“My data don’t look like the simple examples you showed in class...”

- “I don’t have nice vectors of features, each of the same length.”
- Fair enough. Today, an instance of the following strategy:
  1. Identify the prediction you want to make.
  2. Identify the information you need to make each prediction.
  3. Summarize that information into a feature vector.
  4. Pass the result to an SVM (or your other favourite classifier) We will see how to learn SVMs and SVM-like things on large datasets with many features.
Cat or dog?

1. Identify the prediction you want to make.  
   Is this picture of a cat or of a dog?

2. Identify the information you need to make each prediction.  
   Picture from the internet

3. Summarize that information into a feature vector.  
   Yikes...

4. Pass the result to an SVM (or your other favourite classifier).  
   Really not the scary part, is it?
Image features

3. Summarize that information into a feature vector.

- Do I need to summarize? Can I just use pixels?
  - The average lolcat has 250,000 pixels
  - Feature selection clearly can’t save you here. (Unless maybe your cats are always located in the same pixels.)
  - Pixels are affected by many non-cat-related issues, including:
    - Color of cat
    - Distance to cat
    - Pose of cat
    - Illumination
    - Background
    - ...

- Asking an SVM to identify from your training data that the important difference is the cat-dog difference rather than some other accidental difference is completely unreasonable.
What is an image feature?

- A function that given an image produces some (relatively) low-dimensional output but retains some (relatively) interesting information

- “Global” image features:
  - Mean (median, mode) pixel intensity - very low-dimensional, super boring, probably not useful
  - Colour histogram(s) - 256(×3)-dimensional vector, no spatial information, might help find some objects (e.g. Canadian flag vs. American flag?)
  - Image “gradient” - 2D vector pointing at direction of increasing brightness
  - ...

- “Local” image features:
  - Global features applied to little patches of the big image
  - **Dense** if we pre-determine the patches (say using a grid)
  - **Sparse** if we decide which ones to compute based on the image itself
Dense vs. Sparse

- Dense seems good – fixed-length feature vector!
- What if the important information is between grid cells?
- Too fine a grid is impractical.
- Next up: Overview of The Scale-Invariant Feature Transform (SIFT),
- For constructing sparse, local “features” of images (but has been applied in the “dense” setting as well)
Identifying Interesting Points ("keypoints")

- A point is interesting if it is much darker or brighter than its "neighbors"
- Measured how? "Difference-of-gaussians" linear filter

Pictures courtesy SadaraX at en.wikipedia
The Scale of an Interesting Point

- “Interestingness” will depend on both the (x,y) location in the image, and on the chosen scale $\sigma$.
- The “scale” of a point is simply the $\sigma$ that makes it most interesting.
The Orientation of an Interesting Point

- Based on nearby image gradients

- We can now identify interesting points, assign a scale and an orientation.

- Goal: ability to “match up” interesting points between two different images
A “signature” for an interesting point (“descriptor”)

- A 128-dim vector summarizing the image gradients at nearby pixels, relative to interesting point orientation and scale

**Hope:** in a different image, the “same” interesting point will have the same descriptor, even with different lighting, orientation and scale.
SIFT: Scale-Invariant Feature Transform

- Produces a set of “interesting” points, with the following:
  - A 4-element “frame”: \((x, y)\) location, scale, orientation
  - 128-element descriptor
- Are these “features?”
- Goal of sifts: find “the same” points in different pictures. (They will have similar descriptors, possibly different frames.)
SIFTs for Information Retrieval

- Build SIFTs for all images in your database
- Build SIFTs for a query image
- For each image in the database, for each SIFT descriptor in the query
  - Find closest SIFT descriptor in the current DB image, according to Euclidean distance
  - If it’s “close enough”, call that a “match”
- If enough SIFTs from the query are found in a DB image, return that DB image.
The Goal of SIFTS

- Find **specific** objects in a database of images
- **NOT** to use in machine-learning methods.
- Note: does not produce a fixed-length feature vector given an image
- **But** they have properties we want: invariance to position, scale, rotation, (some) 3D pose, fair bit of lighting
- About 1000-ish features are produced for an average-sized image
- But there are different numbers of features for each image, and the sequence has no meaning.
Aside: Bag-of-words Representation of Documents

- What about representing a document as a feature vector?
- Documents have different lengths
- Fix a vocabulary $\mathcal{V}$. $x_k^j$ represents how many times word $k$ appears in document $j$
- We might have $p = 40,000$ for English documents, but for any document, most $x_k^j$ are zero!
- Specialized classification/regression methods can work well in this scenario
- Most are perceptron- or SVM-like.
Aside: Vector quantization

- Construct a *dictionary* of vectors labeled \{1, 2, \ldots, K\}
- Given any vector, we can *encode* it by finding the closest (in whatever distance metric) vector in the dictionary
- We can encode any set of vectors into a histogram
- **The histograms (counts) output by VQ are fixed-length feature vectors.**
- Much like bag-of-words. “How many times does a feature close to vector $k$ appear in my image?”
- Possibly as sparse as bag-of-words.
SIFTs as “bag-of-words”

- Hope: If we apply VQ to sifts, images with “similar” shapes/objects in them will have “similar” feature vectors (histograms)
- We can now use standard ML methods (including possibly feature selection) for classification of images
- Similar ideas could apply in other domains where the “raw data” are not in a format conducive to standard methods.

**But how do we make the dictionary for VQ?**
What is clustering?

- Clustering is grouping similar objects together.
  - To establish prototypes, or detect outliers.
  - To simplify data for further analysis/learning.
  - To visualize data (in conjunction with dimensionality reduction)

- Clusterings are usually not “right” or “wrong” – different clusterings/clustering criteria can reveal different things about the data.

- Some clustering criteria/algorithms have natural probabilistic interpretations

- Clustering algorithms:
  - Employ some notion of distance between objects
  - Have an explicit or implicit criterion defining what a good cluster is
  - Heuristically optimize that criterion to determine the clustering
**$K$-means clustering**

- One of the most commonly-used clustering algorithms, because it is easy to implement and quick to run.
- Assumes the objects (instances) to be clustered are $p$-dimensional vectors, $\mathbf{x}_i$.
- Uses a distance measure between the instances (typically Euclidean distance)
- The goal is to *partition* the data into $K$ disjoint subsets
**K**-means clustering with real-valued data

- **Inputs:**
  - A set of $p$-dimensional real vectors $\{x_1, x_2, \ldots, x_n\}$.
  - $K$, the desired number of clusters.

- **Output:** A mapping of the vectors into $K$ clusters (disjoint subsets), $C : \{1, \ldots, n\} \mapsto \{1, \ldots, K\}$.

1. Initialize $C$ randomly.
2. Repeat
   1. Compute the **centroid** of each cluster (the mean of all the instances in the cluster)
   2. Reassign each instance to the cluster with closest centroid
      until $C$ stops changing.
Example: initial data
Example: assign into 3 clusters randomly
Example: compute centroids
Example: reassign clusters
Example: recompute centroids
Example: reassign clusters
Example: recompute centroids – done!
What if we do not know the right number of clusters?
Example: assign into 4 clusters randomly
Example: compute centroids
Example: reassign clusters
Example: recompute centroids
Example: reassign clusters
Example: recompute centroids – done!
Assessing the quality of the clustering

- If the clustering is used as a pre-processing step for supervised learning, measure the performance of the supervised learner.
- Measure the “tightness” of the clusters: points in the same cluster should be close together, points in different clusters should be far apart.
- Tightness can be measured by the minimum distance, maximum distance or average distance between points.
- Problem: these measures usually favour large numbers of clusters, so some form of regularization/description length penalty is necessary.
Typical applications of clustering

- Pre-processing step for supervised learning
- Data inspection/experimental data analysis
- Discretizing real-valued variables in non-uniform buckets.
- Data compression
Example application: Color quantization

• Suppose you have an image stored with 24 bits per pixel
• You want to compress it so that you use only 8 bits per pixel (256 colors)
• You want the compressed image to look as similar as possible to the original image

⇒ Perform $K$-means clustering on the original set of color vectors with $K = 256$ colors.
  • Cluster centers (rounded to integer intensities) form the entries in the 256-color colormap
  • Each pixel represented by 8-bit index into colormap
Example (Bishop)

- $K = 2$
- $K = 3$
- $K = 10$
- Original image
More generally: Vector quantization with Euclidean loss

- Suppose we want to send all the instances over a communication channel
- In order to compress the message, we cluster the data and *encode each instance as the center of the cluster* to which it belongs
- The *reconstruction error* for real-valued data can be measured as Euclidian distance between the true value and its encoding
- An optimal $K$-means clustering minimizes the squared reconstruction error among all possible codings of the same type
Questions

- Will $K$-means terminate?
- Will it always find the same answer?
- How should we choose the initial cluster centers?
- Can we automatically choose the number of centers?
Does $K$-means clustering terminate?

- For given data $\{x_1, \ldots, x_n\}$ and a clustering $C$, consider the sum of the squared Euclidean distance between each vector and the center of its cluster:

$$J = \sum_{i=1}^{n} \|x_i - \mu_{C(i)}\|^2,$$

where $\mu_{C(i)}$ denotes the centroid of the cluster containing $x_i$.

- There are finitely many possible clusterings: at most $K^n$.

- Each time we reassign a vector to a cluster with a nearer centroid, $J$ decreases.

- Each time we recompute the centroids of each cluster, $J$ decreases (or stays the same.)

- Thus, the algorithm must terminate.
Does $K$-means always find the same answer?

- $K$-means is a version of coordinate descent, where the parameters are the cluster center coordinates, and the assignments of points to clusters.
- It minimizes the sum of squared Euclidean distances from vectors to their cluster centroid.
- This error function has many local minima!
- The solution found is \textit{locally optimal}, but \textit{not globally optimal}
- Because the solution depends on the initial assignment of instances to clusters, random restarts will give different solutions
Example

\[ J = 0.22870 \]

\[ J = 0.3088 \]
Finding good initial configurations

- The initial configuration can influence the final clustering.

- Assigning each item to random cluster in \( \{1, \ldots, K\} \) is unbiased... but typically results in cluster centroids near the centroid of all the data in the first round.

- A different heuristic tries to spread the initial centroids around as much as possible:
  - Place first center on top of a randomly chosen data point
  - Place second center on a data point as far away as possible from the first one
  - Place the \( i \)-th center as far away as possible from the closest of centers 1 through \( i - 1 \)

- \( K \)-means clustering typically runs quickly. With a randomized initialization step, you can run the algorithm multiple times and take the clustering with smallest \( J \).
Choosing the number of clusters

- A difficult problem, ideas are floating around. Our own Shai Ben-David has done significant work. 
  http://www.cs.uwaterloo.ca/~shai/
- Delete clusters that cover too few points
- Split clusters that cover too many points
- Add extra clusters for "outliers"
- Minimum description length: minimize loss + complexity of the clustering
- Use a hierarchical method first (see in a bit)
Why the sum of squared Euclidean distances?

Subjective reason: It produces nice, round clusters.
Why *not* the sum of squared Euclidean distances?

1. It produces nice round clusters!

![Cluster Diagram](image)

2. Differently scaled axes can dramatically affect results.

3. There may be symbolic attributes, which have to be treated differently.
Agglomerative clustering

- Input: Pairwise distances $d(x, x')$ between a set of data objects $\{x_i\}$.
- Output: A hierarchical clustering
- Algorithm:
  1. Assign each instance as its own cluster on a working list $W$.
  2. Repeat
     1. Find the two clusters in $W$ that are most “similar”.
     2. Remove them from $W$.
     3. Add their union to $W$.
     until $W$ contains a single cluster with all the data objects.
  3. Return *all clusters* appearing in $W$ at any stage of the algorithm.
How many clusters?

- How many clusters are generated by the agglomerative clustering algorithm?
- Answer: $2n - 1$, where $n$ is the number of data objects.
- Why?
  - The working list $W$ starts with $n$ singleton clusters
  - Each iteration removes two clusters from $W$ and adds one new cluster
  - The algorithm stops when $W$ has one cluster, which is after $n - 1$ iterations
How do we measure dissimilarity between clusters?

- Distance between nearest objects ("Single-linkage" agglomerative clustering, or "nearest neighbor"):
  \[ \min_{x \in C, x' \in C'} d(x, x') \]

- Distance between farthest objects ("Complete-linkage" agglomerative clustering, or "furthest neighbor"):
  \[ \max_{x \in C, x' \in C'} d(x, x') \]

- Average distance between objects ("Group-average" agglomerative clustering):
  \[ \frac{1}{|C||C'|} \sum_{x \in C, x' \in C'} d(x, x') \]
Example 1: Data

- Single
- Average
- Complete
Example 1: Iteration 30

Single

Average

Complete
Example 1: Iteration 60

Single

Average

Complete
Example 1: Iteration 70

![Single](image1)

![Average](image2)

![Complete](image3)
Example 1: Iteration 78

Single

Average

Complete
Example 1: Iteration 79

Single

Average

Complete
Example 2: Data

Single

Average

Complete
Example 2: Iteration 50

Single

Average

Complete
Example 2: Iteration 80

Single

Average

Complete
Example 2: Iteration 90

Single

Average

Complete
Example 2: Iteration 95
Example 2: Iteration 99

Single

Average

Complete
Intuitions about cluster similarity

- Single-linkage
  - Favors spatially-extended / filamentous clusters
  - Often leaves singleton clusters until near the end

- Complete-linkage favors compact clusters

- Average-linkage is somewhere in between
Summary of clustering

- **$K$-means**
  - Fast way of partitioning data into $K$ clusters
  - It minimizes the sum of squared Euclidean distances to the clusters centroids
  - Different clusterings can result from different initializations
  - Can be interpreted as fitting a mixture distribution

- **Hierarchical clustering**
  - Organizes data objects into a tree based on similarity.
  - Agglomerative (bottom-up) tree construction is most popular.
  - There are several choices of distance metric (linkage criterion)
  - Monotonicity allows us to draw dendrograms in which the height of a node corresponds to the dissimilarity of the clusters merged.
  - Trees can be cut off at some level, to generate a flat partitioning of the data.