrees:
    - $[x_1, x_2] \rightarrow [1, x_1, x_2, x_1^2, x_1 x_2, x_2^2]$

In [ ]:
```
X = [[0,1], [1,2], [3,4]]
pf = preprocessing.PolynomialFeatures(degree=2)
pf.fit(X)
pf.transform(X)
```

2. Classifier

General Classification Problem

- Observation $\mathbf{x}$ (i.e., features)
  - typically a real vector, $\mathbf{x} \in \mathbb{R}^d$.
  - **Example**: a 2-dim vector containing the petal length and sepal width.
    - $\mathbf{x} = \begin{bmatrix} \text{petal length} \\ \text{sepal width} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
- Class y
  - takes values from a set of possible class labels $\mathcal{Y}$.
  - **Example:** $\mathcal{Y} = \{\text{"versicolor", "virginica"}\}$.
    - or equivalently as numbers, $\mathcal{Y} = \{1, 2\}$.
- **Goal**: given an observed features $\mathbf{x}$, predict its class y.

2.1 Linear Classfier

2.1.1 Bayes model

In [ ]:
```
from numpy import *
from sklearn import *
from scipy import stats
```

Probabilistic model

- Model *how* the data is generated using probability distributions.
  - called a **generative model**.
- Generative model
  - 1) The world has objects of various classes.
  - 2) The observer measures features/observations from the objects.
  - 3) Each class of objects has a particular distribution of features.

- in the world, the frequency that class y occurs is given by the probability distribution $p(y)$.
  - $p(y)$ is called the **prior distribution**.

- the observation is drawn according to the distribution $p(\mathbf{x}|y)$.
  - $p(x|y)$ is called the **class conditional distribution**
    - "probability of observing a particular feature vector $\mathbf{x}$ given the object is class y"
    - can "smooth out the samples" or "fill-in" values between samples.

Gaussian distribution (normal distribution)

- Each class is modeled as a separate Gaussian distribution of the feature value
  - $p(x|y=c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} e^{-\frac{1}{2\sigma_c^2}(x-\mu_c)^2}$
  - Each class has its own mean and variance parameters (μ_c, σ_c^2).

Maximum likelihood estimation (MLE)

- set the parameters (μ, σ^2) to maximize the likelihood (probability) of the samples for that class.
- Let $\{\mathbf{x}_i, y_i\}_{i=1}^N$ be the data for one class:

$$(\hat{\mu}, \hat{\sigma}^2) = \underset{\mu,\sigma^2}{\operatorname{argmax}} \sum_{i=1}^{N} \log p(\mathbf{x}_i|y_i)$$

- Solution:
  - sample mean: $\hat{\mu} = \frac{1}{N}\sum_{i=1}^N \mathbf{x}_i$
  - sample variance: $\hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^N (\mathbf{x}_i - \hat{\mu})^2$

Bayesian Decision Rule

- The Bayesian decision rule (BDR) makes the optimal decisions on problems involving probability (uncertainty).
  - minimizes the *probability of making a prediction error*.
- **Bayes Classifier**
  - Given observation x, pick the class c with the *largest posterior probability*, $p(y=c|x)$.
  - **Example:**
    - if $p(y=1|x) > p(y=2|x)$, then choose Class 1
    - if $p(y=1|x) < p(y=2|x)$, then choose Class 2

Bayes' Rule

- The posterior probability can be calculated using Bayes' rule:
$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$
  - The denominator is the probability of x:
    - $p(x) = \sum_{y \in \mathcal{Y}} p(x|y)p(y)$
  - The denominator makes $p(y|x)$ sum to 1.

- Bayes' rule:
$$p(y|x) = \frac{p(x|y)p(y)}{p(x|y=1)p(y=1) + p(x|y=2)p(y=2)}$$

- **Example:**
  - BDR using joint likelihoods:
    - if $p(x|y=1)p(y=1) > p(x|y=2)p(y=2)$, then choose Class 1
    - otherwise, choose Class 2

- Can also apply a monotonic increasing function (like $\log$) and do the comparison.
  - Using log likelihoods:
    - $\log p(x|y=1) + \log p(y=1) > \log p(x|y=2) + \log p(y=2)$
  - This is more numerically stable when the likelihoods are small.

- **Training:**

  1. Collect training data from each class.
  2. For each class c, estimate the class conditional densities $p(x|y=c)$:
     A. select a form of the distribution (e.g. Gaussian).
     B. estimate its parameters with MLE.
  3. Estimate the class priors $p(y)$ using MLE.
- **Classification:**

  1. Given a new sample x^*, calculate the likelihood $p(x^*|y=c)$ for each class c.
  2. Pick the class c with largest posterior probability $p(y=c|x)$.
     - (equivalently, use $p(x|y=c)p(y=c)$ or $\log p(x|y=c) + \log p(y=c)$)

2.1.1.1 Gaussian NB model

In [ ]:
```
# get the NB Gaussian model from sklearn
model = naive_bayes.GaussianNB()

# fit the model to training data
model.fit(trainX, trainY)
```

In [ ]:
```
# predict from the model
predY = model.predict(testX)
# calculate accuracy
acc = metrics.accuracy_score(testY, predY)
```

Naive Bayes model for Boolean vectors

- Model each word independently
  - absence/presence of a word w_j in document
  - Bernoulli distribution
    - present: $p(x_j = 1|y) = \pi_j$
    - absent: $p(x_j = 0|y) = 1 - \pi_j$
  - MLE parameters: $\pi_j = N_j/N$,
    - N_j is the number of documents in class y that contain word j.
    - N is the number of documents in class y.

- Class-conditional distribution

$$p(x_1, \cdots, x_V|y = \text{spam}) = \prod_{j=1}^{V} p(x_j|y = \text{spam})$$

$$\log p(x_1, \cdots, x_V|y = \text{spam}) = \sum_{j=1}^{V} \log p(x_j|y = \text{spam})$$

- for a document, the log-probabilities of the words being in a spam message adds.
  - accumulate evidence over all words in the document.
  - more words that are associated with spam --> more likely the document is spam

2.1.1.2 Bernoulli NB model

In [ ]:
```
# fit the NB Bernoulli model.
# the model automatically converts count vector into binary vector
bmodel = naive_bayes   .BernoulliNB(alpha=0.0)
bmodel.fit(trainX, trainY)
```

In [ ]:
```
# prediction
predY = bmodel.predict(testX)
# calculate accuracy
acc = metrics.accuracy_score(testY, predY)
```

Smoothing

- Some words are not present in any documents for a given class.
  - $N_j = 0$, and thus $\pi_j = 0$.
    - i.e., the document in the class **definitely** will not contain the word.
    - can be a problem since we simply may not have seem an example with that word.
- Smoothed MLE
  - add a smoothing parameter α that adds a "virtual" count
  - parameter: $\pi_j = (N_j + \alpha)/(N + 2\alpha)$,
  - this is called *Laplace smoothing*
- In general, *regularizing* or *smoothing* of the estimate helps to prevent *overfitting* of the parameters.

Summary

- **Generative classification model**
  - estimate probability distributions of features generated from each class.
  - given feature observation predict class with largest posterior probability.
- **Advantages:**
  - works with small amount of data.
  - works with multiple classes.
- **Disadvantages:**
  - accuracy depends on selecting an appropriate probability distribution.
    - if the probability distribution doesn't model the data well, then accuracy might be bad.

- Terminology
  - **"Discriminative"** - learn to directly discriminate the classes apart using the features.
  - **"Generative"** - learn model of how the features are generated from different classes.

2.1.2 Logistic regression

- Use a probabilistic approach
- Need to map the function values $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$ to probability values between 0 and 1.
  - *sigmoid* function maps from real number to interval [0,1]
  - $\sigma(z) = \frac{1}{1+e^{-z}}$

- Given a feature vector x, the probability of a class is:
  - $p(y = +1|\mathbf{x}) = \sigma(f(\mathbf{x}))$
  - $p(y = -1|\mathbf{x}) = 1 - \sigma(f(\mathbf{x}))$
- Note: here we are directly modeling the class posterior probability!
  - not the class-conditional $p(\mathbf{x}|y)$

`Learning the parameters`

- Given training data $\{\mathbf{x}_i, y_i\}_{i=1}^N$, learn the function parameters $(\mathbf{w}, b)$ using maximum likelihood estimation.
- maximize the likelihood of the data $\{\mathbf{x}_i, y_i\}$:

$$(\mathbf{w}^*, b^*) = \underset{\mathbf{w},b}{\operatorname{argmax}} \sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i)$$

- to prevent *overfitting*, add a prior distribution on $\mathbf{w}$.
  - assume Gaussian distribution on $\mathbf{w}$ with variance $1/C$

$$(\mathbf{w}^*, b^*) = \underset{\mathbf{w},b}{\operatorname{argmax}} \log p(\mathbf{w}) + \sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i)$$

- Equivalently,

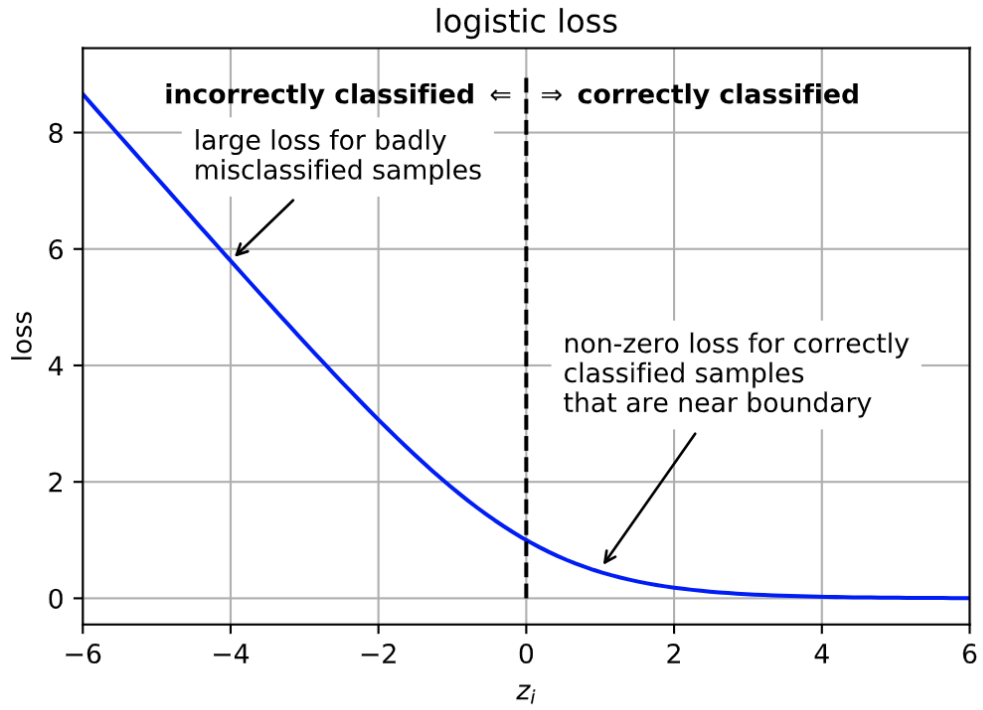
$$(\mathbf{w}^*, b^*) = \underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{C}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{N} \log(1 + \exp(-y_i(\mathbf{w}^T\mathbf{x}_i + b)))$$

- the first term is the *regularization term*
  - Note: $\mathbf{w}^T\mathbf{w} = \sum_{j=1}^{d} w_j^2$
  - penalty term that keeps entries in $\mathbf{w}$ from getting too large.
  - C is the regularization *hyperparameter*
    - larger C value allow large values in $\mathbf{w}$.
    - smaller C value discourage large values in $\mathbf{w}$.

$$(\mathbf{w}^*, b^*) = \underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{C}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{N} \log(1 + \exp(-y_i(\mathbf{w}^T\mathbf{x}_i + b)))$$

- the second term is the *data-fit term*
  - wants to make the parameters $(\mathbf{w}, b)$ to well fit the data.
  - Define $z_i = y_i f(\mathbf{x}_i)$
    - Interesing observation:
      - $z_i > 0$ when sample $\mathbf{x}_i$ is classified correctly
      - $z_i < 0$ when sample $\mathbf{x}_i$ is classified incorrectly
      - $z_i = 0$ when sample is on classifier boundary
  - logistic loss function: $L(z_i) = \log(1 + \exp(-z_i))$

- **no closed-form solution**
  - use an iterative optimization algorithm to find the optimal solution
  - e.g. *gradient descent* - step downhill in each iteration.
    - $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{dE}{d\mathbf{w}}$
    - where E is the objective function
    - η is the *learning rate* (how far to step in each iteration).



In [ ]:
```
# learn logistic regression classifier
# (C is a regularization hyperparameter)
logreg = linear_model.LogisticRegression(C=100)
logreg.fit(trainX, trainY)
```

In [ ]:
```
# predict from the model
predY = logreg.predict(testX)

# calculate accuracy
acc = metrics.accuracy_score(testY, predY)
```

cross-validation

In [ ]:
```
# learn logistic regression classifier usinc CV
#  Cs is an array of possible C values
#  cv is the number of folds
#  n_jobs is the number of parallel jobs to run (makes it faster)
#    -1 means use all cores
logreg = linear_model.LogisticRegressionCV(Cs=logspace(-4,4,20), cv=5, n
_jobs=-1)
logreg.fit(trainX, trainY)
```

Multi-class classification

- For more than 2 classes, split the problem up into several binary classifier problems.
  - **1-vs-rest**
    - *Training:* for each class, train a classifier for that class versus the other classes.
      - For example, if there are 3 classes, then train 3 binary classifiers: 1 vs {2,3}; 2 vs {1,3}; 3 vs {1,2}
    - *Prediction:* calculate probability for each binary classifier. Select the class with highest probability.

In [ ]:
```
# learn logistic regression classifier (one-vs-all)
mlogreg = linear_model.LogisticRegression(C=10) # Same
mlogreg.fit(trainX, trainY)
```

Multiclass logistic regression

- Another way to get a multi-class classifier is to define a multi-class objective.
  - One weight vector $\mathbf{w}_c$ for each class c.
- Define probabilities with **softmax** function
  - analogous to sigmoid function for binary logistic regression.
  - $p(y = c|\mathbf{x}) = \frac{\exp(\mathbf{w}_c^T\mathbf{x})}{\exp(\mathbf{w}_1^T\mathbf{x})+\cdots+\exp(\mathbf{w}_K^T\mathbf{x})}$
  - The class with largest reponse of $\mathbf{w}_c^T\mathbf{x}$ will have the highest probability.
- Estimate the $\{\mathbf{w}_j\}$ parameters using MLE as before.

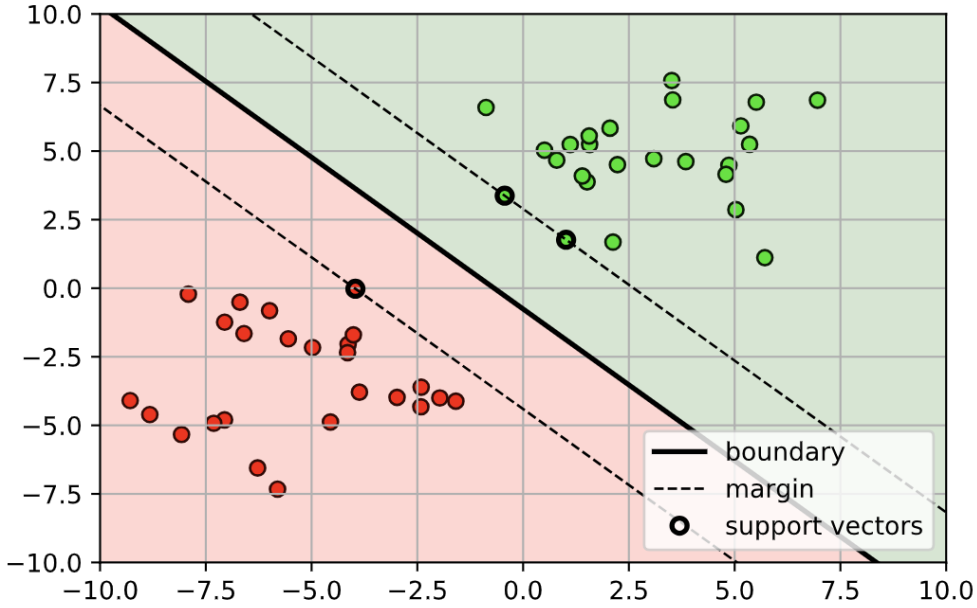
In [ ]:
```
# learn logistic regression classifier
mlogreg = linear_model.LogisticRegression(C=10,
            multi_class='multinomial', solver='lbfgs')
            # use multi-class and corresponding solver
mlogreg.fit(trainX, trainY)
```

2.1.3 Support vector machine

Maximum margin

- Define the space between the separating line and the closest point as the *margin*.
  - think of this space as the "amount of wiggle room" for accomodating errors in estimating $\mathbf{w}$.

- **Idea:** the best separating line is the one that *maximizes the margin*.
  - i.e., puts the most distance between the closest points and the decision boundary.



SVM Training

- given a training set $\{\mathbf{x}_i, y_i\}_{i=1}^N$, optimize:

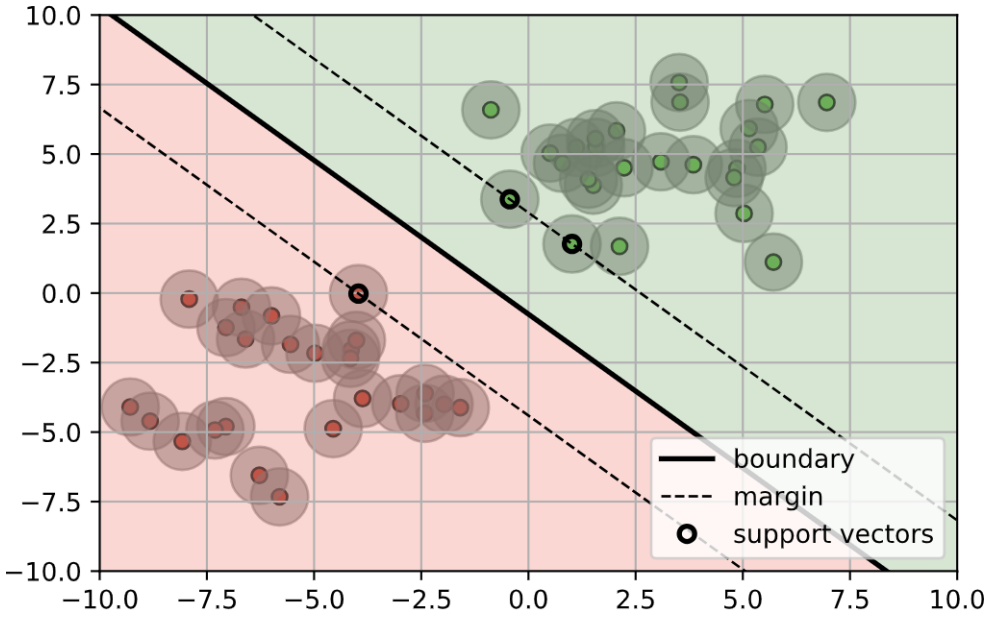
$$\underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

$$\text{s.t. } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad 1 \leq i \leq N$$

  - the objective minimizes the inverse of the margin distance, i.e., maximizes the margin.
  - the inequality constraints ensure that all points are either on or outside of the margin.
    - the margin is set to be distance of 1 from the boundary.

SVM Prediction

- given a new data point $\mathbf{x}_*$, use sign of linear function to predict class
  - $y_* = \text{sign}(\mathbf{w}^T\mathbf{x}_* + b)$ #### Why is maximizing the margin good?
- the true $\mathbf{w}$ is uncertain
  - maximizing the margin allows the most uncertainty (wiggle room) for $\mathbf{w}$, while keeping all the points correctly classified.
- the data points are uncertain
  - maximizing the margin allows the most wiggle of the points, while keeping all the points correctly classified.



What about non-separable data?

- use the same linear classifier
  - allow some training samples to violate margin
    - i.e., are inside the margin (or even mis-classified)
  - Define "slack" variable $\xi_i \geq 0$
    - $\xi_i = 0$ means sample is outside of margin area (no slack)
    - $\xi_i > 0$ means sample is inside of margin area (slack)

- introduce a parameter C which is the penalty for each training sample that violates the margin.
  - smaller value means allow more violations (less penalty)
  - larger value means don't allow violations (more penalty)

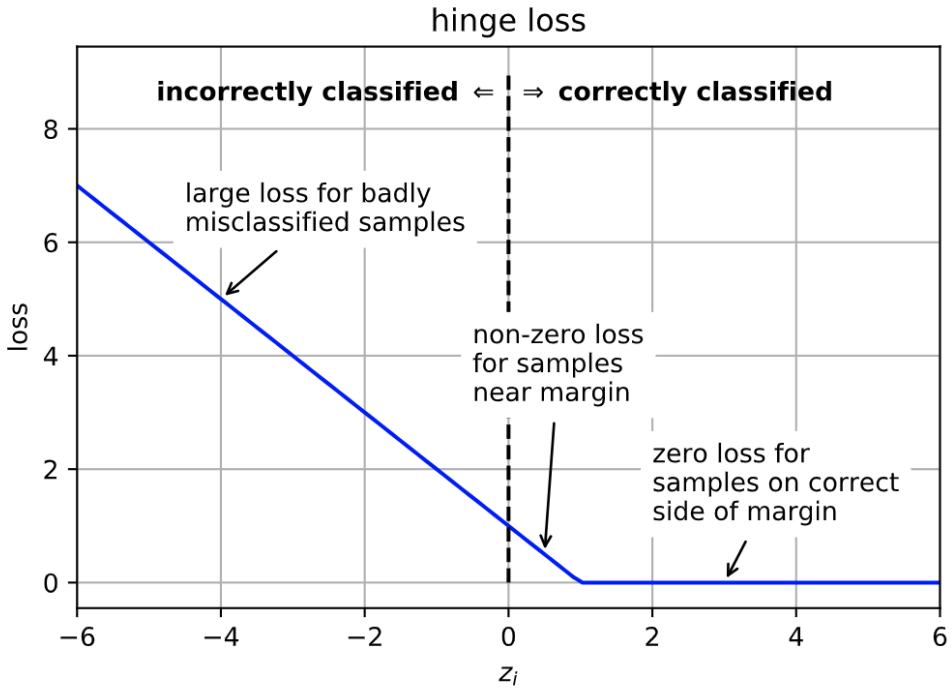
$$\begin{aligned} &\underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{N}\xi_i \\ &\text{s. t. } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \quad 1 \leq i \leq N \\ &\xi_i \geq 0 \end{aligned}$$

Loss function

- After some massaging, the objective function is:

$$\underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{C}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{N} \max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b))$$

  - hinge loss function: $L(z_i) = \max(0, 1 - z_i)$
    - Note: $\max(a, b)$ returns whichever value (a or b) is largest.



- SVM doesn't have it's own dedicated cross-validation function
- Use the `GridSearchCV` to run cross-validation for a list of parameters
  - calculate average accuracy for each parameter
  - select parameter with highest accuracy, retrain model with all data
  - Speed up: each parameter can be trained/tested separately, specify number of parallel jobs using `n_jobs`

In [ ]:
```
# setup the list of parameters to try
paramgrid = {'C': logspace(-3,3,13)}

# setup the cross-validation object
# pass the SVM object, parameter grid, and number of CV folds
# set number of parallel jobs to -1 (use all cores)
svmcv = model_selection.GridSearchCV(svm.SVC(kernel='linear'), paramgrid
, cv=5,
                                     n_jobs=-1, verbose=True)

# run cross-validation (train for each split)
svmcv.fit(trainX, trainY);
```

In [ ]:
```
# view best results and best retrained estimator
svmcv.best_params_
svmcv.best_score_
svmcv.best_estimator_
```

In [ ]:
```
# Directly use svmcv to make predictions
predY = svmcv.predict(testX)

acc = metrics.accuracy_score(testY, predY)
```

Multi-class SVM

- In sklearn, `svm.SVC` implements "1-vs-1" multi-class classification.
  - Train binary classifiers on all pairs of classes.
    - 3-class Example: 1 vs 2, 1 vs 3, 2 vs 3
  - To label a sample, pick the class with the most votes among the binary classifiers.
- Problem:
  - 1v1 classification is very slow when there are a large number of classes.
    - if there are C classes, need to train $C(C-1)/2$ binary classifiers!

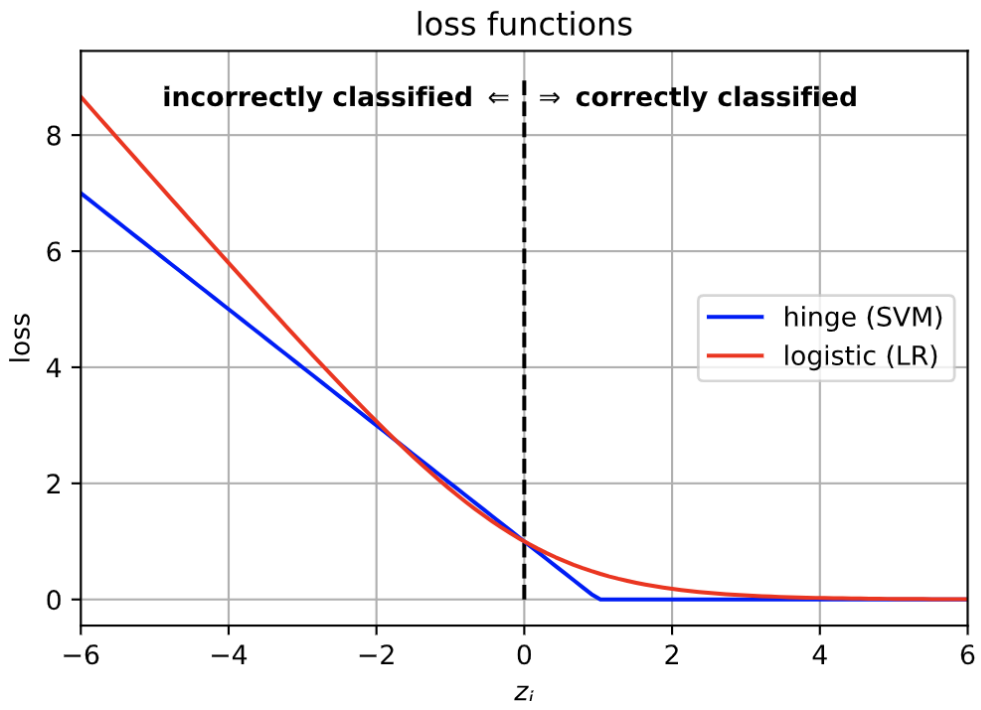
1-vs-all SVM

- Use the `multiclass.OneVsRestClassifier` to build a 1-vs-all classifier from any binary classifier.
  - For `GridSearchCV`, use `'estimator__C'` as the parameter label for `C` in the SVM.

In [ ]:
```
msvm = multiclass.OneVsRestClassifier(svm.SVC(kernel='linear'))

# setup the parameters and run CV
paramgrid = {'estimator__C': logspace(-3,3,13)}
msvmcv = model_selection.GridSearchCV(msvm, paramgrid, cv=5, n_jobs=-1,
verbose=True)
msvmcv.fit(trainX, trainY)
```

Summary for LR & SVM

- **Linear classifiers:**
  - separate the data using a linear surface (hyperplane).
  - $y = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$
- **Two formulations:**
  - logistic regression - maximize the probability of the data
  - support vector machine - maximize the margin of the hyperplane
- **Loss functions**
  - SVM - ensure a margin of 1 between boundary and closest point
  - LR - push the classification boundary as far as possible from all points

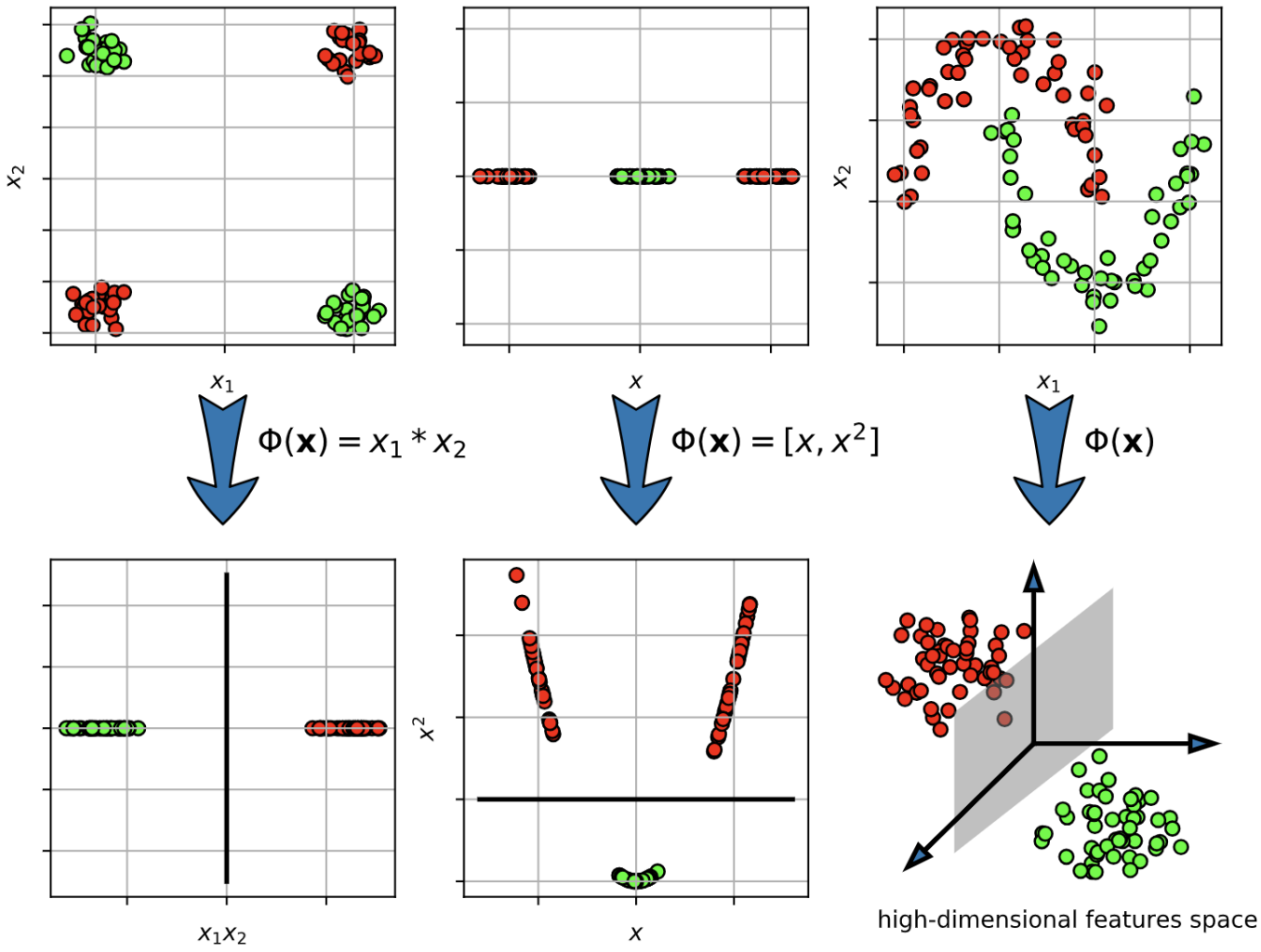


- **Advantages:**
  - SVM works well on high-dimensional features (d large), and has low generalization error.
  - LR has well-calibrated probabilities.
- **Disadvantages:**
  - decision surface can only be linear!
    - Next lecture we will see how to deal with non-linear decision surfaces.

2.2 Non-Linear Classifiers

Idea - transform the input space

- map from input space $\mathbf{x} \in \mathbb{R}^d$ to a new high-dimensional space $\mathbf{z} \in \mathbb{R}^D$.
  - $\mathbf{z} = \Phi(\mathbf{x})$, where $\Phi(\mathbf{x})$ is the transformation function.
- learn the linear classifier in the new space
  - if dimension of new space is large enough ($D > d$), then the data should be linearly separable



SVM with transformed input

- Given a training set $\{\mathbf{x}_i, y_i\}_{i=1}^N$, the original SVM training is:

$$\underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{2}\mathbf{w}^T\mathbf{w} \quad \text{s.t. } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad 1 \leq i \leq N$$

- Apply high-dimensional transform to input $\mathbf{x} \rightarrow \Phi(\mathbf{x})$:

$$\underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{2}\mathbf{w}^T\mathbf{w} \quad \text{s.t. } y_i(\mathbf{w}^T\Phi(\mathbf{x}_i) + b) \geq 1, \quad 1 \leq i \leq N$$

SVM Dual Problem

- Using some convex optimization theory, the SVM problem can be rewritten as a *dual* problem:

$$\underset{\alpha}{\operatorname{argmax}} \sum_i \alpha_i - \frac{1}{2}\sum_{i=1}^N\sum_{j=1}^N \alpha_i\alpha_j y_i y_j \Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$$

$$\text{s.t. } \sum_{i=1}^N \alpha_i y_i = 0, \quad \alpha_i \geq 0$$

- The new variable α_i corresponds to each training sample $(\mathbf{x}_i, y_i)$.
- Recover the hyperplane w as a sum of the margin points:
  - $\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \Phi(\mathbf{x}_i)$
- Classify a new point $\mathbf{x}_*$,

$$y_* = \operatorname{sign}(\mathbf{w}^T\Phi(\mathbf{x}_*) + b)$$

$$= \operatorname{sign}(\sum_{i=1}^N \alpha_i y_i \Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_*) + b)$$

- Interpretation of α_i
  - $\alpha_i = 0$ when the sample $\mathbf{x}_i$ is not on the margin.
  - $\alpha_i > 0$ when the sample $\mathbf{x}_i$ is on the margin (or violates).
    - i.e., the sample $\mathbf{x}_i$ is a support vector.

Kernel function

- the SVM dual problem is completely written in terms of *inner product* between the high-dimensional feature vectors: $\Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$
- So rather than explicitly calculate the high-dimensional vector $\Phi(\mathbf{x}_i)$,
  - we only need to calculate the inner product between two high-dim feature vectors.
- We call this a **kernel function**
  - $k(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$
  - calculating the kernel will be less expensive than explicitly calculating the high-dimensional feature vector and the inner product.

Example: Polynomial kernel

- input vector $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \in \mathbb{R}^d$
- kernel between two vectors is a p-th order polynomial:
  - $k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T\mathbf{x}')^p = (\sum_{i=1}^d x_i x_i')^p$
- For example, $p = 2$,

$$k(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T\mathbf{x}')^2 = (\sum_{i=1}^d x_i x_i')^2$$

$$= \sum_{i=1}^d \sum_{j=1}^d (x_i x_i' x_j x_j') = \Phi(\mathbf{x})^T \Phi(\mathbf{x}')$$

  - transformed feature space is the quadratic terms of the input vector:

$$\Phi(\mathbf{x}) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ \vdots \\ x_2 x_1 \\ x_2 x_2 \\ \vdots \\ x_d x_1 \\ \vdots \\ x_d x_d \end{bmatrix}$$

- Comparison of number of multiplications
  - for kernel: $O(d)$
  - explicit transformation Φ: $O(d^2)$

Kernel trick

- Replacing the inner product with a kernel function in the optimization problem is called the **kernel trick**.
  - *turns a linear classification algorithm into a non-linear classification algorithm*.
  - the shape of the decision boundary is determined by the kernel.

2.2.1 Kernel SVM

- Replace inner product in linear SVM with kernel function:

$$\underset{\alpha}{\text{argmax}} \sum_i \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$$

$$\text{s.t.} \sum_{i=1}^N \alpha_i y_i = 0, \quad \alpha_i \geq 0$$

- Prediction
  - $y_* = \text{sign}(\sum_{i=1}^N \alpha_i y_i k(\mathbf{x}_i, \mathbf{x}_*) + b)$

RBF kernel

- RBF kernel (radial basis function)
  - $k(\mathbf{x}, \mathbf{x}') = e^{-\gamma \|\mathbf{x}-\mathbf{x}'\|^2}$
  - similar to a Gaussian
- gamma $\gamma > 0$ is the inverse bandwidth parameter of the kernel
  - controls the smoothness of the function
  - small $\gamma \rightarrow$ wider Gaussian $\rightarrow$ smooth functions
  - large $\gamma \rightarrow$ thin Gaussian $\rightarrow$ wiggly function

In [ ]:
```
# setup the list of parameters to try
paramgrid = {'C': logspace(-2,3,20),
             'gamma': logspace(-4,3,20) }

# setup the cross-validation object
# pass the SVM object w/ rbf kernel, parameter grid, and number of CV fo
lds
svmcv = model_selection.GridSearchCV(svm.SVC(kernel='rbf'), paramgrid, c
v=5,
                                     n_jobs=-1, verbose=True)

# run cross-validation (train for each split)
svmcv.fit(trainX, trainY);
```

In [ ]:
```
# setup the list of parameters to try
paramgrid = {'C': logspace(-2,3,20),
             'degree': [2,3,4] }

# setup the cross-validation object
# pass the SVM object w/ rbf kernel, parameter grid, and number of CV fo
lds
svmcv = model_selection.GridSearchCV(svm.SVC(kernel='poly'), paramgrid,
cv=5,
                                     n_jobs=-1, verbose=True)

# run cross-validation (train for each split)
svmcv.fit(trainX, trainY);
```

- **Advantages:**
  - non-linear decision boundary for more complex classification problems
  - some intuition from the type of kernel function used.
  - kernel function can also be used to do non-vector data.
- **Disadvantages:**
  - sensitive to the kernel function used.
  - sensitive to the C and kernel hyperparameters.
  - computationally expensive to do cross-validation.
  - need to calculate the kernel matrix
    - N^2 terms where N is the size of the training set
    - for large N, uses a large amount of memory and computation.

Kernels on other types of data

- **Histograms:** $\mathbf{x} = [x_1, \cdots, x_d]$, x_i is a bin value.
  - Bhattacharyya:
$$k(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{d} \sqrt{x_i x_i'}$$
  - histogram intersection:
$$k(\mathbf{x}, \mathbf{x}') = \sum_{i} \min(x_i, x_i')$$
  - χ^2-RBF:
$$k(\mathbf{x}, \mathbf{x}') = e^{-\gamma \chi^2(\mathbf{x}, \mathbf{x}')}$$
    - γ is a inverse bandwidth parameter
    - χ^2 distance: $\chi^2(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{d} \frac{(x_i - x_i')^2}{\frac{1}{2}(x_i + x_i')}$

- **Strings**: $\mathbf{x}$ = "...." (strings can be different sizes)

$$k(\mathbf{x}, \mathbf{x}') = \sum_s w_s \phi_s(\mathbf{x}) \phi_s(\mathbf{x}')$$

  - $\phi_s(\mathbf{x})$ is the number of times substring s appears in $\mathbf{x}$.
  - $w_s > 0$ is a weight.
- **Sets**: $\mathbf{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$ (sets can be different sizes)
  - intersection kernel:

$$k(\mathbf{X}, \mathbf{X}') = 2^{|\mathbf{X} \cap \mathbf{X}'|}$$

    - $|\mathbf{X} \cap \mathbf{X}'|$ = number of common elements.

- Ensemble methods aim to combine multiple classifiers together to form a better classifier.
- Examples:
  - **boosting** - training multiple classifiers, each focusing on errors made by previous classifiers.
  - **bagging** - training multiple classifiers from random selection of training data

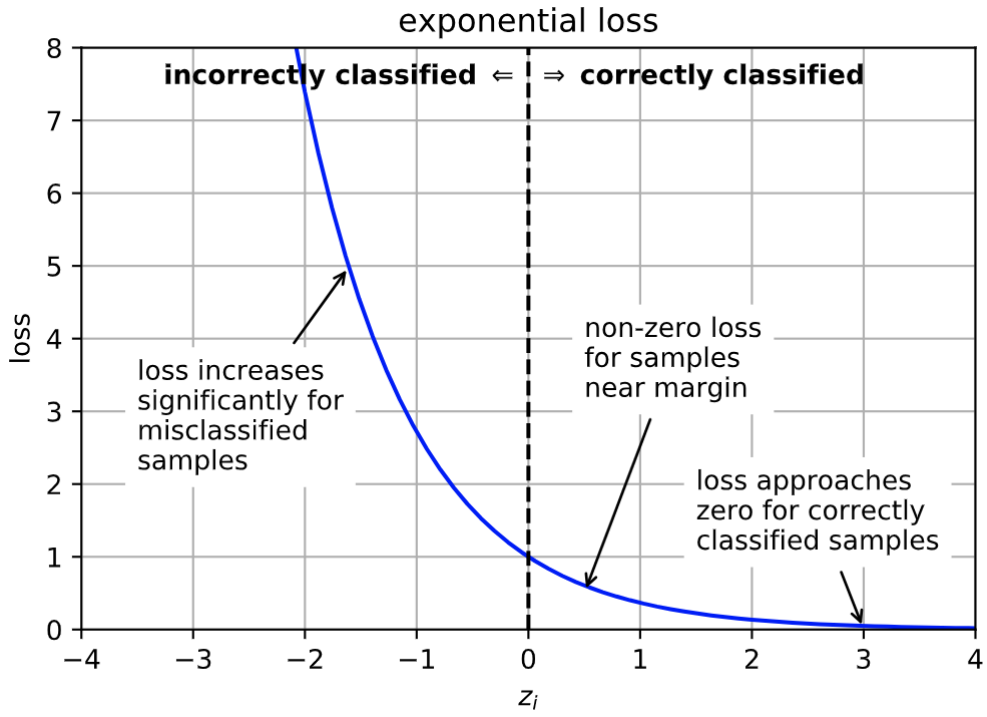
2.2.2 AdaBoost - Adaptive Boosting

- Base classifier is a "weak learner"
  - A simple classifier that can be slightly better than random chance (>50%)
  - Example: *decision stump classifier*
    - check if feature value is above (or below) a threshold.
    - $y = f(x) = \begin{cases} +1, & x_j \geq T \\ -1, & x_j < T \end{cases}$

- **Idea:** train weak classifiers sequentially
- In each iteration,
  - Pick a weak learner $f_t(\mathbf{x})$ that best carves out the input space.
  - The weak learner should focus on data that is misclassified.
    - Apply weights to each sample in the training data.
    - Higher weights give more priority to difficult samples.

Adaboost loss function

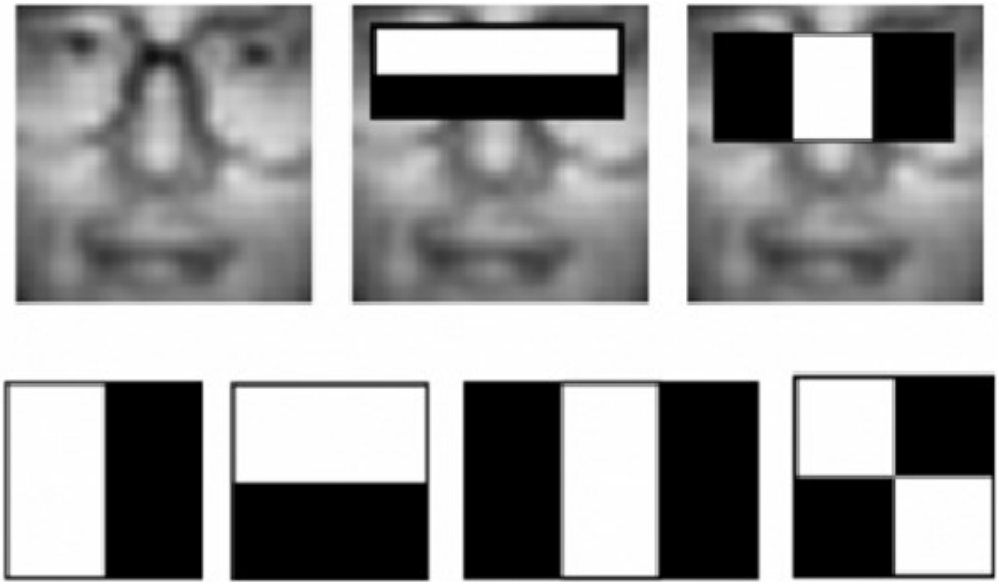
- exponential loss
  - $L(z_i) = e^{-z_i}$
    - $z_i = y_i f(\mathbf{x}_i)$
  - very sensitive to misclassified outliers.



In [ ]:
```
# setup the list of parameters to try
paramgrid = {'n_estimators': array([1, 2, 3, 5, 10, 15, 20, 25, 50, 100,
 200, 500, 1000]) }

# setup the cross-validation object
# (NOTE: using parallelization in GridSearchCV, not in AdaBoost)
adacv = model_selection.GridSearchCV(ensemble.AdaBoostClassifier(random_
state=4487),
                                     paramgrid, cv=5, n_jobs=-1)

# run cross-validation (train for each split)
adacv.fit(trainX, trainY);
```

In [ ]:
```
# predict from the model
predY = adacv.predict(testX)

# calculate accuracy
acc      = metrics.accuracy_score(testY, predY)
```

- Boosting can do feature selection
  - each decision stump classifier looks at one feature
- One of the original face detection methods (Viola-Jones) used Boosting.
  - extract a lot of image features from the face
  - during training, Boosting learns which ones are the most useful.



Summary

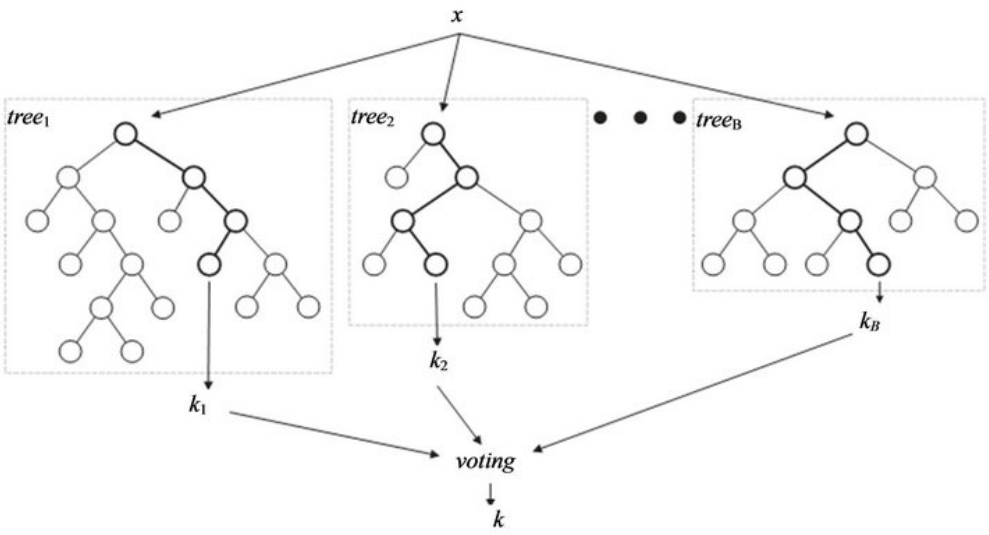
- **Ensemble Classifier:**
  - Combine the outputs of many "weak" classifiers to make a "strong" classifier
- **Training:**
  - In each iteration,
    - training data is re-weighted based on whether it is correctly classified or not.
    - weak classifier focuses on misclassified data from previous iterations.
  - Use cross-validation to pick number of weak learners.
- **Advantages:**
  - Good generalization performance
  - Built-in features selection - decision stump selects one feature at a time.
- **Disadvantages:**
  - Sensitive to outliers.

Decision Tree

- Simple "Rule-based" classifier
  - At each node, move down the tree based on that node's criteria.
  - leaf node contains the prediction
- **Advantage:** can create complex conjunction of rules
- **Disadvantage:** easy to overfit by itself
  - can fix with bagging!

2.2.3 Random Forest Classifier

- Use **bagging** to make an ensemble of Decision Tree Classifiers
  - for each *Decision Tree Classifier*
    - create a new training set by randomly sampling from the training set
    - for each split in a tree, select a random subset of features to use
- for a test sample, the prediction is aggregated over all trees.



In [ ]:
```
# learn a RF classifier
# use 4 trees
clf = ensemble.RandomForestClassifier(n_estimators=4, random_state=4487,
 n_jobs=-1)
clf.fit(X3, Y3)
```

In [ ]:
```
clfs = {}
for i,n in enumerate([5, 10, 50, 100, 1000, 10000]):
    clfs[n] = ensemble.RandomForestClassifier(n_estimators=n, random_sta
te=4487, n_jobs=-1)
    clfs[n].fit(X3, Y3)
```

In [ ]:
```
# predict from the model
predY = clfs[1000].predict(testX)

# calculate accuracy
acc = metrics.accuracy_score(testY, predY)
```

- Important parameters for cross-validation
  - `max_features` - maximum number of features used for each split
  - `max_depth` - maximum depth of a decision tree

Unbalanced Data

- For some classification tasks that data will be unbalanced
  - many more examples in one class than the other.
- **Example:** detecting credit card fraud
  - credit card fraud is rare
    - 50 examples of fraud, 5000 examples of legitimate transactions.
- **Solution:** apply weights on the classes during training.
  - weights are inversely proportional to the class size.

In [ ]:
```
clfw = svm.SVC(kernel='linear', C=10,  class_weight='balanced')
clfw.fit(X, Y)
```

Classifier Imbalance

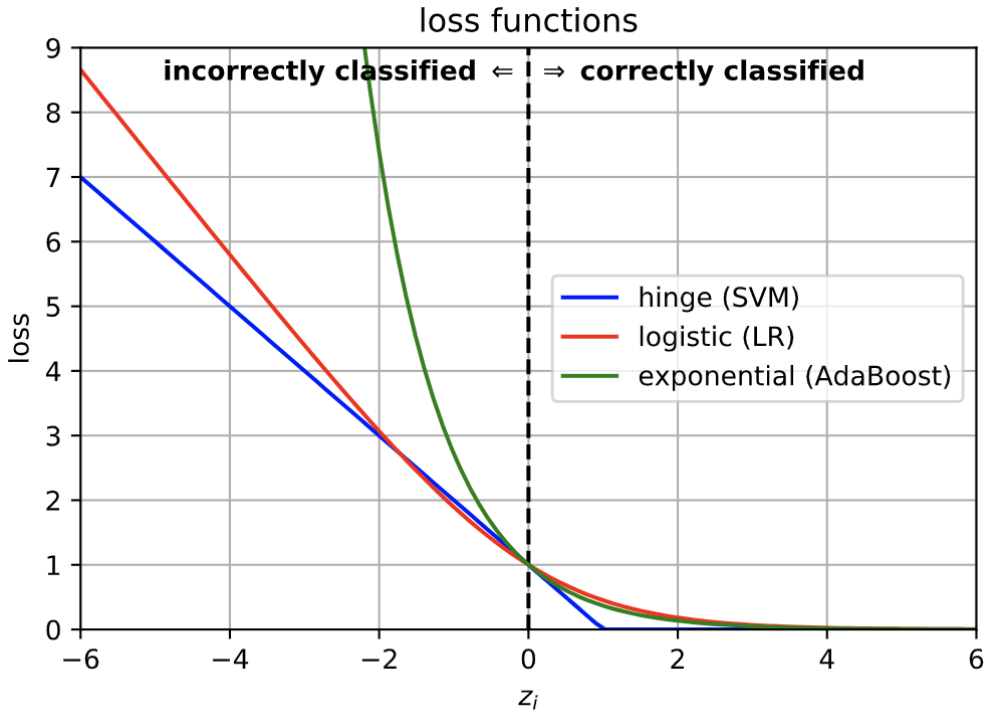
- In some tasks, errors on certain classes cannot be tolerated.
- **Example:** detecting spam vs non-spam
  - non-spam should *definitely not* be marked as spam
    - okay to mark some spam as non-spam
- Class weighting can be used to make the classifier focus on certain classes
  - e.g., weight non-spam class higher than spam class
    - classifier will try to correctly classify all non-spam samples, at the expense of making errors on spam samples.

In [ ]:
```
# dictionary (key,value) = (class name, class weight)
cw = {0: 0.2,
      1:  5}  # class 1 is 25 times more important!

clfw = svm.SVC(kernel='linear', C=10,  class_weight=cw)
clfw.fit(X, Y);
```

2.3 Classification Summary

- **Classification task**
  - Observation $\mathbf{x}$: typically a real vector of feature values, $\mathbf{x} \in \mathbb{R}^d$.
  - Class y: from a set of possible classes, e.g., $\mathcal{Y} = \{0, 1\}$
  - **Goal:** given an observation $\mathbf{x}$, predict its class y.

Name	Type	Classes	Decision function	Training	Advantages	Disadvantages
Bayes' classifier	generative	multi-class	non-linear	estimate class-conditional densities $p(x\|y)$ by maximizing likelihood of data.	- works well with small amounts of data. - multi-class. - minimum probability of error if probability models are correct.	- depends on the data correctly fitting the class-conditional.
logistic regression	discriminative	binary	linear	maximize likelihood of data in $p(y\|x)$.	- well-calibrated probabilities. - efficient to learn.	- linear decision boundary. - sensitive to C parameter.
support vector machine (SVM)	discriminative	binary	linear	maximize the margin (distance between decision surface and closest point).	- works well in high-dimension. - good generalization.	- linear decision boundary. - sensitive to C parameter.
kernel SVM	discriminative	binary	non-linear (kernel function)	maximize the margin.	- non-linear decision boundary. - can be applied to non-vector data using appropriate kernel.	- sensitive to kernel function and hyperparameters. - high memory usage for large datasets
AdaBoost	discriminative	binary	non-linear (ensemble of weak learners)	train successive weak learners to focus on misclassified points.	- non-linear decision boundary. can do feature selection. - good generalization.	- sensitive to outliers.
Random Forest	discriminative	multi-class	non-linear (ensemble of decision trees)	aggregate predictions over several decision trees, trained using different subsets of data.	- non-linear decision boundary. can do feature selection. - good generalization. - fast	- sensitive to outliers.



Regularization and Overfitting

- Some models have terms to prevent overfitting the training data.
  - this can improve *generalization* to new data.
- There is a parameter to control the regularization effect.
  - select this parameter using cross-validation on the training set.

- *Multiclass classification*
  - can use binary classifiers to do multi-class using *1-vs-rest* formulation.
- *Feature normalization*
  - normalize each feature dimension so that some feature dimensions with larger ranges do not dominate the optimization process.
- *Unbalanced data*
  - if more data in one class, then apply weights to each class to balance objectives.
- *Class imbalance*
  - mistakes on some classes are more critical.
  - reweight class to focus classifier on correctly predicting one class at the expense of others.

3. Regression

- Supervised learning
  - Input observation $\mathbf{x}$, typically a vector in $\mathbb{R}^d$.
  - Output $y \in \mathbb{R}$, a real number.
- **Goal:** predict output y from input $\mathbf{x}$.
  - i.e., learn the function $y = f(\mathbf{x})$.

Linear Regression

- **1-d case:** the output y is a linear function of input feature x
  - $y = w * x + b$
  - w is the slope, b is the intercept.

- **d-dim case**: the output y is a linear combination of d input variables $x_1, \cdots, x_d$:
  - $y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$
- Equivalently,
  - $y = w_0 + \mathbf{w}^T \mathbf{x} = w_0 + \sum_{j=1}^{d} w_j x_j$
  - $\mathbf{x} \in \mathbb{R}^d$ is the vector of input values.
  - $\mathbf{w} \in \mathbb{R}^d$ are the weights of the linear function, and w_0 is the intercept (bias term).

3.1 Ordinary Least Squares (OLS)

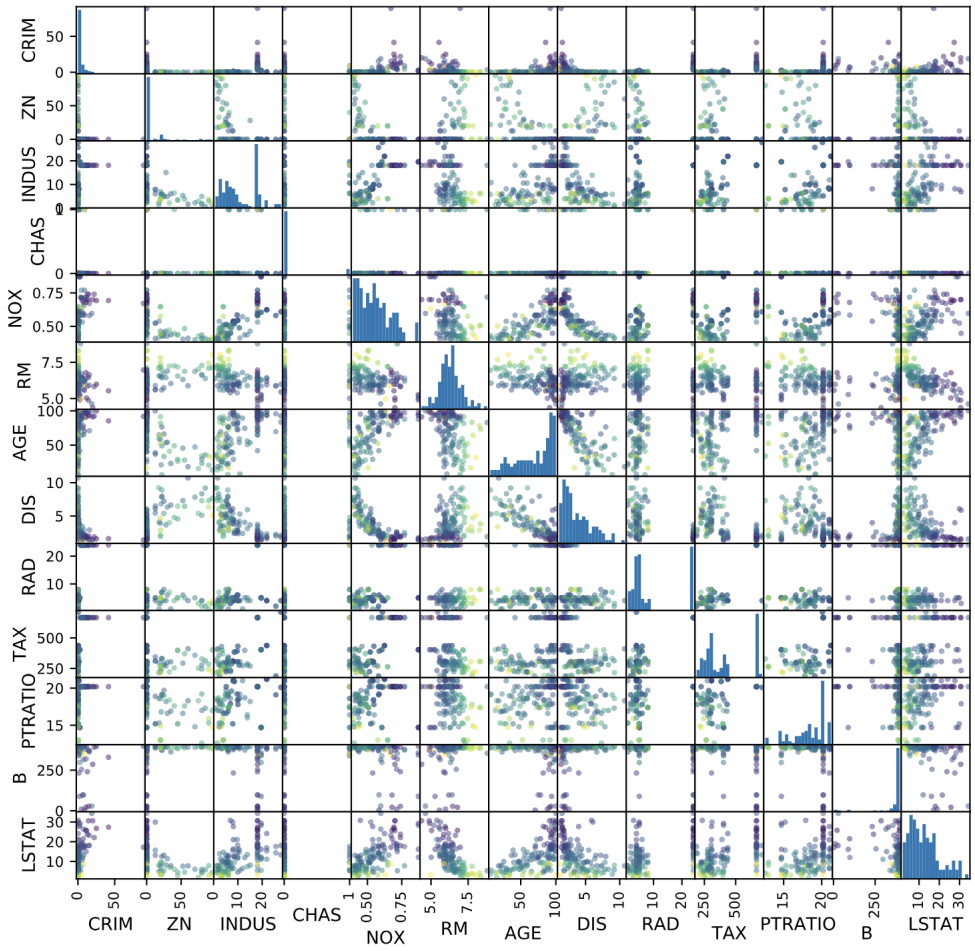
- The linear function has form $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$.
- *How to estimate the parameters* $(\mathbf{w}, b)$ *from the data?*
- Fit the parameters by minimizing the squared prediction error on the training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$:

$$\min_{\mathbf{w},b} \sum_{i=1}^{N} (y_i - f(\mathbf{x}_i))^2 = \min_{\mathbf{w},b} \sum_{i=1}^{N} (y_i - (\mathbf{w}^T\mathbf{x}_i + b))^2$$

- closed-form solution: $\mathbf{w}^* = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y}$
  - where $\mathbf{X} = [\mathbf{x}_1 \cdots \mathbf{x}_N]$ is the data matrix
  - and $\mathbf{y} = [y_1, \cdots, y_N]^T$ is vector of outputs.
  - Note: $(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}$ is also called the *pseudo-inverse* of $\mathbf{X}$.

In [ ]:
```
# fit using ordinary least squares
ols = linear_model.LinearRegression()
ols.fit(linX, linY)
```

In [ ]:
```
import pandas as pd
boston_feature_names = [x[0] for x in bostonAttr]
boston_df = pd.DataFrame(bostonX, columns=boston_feature_names)

tmp=pd.plotting.scatter_matrix(boston_df, c=bostonY, figsize=(9, 9),
                               marker='o', hist_kwds={'bins': 20}, s=10,
                               alpha=.5)
```


CRIM
ZN
INDUS
CHAS
NOX
RM
AGE
DIS
RAD
TAX
PTRATIO
B
LSTAT

Shrinkage

- Add a *regularization* term to "shrink" some linear weights to zero.
  - features associated with zero weight are not important since they aren't used to calculate the function output.
  - $y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$

3.2 Ridge Regression

- Add regularization term to OLS:

$$\min_{\mathbf{w},b} \alpha ||\mathbf{w}||^2 + \sum_{i=1}^{N} (y_i - f(\mathbf{x}_i))^2$$

- the first term is the *regularization term*
  - $||\mathbf{w}||^2 = \sum_{j=1}^{d} w_j^2$ penalizes large weights.
  - α is the hyperparameter that controls the amount of shrinkage
    - larger α means more shrinkage.
    - $\alpha = 0$ is the same as OLS.
- the second term is the *data-fit term*
  - sum-squared error of the prediction.
- Also has a closed-form solution:
  - $\mathbf{w}^* = (\mathbf{X}\mathbf{X}^T + \alpha I)^{-1}\mathbf{X}\mathbf{y}$
  - (The term "ridge regression" comes from the closed-form solution, where a "ridge" is added to the diagonal of the covariance matrix)

In [ ]:
```
# train RR with cross-validation
rr = linear_model.RidgeCV(alphas=alphas, cv=5)
rr.fit(trainXn, trainY)
```

Interpretation

- Which weights are most important?
  - negative weights indicate factors that decrease the house price
    - *Examples:* LSTAT (having higher percentage of lower status population), DIS (distance to business areas), PTRATIO (higher student-teacher ratio)
  - positive weights indicate factors that increase the house price
    - *Examples:* RM (having more rooms), RAD (proximity to highways)

Better shrinkage

- With ridge regression, some weights are small but still non-zero.
  - these are less important, but somehow still necessary.
- To get better shrinkage to zero, we can change the regularization term to encourage more weights to be 0.

3.3 LASSO

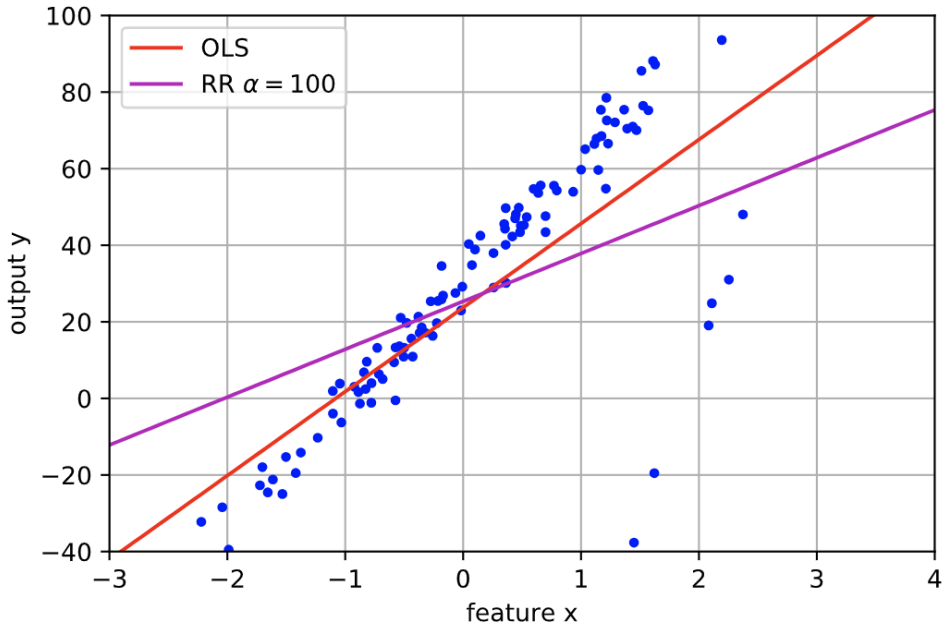
- LASSO = "Least absolute shrinkage and selection operator"
- keep the same data fit term, but change the regularization term:
  - sum of absolute weight values: $\sum_{j=1}^{d} |w_j|$
  - when a weight is close to 0, the regularization term will force it to be equal to 0.

$$\min_{\mathbf{w},b} \alpha \sum_{j=1}^{d} |w_j| + \sum_{i=1}^{N} (y_i - f(\mathbf{x}_i))^2$$

In [ ]:
```
# fit with cross-validation (alpha range is determined automatically)
las = linear_model.LassoCV()
las.fit(trainXn, trainY)

MSE = metrics.mean_squared_error(testY, las.predict(testXn))
las.alpha_
plas.coef_
```

Outliers

- Too many outliers in the data can affect the squared-error term.
  - regression function will try to reduce the large prediction error for outliers, at the expense of worse prediction for other points



3.4 RANSAC

- **RAN**dom **SA**mple **C**onsensus
  - attempt to robustly fit a regression model in the presence of corrupted data (outliers).
  - works with any regression model.
- **Idea:**
  - split the data into inliers (good data) and outliers (bad data).
  - learn the model only from the inliers

Random sampling

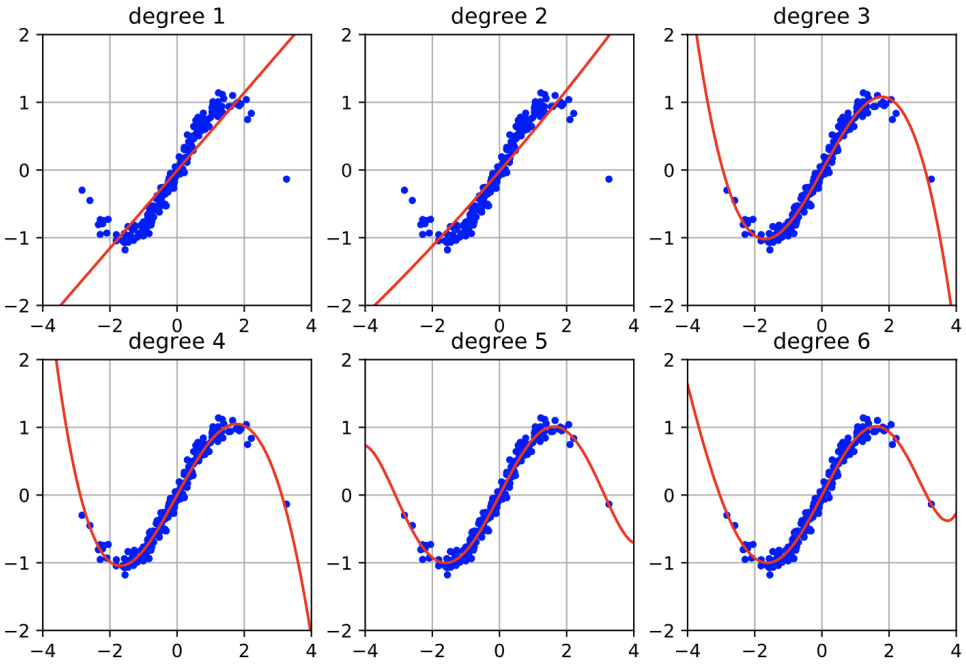
- Repeat many times...
  - randomly sample a subset of points from the data. Typically just enough to learn the regression model
  - fit a model to the subset.
  - classify all data as inlier or outlier by calculating the residuals (prediction errors) and comparing to a threshold. The set of inliers is called the *consensus set*.
  - save the model with the highest number of inliers.
- Finally, use the largest consensus set to learn the final model.

- More iterations increases the probability of finding the correct function.
  - higher probability to select a subset of points contains all inliers.
- Threshold typically set as the median absolute deviation of y.

In [ ]:
```
# use RANSAC model (defaults to linear regression)
rlin = linear_model.RANSACRegressor(random_state=1234)
rlin.fit(outlinX, outlinY)

inlier_mask = rlin.inlier_mask_
outlier_mask = logical_not(inlier_mask)
```

3.5 Polynomial regression

- p-th order Polynomial function
  - $f(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \cdots + w_p x^p$
- Collect the terms into a vector
  - $f(x) = \begin{bmatrix} w_0 & w_1 & w_2 & \cdots & w_p \end{bmatrix} * \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^p \end{bmatrix} = \mathbf{w}^T \phi(x)$
  - weight vector $\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_p \end{bmatrix}$; polynomial feature vector: $\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^p \end{bmatrix}$
- Now it's a linear function, so we can use the same linear regression!



In [ ]:
```
# make the pipeline
#  each entry is a tuple with the name and transformer (implements fit,
 transform)
#  the last entry should be a model (implements fit)
polylin = pipeline.Pipeline([
        ('polyfeats', preprocessing.PolynomialFeatures(degree=1)),
        ('linreg',    linear_model.LinearRegression())
    ])
```

In [ ]:
```
# set the parameters for grid search
# the parameters in each stage are named: <stage>__<parameter>
paramgrid = {
    "polyfeats__degree": array([1, 2, 3, 4, 5, 6]),
}

# do the cross-validdation search
plincv = model_selection.GridSearchCV(polylin, paramgrid, cv=5, n_jobs=-
1,
                                      scoring='neg_mean_squared_error')
plincv.fit(bostonX, bostonY)
```

3.6 Kernel Ridge Regression

- Apply *kernel trick* to ridge regression
  - turn linear regression into non-linear regression
- Closed form solution:
  - for an input point $\mathbf{x}_*$,
    - prediction: $y_* = \mathbf{k}_*(\mathbf{K} + \alpha I)^{-1}\mathbf{y}$
      - $\mathbf{K}$ - the kernel matrix ($N \times N$)
      - $\mathbf{k}_*$ - vector containing the kernel values between $\mathbf{x}_*$ and all training points $\mathbf{x}_i$.

In [ ]:
```
# parameters for cross-validation
paramgrid = {'alpha': logspace(-3,3,10),
             'gamma': logspace(-3,3,10)}

# do cross-validation
krrcv = model_selection.GridSearchCV(
  kernel_ridge.KernelRidge(kernel='rbf'),  # estimator
  paramgrid,                               # parameters to try
  scoring='neg_mean_squared_error',        # score function
  cv=5,                                    # number of folds
  n_jobs=-1, verbose=True)
krrcv.fit(bostonX, bostonY)
```

In [ ]:
```
# parameters for cross-validation
paramgrid = {'alpha': logspace(-3,3,10),
             'degree': [2,3,4]}

# do cross-validation
krrcv = model_selection.GridSearchCV(
  kernel_ridge.KernelRidge(kernel='poly'),  # estimator
  paramgrid,                               # parameters to try
  scoring='neg_mean_squared_error',        # score function
  cv=5,                                    # number of folds
  n_jobs=-1, verbose=True)
krrcv.fit(bostonX, bostonY)
```

3.7 Support Vector Regression (SVR)

- Borrow ideas from classification
  - Suppose we form a "band" of width ϵ around the function:
    - if a point is inside, then it is "correctly" predicted
    - if a point is outside, then it is incorrectly predicted

- Allow some points to be outside the "tube".
  - penalty of point outside tube is controlled by C parameter.
- SVR objective function:

$$\min_{\mathbf{w},b} \sum_{i=1}^{N} \left| y_i - (\mathbf{w}^T \mathbf{x}_i + b) \right|_\epsilon + \frac{1}{C} ||\mathbf{w}||^2$$

- epsilon-insensitive error:
  - $|z|_\epsilon = \begin{cases} 0, & |z| \le \epsilon \\ |z| - \epsilon, & |z| > \epsilon \end{cases}$
- Similar to SVM classifier, the points on the band will be the *support vectors* that define the function.

In [ ]:
```
epsilon = 11.5
svr = svm.SVR(C=1000, kernel='linear', epsilon=epsilon)
svr.fit(linX, linY)
```

Kernel SVR

- Support vector regression can also be kernelized similar to SVM
  - turn linear regression to non-linear regression
- Polynomial Kernel:

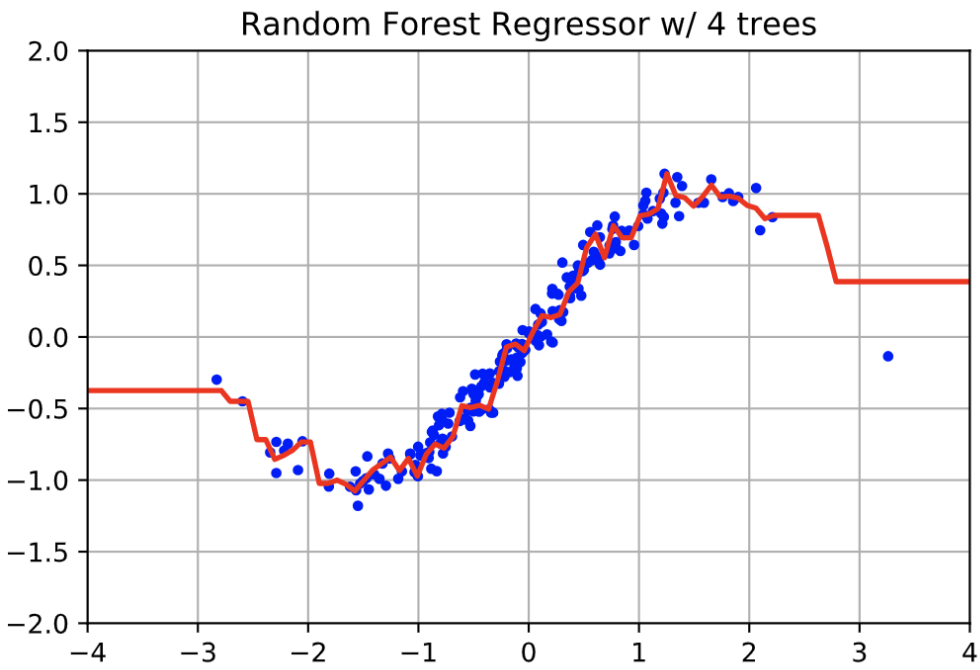
In [ ]:
```
# parameters for cross-validation
paramgrid = {'C':       logspace(-3,3,10),
             'gamma':   logspace(-3,3,10),
             'epsilon': logspace(-2,2,10)}

# do cross-validation
svrcv = model_selection.GridSearchCV(
    svm.SVR(kernel='rbf'),  # estimator
    paramgrid,                      # parameters to try
    scoring='neg_mean_squared_error',  # score function
    cv=5,
    n_jobs=-1, verbose=1)                # show progress
svrcv.fit(bostonX, bostonY)
```

In [ ]:
```
# parameters for cross-validation
paramgrid = {'C':       logspace(-3,3,10),
             'degree':  [2.3.4],
             'epsilon': logspace(-2,2,10)}

# do cross-validation
svrcv = model_selection.GridSearchCV(
    svm.SVR(kernel='poly'),  # estimator
    paramgrid,                      # parameters to try
    scoring='neg_mean_squared_error',  # score function
    cv=5,
    n_jobs=-1, verbose=1)               # show progress
svrcv.fit(bostonX, bostonY)
```

3.8 Random Forest Regression

- Similar to Random Forest Classifier
  - Average predictions over many Decision Trees
    - Each decision tree sees a random sampling of the Training set
    - Each split in the decision tree uses a random subset of features
    - Leaf node of tree contains the predicted value.

In [ ]:
```
rf = ensemble.RandomForestRegressor(n_estimators=4, random_state=4487, n
_jobs=-1)
rf.fit(polyX, polyY)
```



In [ ]:
```
# parameters for cross-validation
paramgrid = {'max_depth': array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15]),
             }

# do cross-validation
rfcv = model_selection.GridSearchCV(
    ensemble.RandomForestRegressor(n_estimators=100, random_state=4487),
  # estimator
    paramgrid,                      # parameters to try
    scoring='neg_mean_squared_error',  # score function
    cv=5,
    n_jobs=-1, verbose=True
)
rfcv.fit(bostonX, bostonY)
```

3.9 Regression Summary

- **Goal:** predict output $y \in \mathbb{R}$ from input $\mathbf{x} \in \mathbb{R}^d$.
  - i.e., learn the function $y = f(\mathbf{x})$.

Name	Function	Training	Advantages	Disadvantages
Ordinary Least Squares	linear	minimize square error between observation and predicted output.	- closed-form solution.	- sensitive to outliers and overfitting.
ridge regression	linear	minimize squared error with $\|\|w\|\|^2$ regularization term.	- closed-form solution; - shrinkage to prevent overfitting.	- sensitive to outliers.
LASSO	linear	minimize squared error with $\sum_{j=1}^{d} \|w_j\|$ regularization term.	- feature selection (by forcing weights to 0)	- sensitive to outliers.
RANSAC	same as the base model	randomly sample subset of training data and fit model; keep model with most inliers.	- ignores outliers.	- requires enough iterations to find good consensus set.
kernel ridge regression	non-linear (kernel function)	apply "kernel trick" to ridge regression.	- non-linear regression. - Closed-form solution.	- requires calculating kernel matrix $O(N^2)$. - cross-validation to select hyperparameters.
kernel support vector regression	non-linear (kernel function)	minimize squared error, insensitive to epsilon-error.	- non-linear regression. - faster predictions than kernel ridge regression.	- requires calculating kernel matrix $O(N^2)$. - iterative solution (slow). - cross-validation to select hyperparameters.
random forest regression	non-linear (ensemble)	aggregate predictions from decision trees.	- non-linear regression. - fast predictions.	- predicts step-wise function. - cannot learn a completely smooth function.

Other Things

- *Feature normalization*
  - feature normalization is typically required for regression methods with regularization.
  - makes ordering of weights more interpretable (LASSO, RR).
- *Output transformations*
  - sometimes the output values y have a large dynamic range (e.g., 10^{-1} to 10^{5}).
    - large output values will have large error, which will dominate the training error.
  - in this case, it is better to transform the output values using the logarithm function.
    - $\hat{y} = \log_{10}(y)$
  - For example, see the tutorial.

4. Clustering

- Each data point is a vector $\mathbf{x} \in \mathbb{R}^d$.
- Data is set of vectors $\{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$
- **Goal:** group similar data together.
  - groups are also called clusters.
  - each data point is assigned with a cluster index ($y \in \{1, \cdots, K\}$)
    - K is the number of clusters.

4.1 K-Means Clustering

- **Idea:**
  - there are K clusters.
  - each cluster is represented by a *cluster center*.
    - $\mathbf{c}_j \in \mathbf{R}^d, j \in \{1, \cdots, K\}$
  - assign each data point to the closest cluster center.
    - according to Euclidean distance: $||\mathbf{x}_i - \mathbf{c}_j||$
- *How to pick the cluster centers?*
  - Assume there are K clusters
  - Pick the cluster centers that minimize the squared distance to all its cluster members.

$$\min_{\mathbf{c}_1, \cdots, \mathbf{c}_K} \sum_{i=1}^{n} ||\mathbf{x}_i - \mathbf{c}_{z_i}||^2$$

  - where z_i is the index of the closest cluster center to $\mathbf{x}_i$.
    - $z_i = \text{argmin}_{j=\{1,\cdots,K\}} ||\mathbf{x}_i - \mathbf{c}_j||$
    - i.e., the assignment of point $\mathbf{x}_i$ to its closest cluster.
- Solution:
  - if the assignments $\{z_i\}$ are known...
    - let C_j be the set of points assigned to cluster j
      - $C_j = \{\mathbf{x}_i | z_i = j\}$
    - cluster center is the mean of the points in the cluster
      - $\mathbf{c}_j = \frac{1}{|C_j|} \sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i$

K-means Algorithm

- Pick initial cluster centers
- Repeat:
  - 1) calculate assignment z_i for each point $\mathbf{x}_i$: closest cluster center using Euclidean distance.
  - 2) calculate cluster center $\mathbf{c}_j$ as average of points assigned to cluster j.
- This procedure will converge eventually.

Important Note

- The final result depends on the initial cluster centers!
  - Some bad initializations will yield poor clustering results!
  - (Technically, there are multiple local minimums in the objective function)

- **Solution:**
  - Try several times using different initializations.
  - Pick the answer with lowest objective score.
- In scikit-learn,
  - automatically uses multiple random initializations.
  - also uses a smart initialization method called "k-means++"
  - can run initialization runs in parallel (`n_jobs`)

In [ ]:
```
# K-Means with 3 clusters
# (automatically does 10 random initializations)
km = cluster.KMeans(n_clusters=3, random_state=4487, n_jobs=-1)
Yp = km.fit_predict(X)   # cluster data, and return labels

cc = km.cluster_centers_   # the cluster centers
cl = km.labels_            # labels also stored here
```

Circular clusters

- One problem with K-means is that it assumes that each cluster has a circular shape.
  - based on Euclidean distance to each center
  - Kmeans cannot handle skewed (elliptical) clusters.

4.2 Gaussian mixture model (GMM)

- A multivariate Gaussian can model a cluster with an elliptical shape.
  - the ellipse shape is controlled by the covariance matrix of the Gaussian
  - the location of the cluster is controlled by the mean.
- Gaussian mixture model is a weighted sum of Gaussians

$$p(\mathbf{x}) = \sum_{j=1}^{K} \pi_j N(\mathbf{x}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$$

  - Each Gaussian represents one elliptical cluster
    - $\boldsymbol{\mu}_j$ is the mean of the j-th Gaussian. (the location)
    - $\boldsymbol{\Sigma}_j$ is the covariance matrix of the j-th Gaussian. (the ellipse shape)
    - π_j is the prior weight of the j-th Gaussian. (how likely is this cluster)

Clustering with GMMs

- Using the data, learn a GMM using maximum likelihood estimation:

$$\max_{\pi,\boldsymbol{\mu},\boldsymbol{\Sigma}} \sum_{i=1}^{N} \log \sum_{j=1}^{K} \pi_j N(\mathbf{x}_i|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$$

- Results in an algorithm similar to K-means:
  - 1) Calculate cluster membership
    - uses "soft" assignment - a data point can have a fractional contribution to different clusters.
    - contribution of point i to cluster j
      - $z_{ij} = p(z_i = j|\mathbf{x}_i)$
  - 2) Update each Gaussian cluster (mean, covariance, and weight)
    - uses "soft" weighting
      - "soft" count of points in cluster j: $N_j = \sum_{i=1}^{N} z_{ij}$
      - weight: $\pi_j = N_j/N$
      - mean: $\boldsymbol{\mu}_j = \frac{1}{N_j} \sum_{i=1}^{N} z_{ij}\mathbf{x}_i$
      - variance: $\Sigma_j = \frac{1}{N_j} \sum_{i=1}^{N} z_{ij}(\mathbf{x}_i - \boldsymbol{\mu}_j)^2$

In [ ]:
```
# fit a GMM
gmm = mixture.GaussianMixture(n_components=4, random_state=4487, n_init=
10)

gmm.fit(X)
Y = gmm.predict(X)

cc = gmm.means_     # the cluster centers
```

Covariance matrix

- The covariance matrix is a $d \times d$ matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

- For high-dimensional data, it can be very large.
  - requires a lot of data to learn effectively.

- Solution:
  - use *diagonal* covariance matrices (d parameters):

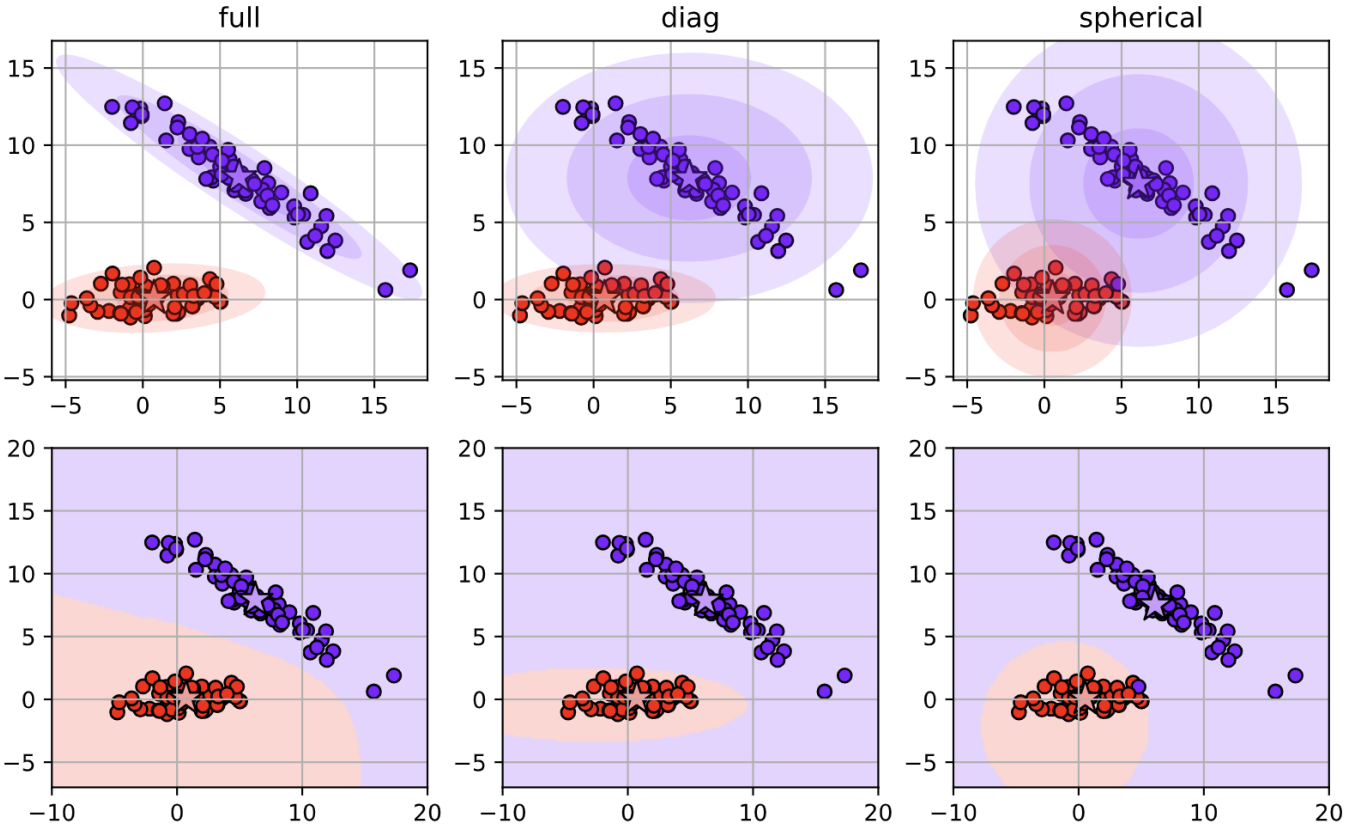
$$\begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix}$$

    - Axes of ellipses will be aligned with the axes.
  - use *spherical* covariance matrices (1 parameter)

$$\begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix}$$

    - Clusters will be circular (similar to K-means)

In [ ]:
```
# full covariance (d*d parameters)
gmmf = mixture.GaussianMixture(n_components=2, covariance_type='full',
                               random_state=4487, n_init=10)
gmmf.fit(X)

# diagonal convariance (d parameters)
gmmd = mixture.GaussianMixture(n_components=2, covariance_type='diag',
                               random_state=4487, n_init=10)
gmmd.fit(X)

# spherical covariance (1 parameter)
gmms = mixture.GaussianMixture(n_components=2, covariance_type='spherica
l',
                               random_state=4487, n_init=10)
gmms.fit(X)
```



4.3 Dirichlet Process GMM

- GMM is extended to automatically select the value of K
  - use a "Dirichlet Process" to model K as a random variable.
- *concentration* parameter α controls the range of K values that are preferred
  - higher values encourage more clusters
  - lower values encourage less clusters
  - expected number of clusers is $\alpha \log N$, where N is the number of points.

In [ ]:
```
# alpha = concentration parameter
# n_components = the max number of components to consider
dpgmm = mixture.BayesianGaussianMixture(covariance_type='full',
                    weight_concentration_prior=1,
                    n_components=5, max_iter=100, random_state=4487)
dpgmm.fit(X)
Y = dpgmm.predict(X)
cl = unique(Y)          # find active clusters
newK = len(cl)          # number of clusters
cc = dpgmm.means_[cl]  # get means
```

4.4 Mean-shift algorithm

- Clustering algorithm that also automatically selects the number of clusters.
- **Idea:** iteratively shift towards the largest concentration of points.
  - start from an initial point $\mathbf{x}$ (e.g., one of the data points).
  - repeat until $\mathbf{x}$ is unchanged:
    - 1) find the nearest neighbors to $\mathbf{x}$ within some radius (bandwidth)
    - 2) set $\mathbf{x}$ to be the mean of the neighbor points.

Getting the clusters

- Run the mean-shift algorithm for many initial points $\{x_i\}$.
  - the set of converged points contain the cluster centers.
    - need to remove the duplicate centers.
  - data points that converge to the same center belong to the same cluster.
  - different initializations can run in parallel (`n_jobs`)

In [ ]:
```
# bin_seeding=True -- coarsely uses data points as initial points
ms = cluster.MeanShift(bandwidth=5, bin_seeding=True, n_jobs=-1)
Y = ms.fit_predict(X)

cc = ms.cluster_centers_  # cluster centers
```

Number of clusters

- Number of clusters is implicitly controlled by the bandwidth (radius of the nearest-neighbors)
  - larger bandwidth creates less clusters
    - focuses on global large groups
  - smaller bandwidth creates more clusters
    - focuses on local groups.

Non-compact clusters

- K-means, GMM, and Mean-Shift assume that all clusters are compact.
  - i.e., circles or ellipses
- What about clusters of other shapes?
  - e.g., clusters not defined by compact distance to a "center"

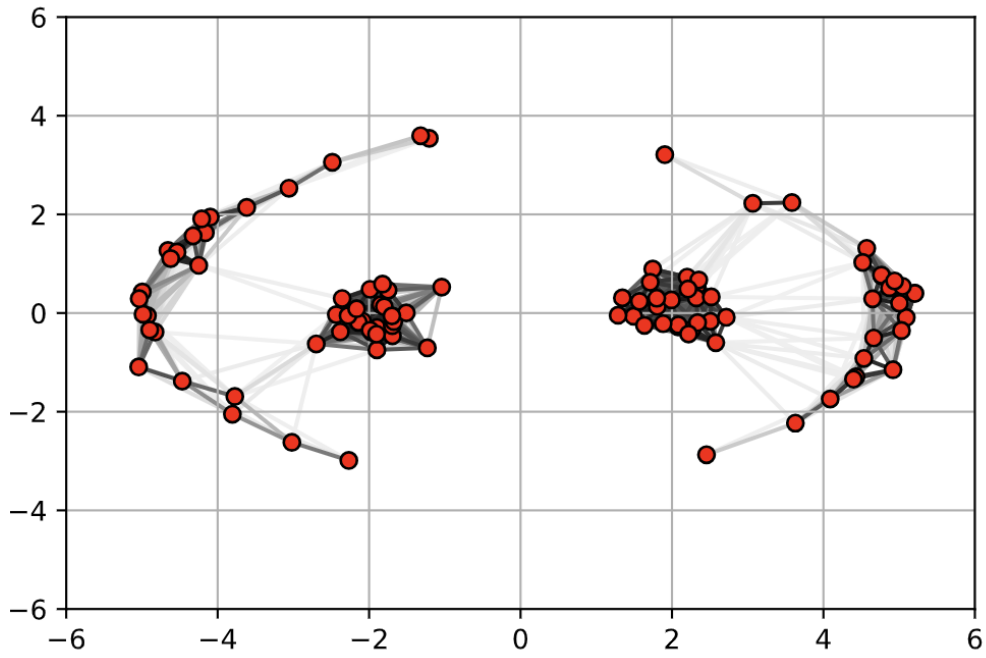
4.5 Spectral Clustering

- Estimate the clusters using the pair-wise affinity between points.
- Affinity (similarity) between points
  - kernel function: $k(\mathbf{x}_i, \mathbf{x}_j)$ -- RBF kernel
  - number of nearest neighbors within a radius (bandwidth)

In [ ]:
```
sc = cluster.SpectralClustering(n_clusters=4, affinity='rbf', gamma=1.0,
 assign_labels='discretize', n_jobs=-1)
Y = sc.fit_predict(X)  # cluster and also return the labels Y
```

Spectral Clustering

- **Idea:** clustering with a graph formulation
  - each data point is a node in a graph
  - edge weight between two nodes is the affinity $k(\mathbf{x}_i, \mathbf{x}_j)$
    - (darker colors indicate stronger weights)



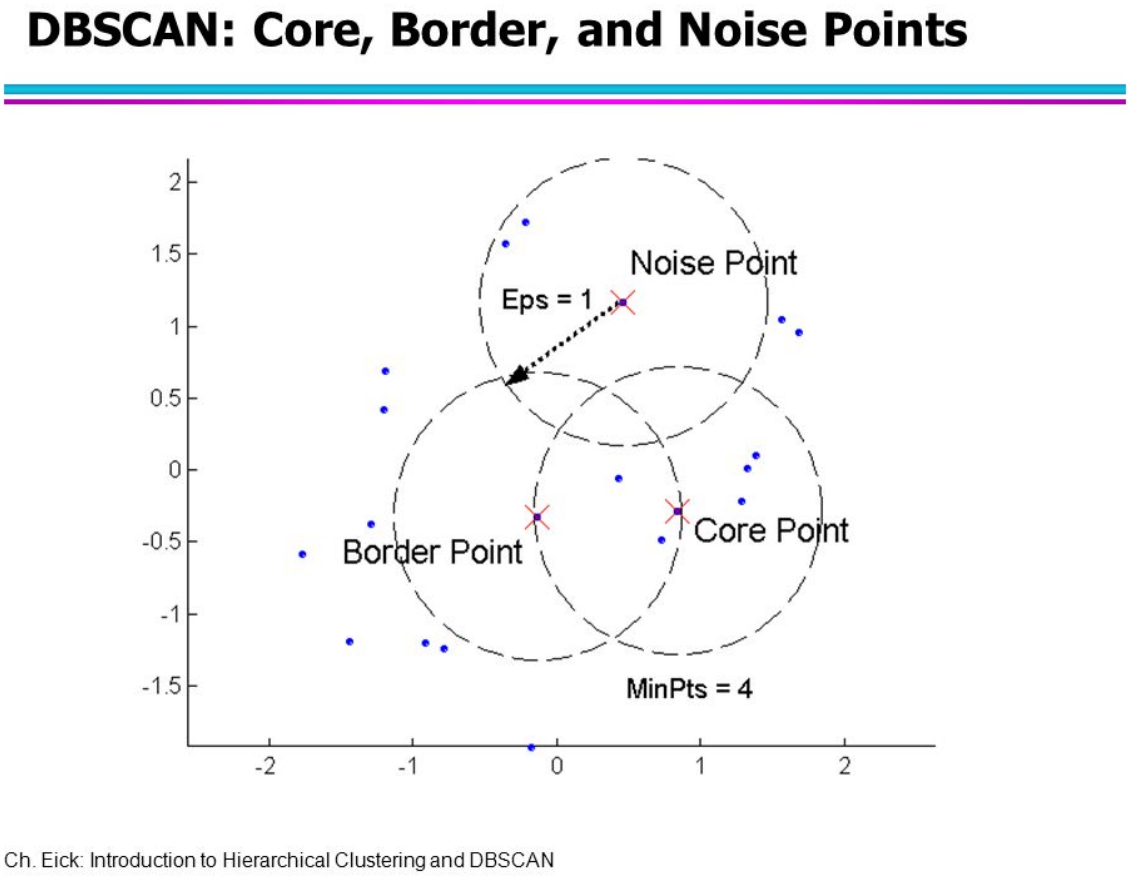
Sensitivity to gamma

- gamma controls which structures are important
  - small gamma - far away points are still considered similar
  - large gamma - close points are not considered similar

4.6 DBSCAN

- "Density-Based Spatial Clustering of Applications with Noise"
  - Assumption: clusters are regions of high density of points separated by areas of low density.
  - Algorithm Idea:
    - Find a *core* point of high density.
    - Recursively label the neighbors as core points.
    - Neighbors that are not core samples are called *boundary* or *non-core* points.
    - Points that are not boundary and not core points are *outliers*.

- Define two parameters:
  - `eps` : the maximum distance to be considered a neighbor.
  - `min_samples` : the minimum number of neighbors (including point itself) to be considered a core sample.



Ch. Eick: Introduction to Hierarchical Clustering and DBSCAN

In [ ]:
```
# eps = the max distance to be considered a neighbor
# min_samples = min number of neighbors to be a core sample
dbs = cluster.DBSCAN(eps=0.5, min_samples=5, n_jobs=-1)
Y = dbs.fit_predict(X)

# labels: -1 means outlier
print(Y)
```

- Effect of `eps`
  - smaller `eps` - high density required to make a single cluster
  - larger `eps` - encourages collapsing of clusters

- Effect of `min_samples`
  - smaller `min_samples` - encourages forming small clusters
  - larger `min_samples` - forms clusters only in very high-density regions.

4.7 Clustering Summary

- **Goal:** given set of input vectors $\{\mathbf{x}_i\}_{i=1}^n$, with $\mathbf{x}_i \in \mathbb{R}^d$, group similar x_i together into clusters.
  - estimate a cluster center, which represents the data points in that cluster.
  - predict the cluster for a new data point.

Name	Cluster Shape	Principle	Advantages	Disadvantages
K-Means	circular	minimize distance to cluster center	- scalable (MiniBatchKMeans)	- sensitive to initialization; could get bad solutions due to local minima. - need to choose K.
Gaussian Mixture Model	elliptical	maximum likelihood	- elliptical cluster shapes.	- sensitive to initialization; could get bad solutions due to local minima. - need to choose K.
Dirichlet Process GMM	elliptical	maximum likelihood	- automatically selects K via concentration parameter.	- can be slow. - sensitive to initialization; could get bad solutions due to local minima.
Mean-Shift	concentrated compact	move towards local mean	- automatically selects K via bandwidth parameter.	- can be slow.
Spectral clustering	irregular shapes	graph-based	- can handle clusters of any shape, as long as connected.	- need to choose K. - cannot assign novel points to a cluster. - can be slow (kernel matrix)
DBSCAN	irregular shapes	density-based	- can handle clusters of any shape, as densely sampled. - can detect outliers	- sensitive to parameters - cannot assign novel points to a cluster.

Other Things

- *Feature normalization*
  - feature normalization is typically required clustering.
  - e.g., algorithms based on Euclidean distance (Kmeans, Mean-Shift, Spectral Clustering)

5. Dimensionality Reduction

- **Goal:** Transform high-dimensional vectors into low-dimensional vectors.
  - Dimensions in the low-dim data represent co-occuring features in high-dim data.
  - Dimensions in the low-dim data may have semantic meaning.
- **For example:** document analysis
  - high-dim: bag-of-word vectors of documents
  - low-dim: each dimension represents similarity to a topic.

Reasons for Dimensionality Reduction

- Preprocessing - make the dataset easier to use
- Reduce computational cost of running machine learning algorithms
- Remove noise
- Make the results easier to understand (visualization)

5.1 Linear Dimensionality Reduction for Vectors

- Project the original data onto a lower-dimensional hyperplane (e.g., line, plane).
  - I.e, Move and rotate the coordinate axis of the data
- Represent the data with coordinates in the new component space.

- Equivalently, approximate the data point $\mathbf{x}$ as a linear combination of basis vectors (components) in the original space.
  - original data point $\mathbf{x} \in \mathbb{R}^d$
  - approximation: $\hat{\mathbf{x}} = \sum_{j=1}^{p} w_j \mathbf{v}_j$
    - $\mathbf{v}_j \in \mathbb{R}^d$ is a basis vector and $w_j \in \mathbb{R}$ the corresponding weight.
  - the data point $\mathbf{x}$ is then represented its corresponding weights
    - $\mathbf{w} = [w_1, \cdots, w_P] \in \mathbb{R}^p$

- Several methods for linear dimensionality reduction.
- **Differences:**
  - goal (reconstruction vs classification)
  - unsupervised vs. supervised
  - constraints on the basis vectors and the weights.
  - reconstruction error criteria

5.1.1 Principal Component Analysis (PCA)

- Unsupervised method
- **Goal:** preserve the variance of the data as much as possible
  - choose basis vectors along the maximum variance (longest extent) of the data.
  - the basis vectors are called *principal components* (PC).

- **Goal:** Equivalently, minimize the reconstruction error over all the data points $\{\mathbf{x}_i\}_{i=1}^N$.
  - reconstruction: $\hat{\mathbf{x}}_i = \sum_{j=1}^p w_{i,j}\mathbf{v}_j$

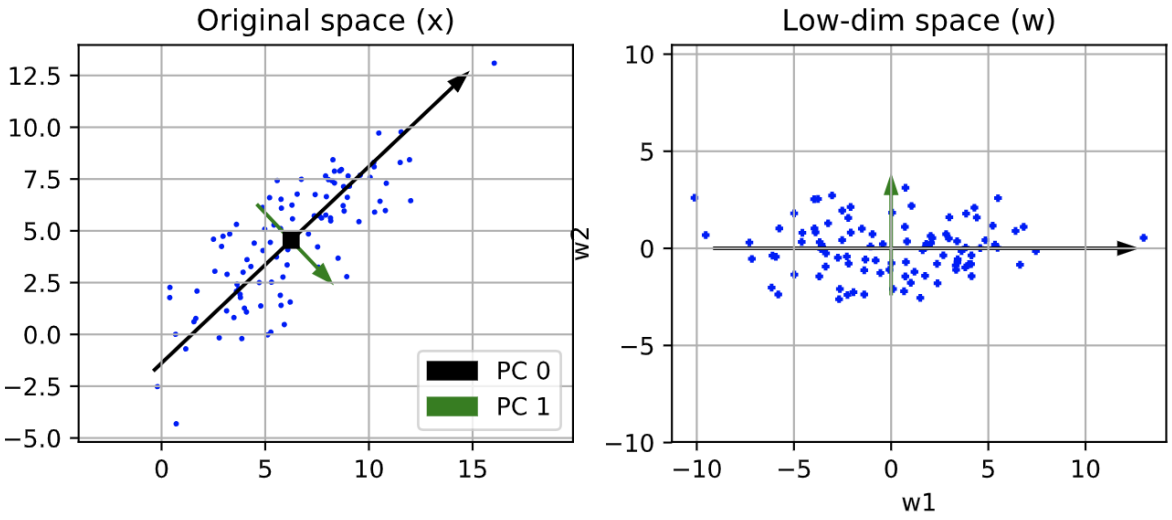
$$\min_{w,\mathbf{v}} \sum_{i=1}^{N} ||\mathbf{x}_i - \hat{\mathbf{x}}_i||^2$$

  - *constraint:* principal components $\mathbf{v}_j$ are orthogonal (perpendicular) to each other.

PCA algorithm

- 1) subtract the mean of the data
- 2) the first PC v_1 is the direction that explains the most variance of the data.
- 3) the second PC v_2 is the direction perpendicular to v_1 that explains the most variance.
- 4) the third PC v_3 is the direction perpendicular to $\{v_1, v_2\}$ that explains the most variance.
- 5) ...

In [ ]:
```
# run PCA
pca = decomposition.PCA(n_components=1)
W   = pca.fit_transform(X)  # returns the coefficients
Wt = pca.transform(testX)  # use the pca model to transform the test set

v = pca.components_  # the principal component vector
m = pca.mean_        # the data mean
```



How to choose the number of principal components?

- Two methods to set the number of components p:
  - preserve some percentage of the variance (e.g., 95%).
  - whatever works well for our final task (e.g., classification, regression).

Explained variance

- each PC explains a percentage of the original data
  - this is called the *explained variance*.
  - PCs are already sorted by explained variance from highest to lowest
- pick the number of PCs to get a certain percentage of explained variance
  - typically 95%

In [ ]:
```
ev    = pca.explained_variance_ratio_  # variance explained by each comp
onent
cumev = cumsum(ev)                     # cumulative explained variance
```

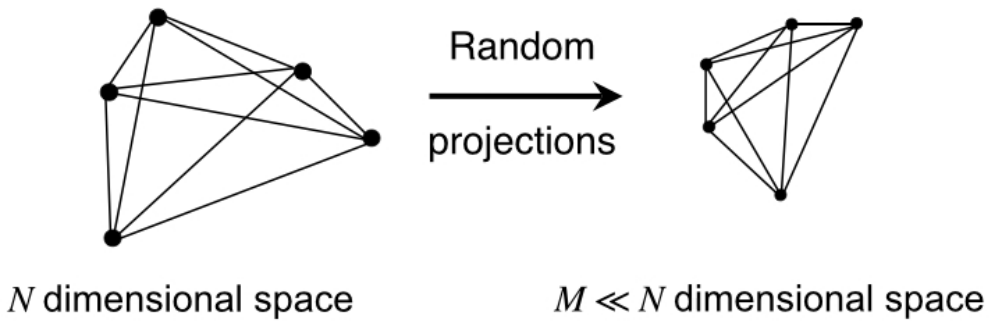
5.1.2 Random Projections

- If the data is very high-dimensional, then it might take too many calculations to do PCA.
  - Complexity: $O(dk^2)$, d is the dimension, k is the number of components
- Do we really need to estimate the principal components to reduce the dimension?

- **Solution:**
  - We can generate random basis vectors and use those.
    - Each entry of $\mathbf{v}_j$ sampled from a Gaussian.
  - This will save a lot of time.
  - Random Projections can reduce computation at the expense of losing some accuracy in the points (adding noise).

In [ ]:
```
# project the digits data with Random Projection
rp = random_projection.GaussianRandomProjection(n_components=2, random_s
tate=4487)
Wrp = rp.fit_transform(trainX)
```

- Okay, but is it good?
  - One way to measure "goodness" is to see if the structure of the data is preserved.
  - In other words, are distances between points preserved in the transformed data?



- **Answer:**
  - Yes!
  - According to the *Johnson-Lindenstrauss lemma*, carefully selecting the distribution of the random projection matrices will preserve the preserve the pairwise distances between any two samples of the dataset, within some error *epsilon*.
    - $(1-\epsilon)\|\mathbf{x}_i - \mathbf{x}_j\|^2 < \|\mathbf{w}_i - \mathbf{w}_j\|^2 < (1+\epsilon)\|\mathbf{x}_i - \mathbf{x}_j\|^2$
    - the minimum reduced dimension p to gaurantee ϵ error depends on the number of samples.
      - (actually, this is fairly conservative)

5.1.3 Sparse Random Projection

- More computation can be saved by using a *sparse* random projection matrix
  - "sparse" means that many entries in the basis vector are zero, so we can ignore those entries when multiplying.

In [ ]:
```
# project the digits data with Random Projection
srp = random_projection.SparseRandomProjection(n_components=500, random_
state=4487)
Wsrp = srp.fit_transform(X)
```

Question

- Suppose we have data for the below classification problem...
- We want to reduce the data to 1 dimension using PCA.
  - What is the first PC?

Answer

- first PC is along the direction of most variance.
  - collapses the two classes together!

Problem with Unsupervised Methods

- If our end goal is classification, preserving the variance sometimes won't help!
  - PCA doesn't consider which class the data belongs to.
  - When the "classification" signal is less than the "noise", PCA will make classification more difficult.

5.1.4 Fisher's Linear Discriminant (FLD)

- Supervised dimensionality reduction
- Also called "*Linear Discriminant Analysis*" (LDA)
- **Goal:** find a lower-dim space so as to minimize the class overlap (or maximize the class separation).
  - data from each class is modeled as a Gaussian.
  - requires the class labels

In [ ]:
```
# example of FLD projection (using LDA name)
fld = discriminant_analysis.LinearDiscriminantAnalysis(n_components=1)
W  = fld.fit_transform(X, Y)

v = fld.coef_   # the basis vectors
```

5.2 Linear Dimensionality Reduction for Tex

5.2.1 Latent Semantic Analysis (LSA)

- Also called *Latent Semantic Indexing*
- Consider a bag-of-word representation (e.g., TF, TF-IDF)
  - document vector $\mathbf{x}_i$
  - $x_{i,j}$ is the frequency of word j in document i

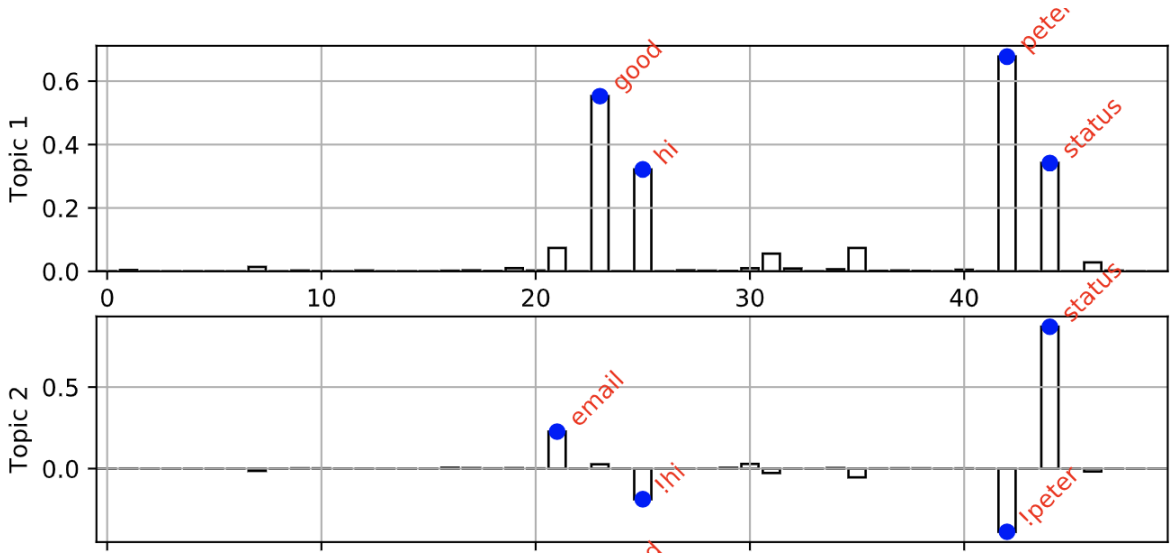
- Approximate each document vector as a weighted sum of topic vectors.
  - $\hat{\mathbf{x}} = \sum_{n=1}^{p} w_p \mathbf{v}_p$
  - Topic vector $\mathbf{v}_p$ contains co-occuring words.
    - corresponds to a particular *topic* or *theme*.
  - Weight w_p represents similarity of the document to the p-th topic.
- Objective:
  - minimize the squared reconstruction error (Similar to PCA):
  - $\min_{\mathbf{v},\mathbf{w}} \sum_i ||\mathbf{x}_i - \hat{\mathbf{x}}_i||^2$

- Represent each document by its topic weights.
  - Apply other machine learning algorithms...
- **Advantage:**
  - Finds relations between terms (synonymy and polysemy).
  - distances/similarities are now comparing topics rather than words.
    - higher-level semantic representation

In [ ]:
```
lsa = decomposition.TruncatedSVD(n_components=5, random_state=4487)
Wlsa = lsa.fit_transform(Xtf)

# components
V = lsa.components_
```

Topic vectors

- topic vectors contain frequent co-occuring words



Problem with LSA

- In the topic vector, the "frequency" of a word can be negative!
  - Doesn't really make sense for document bag-of-words model.

Problems with LSA

- The weights for each topic can be negative!
  - Topics should only be "additive"
    - Topics should increase probability of some topic-related words, but not decrease probability of other words.
    - It doesn't make sense to "remove" a topic using a negative topic weight.

5.2.2 Non-negative Matrix Factorization (NMF)

- **Solution:** constrain the topic vector and weights to be non-negative.
- Similar to LSA
  - Approximate each document vector as a weighted sum of topic vectors.
    - $\hat{\mathbf{x}}_j = \sum_{n=1}^{p} w_p \mathbf{v}_p$
    - But now, each entry of topic vector $\mathbf{v}_p \geq 0$ and topic weight $w_p \geq 0$
  - Objective: minimize the squared reconstruction error
    - $\min_{\mathbf{v},\mathbf{w}} \sum_j ||\mathbf{x}_j - \hat{\mathbf{x}}_j||^2$
    - subject to the non-negative constraints.

In [ ]:
```
# Run NMF
nmf = decomposition.NMF(n_components=5)
Wnmf = nmf.fit_transform(Xtf)

# components
V = nmf.components_
```

Sparseness

- For NMF representation, most topic weights for a document are zero.
  - this is called a *sparse* representation.
  - each document is only composed of a few topics.

Problem with NMF

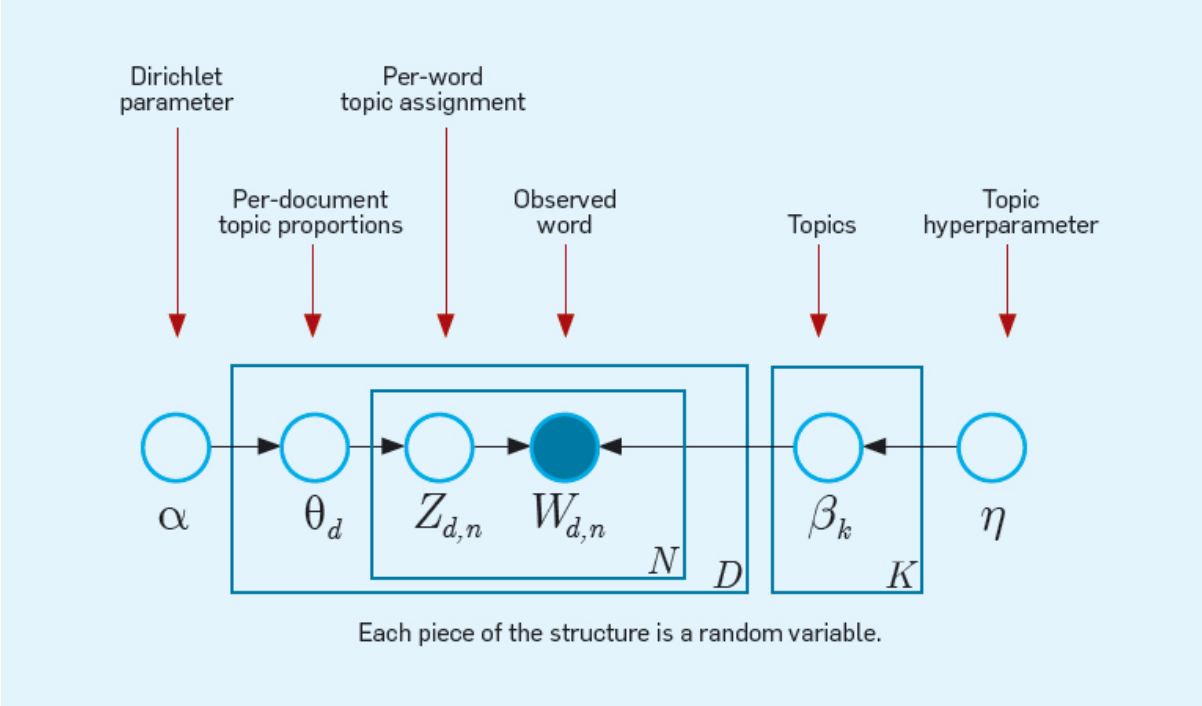
- *While the weights and component vectors are non-negative, NMF does not enforce them to be probabilities.*
- TF/TFIDF is a probabilistic model of words in a document
  - the vector of probabilities sums to 1
  - probabilities are between 0 and 1
- The NMF components and weights are difficult to interpret.

5.2.3 Latent Dirichlet Allocation (LDA)

- Use a generative probabilistic framework to model topics and documents.
- A document is composed of a mixture of topics.
  - Each topic has its own distribution of words (topic vector $\boldsymbol{\beta}_k$).
    - Topic vectors are shared among documents.
  - Each document has its own topic weighting.
    - The d-th document: $\hat{\mathbf{x}}_d = \sum_{k=1}^{K} \theta_{d,k}\boldsymbol{\beta}_k$
      - $\theta_{d,k}$ is the probability of the k-th topic occurring in the d-th document.
      - $\boldsymbol{\beta}_k$ is the topic vector for the k-th topic.

LDA graphical model

- Each node is a random variable.
- Plates (boxes) denote a vector of random variables.
  - The size is given in the bottom-right corner.



- LDA generative model
  1. For each topic $k = 1 \dots K$:
     A. Sample the topic vector $\boldsymbol{\beta}_k$ from a Dirichlet distribution.
  2. For each document $d = 1 \dots D$:
     A. Sample the topic weights θ_d from a Dirichlet distribution.
     B. For each word-position $n = 1 \dots N$:
        a. Sample a topic $z_{d,n}$.
        b. Sample a word $w_{d,n}$ from topic $z_{d,n}$.

LDA implementation

- Input X is the word counts from `CountVectorizer`.
- Important parameters:
  - `n_components` - The number of topics (K).
  - `doc_topic_prior` - Smoothing parameter (α) for the topic weights θ.
  - `topic_word_prior` - Smoothing parameter (η) for the topic vector β.
- Note: in the sklearn implementation,
  - the topic weights are not normalized on output.
  - the topic vectors are not normalized in `components_`
  - can be parallelized (`n_jobs`)

In [ ]:
```
# fit LDA model
lda = decomposition.LatentDirichletAllocation(
    n_components=5,
    doc_topic_prior=0.01,
    topic_word_prior=0.01,
    random_state=4487, max_iter=25, n_jobs=-1)
# fit with X
Wlda = lda.fit_transform(X)

# normalize rows to be probabilities
Wlda /= sum(Wlda, axis=1)[:,newaxis]
```

- Topic vectors
  - Note that there is a small probability for each word (controlled by smoothing parameter η).
- Document vector
  - Note the small probability for each topic (controlled by smoothing parameter α).

5.3 Linear Dimensionality Reduction - Summary

- **Goal:** given set of input vectors $\{\mathbf{x}_i\}_{i=1}^n$, with $\mathbf{x}_i \in \mathbb{R}^d$, represent each input vector as lower-dimensional vector $\mathbf{w}_i \in \mathbb{R}^p$.
  - Approximate $\mathbf{x}$ as a weighted sum of basis vectors $\mathbf{v}_j \in \mathbb{R}^d$
    - $\hat{x} = \sum_{j=1}^p w_j \mathbf{v}_j$
    - minimize the reconstruction error of $\hat{\mathbf{x}}$.
  - enables faster processing, or reduces noise.

Name	Objective	Advantages	Disadvantages
Principal component analysis (PCA)	minimize reconstruction error; preserve the most variance of data	- captures correlated dimensions, removes redundant dimensions, removes noise. - closed-form solution	- does not consider end goal (e.g., classification)
Random Projections	sample random basis vectors.	- fast. - preserves pairwise distances between points (up to accuracy factor).	- adds noise to the pairwise distances.
Fisher's Linear Discriminant (FLD)	maximize class separation	- preserves class separation	- requires class information
Latent Semantic Analysis (LSA)	minimize reconstruction error	- topic vectors have semantic meaning (co-occuring words) - closed-form solution	- topic weights and topic vectors can be negative - does not consider end goal (e.g., classification)
Non-negative Matrix Factorization (NMF)	minimize reconstruction error; non-negative weights and basis vectors.	- "additive" topic/parts model for text or images - sparse topic weights.	- solution requires iterative algorithm. - does not consider end goal (e.g., classification)
Latent Dirichlet Allocation (LDA)	document is a mixture of topics.	- generative probabilistic model. - robust when dataset size is small.	- inference/training can be slow for larger datasets.

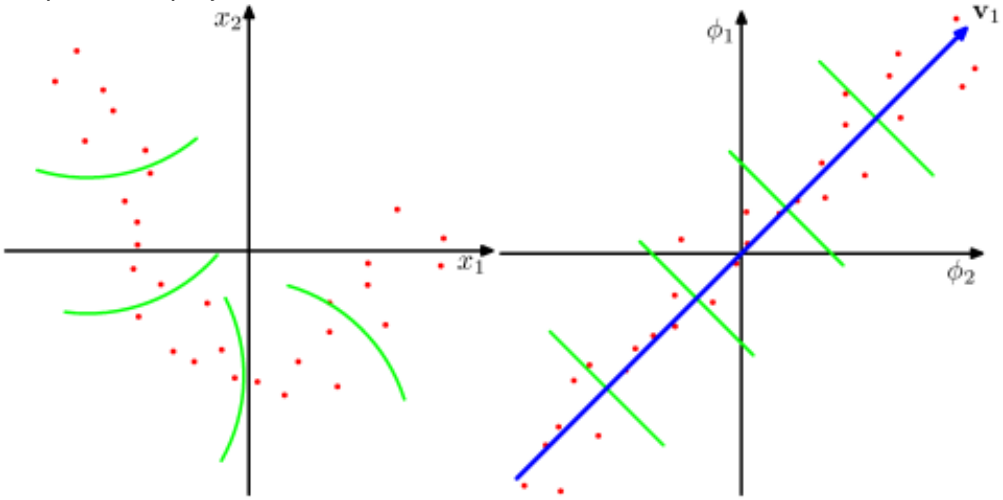
Other things

- *Feature Normalization*
  - PCA and LDA are based on the covariance between input dimensions.
  - applying *per-feature* normalization will yield a different PCA result!
    - normalizing each input dimension changes the relative covariances.

5.4 Non-Linear Dimensionality Reduction

Kernel PCA

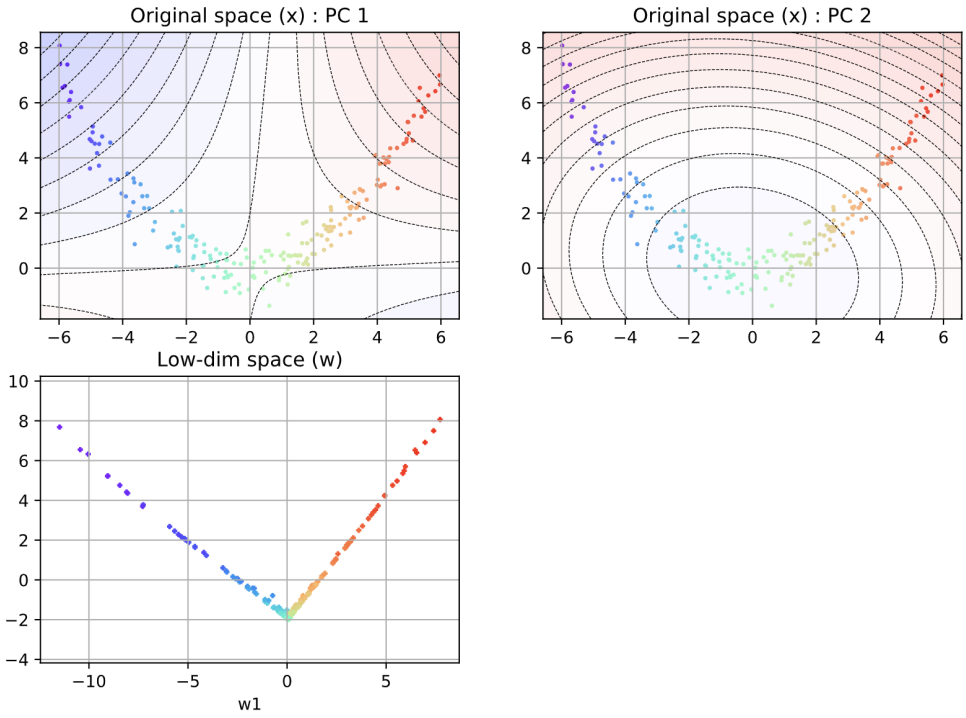
- *How to project to a non-linear surface?*
  - apply a high-dimensional feature transformation to the data
    - $\mathbf{x}_i \Rightarrow \phi(\mathbf{x}_i)$
  - project high-dim data to a linear surface
    - i.e. run PCA on $\phi(\mathbf{x}_i)$
  - in the original space, the projection will be non-linear



Kernel principal components

- kernel principal component $\mathbf{v}$ is a linear combination of high-dim vectors
  - $\mathbf{v} = \sum_{i=1}^{n} \alpha_i \phi(\mathbf{x}_i)$
  - where α_i are learned weights.
- For a new point $\mathbf{x}_*$, the KPCA coefficient for $\mathbf{v}$ is
  - $w = \phi(\mathbf{x}_*)^T \mathbf{v} = \sum_{i=1}^{n} \alpha_i \phi(\mathbf{x}_*)^T \phi(\mathbf{x}_i) = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_*, \mathbf{x}_i)$
  - coefficient is based on similarity to data points belonging to $\mathbf{v}$.
  - using the kernel trick saves computation.

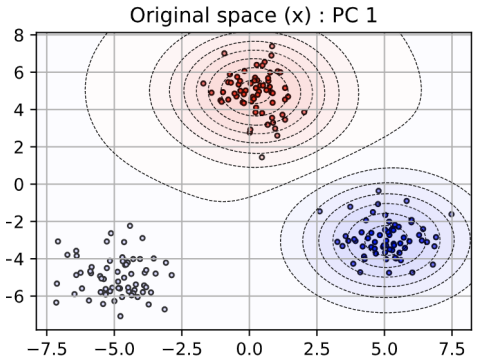
In [ ]:
```
# run KPCA
kpca = decomposition.KernelPCA(n_components=1, kernel='poly', gamma=0.15
, degree=2, coef0=0, n_jobs=-1)
W = kpca.fit_transform(X)
testW = kpca.transform(testX)
```

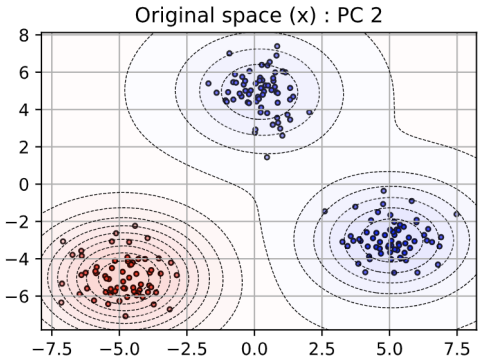


RBF kernel

- principal components separate the data into clusters
- coefficient is distance to clusters

In [ ]:
```
# run KPCA
kpca = decomposition.KernelPCA(n_components=8, kernel='rbf', gamma=0.15,
 n_jobs=-1)
W = kpca.fit_transform(X)
testW = kpca.transform(testX)
```





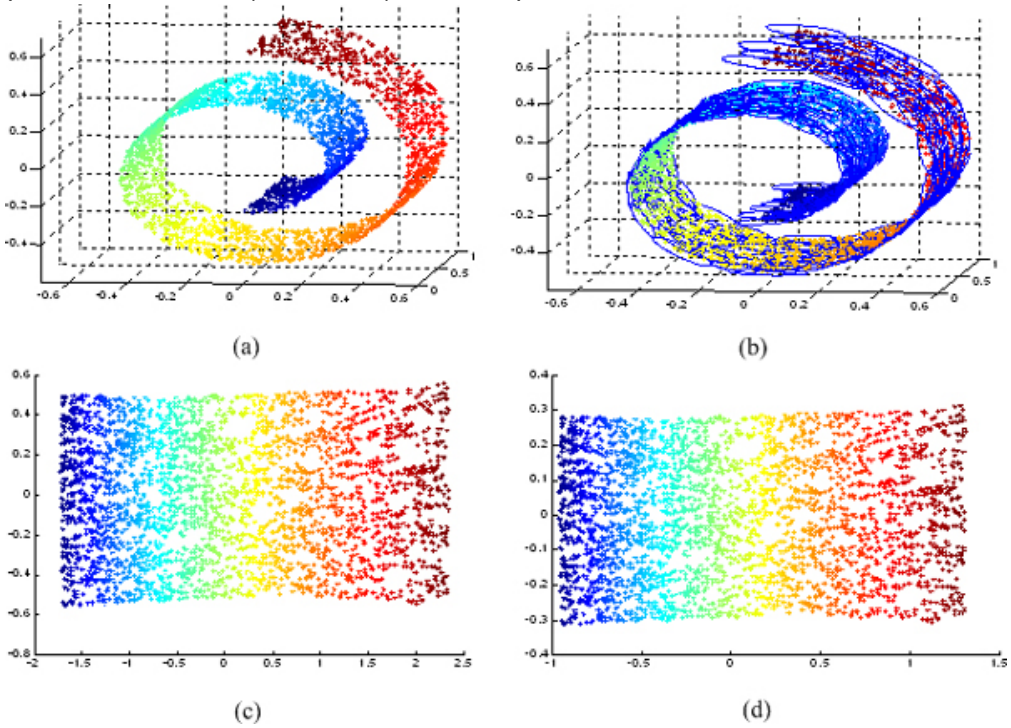
KPCA Summary

- Use kernel trick to perform PCA in high-dimensional space.
  - Coefficients are based on a non-linear projection of the data.
  - The type of projection is based on the kernel function selected.
- Using RBF kernel, KPCA can split the data into clusters.

5.5 Manifold Embedding

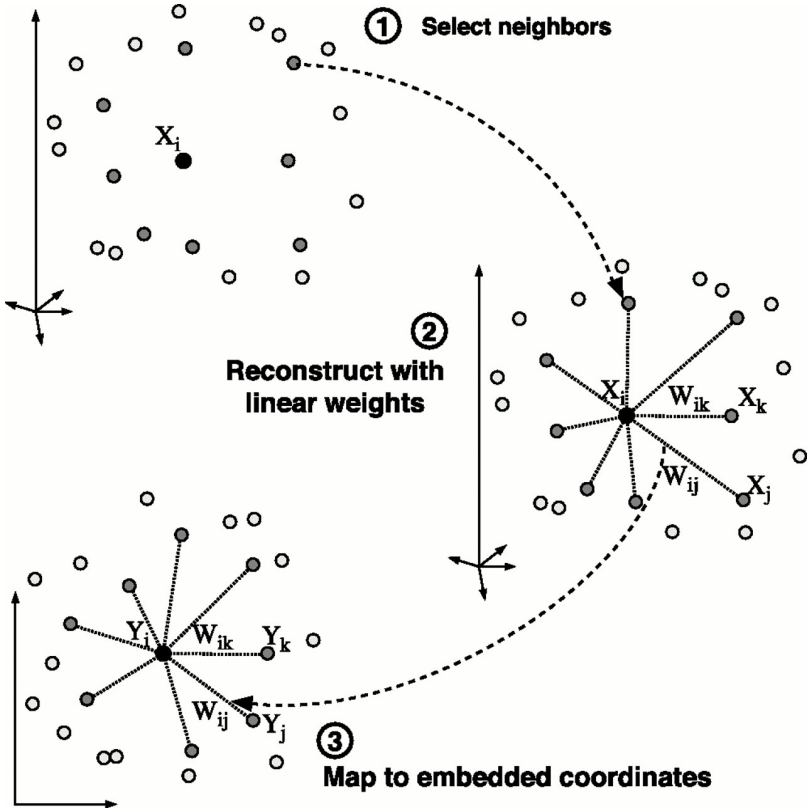
Manifold Embedding

- Reduce high-dimensional data to 2 or 3 dimensions for visualization
- Try to preserve the inherent structure of the data
  - find a set of lower-dim points that optimize some criteria.
- Two types:
  - 1) preserve local neighborhood structure
    - assumes thata lies in a lower-dim manifold; unfold the manifold
  - 2) preserve pairwise distances (similarities) between points

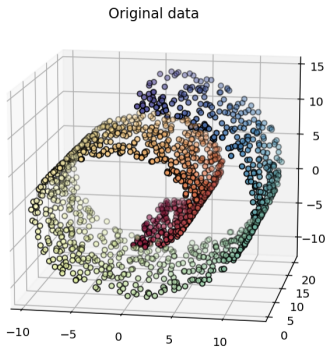


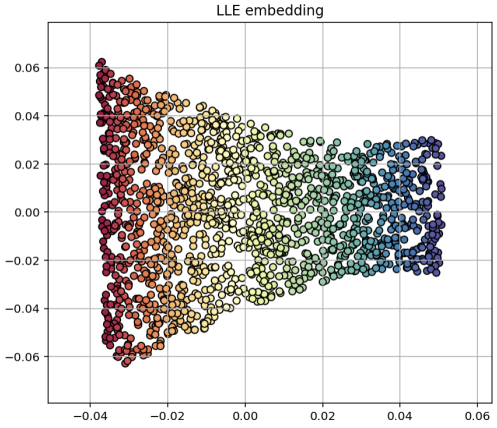
5.5.1 Locally-linear Embedding (LLE)

- **Idea:** preserve linearity within local neighborhoods defined by K nearest neighbors.
  - 1) a point $\mathbf{x}_i$ can be reconstructed by a linear combination of its neighbors N_i.
    - find the weights for the best reconstruction
      - $W^* = \text{argmin}_W \sum_i \|\mathbf{x}_i - \sum_{j \in N_i} w_{i,j}\mathbf{x}_j\|^2$
  - 2) the embedded point $\mathbf{y}_i$ should also have the same local linearity.
    - find the embedded points that best preserve the linearity
      - $Y^* = \text{argmin}_Y \sum_i \|\mathbf{y}_i - \sum_{j \in N_i} w_{i,j}\mathbf{y}_j\|^2$



In [ ]:
```
# n_neighbors = number of nearest neighbors to use for local neighborhood
# n_components = number of dimensions of manifold embedding
lle = manifold.LocallyLinearEmbedding(n_neighbors=12, n_components=2, random_state=121, n_jobs=-1)
Xr = lle.fit_transform(X)
```





Variants of LLE

- **Hessian LLE (HLLE)**: LLE that also uses local curvature information

In [ ]:
```
# set method to 'hessian'
hlle = manifold.LocallyLinearEmbedding(method='hessian', n_neighbors=12,
 n_components=2, random_state=121, n_jobs=-1)
Xr = hlle.fit_transform(X)
```

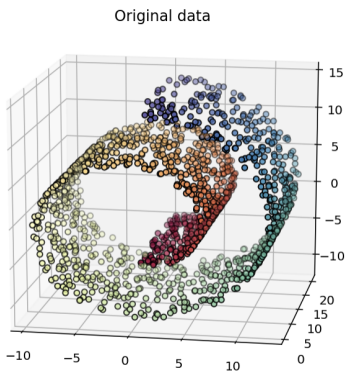
- **Local tangent space alignment (LTSA)**: rather than preserve distances, align local tangent spaces of the neighborhoods.

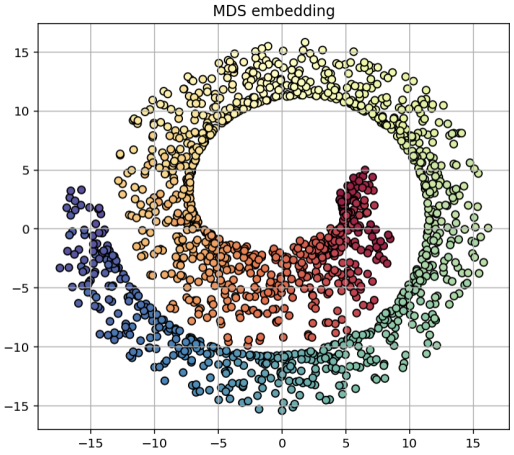
In [ ]:
```
# set method to 'ltsa'
ltsa = manifold.LocallyLinearEmbedding(method='ltsa', n_neighbors=12, n_
components=2, random_state=121, n_jobs=-1)
Xr = ltsa.fit_transform(X)
```

5.5.2 Multidimensional Scaling

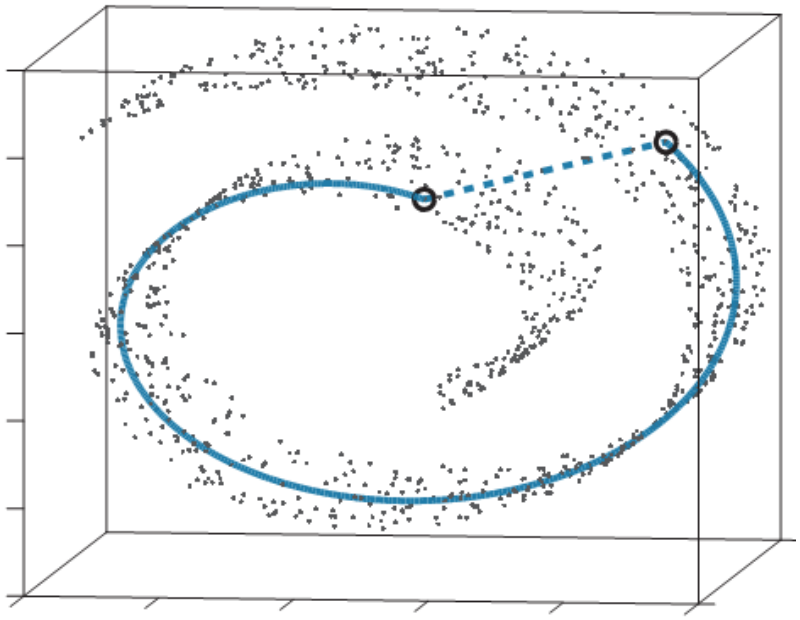
- **Idea:** find a low-dimensional embedding that preserves the pairwise distances between points in original high-dim space.
  - $Y^* = \text{argmin}_Y \sum_{i,j}(d(\mathbf{x}_i, \mathbf{x}_j) - d(\mathbf{y}_i, \mathbf{y}_j)))^2$
    - $\mathbf{x}_i$ are points in original space
    - $\mathbf{y}_i$ are points in the embedding space

In [ ]:
```
mds = manifold.MDS(n_components=2, random_state=1234, n_jobs=-1)
Xr = mds.fit_transform(X)
```



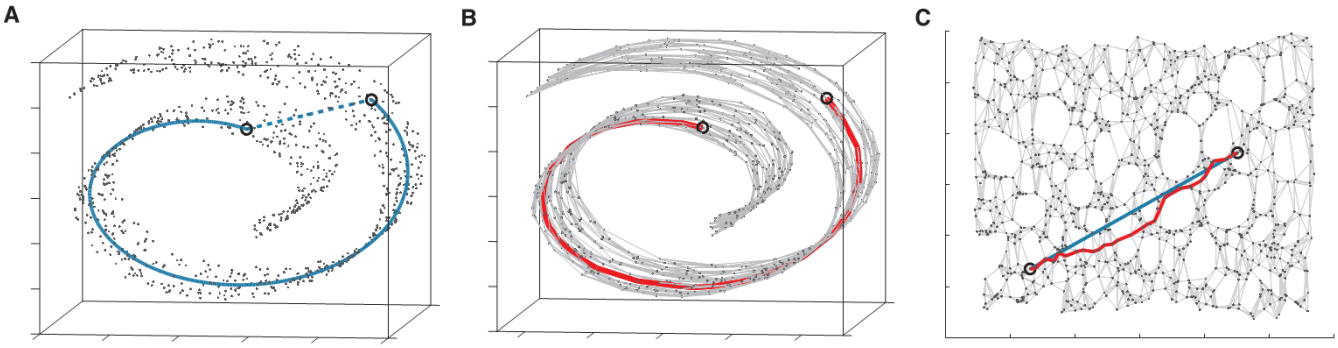


- **Problem**
  - MDS tries to preserve the Euclidean distance between the points.
  - For a manifold, two points may be far away along the manifold (geodesic distance), but close in Euclidean distance.
    - dashed line = Euclidean distance
    - solid line = Geodesic distance

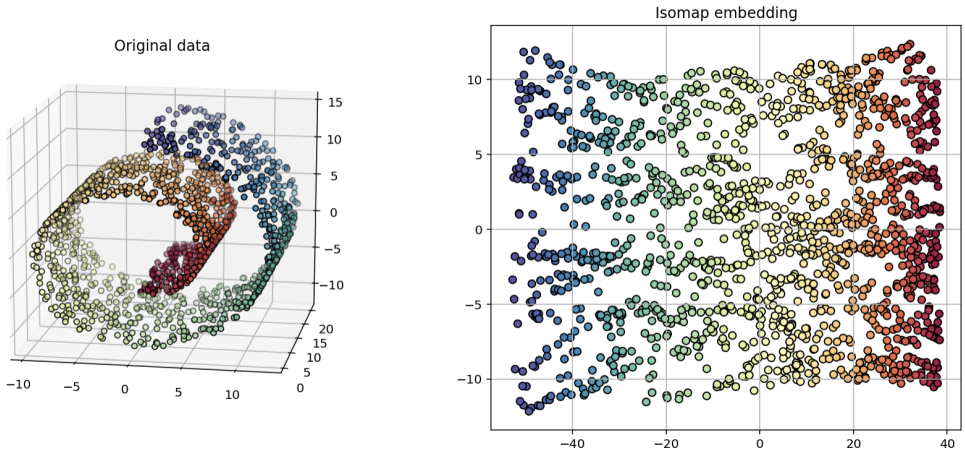


5.5.3 Isomap (Isometric Mapping)

- Find embedding that preserves geodesic distances between points
  - geodesic distance = distances moving on the manifold (figure **A**)
- Approximate manifold by forming a graph (figure **B**)
  - each data point is a node
  - edge is added between nodes if they are K-nearest-neighbors.
  - calculate geodesic distance between 2 points using shortest-paths algorithm (Dijkstra's algorithm).
- Use MDS on the geodesic distances to calculate the embedding (figure **C**).



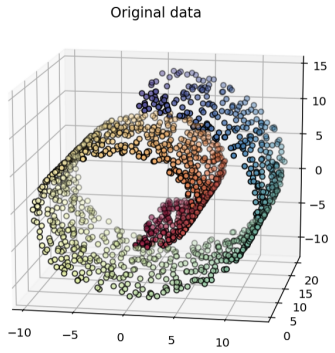
In [ ]:
```
iso = manifold.Isomap(n_neighbors=12, n_components=2, n_jobs=-1)
Xr = iso.fit_transform(X)
```

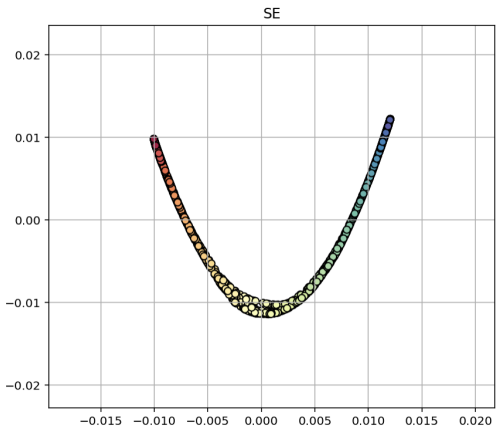


5.5.4 Spectral Embedding (Laplacian Eigenmaps)

- Preserve the neighborhood structure
  - 1) form an affinity (adjacency) matrix W
    - using K nearest neighbors
      - $W_{i,j} = \begin{cases} 1, & \mathbf{x}_i \text{ and } \mathbf{x}_j \text{ are neighbors} \\ 0, & \text{they are not neighbors} \end{cases}$.
    - using RBF kernel
      - $k(\mathbf{x}_i, \mathbf{x}_j)$ represents how close a neighbor.
  - 2) keep neighboring points close together in the embedding.
    - penalize embedding points that are far apart, but should be neighbors
      - $Y^* = \text{argmin}_Y \sum_{i,j} W_{i,j}(y_i - y_j)^2$

In [ ]:
```
# using nearest-neighbors for affinity
spe = manifold.SpectralEmbedding(n_components=2, affinity='nearest_neigh
bors',
                                 random_state=1234, n_neighbors=12, n_jo
bs=-1)
Xr = spe.fit_transform(X)
```





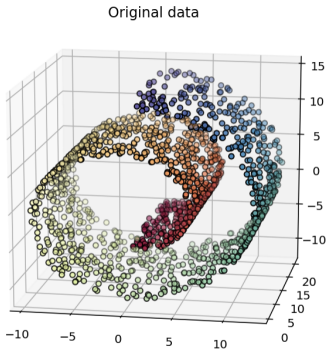
In [ ]:
```
# using RBF kernel for affinity
# gamma is the inverse bandwidth
spe = manifold.SpectralEmbedding(n_components=2, affinity='rbf',
                                 gamma=0.2, random_state=1234)
Xr = spe.fit_transform(X)
```

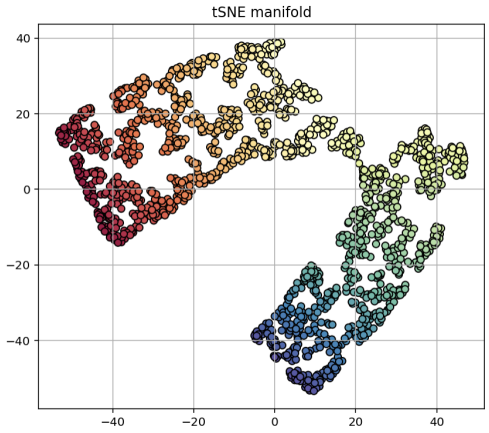
5.5.5 t-Distributed Stochastic Neighbor Embedding (t-SNE)

- Preserve pairwise similarities
  - 1) treat similarities in original space as probabilities
    - the probability that j is a neighbor of i: $p_{j|i} = \frac{k_i(\mathbf{x}_i,\mathbf{x_j})}{\sum_{k\neq i} k_i(\mathbf{x}_i,\mathbf{x}_k)}$
      - $k_i(\mathbf{x}_i,\mathbf{x}_j)$ is the RBF kernel with inverse bandwidth γ_i.
    - the similarity between i and j: $p_{i,j} = \frac{p_{j|i}+p_{i|j}}{2N}$
  - 2) similarities in the embedding space (student-t distribution)
    - the probability that i and j are neighbors: $q_{i,j} = \frac{(1+\|\mathbf{y}_i-\mathbf{y_j}\|^2)^{-1}}{\sum_{k\neq l}(1+\|\mathbf{y}_l-\mathbf{y_k}\|^2)^{-1}}$
  - 3) find the embedding points that preserve the probability structure:
    - $Y = \text{argmin}_Y \sum_{i\neq j} p_{i,j} \log \frac{p_{i,j}}{q_{i,j}}$
    - the objective is the Kullback-Leibler (KL) divergence, which is a measure of similarity between two probability distributions.

- Tends to group together similar items
  - student-t distribution is heavy-tailed
  - "moderate" distances in original space are converted to "large" distances in embedding space

In [ ]:
```
# perplexity = similar to number of neighbors
tsne = manifold.TSNE(n_components=2, perplexity=30.0, random_state=11)
Xr = tsne.fit_transform(X)
```





- The `perplexity` parameter controls the number of groups (size of groups).
  - small perplexity: form more groups
  - large perplexity: form less groups

5.6 Non-Linear Dimensionality Reduction - Summary

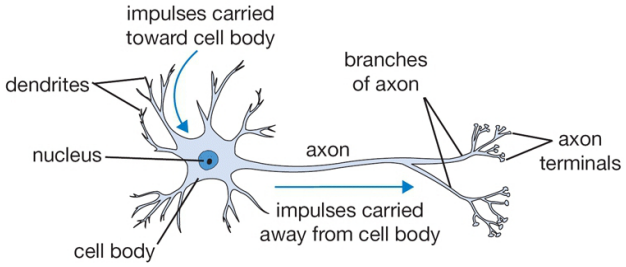
- **Goal:** given high-dim data, find a low-dim representation.
  - try to preserve inherent structure of data
- Two types of approaches:
  - *Non-linear dimensionality reduction*
    - calculate low-dim coefficients using non-linear projections (kernel).
  - *Manifold embedding*
    - Optimize over the embedding (low-dim) points to preserve some property from the high-dim points.
    - After training, manifold embedding methods cannot transform a novel (new) point.

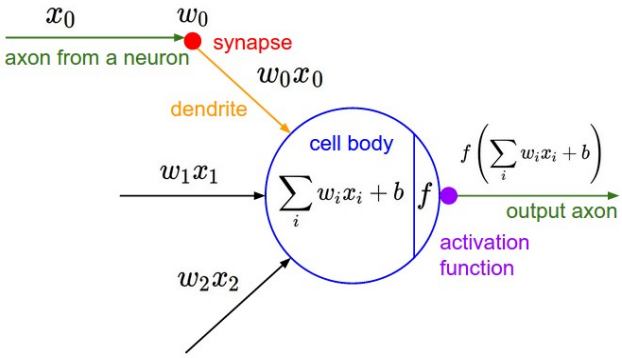
Name	Objective	Advantages	Disadvantages
Kernel principal component analysis (KPCA)	PCA in high-dim feature space (kernel trick)	- can transform novel data	- need to calculate kernel matrix; - need to keep training data around for embedding new points.
Locally-Linear Embedding (LLE)	preserve local neighborhood distances	- good for single low-dim manifold	- cannot embed a novel point.
Hessian LLE	w/ local curvature information.	- good for single low-dim manifold	- cannot embed a novel point.
Local tangent space alignment (LTSA)	align tangent spaces	- good for single low-dim manifold	- cannot embed a novel point.
Multi-dimensional scaling (MDS)	preserve pairwise distances	- good for viewing the space.	- cannot embed a novel point.
Isometric mapping (Isomap)	preserve geodesic distances	- good for single low-dim manifold	- cannot embed a novel point.
Spectral embedding (SE)	"PCA" on graph representation	- good for single low-dim manifold	- cannot embed a novel point.
t-distributed Stochastic Neighbor Embedding (t-SNE)	preserve pairwise similarities	- can discover similar items	- cannot embed a novel point.

6. Deep Learning

Original idea

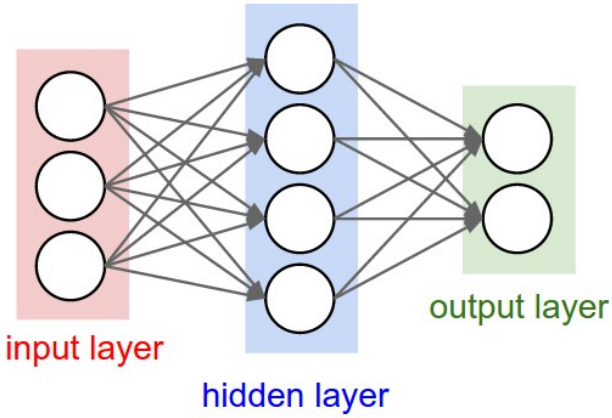
- *Perceptron*
  - Warren McCulloch and Walter Pitts (1943), Rosenblatt (1957)
  - Simulate a neuron in the brain
    - 1) take binary inputs (input from nearby neurons)
    - 2) multiply by weights (synapses, dendrites)
    - 3) sum and threshold to get binary output (output axon)
  - Train weights from data.

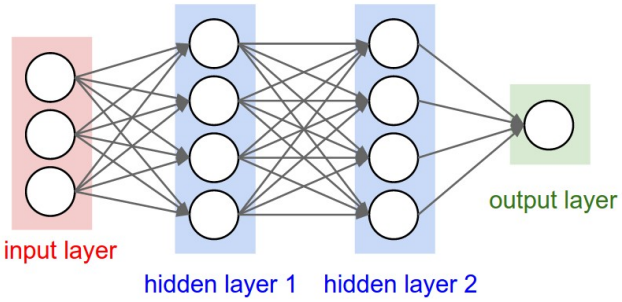




6.1 Multi-layer Perceptron

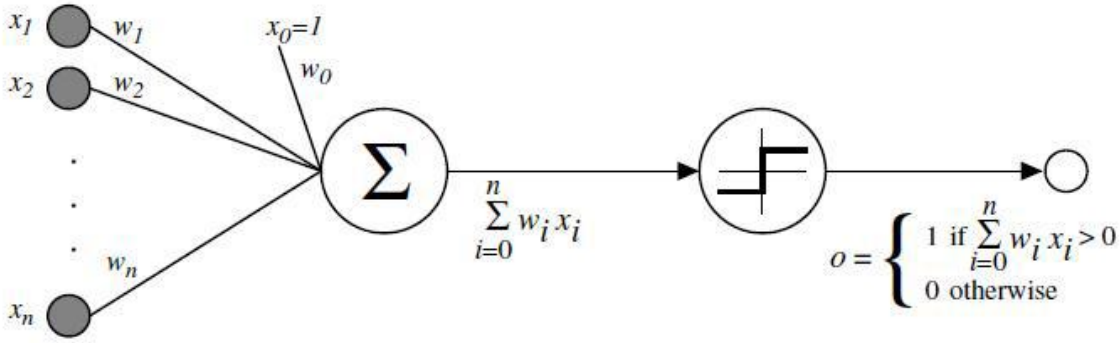
- Add *hidden* layers between input and output neurons
  - each layer extracts some features from the previous layers
  - can represent complex non-linear functions
  - train weights using *backpropagation* algorithm. (1970-80s)
  - (now called a *neural network*)





- **Problem:**
  - difficult to train.
  - sensitive to initialization.
  - computationally expensive (at the time).

- Formally,
  - $y = f(\sum_{j=0}^{d} w_j x_j) = f(\mathbf{w}^T \mathbf{x})$
  - $\mathbf{w}$ is the weight vector.
  - $f(a)$ is the activation function
    - $f(a) = \begin{cases} 1, & a > 0 \\ 0, & \text{otherwise} \end{cases}$



Perceptron training criteria

- Train the perceptron on data $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$
- Only look at the points that are misclassified.
  - Loss is based on how badly misclassified
  - $E(\mathbf{w}) = \sum_{i=1}^{N} \begin{cases} -y_i \mathbf{w}^T \mathbf{x}_i, & \mathbf{x}_i \text{ is misclassified} \\ 0, & \text{otherwise} \end{cases}$
- Minimize the loss: $\mathbf{w}^* = \text{argmin}_{\mathbf{w}} E(\mathbf{w})$

Training algorithm

- Computers were slow back then...only look at one data point at a time and use gradient descent.
- **Perceptron Algorithm**
  - For each point $\mathbf{x}_i$,
    - If the point $\mathbf{x}_i$ is misclassified,
      - Update weights: $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$
  - Repeat until no more points are misclassified

Perceptron Algorithm

- Fails to converge if data is not linearly separable
- Rosenblatt proved that the algorithm will converge if the data is linearly separable.
  - the number of iterations is inversely proportional to the separation (margin) between classes.
  - *This was one of the first machine learning results!*
- Different initializations can yield different weights.

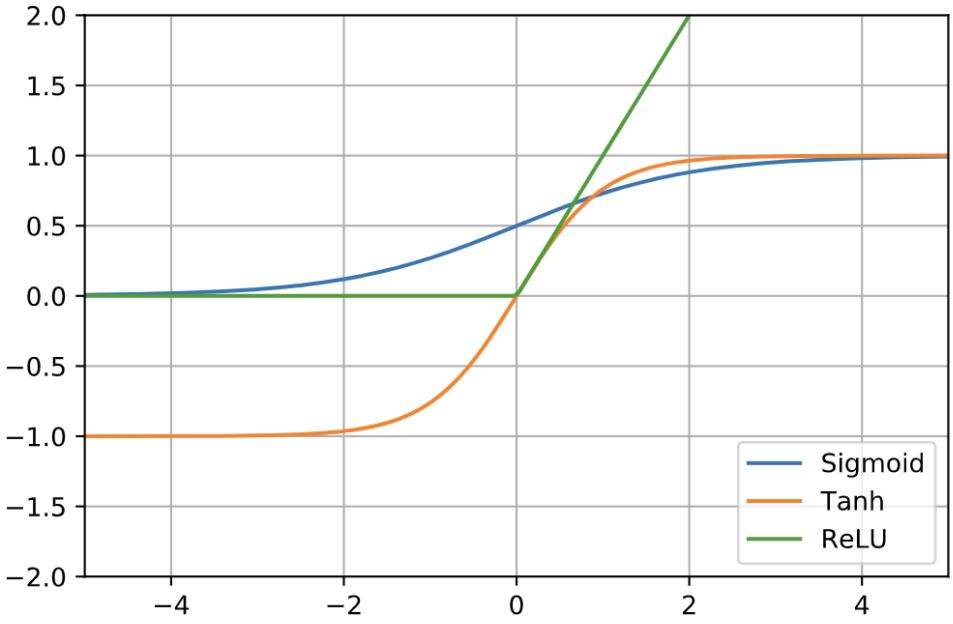
Multi-layer Perceptron

- Add hidden layers between the inputs and outputs
  - each hidden node is a Perceptron (with its own set of weights)
    - its inputs are the outputs from previous layer
    - extracts a feature pattern from the previous layer
  - can model more complex functions

- Formally, for one layer:
  - $\mathbf{h} = f(\mathbf{W}^T\mathbf{x})$
    - Weight matrix $\mathbf{W}$ - one column for each node
    - Input $\mathbf{x}$ - from previous layer
    - Output $\mathbf{h}$ - to next layer
    - $f(a)$ is the activation function - applied to each dimension to get output

Activation functions

- There are different types of activation functions:
  - *Sigmoid* - output [0,1]
  - *Tanh* - output [-1,1]
  - *Rectifier Linear Unit (ReLU)* - output $[0,\infty]$



- Activation functions specifically for output nodes:
  - *Linear* - output for regression
  - *Softmax* - output for classification (same as multi-class logistic regression)
- Each layer can use a different activation function.

Which activation function is best?

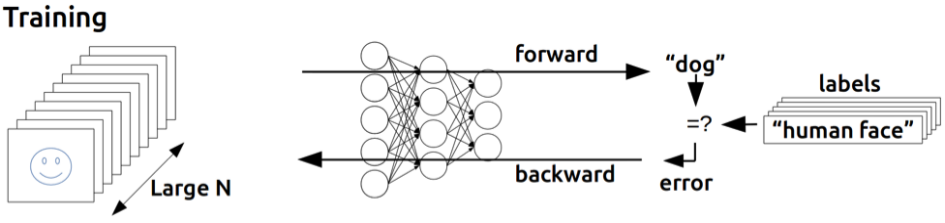
- In the early days, only the Sigmoid and Tanh activation functions were used.
  - these were notoriously hard to train with.
    - "vanishing gradient" problem
- Recently, ReLU has become very popular.
  - easier to train with - no "vanishing gradient"
  - faster - don't need to calculate exponential
  - sparse representation - most nodes will output zero.

Training an MLP

- For classification, we use the cross-entropy loss
  - $E = -\sum_{j=1}^{K} y_j \log \hat{y}_j$
    - y_j is 1 for the true class, and 0 otherwise
    - $\hat{y}_j$ is the softmax output for the j-th class
- Use gradient descent as before:
  - $w_{ij} \leftarrow w_{ij} - \eta \frac{dE}{dw_{ij}}$
    - layer i, node j
  - η is the learning rate
    - controls convergence rate
      - too small --> converges very slowly
      - too large --> possibly doesn't converge

Backpropagation (backward propagation)

- Do a forward pass to calculate the prediction
- Do a backward pass to update weights that were responsible for an error



Gradient descent with the chain-rule

- Suppose we have a 2-layer network
  - E is the cost function
  - g_1 , g_2 are the output functions of the two layers
    - $g_j(\mathbf{x}) = f(\mathbf{W}_j^T \mathbf{x})$
    - $\mathbf{W}_1$, $\mathbf{W}_2$ are the weight matrices
- Prediction for input $\mathbf{x}$: $y = g_2(g_1(\mathbf{x}))$
- Cost for input $\mathbf{x}$: $E(\mathbf{x}) = E(g_2(g_1(\mathbf{x})))$

- Apply the chain rule to get the gradients of weights in layer
  - $\frac{dE(\mathbf{x})}{d\mathbf{W}_2} = \frac{dE}{dg_2}\frac{dg_2}{d\mathbf{W}_2}$
  - $\frac{dE(\mathbf{x})}{d\mathbf{W}_1} = \frac{dE}{dg_2}\frac{dg_2}{dg_1}\frac{dg_1}{d\mathbf{W}_1}$
- Defines a set of recursive relationships
  - 1) calculate the output of each node from first to last layer
  - 2) calculate the gradient of each node from last to first layer
- NOTE: the gradients multiply in each layer!
  - if two gradients are small (<1), their product will be even smaller. This is the "vanishing gradient" problem.

Stochastic Gradient Descent (SGD)

- The datasets needed to train NN are typically very large
- Use SGD so that only a small portion of the dataset is needed at a time
  - Each small portion is called a *mini-batch*
  - Use a *momentum* term, which averages the current gradient with those from previous mini-batches.
  - One complete pass through the data is called an *epoch*.

Other Tricks

- Normalize the inputs to [-1,1] or [0,1]
  - improves numerical stability.
- Separate the training set into training and validation
  - use the training set to run backpropagation
  - test the NN on the validation set for diagnostics
    - check for convergence - adjust learning rate if necessary
    - check for diverging loss - adjust learning rate
    - stopping criteria - stop when no change in the validation error.
    - decay learning rate after each epoch.

In [ ]:
```
# use TensorFlow backend
%env KERAS_BACKEND=tensorflow
import keras
import tensorflow
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Dropout, Conv2D, Flatten, In
put
import logging
logging.basicConfig()
import struct
```

In [ ]:
```
# convert class labels to binary indicators (required for Keras)
Yb = keras.utils.np_utils.to_categorical(Y)
```

In [ ]:
```
# initialize random seed
random.seed(4487); tensorflow.set_random_seed(4487)

# build the network
nn = Sequential()
nn.add(Dense(units=20, input_dim=2, activation='relu'))
nn.add(Dense(units=2, activation='softmax'))

# compile and fit the network
nn.compile(loss=keras.losses.categorical_crossentropy,
    optimizer=keras.optimizers.SGD(lr=0.3, momentum=0.9, nesterov=True))
history = nn.fit(X, Yb, epochs=100, batch_size=32, validation_split=0.1,
 verbose=False)
```

In [ ]:
```
predY = nn.predict_classes(testX, verbose=False)
```

Overfitting

- Continuous training will sometimes lead to overfitting
  - the training loss decreases, but the validation loss increases

Early stopping

- Training can stopped when the validation loss is stable for a number of iterations
  - stable means change below a threshold
  - this is to prevent overfitting the training data.
  - we can limit the number of iterations.

In [ ]:
```
# setup early stopping callback function
earlystop = keras.callbacks.EarlyStopping(
    monitor='val_loss',     # look at the validation loss
    min_delta=0.0001,       # threshold to consider as no change
    patience=5,             # stop if 5 epochs with no change
    verbose=1, mode='auto'
)
callbacks_list = [earlystop]
```

Universal Approximation Theorem

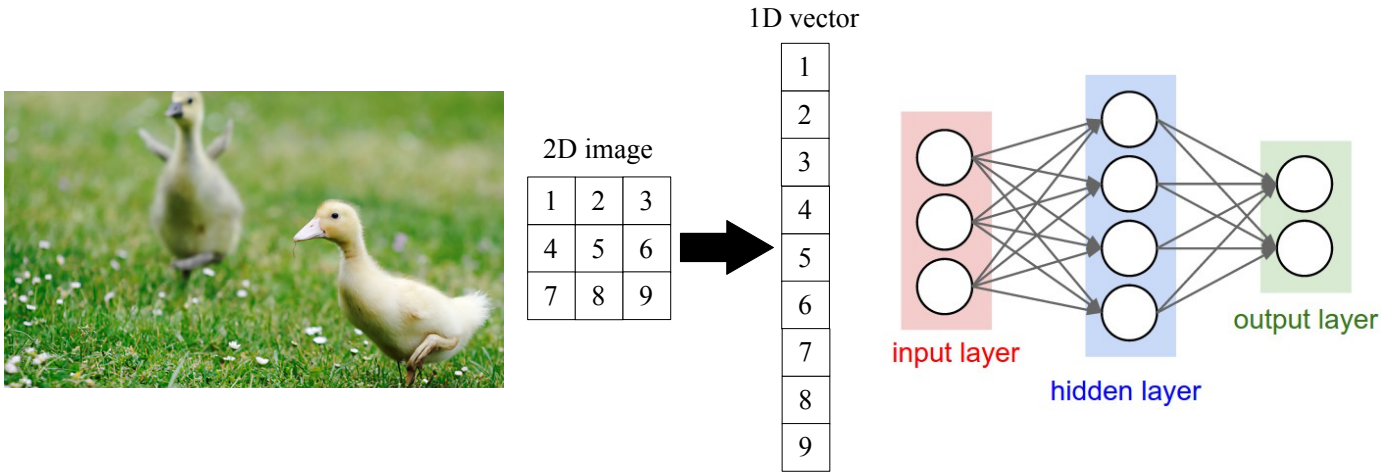
- Cybenko (1989), Hornik (1991)
  - *A multi-layer perceptron with a single hidden layer and a finite number of nodes can approximate any continuous function.*
    - The number of nodes needed might be very large.
    - Doesn't say anything about how difficult it is to train it.
- Deep learning corrolary
  - A deep network can learn the same function using less nodes.
  - Given the same number of nodes, a deep network can learn more complex functions.
    - Doesn't say anything about how difficult it is to train it.

In [ ]:
```
params = nn.get_layer(index=0).get_weights()
```

6.2 Convolutional neural network (CNN)

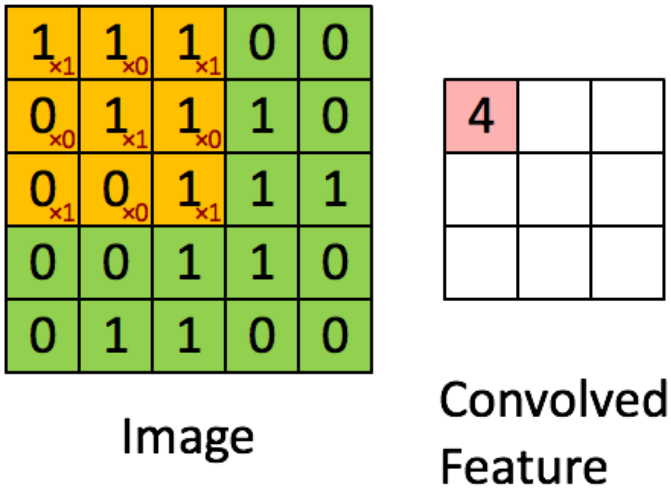
Image Inputs and Neural Networks

- In the MLP, each node takes inputs from all other nodes in the previous layer.
  - For image input, we transform the image into a vector, which is the input into the MLP.

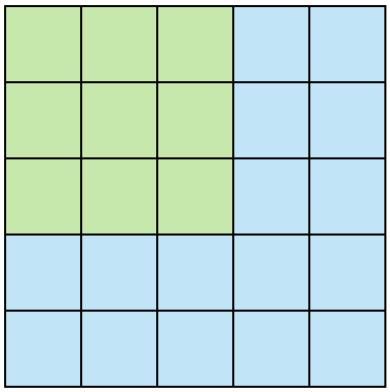


Convolutional Neural Network (CNN)

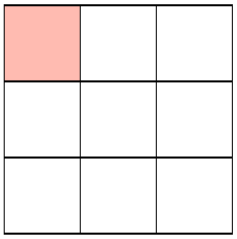
- Use the spatial structure of the image
- 2D convolution filter
  - the weights $\mathbf{W}$ form a 2D filter template
  - filter response: $h = f(\sum_{x,y} W_{x,y} P_{x,y})$
    - $\mathbf{P}$ is an image patch with the same size as $\mathbf{W}$.
- Convolution feature map
  - pass a sliding window over the image, and apply filter to get a *feature map*.



- Convolution modes
  - **"valid"** mode - only compute feature where convolution filter has valid image values.
    - size of feature map is reduced.

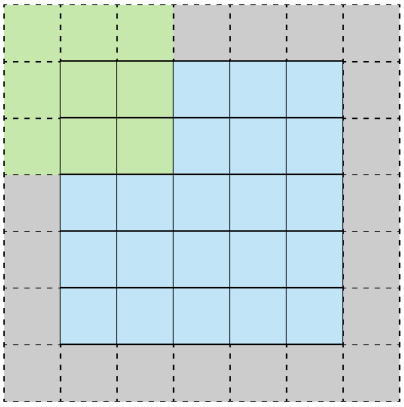


Stride 1

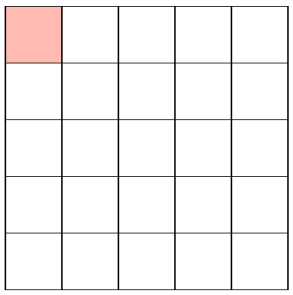


Feature Map

- Convolution modes
  - **"same"** mode - zero-pad the border of the image
    - feature map is the same size as the input image.

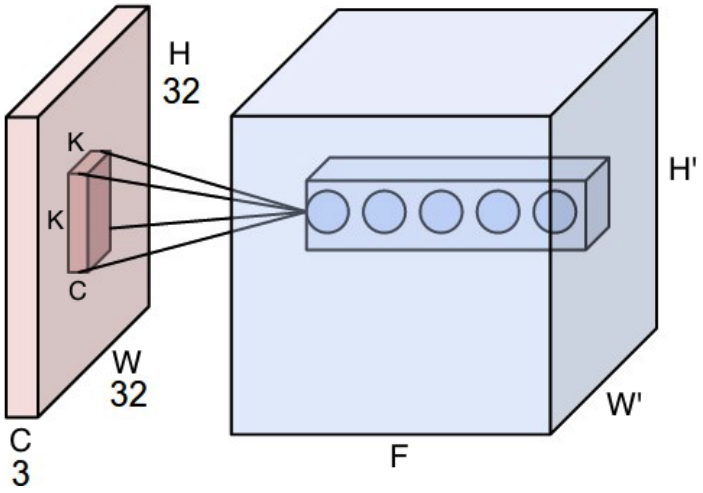


Stride 1 with Padding

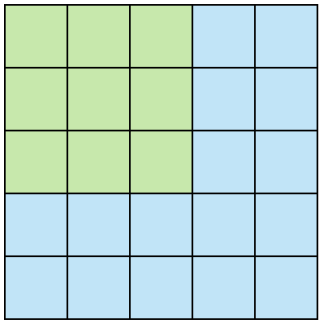


Feature Map

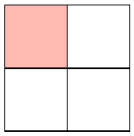
- `(Usually "same" is better since it looks at structures around border)`
- Convolutional layer
  - **Input:** HxW image with C channels
    - For example, in the first layer, C=3 for RGB channels.
    - defines a 3D volume: C x H x W
  - **Features:** apply F convolution filters to get F feature maps.
    - Uses 3D convolution filters: weight volume is C x K x K
    - K is the spatial extent of the filter
  - **Output:** a feature map with F channels
    - defines a 3D volume: F x H' x W'



- Spatial sub-sampling
  - reduce the feature map size by subsampling feature maps between convolutional layers
    - *stride* for convolution filter - step size when moving the windows across the image.

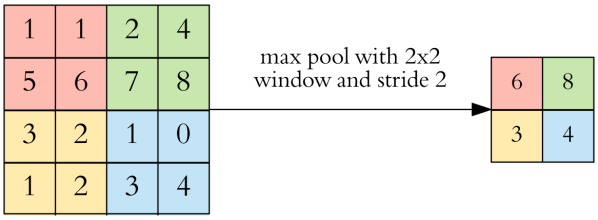


Stride 2



Feature Map

    - *max-pooling* - use the maximum over the pooling window
      - gathers features together, makes it robust to small changes in configuration of features



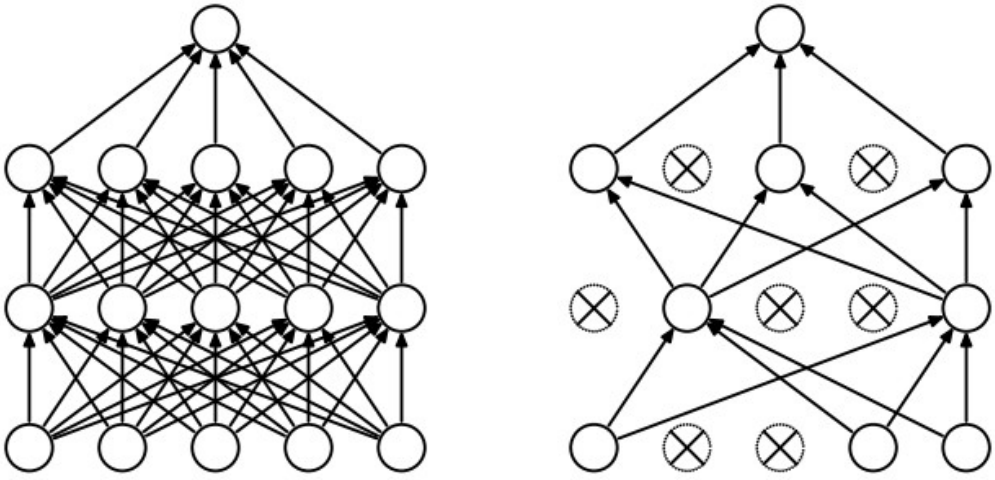
Advantages of Convolution Layers

- The convolutional filters extract the same features throughout the image.
  - Good when the object can appear in different locations of the image.
- Pooling makes it robust to changes in feature configuration, and translation of the object.
- The number of parameters is small compared to Dense (Fully-connected) layer
  - Example: input is C x H x W, and output is F x H x W
    - Number of MLP parameters: (CHW+1) x (FHW)
    - Number of CNN parameters: (CKK+1) x (FKK)

In [ ]:
```
nn.add(Conv2D(10, (5,5), strides=(2,2), activation='relu',
              input_shape=(1,28,28),
              padding='same', data_format='channels_first'))
```

Regularization with "Dropout"

- During training, randomly "drop out" each node with probability p
  - a dropped-out node is not used for calculating the prediction or weight updating.
  - trains a reduced network in each iteration.



(a) Standard Neural Net (b) After applying dropout.

- During test time, use all the nodes for prediction and scale output by p.
  - Similar to creating an ensemble of networks, and then averaging the predictions.
- Prevents overfitting
  - also improves the training time.

In [ ]:
```
nn.add(Dropout(rate=0.5, seed=44))   # dropout layer! (need to specify t
he seed)
```

Regularization with Weight Decay

- Another way to regularize the network is to use "weight decay"
  - Add a penalty term to the loss function
    - larger weights impose higher penalty
    - $L = Loss + \alpha \sum_i w_i^2$

In [ ]:
```
nn.add(Conv2D(80, (5,5), strides=(1,1), activation='relu',
              kernel_regularizer=keras.regularizers.l2(0.0001),  # L2 re
gularizer
              padding='same', data_format='channels_first'))
```

Data augmentation

- artificially permute the data to increase the dataset size
  - goal: make the network invariant to the permutations
  - examples: translate image, flip image, add pixel noise, rotate image, deform image, etc.

In [ ]:
```
from keras.preprocessing.image import ImageDataGenerator

# build the data augmenter
datagen = ImageDataGenerator(
    rotation_range=10,          # image rotation
    width_shift_range=0.2,      # image shifting
    height_shift_range=0.2,     # image shifting
    shear_range=0.1,            # shear transformation
    zoom_range=0.1,             # zooming
    data_format='channels_first')

# fit (required for some normalization augmentations)
datagen.fit(vtrainI)
```

In [ ]:
```
# pass data through augmentor and fit
# runs data-generator and fit in parallel
history = nn.fit_generator(
            datagen.flow(vtrainI, vtrainYb, batch_size=50),  # data from
 generator
            steps_per_epoch=len(vtrainI)/50,    # should be number of ba
tches per epoch
            epochs=100,
            callbacks=callbacks_list,
            validation_data=validsetI, verbose=False)
```

Data augmentation with noise

- Also add per-pixel noise to the image for data augmentation.
  - define a function to add noise
  - set it as the `preprocessing_function`

In [ ]:
```
def add_gauss_noise(X, sigma2=0.05):
    # add Gaussian noise with zero mean, and variance sigma2
    return X + random.normal(0, sigma2, X.shape)

# build the data augmenter
datagen = ImageDataGenerator(
    rotation_range=10,          # image rotation
    width_shift_range=0.2,      # image shifting
    height_shift_range=0.2,     # image shifting
    shear_range=0.1,            # shear transformation
    zoom_range=0.1,             # zooming
    preprocessing_function=add_gauss_noise,
    data_format='channels_first')

# fit (required for some normalization augmentations)
datagen.fit(vtrainI)
```

Summary

- **Different types of neural networks**
  - *Perceptron* - single node (similar to logistic regression)
  - *Multi-layer perceptron (MLP)* - collection of perceptrons in layers
    - also called *Fully-connected layer*
  - *Convolutional neural network (CNN)* - convolution filters for extracting local image features
- **Training**
  - optimize loss function using stochastic gradient descent
- **Advantages**
  - lots of parameters - large capacity to learn from large amounts of data
- **Disadvantages**
  - lots of parameters - easy to overfit data
    - need to regularize parameters (dropout, L2)
    - need to monitor the training process
  - sensitive to initialization, learning rate, training algorithm.

6.3 Image Classification and Deep Architectures

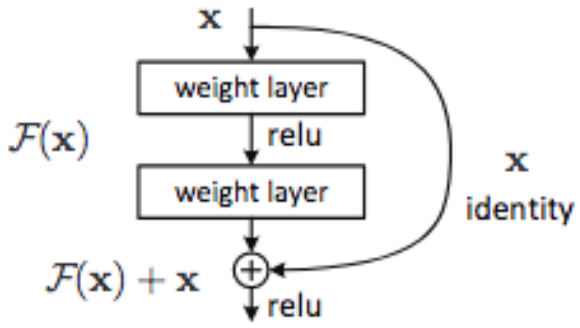
Image Classifiers in Keras

- Keras includes several pre-trained image classifier networks
  - VGG16, VGG19, ResNet50, InceptionV3, ...
  - already trained on ImageNet
- Can use these networks to:
  - perform image classification (for 1000 classes only)
  - extract image features for our own task
  - transfer learning - modify/adapt a pre-trained network for our own task

In [ ]:
```
# use TensorFlow backend
%env KERAS_BACKEND=tensorflow
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Dropout, Conv2D, Flatten, \
                 Input, MaxPooling2D, UpSampling2D, Lambda, Reshape, Batc
hNormalization, \
                 GlobalAveragePooling2D
import keras
import tensorflow
import logging
logging.basicConfig()
import struct

# use channels first representation for images
from keras import backend as K
K.set_image_data_format('channels_first')

from keras.callbacks import TensorBoard
```

Residual Learning

- The network is learning a function (image to class)
  - build the function block-by-block
  - each block learns a residual, which is added to the previous block
    - keep all the previous information and make small changes with the residual



- Create an instance of the ResNet50 model
  - trained on ImageNet
  - will download the model if loading for the first time.

In [ ]:
```
# contains the model and support functions for ResNet50
import keras.applications.resnet50 as resnet
from keras.preprocessing import image


# create an instance of the model
model = resnet.ResNet50(weights='imagenet')
model.summary()
```

Computing Image Features

- Use the pre-trained network as a feature extractor
  - remove the last layer (the classifier)
  - apply average pooling on the feature map
    - 7x7x2048 --> 1x2048

In [ ]:
```
# create an instance of the model w/o the last layer
model_f = resnet.ResNet50(weights='imagenet', include_top=False, pooling
='avg')
```

Transfer learning and fine-tuning

- The network can also be "fine-tuned" for a new image classification task.
  - This is called transfer learning.
- Rather than retrain the whole network,
  - fix the lower layers that extract features (no need to train them since they are good features)
  - train the last few layers to perform the new task
- Does not need as much data, compared to the original ImageNet.

In [ ]:
```
random.seed(4487); tensorflow.set_random_seed(4487)

# create the base pre-trained model with-out the classifier
base_model = resnet.ResNet50(weights='imagenet', include_top=False)

# start with the output of the ResNet50 (7x7x2048)
x = base_model.output

# for each channel, average all the features (1x2048)
x = GlobalAveragePooling2D()(x)

# fully-connected layer (1 x32)
# (only two classes so don't need so many)
x = Dense(32, activation='relu')(x)

# finally, the softmax for the classifier (2 classes)
predictions = Dense(2, activation='softmax')(x)
```

- Create the Model
  - specify the input and output layers
  - the network is everything in between
- Fix the weights of the layers of ResNet so they will not change during training

In [ ]:
```
# build the model for training
# - need to specify the input layer and the output layer
model_ft = Model(inputs=base_model.input, outputs=predictions)

# fix the layers of the ResNet50.
for layer in base_model.layers:
    layer.trainable = False

# compile the model - only the layers that we added will be trained
model_ft.compile(optimizer='rmsprop',
                 loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
from keras.preprocessing.image import ImageDataGenerator

def add_gauss_noise(X, sigma2=0.05):
    # add Gaussian noise with zero mean, and variance sigma2
    return X + random.normal(0, sigma2, X.shape)

# build the data augmenter
datagen = ImageDataGenerator(
    rotation_range=10,          # image rotation
    width_shift_range=0.2,      # image shifting
    height_shift_range=0.2,     # image shifting
    shear_range=0.1,            # shear transformation
    zoom_range=0.1,             # zooming
    horizontal_flip=True,
    preprocessing_function=add_gauss_noise,
)

# fit (required for some normalization augmentations)
datagen.fit(vtrainXim)
```

In [ ]:
```
# train the model on the new data for a few epochs
bsize = 32
callbacks_list = []
history = model_ft.fit_generator(
            datagen.flow(vtrainXim, vtrainYb, batch_size=bsize),  # data
 from generator
            steps_per_epoch=len(vtrainXim)/bsize,    # should be number
 of batches per epoch
            epochs=20,
            callbacks=callbacks_list,
            validation_data=validset, verbose=True)
```

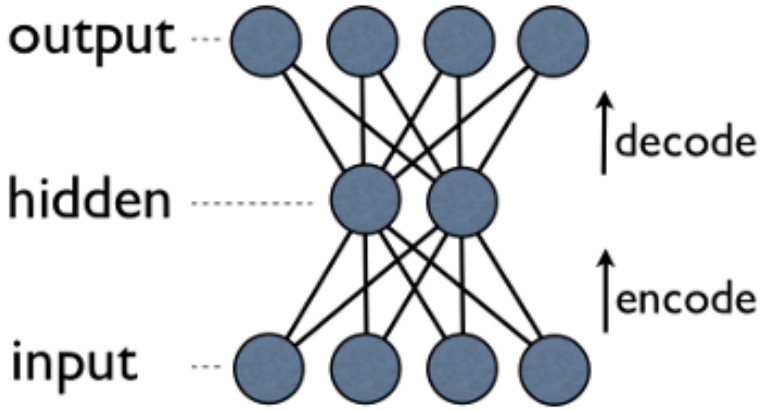
In [ ]:
```
predYscore = model_ft.predict(testXim, verbose=False)
predY = argmax(predYscore, axis=1)
acc = metrics.accuracy_score(testY, predY)
```

6.4 Neural Networks and Unsupervised Learning

- How to use NN for dimensionality reduction or clustering?

Denoising Autoencoder

- Use the hidden layer as the lower-dimensional representation (code)
- Train the network to "encode" and "decode"
  - randomly corrupt the input (by setting values to 0)
  - run it through the encoding-decoding network
  - minimize the difference between the output and the original input



- Train the autoencoder
  - specify the number of hidden nodes
  - corrupt the image using Dropout
    - corruption level = percentage of inputs that are zeroed out.

- Use `Model` class.
  - pass input and output layers.
  - The model consists of everything between input and output.

In [ ]:
```
# initialize random seed
random.seed(4487); tensorflow.set_random_seed(4487)

# Build the Encoder model
input_img = Input(shape=(784,))
corrupted_img = Dropout(rate=0.3)(input_img)
encoded = Dense(10, activation='relu')(corrupted_img)
encoder = Model(input_img, encoded)

# Build the Decoder model
encoded_input = Input(shape=(10,))
decoded = Dense(784, activation='sigmoid')(encoded_input)
decoder = Model(encoded_input, decoded)

# build the full autoencoder model
autoencoder = Model(input_img, decoder(encoder(input_img)))
```

In [ ]:
```
# early stopping criteria
earlystop = keras.callbacks.EarlyStopping(monitor='val_loss',
                              min_delta=0.0001, patience=10,
                              verbose=1, mode='auto')

# compile and fit the network
autoencoder.compile(loss=keras.losses.binary_crossentropy,
           optimizer=keras.optimizers.SGD(lr=0.2, momentum=0.9, nesterov
=True))
```

In [ ]:
```
# fit the model: the input and output are the same
# write to a log directory to see training process
history = autoencoder.fit(vtrainXraw, vtrainXraw,
                  epochs=20, batch_size=50,
                  callbacks=[earlystop, TensorBoard(log_dir='./logs/ae'
)],
                  validation_data=(validXraw, validXraw), verbose=False)
```

- Run tensorboard in console: `tensorboard --logdir=./logs.ae`

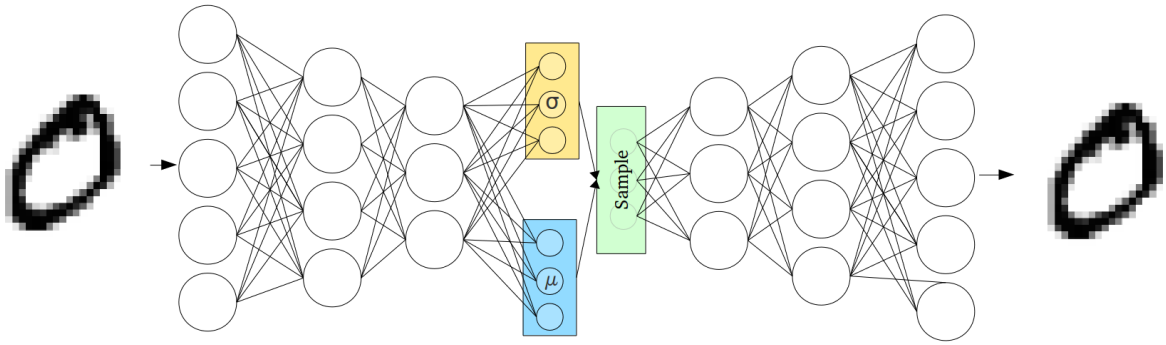
In [ ]:
```
Z = encoder.predict(trainXraw)
Z.shape
```

- Visualize the reconstruction of the input image

In [ ]:
```
testXrecon = decoder.predict(encoder.predict(testXraw))
```

Variational AutoEncoder (VAE)

- The standard autoencoder can have difficulty encoding/decoding new images
  - the decoder never sees (encoded) latent vectors outside of the training set
- VAE fixes this by introducing noise in the latent vectors
  - the noise lets the decoder network see slightly different latent vectors for each training image.
  - improves the ability to interpolate between training samples



- Build the encoder
- Map the input into the mean and log(sigma) of the Gaussian distribution
  - the mean is the encoded vector

In [ ]:
```
# encoder mapping to distribution (mean and log_sigma)
x = Input(shape=(original_dim,))
h = Dense(intermediate_dim, activation='relu')(x)

# the man and log-sigma
z_mean = Dense(latent_dim)(h)
z_log_sigma = Dense(latent_dim)(h)

# encoder, from inputs to latent space
encoder = Model(x, z_mean)
```

- Use the mean and log(sigma) to sample a latent variable z

In [ ]:
```
# sampling function - draw Gaussian random noise
epsilon_std = 0.001
def sampling(args):
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape=K.shape(z_mean), mean=0., stddev=eps
ilon_std)
    return z_mean + K.exp(z_log_sigma) * epsilon

# layer that samples according to mean and sigma
z = Lambda(sampling)([z_mean, z_log_sigma])
```

- Decode the latent variable z
- Construct the whole VAE

In [ ]:
```
# create the layers and assign to a variable, since we need
# to use it later
decoder_h = Dense(intermediate_dim, activation='relu')
decoder_mean = Dense(original_dim, activation='sigmoid')

# connect the latent variable and hidden states
h_decoded = decoder_h(z)
x_decoded_mean = decoder_mean(h_decoded)

# end-to-end variational autoencoder
vae = Model(x, x_decoded_mean)
```

- Construxct the generator (decoder)
  - attach another input to the saved layers, and connect them

In [ ]:
```
# generator, from latent space to reconstructed inputs
decoder_input = Input(shape=(latent_dim,))  # make an input and attach i
t to the hidden state layer
_h_decoded = decoder_h(decoder_input)       # and other layers
_x_decoded_mean = decoder_mean(_h_decoded)

# the generator model
generator = Model(decoder_input, _x_decoded_mean)
```

- VAE uses a special loss function
  - minimize the KL divergence between the distributions

In [ ]:
```
# define the VAE loss
def vae_loss(x, x_decoded_mean):
    # cross-entropy loss
    xent_loss = keras.losses.binary_crossentropy(x, x_decoded_mean)
    # KL divergence loss
    kl_loss = - 0.5 * K.mean(1 + z_log_sigma - K.square(z_mean) - K.exp(
z_log_sigma), axis=-1)
    return xent_loss + kl_loss

# compile the model for optimization
vae.compile(optimizer='rmsprop', loss=vae_loss)
```

In [ ]:
```
history = vae.fit(vtrainXraw, vtrainXraw,
            shuffle=True,
            epochs=epochs,
            batch_size=batch_size,
            validation_data=(validXraw, validXraw),
            callbacks=[TensorBoard(log_dir='./logs/vae')],
            verbose=False
           )
```

Summary

- **Deep architectures**
  - advances of deep learning has been driven by the ImageNet competiton.
  - error rate decreases as the depth increases.
  - as depth increases, need to have a smart architecture design to make training more effective.
- **Unsupervised Learning**
  - Autoencoder - unsupervised dimensionality reduction and clustering.
  - Convolutional autoencoder - AE for images.
  - Variational autoencoder - improve interpolation ability.