```
In [4]:   # Create plots to include in other slides
          rh_ch = recipe_helper.Charts_Helper(visible=False)
          _= rh_ch.create_data()
          _= rh_ch.create_fit()
```

Overview

This is the "trailer" for the course: a brief plot summary and introduction to the key characters you will encounter.

Goals

- Get a high level view of Machine Learning
- Introduce notation
- Preview concepts

How this course is different

Our belief is that Machine Learning should be taught as a *process* for problem solving.

The following picture will be our agenda; each column is one step in the process.

Get the data

- Get the data
- Have a look
- Define Performance Measure
- Creat test set

Exploratory Data Analysis

- Visualization

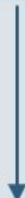## Prepare the data

- Cleaning
- Handle Non-Numeric attributes
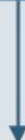- Transformations
- Scaling

Train a model

- Select a model
- Fit
- Validation and Cross Validation
- Error Analysis

Fine tune

- Hyper parameter tuning

In contrast, many approaches focus on a few steps under "Train a model"

- Select a model
- Fit

At two extremes, these approaches either focus on "using an API" or deep math.

This may lead to the ability to construct models but, in our opinion, what distinguishes an adequate Data Scientist from a good one are all the other steps in the process.

To be sure, this course will both teach you how to use an API for Machine Learning and contain a fair amount of math.

But we take an engineer/scientist approach and focus on insight and repeatability (hence, process)

- we view Data Science as an experimental science
- your experiments are implemented via code
- you need to understand enough math to diagnose problems and improve experiments

So *expect to do a lot of coding.*

- You don't need to be a "professional" programmer
- But you do need to be a *disciplined* programmer to ease repeatability
    - Subprograms/classes (methods) versus cut and paste

Also expect some math

- in order to understand why a model is appropriate or not, and to diagnose why it is not working
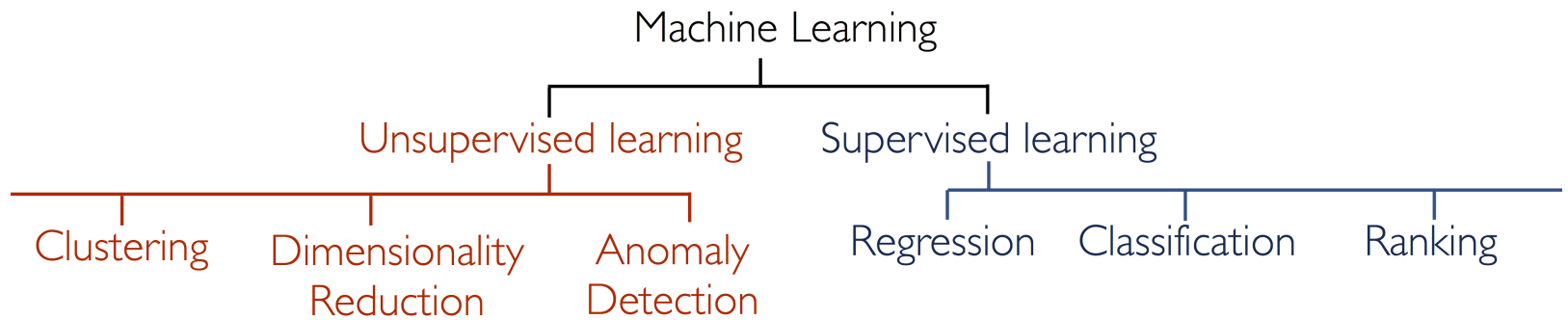- **not** to be able to derive formulae

Classical ML and Deep Learning

There are two main streams in this course

- "Classical ML"
    - somewhat long history
    - somewhat related to Statistics
- "Deep Learning"
    - really took off after 2010
    - more related to Artificial Intelligence than Statistics
        - experimental versus mathematical

This preview is for Classical Machine Learning.

The big picture

Machine Learning

Unsupervised learning · Supervised learning

Clustering · Dimensionality Reduction · Anomaly Detection

Regression · Classification · Ranking

Supervised Learning

Supervised learning is about *informed prediction*.

Let's parse these word

- prediction
- informed

Prediction: what digits do these pixels represent ?

Prediction: Given an image that we haven't seem before, determine which digit it represents.

More formally:

- A single input $\mathbf{x}$ is a vector of length n, i.e., a collection of n *features*.

- A **predictor** is a map from $\mathbf{x}$ to a class (label) $\hat{\mathbf{y}}$.

The previously unseen image: $\mathbf{x}$ consists of $n = 64$ pixels (arranged as an 8×8 grid.

$\hat{\mathbf{y}}$ is the digit that we will say the image represents.

For now:

- a class is drawn from a finite set C of potential classes.
- we are describing Classification -- mapping $\mathbf{x}$ to a single class.
- we will extend to Regression: outputs are from a continous universe (e.g., numbers)

An example is a pair $(\mathbf{x}, c)$ of a feature vector $\mathbf{x}$ and a class $c \in C$ (the target).

Informed prediction is when the probability of the predictor making a correct prediction is greater than $\frac{1}{||C||}$.

- Consider a single example $(\mathbf{x}, c)$.

- A simple but naive predictor would map $\mathbf{x}$ to a random $c' \in C$.

The probability of the predictor being correct ($c' = c$) is $\frac{1}{||C||}$.

How do we achieve this ?

Supervised Learning makes the prediction based on having seen multiple, correctly labelled examples.

- It tries to *generalize*: find some pattern in the examples that is associated with the label.

- Perhaps the individual features (elements of $\mathbf{x}$) are associated with the correct class c.

- The aim of Supervised Learning is to create a function (predictor) that maps an $\mathbf{x}$ to the correct c.

Notation

- Supervised Learning involves supplying a number (m) of examples.

- Each example is a pair consisting of

    - vector $\mathbf{x}$ consisting of $n \geq 1$ *features* (attributes)
    - scalar (sometimes a vector) $\mathbf{y}$
        - refered to as the *target* value or *label* associated with $\mathbf{x}$

- we use **bold face** to indicate a vector (e.g, $\mathbf{x}$)

- We use superscript $\mathbf{(i)}$ to denote an element i of a collection of m examples (e.g., $\mathbf{x^{(i)}}$)

- We use subscript j to index element j of a vector, e.g., $\mathbf{x}_j^{(i)}$

- So $\mathbf{x^{(i)}}$ is

$$\mathbf{x^{(i)}} = \begin{pmatrix} \mathbf{x}_1^{(i)} \\ \mathbf{x}_2^{(i)} \\ \vdots \\ \mathbf{x}_n^{(i)} \end{pmatrix}$$

Each element of $\mathbf{x^{(i)}}$ is a "feature"

Not just numbers !

The features *aren't restricted to be numeric* !

In this course, we will deal with data that is

- numeric
- categorical
- text
- image
- sound (not this course)

Of course, you'll have to encode this data as numbers in order for numerical algorithms to handle them.

Training set

- The collection of examples used for fitting (training) a model is called the *training set*:

$$\langle \mathbf{X}, \mathbf{y} \rangle = [\mathbf{x^{(i)}}, \mathbf{y^{(i)}} | 1 \leq i \leq m]$$

where m is the size of training set and each $\mathbf{x^{(i)}}$ is a feature vector of length n.

- By seeing many (m) pairs of feature vectors and associated labels we will try to infer the correct label $\mathbf{y^{(i)}}$ from the features in $\mathbf{x^{(i)}}$

- $\mathbf{X}$ is an $(m \times n)$ matrix and $\mathbf{y}$ is an $(m \times 1)$ vector of targets.

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(m)})^T \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^{(1)} \ldots \mathbf{x}_n^{(1)} \\ \mathbf{x}_1^{(2)} \ldots \mathbf{x}_n^{(2)} \\ \vdots \\ \mathbf{x}_1^{(m)} \ldots \mathbf{x}_n^{(m)} \end{pmatrix}$$

Training set

- We will sometimes add a "constant" feature by setting $\mathbf{x}_0^{(\mathbf{i})} = 1, 0 \leq i \leq m$ so that the first column of $\mathbf{x}$ is 1:

$$
\mathbf{X} = \begin{pmatrix} 1 & \mathbf{x}_1^{(1)} & \cdots & \mathbf{x}_n^{(1)} \\ 1 & \mathbf{x}_1^{(2)} & \cdots & \mathbf{x}_n^{(2)} \\ \vdots & \vdots & \cdots & \vdots \\ 1 & \mathbf{x}_1^{(m)} & \cdots & \mathbf{x}_n^{(m)} \end{pmatrix}
$$

- So each of the m rows is an example and each of the n columns is a feature.

Prediction

- Given training example $\mathbf{x}^{(i)}$, we construct a function h to predict its label

$$\hat{\mathbf{y}}^{(i)} = h(\mathbf{x}^{(i)})$$

- The function h will often be parameterized (by Θ) so, for clarity, we should write

$$\hat{\mathbf{y}}^{(i)} = h(\mathbf{x}^{(i)}; \Theta)$$

- We will often drop Θ for ease of reading.
- Since h is a function, it should also be possible to make a prediction for a vector $\mathbf{x}$ that is **not** part of the training set.
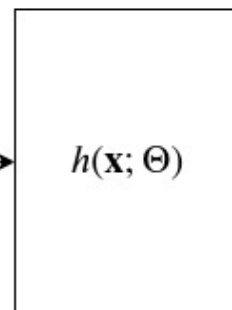- That is, we are able *generalize* to non-training examples: to make out of sample predictions

Example	Model	Prediction

$$\text{Features} \quad \begin{matrix} \mathbf{x}_1^{(i)} \\ \mathbf{x}_2^{(i)} \\ \vdots \\ \mathbf{x}_n^{(i)} \end{matrix} \longrightarrow \boxed{h(\mathbf{x}; \Theta)} \longrightarrow \hat{\mathbf{y}}^{(i)}$$
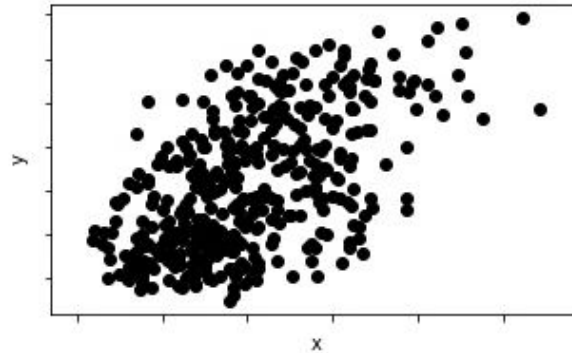
$$\text{Label} \quad \mathbf{y}^{(i)}$$

The key task of Machine Learning is finding the "best" values for parameters Θ.

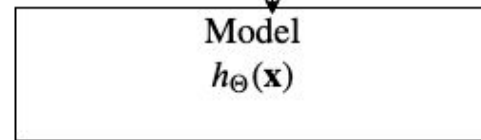The process of using training examples $\mathbf{X}$ to find Θ

- is called *fitting* the model
- is solved as an optimization problem (to be described)

$\langle \mathbf{X}, \mathbf{y} \rangle$ Training examples
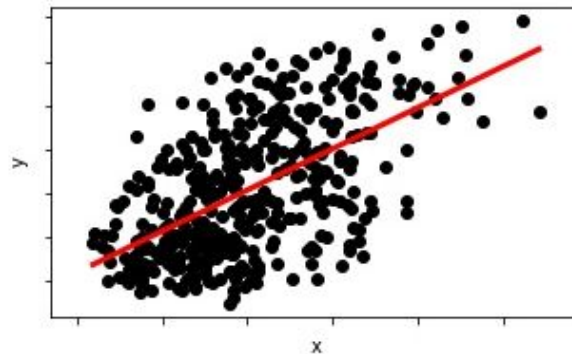
$\mathbf{X} : m \times n$
$\mathbf{y} : m \times 1$

Model
$h_\Theta(\mathbf{x})$

Fitting, training

Fit $\Rightarrow$ Θ

$\Theta : n \times 1$
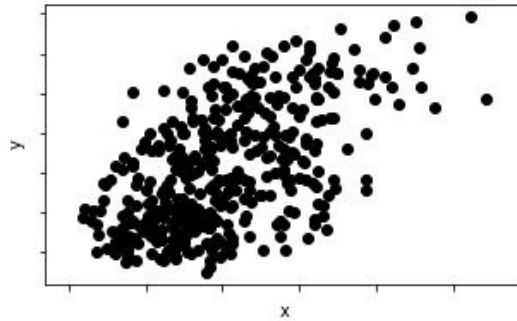
$\langle \mathbf{X}, \mathbf{y} \rangle$ Training examples

$\mathbf{X} : m \times n$
$\mathbf{y} : m \times 1$

Model
$$h_\Theta(\mathbf{x}) = \Theta_0 * \mathbf{x}_0 + \Theta_1 * \mathbf{x}_1$$
$$= \Theta^T \cdot \mathbf{x}$$

Fit $\Rightarrow$

Fitting, training

$$\Theta = [\Theta_0, \Theta_1] = [\text{intercept, slope}]$$

$$\Theta : (n+1) \times 1$$

$$\mathbf{x}^{(i)} = [1, \mathbf{x}_1^{(i)}, \dots, \mathbf{x}_n] : (n+1) \times 1$$

Summary

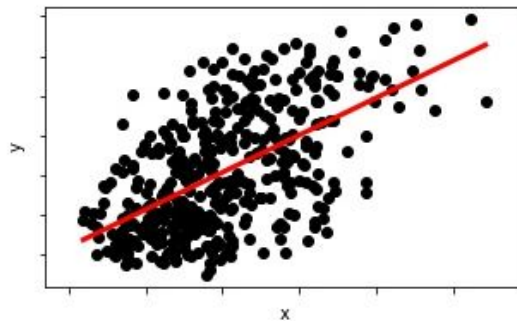- a training example is a pair $(\mathbf{x}^{(\mathbf{i})}, \mathbf{y}^{(\mathbf{i})})$ drawn from training set $\langle \mathbf{X}, \mathbf{y} \rangle$ consisting of
    - a feature vector $\mathbf{x}^{(\mathbf{i})}$ of length n
    - the associated label (target) $\mathbf{y}^{(\mathbf{i})}$
    - $\mathbf{X}$ is of dimension $m \times n$
    - $\mathbf{y}$ is dimension $m \times 1$, i.e., target is a single, continous value per example
- predictions are indicated with a "hat:
    - $\hat{\mathbf{y}}^{(\mathbf{i})}$ is the prediction made given $\mathbf{x}^{(\mathbf{i})}$ as input

Fundamental assumption of Machine Learning

Our goal is to learn (from training examples) to make a good prediction on a never before seen *test* example.

A necessary condition is that the training examples are representative of the future test examples we will encounter.

Let's imagine that there is some true (but unknown) distribution p_{data} of feature/label pairs $(\mathbf{x}, \mathbf{y})$.

In order to learn, we must assume

- That each test example $(\mathbf{x}, \mathbf{y})$ is drawn from p_{data}
- The *training examples* are a sample drawn from p_{data}.

We sometimes call the training data an *empirical* distribution -- it is just a sample, not the "true" distribution.

That is: our model can only generalize based on training examples

- The training examples need to be representative of unseen examples in the wild in order to generalize well
- Larger training sets are preferred as they may be more representative of the true p_{data}
    - They should also be diverse

If the test example $\mathbf{x}$ is not from p_{data}, the model is unconstrained in its prediction.

Making it concrete: Let's predict !

Let's load a dataset to make these concepts concrete

```
In [5]:  import class_helper
         %aimport class_helper

         clh= class_helper.Classification_Helper()
         X_digits,  y_digits = clh.load_digits()
```

- Let's see what m (number of examples), n (number of features) are

```
In [6]:  import numpy as np

         print("m={m:d} training examples".format(m=X_digits.shape[0]))
         print("n={m:d} features per example".format(m=X_digits.shape[1]))
         targets = np.unique(y_digits)
         targets.sort()

         print("{nc:d} classes: {c:s}".format(nc=len(targets), c=", ".join( [ str(t) for
         t in targets ]) ) )
```

```
m=1797 training examples
n=64 features per example
10 classes: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

```python
# Across the features of all examples: what is the min and the max ?
# Let's look at the feature vector for example at index ex_num
ex_num = 0
print("\nExample {n:d}, range({mn:2.2f}, {mx:2.2f}):\n\t ".format(n=ex_num,
                                                                    mn=X_digits.min(),
mx=X_digits.max()
                                                                    ),
        X_digits[ex_num,:]
        )
```

```
Example 0, range(0.00, 1.00):
         [0.      0.      0.3125 0.8125 0.5625 0.0625 0.      0.      0.      0.
 0.8125 0.9375 0.625  0.9375 0.3125 0.      0.      0.1875 0.9375 0.125
 0.      0.6875 0.5    0.      0.      0.25   0.75   0.      0.      0.5
 0.5    0.      0.      0.3125 0.5    0.      0.      0.5625 0.5    0.
 0.      0.25   0.6875 0.      0.0625 0.75   0.4375 0.      0.      0.125
 0.875  0.3125 0.625  0.75   0.      0.      0.      0.      0.375  0.8125
 0.625  0.      0.      0.      ]
```

- The dataset contains a number of examples.

  - Each example $\mathbf{x}^{(i)}$ is a vector of 64 features, which are numbers in the range $[0, 1]$
  - The target $y^{(i)}$ is a digit in the range $[0, 9]$

- In other words: the examples are encodings of images with labels that indicate what the image is.

Since the examples are grey scale values, we can re-arrange them into a square grid and plot:

```
In [8]: fig, axs = clh.plot_digits(X_digits, y_digits)
```
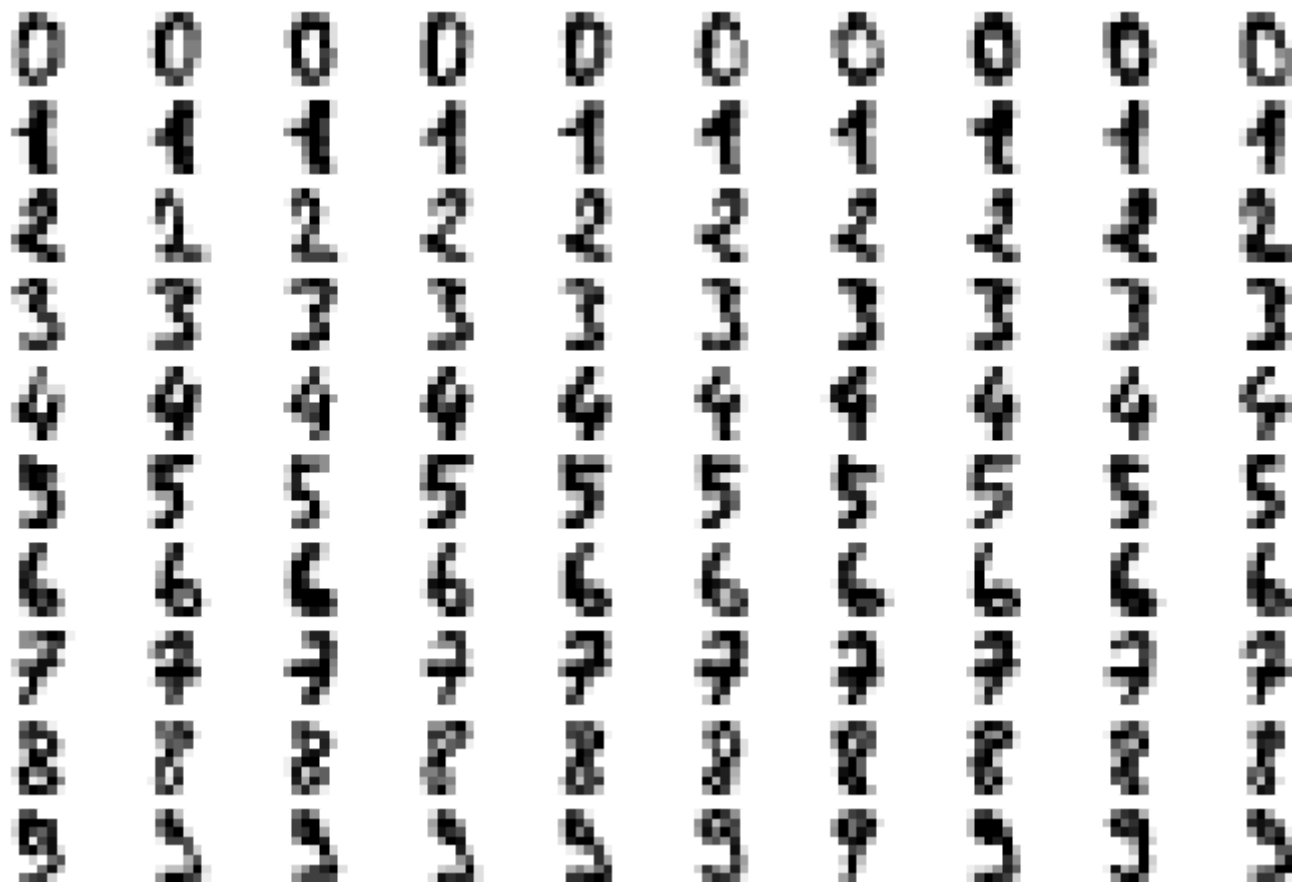
- Our problem is to take an unknown $\mathbf{x}$ and map it (predict) to a label in the range $[0, 9]$.

- This is a *classification* problem as our predictions are from a finite set.

```
In [9]: Xd_train, Xd_test, yd_train, yd_test, models = clh.fit_digits(X_digits, y_digits
        )

        _= clh.predict_digits(models["knn"], Xd_test[:10], yd_test[:10])
```

```
KNN score: 0.990000
LogisticRegression score: 1.000000
```
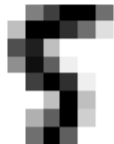

| Correct 6 | Correct 9 | Correct 3 | Correct 7 | Correct 2 |
| Correct 1 | Correct 5 | Correct 2 | Correct 5 | Correct 2 |

- How would **you** predict a label for an image, given the 64 pixel values ?

- We will use a very simple (and inefficient) algorithm called *K Nearest Neighbors (KNN).*

Template matching

- One approach to Classification is to match our input vector $\mathbf{x}$ against a *template*: (a vector of similar length) whose class is known.

- With one template $\mathbf{v}_{(c)}$ for each class $\mathbf{c} \in C$, we could classify $\mathbf{x}$ as being in the class c' whose template was "closest" to $\mathbf{x}$.

- We need a similarity measure that maps $\mathbf{x}$ and $\mathbf{v}_{(c)}$ to a number such that larger means more similar.

Our first predictor: K Nearest Neighbors (KNN)

- Here's one of the simplest Machine Learning algorithms, that leverages template matching.

- In this case, the templates are the feature vectors of the training set.

- Use the similarity measure to find the K training examples closest to $\mathbf{x}$;

- Predict the class that appears most frequently among these K examples.

- Here is our predictor function, given *test* input $\mathbf{x}$:
    - For each training example $\mathbf{x}^{(\mathbf{i})}$, compute the similarity $s^{(\mathbf{i})}$ of $\mathbf{x}$ to $\mathbf{x}^{(\mathbf{i})}$
    - Let S_K be the set of K training examples $i_1, i_2, \dots, i_K$ with greatest similarity to $\mathbf{x}$
        - $Y = [\mathbf{y}^{(j)} \mid j \in S_K]$ be the classes associated with these closest examples
    - Let count_c be the number of elements of Y that are equal to class $c, c \in C$.
    - Predict class c', with the greatest $\mathrm{count}_{c'}$

Test example

$\mathbf{x}$

Training
examples

Targets ordered
in decreasing order of
similarity of $\mathbf{x}^{(i)}$ to $\mathbf{x}$

$\mathbf{x}^{(1)}$ similarity$(\mathbf{x}, \mathbf{x}^{(1)})$ similarity$(\mathbf{x}, \mathbf{x}^{(i_1)})$ $\mathbf{y}^{(i_1)}$

$\mathbf{x}^{(2)}$ similarity$(\mathbf{x}, \mathbf{x}^{(2)})$ similarity$(\mathbf{x}, \mathbf{x}^{(i_2)})$ $\mathbf{y}^{(i_2)}$

    ⋮   ⋮   ⋮   ⋮

$\mathbf{x}^{(m)}$ similarity$(\mathbf{x}, \mathbf{x}^{(m)})$ similarity$(\mathbf{x}, \mathbf{x}^{(i_m)})$ $\mathbf{y}^{(i_m)}$

Compute similarity of
$\mathbf{x}$
to each
$\mathbf{x}^{(i)}$

Sort $\mathbf{x}^{(i)}$
in decreasing order of
similarity to $\mathbf{x}$

Prediction: $\hat{\mathbf{y}} = \text{most frequent}(\mathbf{y}^{(i_1)}, \dots, \mathbf{y}^{(i_k)})$

Here's an illustration of KNN in action:

- training example
$$\mathbf{x}^{(i)} = [\mathbf{x}_1, \mathbf{x}_2], \mathbf{y}^{(i)} \in \{0, 1\}$$
  is plotted as a colored dot, with the color corresponding to $\mathbf{y}^{(i)}$
- we form many test (non-training) examples by creating arbitrary pairs of $\mathbf{x}_1, \mathbf{x}_2$ values in a grid
  - predict for each, fill the grid with a color corresponding to the predicted class

The line separating colors (classes) is called the *separating* or *decision* boundary.

```python
In [10]:   from sklearn.neighbors import KNeighborsClassifier

           classifiers =  [ ("knn (k=1)", KNeighborsClassifier(1)),
                            ("knn (k=3)", KNeighborsClassifier(5))
                          ]
           svmh = svm_helper.SVM_Helper()
           _= svmh.create_kernel_data(classifiers=classifiers)
           fig, axs = svmh.plot_kernel_vs_transform(show_margins=False)
           plt.close()
```

In [11]: `fig`

Out[11]:
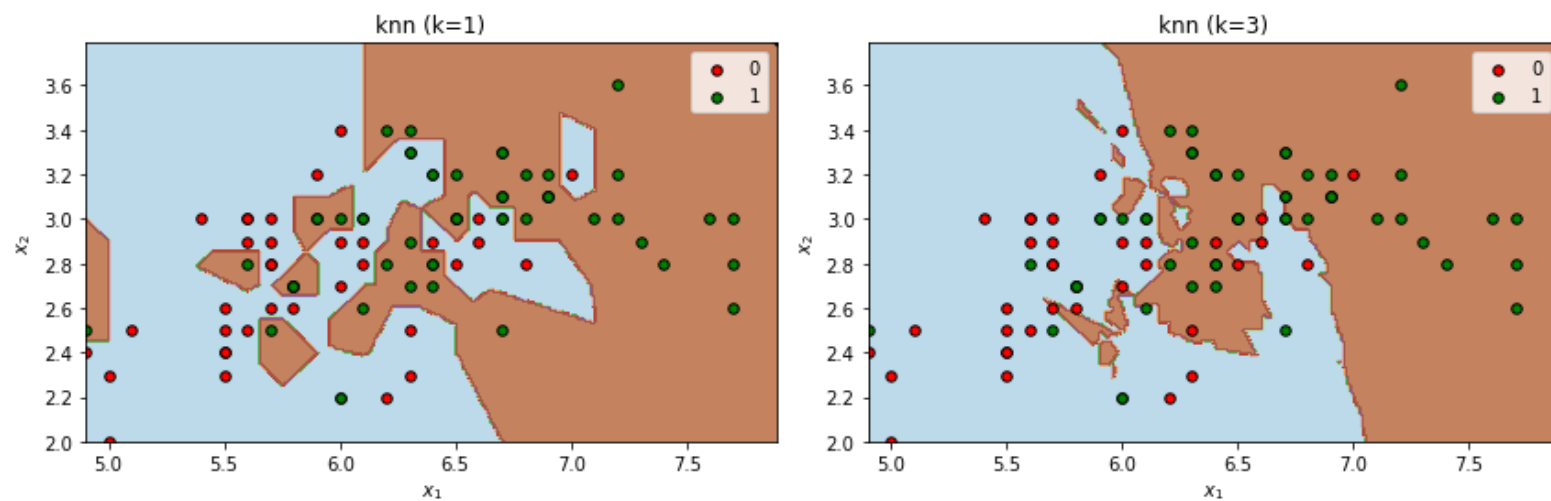
- KNN operates under the assumption (Manifold Hypothesis) that if two vectors are similar, they have the same class.

- If $K = 1$, the predictions are highly sensitive to the training examples; increasing K may increase the prediction accuracy.

Although simple, can you spot the drawback to KNN ?

$$\Theta = \mathbf{X}$$

- The size of Θ (the number of parameters) is proportional to
    - the size of the training set: $m * n$
    - ideally: m is very large, so Θ is big

Note

- *Always* count the number of parameters (size of Θ)
- You may be surprised how many you are estimating in comparsion to the amount of training data

KNN is so simple it's almost embarassing to call it Machine Learning. But it does illustrate the key steps

- the basis of Supervised Learning are training examples
    - the more the better
- the training examples are used to *fit* a predictor
    - we will learn many predictors (models) in this course
- the features of the examples are the key to prediction

- KNN did not make intelligent use of the features: it merely memorized the m examples.

  - That is, it used m templates each of size n so $|\Theta| = m * n$.

- We will see that many ML algorithms, both Classic (e.g., Regression) and Deep Learning, are based on *solving* for Θ -- finding small templates that are effective for prediction.

- A more intelligent basis for prediction would include:

  - finding one (or more) features that are predictive
  - finding relationships among features that are predictive
  - find a subset of features that is *common across all examples* in a class.

- Another issue: perhaps we are using the wrong features ?
  - are $n = 64$ raw pixels the best representation of the input (and template) for learning ?
  - would higher level features (e.g., groups of pixels that form horizontal/vertical lines) be more efficient ?

- This is called Data Transformation or Feature Engineering and will be key concept.

Fitting a predictor (training a model)

- As simple as KNN is, it has far less mathematical basis than most ML algorithms.

- To be more formal, a predictor is a function of feature vectors $\mathbf{x}$ whose behavior is parameterized by Θ.

    - terminology: also called an *estimator*, *fitted model*

- A *cost or loss function* measures how well the predictor performs on the training set, given Θ.

- Fitting (or training) the predictor: solve for the Θ that minimizes the cost function.

Note The loss function is relative to the *training* set

- it is usually similar, but not identical, to the Performance Measure that quantifies how well the predictor peforms out of sample (e.g., on the Test or Validation set).

Loss/Cost, Utility

- The prediction $\hat{\mathbf{y}}^{(i)}$ for example $\mathbf{x}^{(\mathbf{i})}$ is perfect if it matches the true label $\mathbf{y}^{(\mathbf{i})}$

$$\hat{\mathbf{y}}^{(\mathbf{i})} = \mathbf{y}^{(\mathbf{i})}$$

- Perfection is hard (at least at first) so we need a measure for "how far off" the prediction is.

- We will call the distance between $\hat{\mathbf{y}}^{(i)}$, $y^{(i)}$ the *Loss* (or *Cost*) for example i:

$$\mathcal{L}_{\Theta}^{(\mathbf{i})} = L(\ h(\mathbf{x}^{(\mathbf{i})}; \Theta), \mathbf{y}^{(\mathbf{i})}\ ) = L(\hat{\mathbf{y}}^{(i)}, \mathbf{y})$$
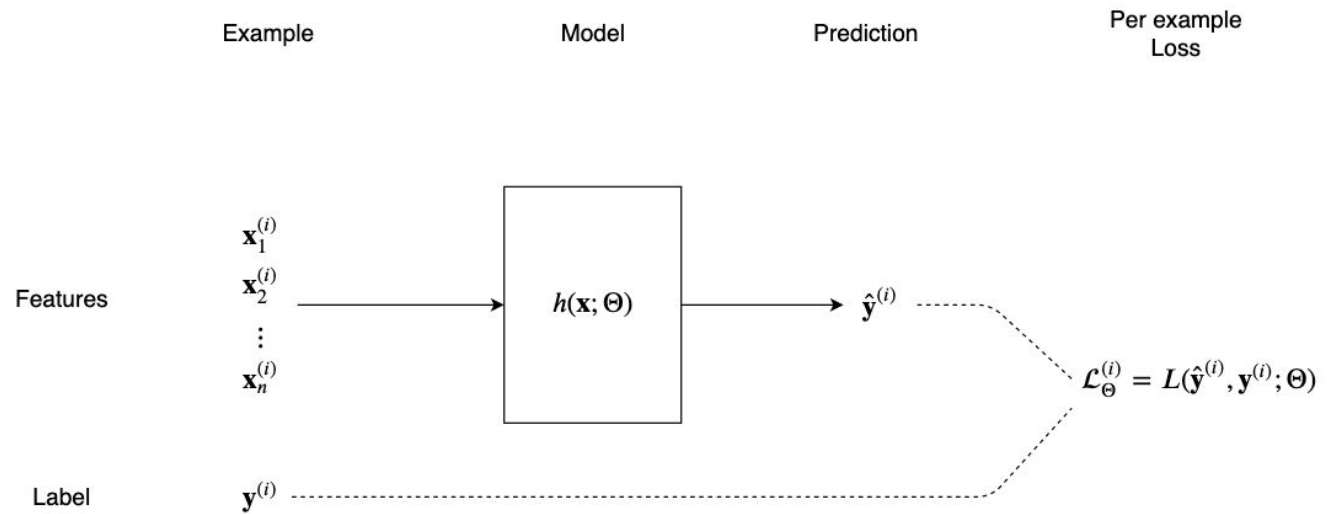
where $L(a, b)$ is a function that is 0 when $a = b$ and increasing as a increasingly differs from b.

Two common forms of L are Mean Squared Error (for Regression) and Cross Entropy Loss (for classification).

The Loss for the entire training set is simply the average (across examples) of the Loss for the example

$$\mathcal{L}_{\Theta} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}_{\Theta}^{(\mathbf{i})}$$

Training Example

Example	Model	Prediction	Per example Loss

Features

$$\mathbf{x}_1^{(i)}$$
$$\mathbf{x}_2^{(i)}$$
$$\vdots$$
$$\mathbf{x}_n^{(i)}$$

$h(\mathbf{x}; \Theta)$

$\hat{\mathbf{y}}^{(i)}$

$$\mathcal{L}_{\Theta}^{(i)} = L(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}; \Theta)$$

Label

$$\mathbf{y}^{(i)}$$

The best (optimal) Θ is the one that minimizes the Average (across trainng examples) Loss

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}} \, \mathcal{L}_\Theta$$

Whereas Loss describes how "bad" our prediction is, we sometimes refer to the converse -- how "good" the prediction is.

We call the "goodness" of the prediction the *Utility* U_{Θ}.

So we could state the optimization objective either as "minimize Cost" or "maximize Utility".

By convention, the DL optimization problem is usually framed as one of minimization (of cost or loss) rather than maximization of utility.

Since Cost is inversely related to Utility, you will sometimes see the minimization objective written as "minimize -1 times Utility".

So be forewarned that you will often see Loss function with leading "negation" signs.

Creating Loss functions is a key part of Deep Learning

As you will come to see, particularly for Deep Learning, the essence of many problems is in creating a Loss Function that captures the objective of your problem.

This is far from a trivial part of the process.

Optimization: Minimize Cost/Loss, Maximize Utility

- The goal of fitting/training is to solve for the Θ that minimizes the training set loss L_Θ (or conversely, maximizes the Utility U_Θ.

- The method for finding Θ is called optimization.

- There is one optimization method that we will study in depth: Gradient Descent.

- We can use this in Classical ML but it will become a key tool once we move on to Deep Learning.

- One focus of this course will be variations on Gradient Descent.

- The difficulty of finding a good "solution" to a problem is

    - creating a loss function that describes your objectives, which often have multiple aspects
    - having a large and diverse set of training examples to estimate Θ.

- Many packages have a method "fit" that takes the training set and performs the optimization.

- These packages usually create a "model" object (containing the Θ among other things)

- We use *predictor* and *model* as synonyms.

The dot product: key tool of linear models

- The "dot product" (special case of inner product) is one function that often appears in template matching

- It measures the similarity of two vectors

$$\mathbf{v} \cdot \mathbf{v}' = \sum_{i=1}^{n} \mathbf{v}_i \mathbf{v}'_i$$

- As a similarity measure (rather than as a distance) high dot product means "more similar".

- There are several intuitions for the dot product

- The dot product is maximized when large (resp., small) values appear in similar positions in both vectors

    - this becomes evern more obvious if we 0-center both vectors such that "small" values become negative
    - this looks like the statistical formula for covariance
        - if we normalize both vectors to unit length, then this looks like correlation

- Geometric. Let α be the angle between vectors.
    - $\mathbf{v} \cdot \mathbf{v}' = ||\mathbf{v}|| * ||\mathbf{v}'|| * \cos(\alpha)$
    - $\frac{\mathbf{v} \cdot \mathbf{v}'}{||\mathbf{v}|| * ||\mathbf{v}'||}$ is called the cosine similarity
        - similarity between normalized vectors
    - similarity is maximized when $\alpha = 0$, that is v and v' are coincident (but perhaps different lengths)
    - similarity is 0 when v, v' are orthogonal
    - similarity is negative when v, v' point in different directions

- We can generalize dot product to higher dimensions by taking the sum of element-wise multiplication (or simply by first flattening both vectors to one dimension)

- We will see the dot product appear repeatedly, particularly in Deep Learning.

Summary

- Machine Learning is a *process* that involves multiple steps
    - It is *not* just learning to use various models (predictors)
    - We will emphasize the process as much as the algorithms
- Supervised Machine Learning depends on the availability of data
    - obtaining, cleaning, augmenting data is important
- An example is a collection of "features"
    - finding/creating/interpretting features is the key skill of a Data Scientist
        - which features are important
        - how do features interact
    - sometimes features are missing or too low level
    - a key skill is creating features than enable learning ML

- A key part of Machine Learning is stating an optimization objective that captures your goal
    - not always obvious

Get the data

- Get the data
  - ↓
- Have a look
  - ↓
- Define Performance Measure
  - ↓
- Creat test set

Exploratory Data Analysis

- Visualization

Prepare the data
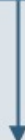
- Cleaning
  - ↓
- Handle Non-Numeric attributes
  - ↓
- Transformations
  - ↓
- Scaling

Train a model

- Select a model
  - ↓
- Fit
  - ↓
- Validation and Cross Validation
  - ↓
- Error Analysis

Fine tune

- Hyper parameter tuning

```
In [12]: print("Done")
```

Done