Overview of a pattern recognition system

Jianxin Wu

LAMDA Group
National Key Lab for Novel Software Technology
Nanjing University, China
wujx2001@gmail.com

March 13, 2017

Contents

1 Introduction

The purpose of this note is to introduce a few components that are common to most (if not all) pattern recognition or machine learning systems, including a few key concepts and building blocks. A few commonly encountered issues are also discussed.

We will use face recognition as an example to introduce these components, and use the nearest neighbor classifier as a simple solution to the face recognition problem.

2 Face Recognition

You are working in a small startup IT company, which uses a face recognition device to record its employees' attendance. In the morning (or noon since it is an IT company), you come to the office and look into this face recognition gadget. It will automatically take a picture of you, and (successfully) recognizes your face. A welcome message "Morning John" is emitted by it and your attendance recorded. The synthesized speech is dumb, you think, "I shall ask the CTO to change it!"

For now let us forget about the dumb voice of the device, and forget about why a startup IT company needs an attendance recording system. You have just passed that face recognition-based attendance recording gadget, and cannot help thinking about this question: how is face recognition accomplished by this small device?

There must be your records pre-stored in that device. You do not need to be Sherlock Holmes to deduce this. How is it possible to recognize your face and your name if that is not the case? In fact, on your first day in this company, it was the same device that took your face pictures and a guy also keyed in your name such that the device can associate these pictures with you. You were told that everybody has 5 pictures stored in that little thing. If you remember that the word "records" is used at the beginning of this paragraph, by "records" we are talking about these photos and their *associated* names.

Then, things become trivial, you think. You are the latest employee and your ID is 24 in this company. You calculated it: 120 ($24 \times 5 = 120$) pictures is even much smaller than the number of images stored in your mobile phone. The device just needs to find out which one of the 120 stored pictures is similar to the one that was taken 10 seconds ago. *Of course* it is one of your 5 pictures that is the most similar, then the gadget knows your name. "That is so simple, and I am so smart."

Your smart solution is in fact a nearest neighbor classifier, which is a classic method in machine learning and pattern recognition. However, in order to turn your idea (which, as you will see, has many vague components) into precise algorithms and a working system, we need to first obtain a precise mathematical description of the task and the idea.

Face recognition is a typical application of nearest neighbor classification. Given an image that contains a human face, the task of face recognition is to find the identity of the human in the image. We are faced with two types of images or pictures. One type is called the training examples (like those 5 pictures you took on your first day), which has labels (your name) associated with them. We call the collection of this type of examples the training set. In this task, the training set is composed of many face images and the identities associated

with them. The second type of examples are called the test examples (like the picture you took just 10 seconds back). It is obvious that the face recognition task is to find the label (e.g., name) of those test set images. The procedure that finds the label for a test example is the central part of machine learning and pattern recognition, which can be abstracted as a mapping (recall this important mathematical concept?) from an input (test) example to its label. A computer vision, machine learning, or pattern recognition method or system needs to find a good *mapping* for a particular task, which can be described by algorithms and then implemented by various programming languages.

3 A simple nearest neighbor classifier

That is to say, we need to formalize every concept that is aforementioned using precise mathematical notations. Only with this formalization can we turn vague ideas into programmable solutions.

3.1 Training or learning

Formally speaking, in order to learn this mapping, we have access to n pairs of entities $\boldsymbol{x}_i$ and y_i. The pair $(\boldsymbol{x}_i, y_i)$ include the i-th *training example* ($\boldsymbol{x}_i$, also called the i-th training instance) and its associated *label* (y_i). The set of all training instances and their associated labels form the *training set*. The first stage in our task is called the *training* or *learning* stage, in which we need to find how we can deduce the label y for any example $\boldsymbol{x}$. The examples $\boldsymbol{x}_i$ comprises of the *features*. We can directly use raw input as features (e.g., directly use the face images captured by a camera) or use feature extraction techniques to process the raw input and obtain features (e.g., coordinates of various facial keypoints). In our example, $n = 24 \times 5 = 120$, and each $\boldsymbol{x}_i$ is a picture stored in the gadget. We can use the employee ID to denote the human identity. Hence, $y_i \in \{1, 2, \ldots, 24\}$, which has a one-to-one correspondence with the employee names.

We can write the mapping formally as $f : \mathbb{X} \mapsto \mathbb{Y}$. The set $\mathbb{X}$ is the space in which all training instances reside in, i.e., $\boldsymbol{x}_i \in \mathbb{X}$ for $1 \leq i \leq n$. If there are any additional training example(s), we also require that they belong to $\mathbb{X}$. Furthermore, we assume that any instance or example in our specific task, no matter they are associated with labels (e.g., training instances) or not (e.g., test examples, which we will introduce soon), should also be in the set $\mathbb{X}$. Hence, given any example $\boldsymbol{x}$ in $\mathbb{X}$, we can apply the mapping f to output our best *guess* of its associated label, as $f(\boldsymbol{x})$.

3.2 Testing or predicting

There are mainly two scenarios in applying the mapping f. Suppose we are given another set of examples $\boldsymbol{x}_i$, $n + 1 \leq i \leq n + m$. If we do not have the labels associated with these new examples, we can apply the mapping to obtain

$\hat{y}_i$ ($n+1 \leq i \leq n+m$), and $\hat{y}_i$ is the prediction produced by the mapping f (which in turn is the product of the learning stage). We are, however, unaware of the quality of these predictions—are they accurate or not?

In the second scenario of applying the learned mapping f, we are given the labels y_i, $n+1 \leq i \leq n+m$, that is, we know the *ground-truth* values of the labels. Then, we can also estimate how accurate our learned mapping is, by comparing y_i and $\hat{y}_i$ for $n+1 \leq i \leq n+m$. For example, the *accuracy* is measured as the percentage of cases when $y_i = \hat{y}_i$. Of course, we want the accuracy to be as high as possible. The process of estimating the quality of the learning mapping is often called *testing*, and the set of examples used for testing, $(\boldsymbol{x}_i, y_i)$ $(n+1 \leq i \leq n+m)$ are called the *test set*. Testing is often the second stage in our task.

If the quality of our mapping f is not satisfactory (e.g., its accuracy is too low), we need to improve it (e.g., by designing or learning better features, or by learning a better mapping). When the mapping has exhibited acceptable quality in the testing, it is ready to be shipped to its users (i.e., the first scenario in applying the mapping f). The deployment of the learned mapping can be considered as the third stage of our task. Deployment may raise new issues and requirements, which will require more iterations of training and testing. Although deployment is often ignored in teaching (e.g., in this note and this course) and researching activities, it is important in real world systems.

We want to emphasize that the labels of test examples are *strictly not allowed* to be used in the first stage (i.e, the training or learning phase). In fact, the test examples (including both the instances $\boldsymbol{x}$ and their ground-truth labels y) can *only* be used for testing. An ideal scenario is to separate the training and testing stages to two groups of people. The first group of people have access to the training data but they have absolutely no access to the test data. After the first group of people learned a mapping f (e.g., implemented as a piece of software), the second group of people will test the performance of f using the test data.

This ideal scenario may be impractical in research activities or small companies, in which the two groups of people may reduce to one single people. However, even in this adverse setup, test examples (both instances $\boldsymbol{x}$ and labels y) are not allowed to be utilized in the learning phase.

3.3 A nearest neighbor classifier

Although tons of details are still missing, we are closer to an algorithmic description of a face recognition method. We have n training examples $(\boldsymbol{x}_i, y_i)$, in which $\boldsymbol{x}_i$ $(1 \leq i \leq n)$ refers to one particular picture stored in the gadget and $n = 120$; $1 \leq y_i \leq 24$ and $y_i \in \mathbb{Z}$ records the employee ID associated with $\boldsymbol{x}_i$. Note that $\mathbb{Z}$ is the set of integers. Given any picture taken by the gadget, we just need to find an i such that $\boldsymbol{x}_i$ is the *most similar* one to it. The question is: how do we decide the similarity between two pictures? We need to turn these words into formal mathematical descriptions that are both precise and *executable*.

The first obstacle is, in fact, how do we represent a picture into $\boldsymbol{x}_i$? You may decide to use the simplest image representation in your first trial: using the pixel intensity values. An image with height H and width W has $H \times W$ pixels, and each pixel contains three channels (RGB) of values. To make the representation simpler, you decide to work with *grayscale* images, which collapse the color (RGB) into one single number (the pixel grayscale, or intensity). Hence, an image is coded as a matrix with H rows and W columns.

Matrix is an essential mathematical concept, and you know a lot of tricks to handle matrices. However, as you have seen (or will see from this course), most learning algorithms deal with *vector* data instead of matrices. Hence, you decide to convert the matrices into vectors. This is pretty simple—just "stretch" the matrix, i.e., a matrix $X \in \mathbb{R}^H \times \mathbb{R}^W$ is converted into a vector $\boldsymbol{x} \in \mathbb{R}^{HW}$, with

$$x_{(i-1)\times W+j} = X_{i,j} \qquad \forall 1 \le i \le H, 1 \le j \le W. \tag{1}$$

This simple image representation strategy applies to all images in this face recognition setup, no matter it is a training or test image.

The second question is, naturally, how do we find the examples in the training set $\boldsymbol{x}_i$ $(1 \le i \le n)$ that is the most similar to a new image $\boldsymbol{x}$? Now that we represent these images as vectors, we can easily compute the distance between any two vectors using the classic Euclidean distance. Suppose $\boldsymbol{x} \in \mathbb{R}^d$ and $\boldsymbol{y} \in \mathbb{R}^d$ are two vectors with the same dimensionality, the distance between them is

$$d(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\| . \tag{2}$$

It is natural to use distance as a measure of the dissimilarity, i.e., two vectors are dissimilar if the distance between them is large. Hence, the most similar training example can be chosen as the example that has the smallest distance with the new test example.

Till now, we have collected all necessary notations and algorithmic details to turn your vague idea into an operational algorithm, which was listed in Algorithm 1.

Algorithm 1 A simple nearest neighbor algorithm for face recognition

1: **Input:** A training set $(\boldsymbol{x}_i, y_i)$, $1 \le i \le n$. The i-th training image is converted to the vector $\boldsymbol{x}_i$.

2: **Input:** A test example $\boldsymbol{x}$, which is converted from a test image.

3: Find the index of the nearest neighbor in the training set, as

$$nn = \arg\min_{1 \le i \le n} \|\boldsymbol{x} - \boldsymbol{x}_i\| . \tag{3}$$

4: **Output:** Return the predicted identity as

$$y_{nn} . \tag{4}$$

Is that fantastic? Your idea now is an executable algorithm, which is possibly the shortest non-trivial algorithm in machine learning and pattern recognition—its main body has only 1 line (Equation 3), and the equation in that line also looks succinct and beautiful!

It is easy to understand Algorithm 1. Given any test image $\boldsymbol{x}$, you first compute the distance between it and every training example. The arg min operator finds out which index corresponds to the smallest distance. For example, if $\boldsymbol{x}_{24}$ has the smallest distance to $\boldsymbol{x}$, Equation 3 will assign 24 to nn. Then, the algorithm terminates by returning the label (human identity) associated with $\boldsymbol{x}_{24}$, i.e., $y_{24} = y_{nn}$.

Simple as Algorithm 1 is, we have a working face recognition algorithm. Its core is Equation 3, which finds the nearest neighbor of an example $\boldsymbol{x}$ in a training set of examples. We call this simple algorithm the nearest neighbor algorithm or abbreviated as NN algorithm, or the nearest neighbor classification method. The operation in Equation 3 is also called the nearest neighbor search. It is also obvious that the NN method can be used in many other tasks so long as the vectors $\boldsymbol{x}_i$ represent instances other than face images. In other words, although being extremely simple, nearest neighbor is a neat and general learning method.

3.4 k-nearest neighbors

One variant of the NN algorithm is k-nearest neighbors (or k-NN). In k-NN, k is an integer value, e.g., $k = 5$. In the nearest neighbor search, k-NN return the k nearest neighbors instead of the single nearest one.

The label that appeared the most frequent in the returned k examples is the prediction of the k-NN algorithm. For example, if $k = 5$ and the 5 nearest neighbors of your picture are with labels $7, 24, 3, 24, 24$, then the k-NN prediction is 24 (which is correct). Although the label of the nearest example is 7, there are three examples out of the 5 nearest neighbors with the correct label (24).

The nearest neighbor algorithm is also called the 1-NN algorithm. When $k > 1$, k-NN may lead to higher accuracy than 1-NN, because it can remove the effect of incorrect nearest neighbor. For instance, in the above example, 1-NN will return 7, but 5-NN predicts the correct result.

4 The ugly details

Unfortunately, making a system working well is never going to be neat. Many difficulties or caveats can be envisioned prior to the actual system building, but many more may appear at any point in a pattern recognition project. In the following we list a few typical difficulties, once again using nearest neighbor-based face recognition as an example to illustrate them. Some of these difficulties may be regarded as unimportant details at first sight. These details are, however, easily becoming ugly or even devastating if they are not taken good care of. The degradation in accuracy caused by any of these improperly handled details can

be much larger than the performance drop caused by a bad learning algorithm (in comparison to a good algorithm).

- **Noisy sensing data.** The raw data in both the training and test sets are mostly obtained from various sensors. Sensors, however, are subject to the effect of various internal and external noise. For example, a drop of water on the camera lens may cause out-of-focus or blurred face images; a defect camera lens or CCD component may cause pepper noise or other weird type of artifacts in the face images. There could also be occlusion in the face images. For example, people may wear sunglasses.

- **Noisy or wrong labels.** Likewise, the labels associated with training examples can also be noisy. The noise can be attributed to many factors, e.g., a typo can occur when the labels are keyed in to the gadget. In tasks where the labels are difficult to obtain, the "ground-truth" labels provided in a dataset might contain errors. For example, a doctor is often asked to determine whether a Computed Tomography (CT) imagery indicates a specific type of decease, but even experienced doctors (i.e., experts) can make wrong judgments.

- **Uncontrolled environment.** We are implicitly assuming some restrictions to the environment. For example, in the face recognition gadget we assume any picture taken by the gadget will have one (and only one) face inside it. We probably also require that the face will be at the center of the image, and its size will be proper. A missing face, or a face that is too large or too small, or two faces in the same picture will inevitably cause trouble in our simple algorithm. Furthermore, we probably also must assume that anybody going through the gadget is an employee of your company (and his or her picture has been stored in the gadget.) We also have to assume that the employees are all dumb enough—that they will not put on the entire set of Spider-man costume to test the limit of the poor gadget! The list of potential assumptions can go on unlimitedly. In short, we need many (either explicit or implicit) assumptions to make a system work.

- **Improper pre-processing.** Suppose the face images stored in the gadget is 100×100, which means $\boldsymbol{x}_i \in \mathbb{R}^{10000}$ for $1 \leq i \leq n$. However, if the gadget takes your photo at resolution 200×200, Algorithm 1 is presented with an input $\boldsymbol{x} \in \mathbb{R}^{40000}$. This will render our algorithm invalid, because Equation 3 is not well defined if $\boldsymbol{x}$ has a different dimensionality than any $\boldsymbol{x}_i$. This issue can be easily solved, though. We can resize the test image $\boldsymbol{x}$ to 100×100 using image processing routines, which can be regarded as a pre-processing of your data. In addition to the learning algorithms, many components in a complete system enforces various assumptions to your raw input data. Hence, the pre-processing step should understand and fulfill the requirements of all other modules in the system.

- **The existence of a semantic gap.** The numeric description and comparison of images are often far away from their meanings. For example, neither the pixel intensity values nor the Euclidean distance in Equation 3 knows the existence of faces. The intensity values are integers between 0 and 255 and every two values are treated as independent to each other by the Euclidean distance. It is very possible that two images of the same person may have a Euclidean distance that is larger than that between images of two different persons. This phenomenon is called the *semantic gap*. The semantic gap occurs not only in recognizing and understanding images. It is also a serious issue in methods and systems that deal with acoustic and many other raw input data.

- **Improper or failed feature extraction.** Feature extraction is the major step responsible for extracting features that describe the semantic information in the raw input data (e.g., "there is a cat lying on a white couch.") Features extracted in classical methods and systems, however, are some statistics (often simple statistics) of the raw input data, which is at most implicitly related to the useful semantic information. Domain experts may describe some semantic properties that are useful for certain tasks, and the feature extraction module will design features that may explicitly or implicitly describe such properties. An expert may list statistics that are useful for face recognition: the shape of eyes, the shape of the face contour, distance between two eyes, etc. The extraction of these properties requires the face contour and the eyes to be precisely detected in a face. The detection of these facial features, however, may be a task that is more difficult than face recognition itself. In short, proper feature extraction is a difficult problem. Imperfect raw input can make the problem even more difficult. The input may be noisy or having missing values (e.g., a large area of the face is occluded by a scarf.)

- **Mismatch between your data and algorithm.** We often make restrictions to change the environment from uncontrolled wilderness to a more civilized one. These restrictions allow us to make some assumptions to the input data of our learning algorithms. As will soon be discussed, assumptions on the input data is essential to the success of learning algorithms and recognition systems. The *no free lunch* theorem for machine learning states that if no assumption is made towards the data, any two machine learning algorithms (under rather weak requirements) will have exactly the same accuracy when averaged over all possible datasets. Hence, tasks with different data assumptions must be equipped with different learning algorithms. The nearest neighbor method may be suitable for face recognition, but inappropriate for searching of time-series data (e.g., stock price in a day). This is why so many machine learning algorithms have been proposed till now, because the characteristics of data appearing in different tasks vary a lot. A mismatch between the data and the learning algorithm may cause serious performance degradation.

- **Thirst for resources.** The above algorithm might work extremely well in your company (which has only 24 employees). But, what will happen if it is migrated to a large company without modification? Let's suppose the face image resolution is 100×100 and there are 10,000 employees now (hence $d = 100 \times 100 = 10000$ and $n = 50000$ if every employee has 5 images stored in the gadget.) The gadget needs to store n 100×100 images, which means 500 megabytes are required to store the face images. And, the gadget is forced to search the entire 500 megabytes to find a match of any test image—which means roughly 20 seconds for attendance recording for any single individual! Algorithms or systems become greedy, requesting huge amount of CPU, storage, and time resources, when the data they handle become big. These requests are highly impractical in real-world systems. Energy is another type of resource that is essential for modern systems. A high energy consumption gadget is likely unwelcome in the market.

- **Improper objectives.** Many objectives should be set forth for a recognition systems, including at least accuracy, running speed, and other resource requirements. If we are talking about objectives for a commercial systems, many more can be added, e.g., system price, complexity of system deployment and maintenance. Shall we expect a system that is 100% correct, requires almost zero CPU / storage / power resources, extremely inexpensive, and requires almost no effort in its maintenance? This type of systems will be the favorite product in the market—so long as they exist! These requirements are contradictory with each other, and we shall be careful to reach a satisfactory tradeoff among all these factors. For example, you may request 3 years and 1 million dollars budget to develop a "perfect" face recognition based attendance gadget that is fast and highly accurate (e.g., with a 99.99% accuracy). But, you CEO only approves 3 months and 100K dollars. You have to compromise the gadget's accuracy and speed. Even with enough budget, manpower and time, you may not reach the 99.99% accuracy objective either. We will show that a learning task has a theoretical upper bound on how accurate it can be. If that upper bound is 97% for your data, you will never reach the 99.99% accuracy target. Other factors (for example, running speed) may also force you to accept a lower recognition accuracy.

- **Improper post-processing.** Making a prediction is probably the last step in many machine learning algorithms, but almost never the last step in a real-world system. We often need to make a decision or choice based on this prediction, and then react to the environment according to the decision. For example, what is a good reaction if the gadget determines that you are recording attendance twice within five minutes? It might be overacting if it sounds an alarm and instructs the security guards to nab you; but it is also bad if it pretends that nothing unusual is happening. A more concrete example is in autonomous driving. What is the proper action to take if the autonomous driving system finds a car too close to

you? An emergency braking, or a sudden lane change, or shunning to the road shoulder? An appropriate reaction may avoid a fatal accident, but a bad one may claim several lives.

- **Improper evaluation.** How do we decide whether an algorithm or system is good or bad? And, how do we know our decision is correct? Accuracy, the simple metric, is not always valid. For example, in an imbalanced binary classification problem in which one class has 9900 training examples but the other has only 100, accuracy is a wrong evaluation metric. A classifier can simply predict any example as belonging to class 1 (the class with 9900 training examples). Although its accuracy is pretty high (99%), this prediction rule may lead to great loss. Suppose the task is to approve or reject credit card applications, and the two classes correspond to safe and risky applicants, respectively. The simple classifier has 99% accuracy, but will approve a credit card to any applicant that sends in an application form!

5 Make assumptions and simplifications

As shown by the above discussions, the performance of a learning or recognition system is determined by many factors. The quality of raw input data are, however, arguably the most important single factor, which are beyond the control of learning algorithms.

5.1 Engineering the environment vs. Designing sophisticated algorithms

Hence, an essential step in building a recognition system is to engineer your environment (and subsequently the raw input data to your system). For example, it might be quite difficult to prevent fraudulent attendance recording if someone uses a 3D printed face mask to cheat the system. A diligent security guard, however, can easily detect such unusual activities and stop someone wearing a mask from accessing the gadget.

Let us compare two face recognition based attendance recording gadgets. One gadget may allow a user to record attendance at any position around the gadget and use sophisticated face detection and face alignment techniques to correctly find and match the face. Because different locations may have varying illumination conditions, some pre-processing have to compensate for this variation in different face images. This gadget (gadget A) might be advertised as high-tech, but will require many sub-systems (detection, alignment, illumination handling, etc.) and use more computing resources.

As a comparison, another gadget (gadget B) might stick a sign on the ground at a fixed location in front of it and require all users to stand on top of the sign and look toward the gadget. Furthermore, because the user's location is fixed, gadget B can ensure the illumination condition under which the face image is

taken. Hence, gadget B may omit the detection and alignment modules and run much faster. Most probably, gadget B will have higher face recognition accuracy than gadget A, because its raw input data is taken under a well-controlled situation.

In short, if it is ever possible, we want to engineer our data collection environment to ensure high-quality and easy-to-process raw input data. With appropriate constraints on the environment, we can assume certain properties hold for the input data our learning algorithm processes, which will greatly simplify the learning process and attain higher performance (e.g., higher speed and higher accuracy). Of course, some environments cannot be easily engineered. For example, for a robot that carries boxes around a storage house, it is reasonable and advantageous to assume the ground is flat. This assumption can be easily implemented, but will greatly simplify the robot's locomotion components—two or four simple rolling wheels are sufficient. But, the same assumption seriously breaks down for a battlefield robot, which unfortunately must move on any terrain. In that case, we have no choice and have to design more complex theory and (hardware and software) systems and algorithms. Autonomously moving a robot in difficult terrains is an open problem till now, although some significant progresses have been achieved (e.g., the BigDog created by Boston Dynamics).[1]

5.2 Assumptions and simplifications

In fact, quite some assumptions have been made in our (somehow informal) description of the training and testing stages, which shall be remembered throughout this course. Similarly, quite some simplifications are also made accordingly. In this section, we will discuss some commonly used assumptions and simplifications. We will also briefly mention some approaches to handle the difficulties listed above.

- If not otherwise specified, we assume the input data (or features extracted from them) are free of noise or other types of errors. This assumption, as you will soon see in this course, makes the design and analysis of algorithms and systems easier. Noise and errors are, however, always existent in real-world data. Various methods have been developed to handle them, but they are beyond the scope of this introductory course.

- We also assume the labels (for both training and test examples) are noise- and error-free.

- We assume the ground-truth and predicted labels in a specific task (or equivalently, in the mapping f) belong to the same set $\mathbb{Y}$. Many practical applications are compatible with this assumption, which is also assumed

[1] More information about the BigDog robot can be found at https://en.wikipedia.org/wiki/BigDog. And, to be precise, autonomously moving a robot is not a machine learning or pattern recognition task (although many machine learning and pattern recognition modules may be required in order to solve this task). We use this example to illustrate the difference between preparing the environment or not.

throughout this course. For example, if only male and female appear as labels in a gender classification task's training set, we do not need to worry about a third type of gender emerging as the ground-truth label for a test example. However, the taxonomy of gender can be more complex than this simple dichotomy. One gender taxonomy uses the X and Y chromosomes, where most people are either XX (biological female) or XY (biological male). Other chromosome combinations such as XXY (Klinefelter syndrome) also exist. Hence, a gender classification system may not follow this assumption. Because XXY does not appear in the gender classification training set, we call it a *novel class* when one example from it appears in the test set. Novel class detection and handling is an advanced topic, which will not be discussed in this course.

- We assume that for an example $\boldsymbol{x}$ and its corresponding label y, there is certain relationship that related $\boldsymbol{x}$ and y. If $\boldsymbol{x}$ and y are not related (or even independent), any learning algorithm or recognition system cannot learn a meaningful mapping f that maps $\boldsymbol{x}$ to y. For example, if $\boldsymbol{x}$ is a digitized version of Claude Monet's famous painting "Impression, Sunrise" and y is tomorrow's NYSE (New York Stock Exchange) closing index value, we cannot build the mapping between them because these two variables are independent to each other.

- In the statistical machine learning formulation, we view $\boldsymbol{x}$ as a random vector and y as a random variable, and require that $\boldsymbol{x}$ and y are not independent. Let $p(\boldsymbol{x}, y)$ be the joint density for $\boldsymbol{x}$ and y. The most commonly used assumption is the i.i.d. (independent and identically distributed) assumption. The i.i.d. assumption states that any example $(\boldsymbol{x}, y)$ is sampled from the same underlying distribution $p(\boldsymbol{x}, y)$ (i.e., *identically distributed*) and any two examples are *independently* sampled from the density (i.e., the generation of any two examples will not interfere with each other). Note that this assumption only states the existence of the joint density, but does not tell us how to model or compute the joint density. The i.i.d. assumption is assumed throughout this course.

    1. To be more specific, the i.i.d. assumption assumes the training and test data are sampled from exactly the same underlying distribution $p(\boldsymbol{x}, y)$. Hence, after we use a training set to learn a mapping f, it can be safely applied to examples in the test set to obtain predictions.
    2. Another implication of the i.i.d. assumption is that the underlying distribution $p(\boldsymbol{x}, y)$ does not change. In some dynamic environments, characteristics of the label y (and sometimes the raw input data $\boldsymbol{x}$ as well) may change. This phenomenon is termed as *concept drift* (that is, the distribution $p(\boldsymbol{x}, y)$ is not stationary). Handling concept drifting is an important problem, but will not be discussed in this course.
    3. To abide by the i.i.d. assumption requires making the environment under control (at least in certain aspects), which is not always pos-

sible. For example, a guest visiting your company may decide to try the gadget. Since his face image has not been stored in the gadget, this test example violates the i.i.d. assumption, and is an *outlier*. The detection and processing of outliers are also important issues in learning systems, although we will not elaborate on it.

- We will introduce three types of preprocessing techniques in this course.

  1. The Principal component analysis (PCA) is good at reducing the effect of white noise (a special type of noise in the input data). It can also be viewed as a simple linear feature extraction method. Since PCA reduces the number of dimensions in the data, it is useful for reducing the CPU and memory footage. We have one note specifically devoted to PCA.
  2. Another simple but useful preprocessing technique is the normalization of features, which helps when the different dimensions in an example's feature vector have different scales. It is also useful when feature vectors of different examples have scale variations.
  3. The third type of preprocessing technique we introduce in this course is the Fisher's Linear Discriminant (FLD), which is often used as a feature extraction method. We also devote one note for FLD.

- After the input data and labels have been fixed, the feature extraction module might be the most important for achieving highly accurate recognition results. Good features should be able to capture useful semantic information from the raw input data as its features. However, extracting semantically meaningful features are very difficult due to the semantic gap. A classic way is to interpret domain experts' knowledge and turn them into a feature extraction method. However, this interpretation process is never easy. And, a new feature extraction method is required for a novel task domain. The extracted features are then subject to the processing of other learning algorithms. A recent breakthrough in machine learning is the rise of deep learning methods. Deep learning is sometimes termed as representation learning, because feature (or representation) learning is the core of deep learning. Deep learning is an advanced topic, and we have a note on deep learning at the end of this course. Different from the separate feature extraction module followed by a machine learning module, deep learning methods are often *end-to-end*, i.e., the input of a deep learning method is the raw input data, while the output of it is the learning target. The feature extraction and learning modules are combined into one.

- The data we need to handle are complex and appear in different formats. Two types of formats are widely used: real numbers and categorical data. A real number is a value that represents a quantity along a line, e.g., 3.14159 (an approximation of π). We can compare any two real number (e.g., $\pi > 3.14159$) and compute the distance between them (e.g., $|\pi - 3.14159| = 0.00000\ 26535\ 89793\ 23846\ldots$). Categorical data refer to

different categories, whose magnitude can not be compared. For example, we can use 1 and 2 to refer to two categories "apple" and "banana", respectively. However, we cannot say "apple (1)" is smaller than "banana (2)".

1. We assume the labels for both training and test examples $y \in \mathbb{Y}$. The task of learning the mapping f is called a regression task if $\mathbb{Y}$ is the set of of real numbers $\mathbb{R}$ (or a subset of it); the task is called classification if $\mathbb{Y}$ is a set of categorical values. Face recognition is a classification task, in which $\mathbb{Y}$ is the set $\{1, 2, \ldots, 24\}$ in our example.
2. Classification is a major task in learning and recognition research. Many algorithms have been proposed on this topic. We will introduce the support vector machine (SVM) and probabilistic classifiers (one note for each method). These methods have their respective assumptions and are suitable for different tasks. The Expectation Maximization (EM) algorithm is a very important tool for probabilistic methods. The EM algorithm might seem slightly advanced for this introductory course, and we introduce it as an advanced topic in one note.
3. The nearest neighbor algorithm uses the Euclidean distance to compute the distance between two faces. Because the Euclidean distance cannot handle categorical data, we need different tools when the input features are categorical. We will briefly introduce the decision tree classifier for this purpose. Decision trees are introduced in a note, sharing the note with a minimal introduction to information theory.
4. Information theory formally studies how *information* can be quantized, stored, and transferred. Although these tasks are not the focus of this course, the mathematical tools developed in information theory have proven themselves very useful in describing various quantities in machine learning and pattern recognition. By viewing our real-valued and categorical data as (respectively) continuous and discrete random variables, we can use these tools to compare our data (e.g., to compute the distance between two sets of categorical feature values).
5. Although we have not explicitly used the term "regression" in this course, many tasks we study in this course has its output label y in the real line. Hence, they are in fact regression tasks. One example is the similarity computation. When we compare two examples, we may prefer a numerical value (e.g., the similarity is 65%) over binary answers (e.g., 0 for dissimilar and 1 for similar). Distance metric learning can learn a suitable similarity (or dissimilarity) metric specifically for a particular task. We introduce a few generalizations of the Euclidean distance, distance metric learning and some feature normalization techniques in a note.

6. We assume the label y is always a real-valued or categorical variable in this course. However, we want to point out that more complex output formats are common in practical applications. The label y can be a vector (e.g., in multi-label classification or vector regression), a matrix of values (e.g., in semantic segmentation of images), a tree (e.g., in natural language analysis), etc.
7. We assume the label y is given for every training example. This is a *supervised* learning setting. *Unsupervised* learning is also widely studied, in which the labels are not available even for the training examples. Suppose you are given 120 images stored in the gadget (24 people, 5 images each) without their associated identity information (i.e., the labels are not present). It is possible to group these images into 24 groups, where the images belonging to the same employee in your company form one group. This process is called *clustering*, which is a typical unsupervised learning task. There are also other types of unsupervised learning tasks, but we will not touch unsupervised learning in this course.
8. We will also discuss three types of special data formats. In the nearest neighbor search algorithm, we assume the same number of dimensions for all examples $\boldsymbol{x}_i$, and the same dimension in all examples is associated with the same semantics. However, some tasks have misaligned data. For example, the input data is a short video clip showing a golf swing action. Two persons may use different speed to finish the swing. Hence, their corresponding videos will have different length, which means that there is no direct correspondence in their semantics, and a simple distance metric (such as the Euclidean distance) is not applicable. In some other tasks, the data are inherently *sparse*, meaning many of its dimensions are zero. We will discuss misaligned data and sparse data in one note.
9. The third type of special data is time series data. In speech recognition, one person reads a sentence and the signal is recorded as a time series of sensor inputs. The temporal relationship among these sensory recordings are vital for the speech recognition task, and requires algorithms that explicitly model these temporal signals. The Hidden Markov Model (HMM) is such a tool and we introduce its basic concepts and algorithms in one note.

- We will introduce the Convolutional Neural Network (CNN), a representative example for deep learning, as an advanced topic in this course. CNN is particularly useful in analyzing images, which is the major sensory input for many recognition tasks. Compared with other methods we introduce in this course, CNN (and other deep learning methods) have many virtues: end-to-end (hence no need to manual feature extraction), handles big data, and very accurate.

- However, CNN also has its limitations. It requires a lot of training data

(much more when compared with other methods), requests a lot of computing resources (CPU and/or GPU instructions, main memory and disk storage, and energy assumption). In a real system development, one should carefully consider all these factors and make compromises, including at least interactions between the pattern recognition module and other system modules, the relationship between overall system objective and the pattern recognition module's objective, etc. For example, approximate nearest neighbor (ANN) search is a good choice for the gadget if the company grows large and an exact NN search becomes too slow. ANN tries to greatly reduce the running time of NN search by tolerating small amount of errors in the search results (hence the name "approximate"). There are also huge research and development efforts to reduce the resource consumption of CNN models.

- The final aspect we want to discuss is how to evaluate a learning or recognition system. Is it accurate? How accurate can it be? Is method A better than method B? We will discuss these issues in the next note.

6 A framework

Till now, we have used the nearest neighbor search algorithm (and the face recognition method for the attendance recording task) as an example to introduce some basic concepts, issues, some possible solutions and some advanced topics. Before we finish this note, we want to revisit how the nearest neighbor algorithm was came up with in this note. To reach Algorithm 1, we have went through the following steps (but some of these steps have not appeared in Algorithm 1 because it is too simple).

1. **Know your problem**. Try to understand the input, desired output and potential difficulties in the task.

2. **Make assumptions** and **formalize your problem**. Decide some restrictions to your task's environment and convert them to assumptions to your input data. Try to use mathematical language to describe your problem and assumptions precisely.

3. **Come up with a good idea**. How will you solve a particular task (or a module inside it)? You may have some intuitions or ideas, either by studying the properties of your data or by observing how human experts solve these tasks.

4. **Formalize the idea**. Turn this idea (or these ideas) into precise mathematical languages. Often they end up as one or a few mathematical optimization problems. Try to make your equations (mathematical descriptions of the idea) as succinct as possible (cf. Equation 3). In some cases you may feel it difficult to write down your idea(s) mathematically,

let alone be succinct. Then you should go back and scrutinize that idea—maybe the idea is not good enough yet.

5. **Start from the simple case**. You may find the mathematical optimization problem very difficult. They are indeed extremely difficult in some problems. In the other problems, however, you can try to simplify your setup (e.g., by making more restrictions and assumptions) *so long as these simplifications do not change the most important characteristic of your problem*. You will possibly find the problem becoming much easier to solve. And, you can solve the original problem by relaxing these assumptions later.

6. **Solve it**.

The above overly simplified way of thinking has turned out to be useful in understanding (and inventing) many learning and recognition algorithms. We encourage you to apply these steps to understand the algorithms in this course. For example, PCA and SVM are two perfect examples to showcase the utility of these steps.

Exercises

1. In this problem we will have a taste of approximate nearest neighbor search. We use the ANN functions provided in the VLFeat software packages (`http://www.vlfeat.org/`).

   (a) Read the installation instructions and install the VLFeat software to your computer. Make sure your installation satisfy the following requirements: installed under the Linux environment; the Matlab interface is installed;[2] the installation is compiled from the source code (rather than using the pre-compiled executables); and finally, before you start the installation, change the Makefile to remove support for OpenMP.

   (b) In Matlab (or Octave), use `x=rand(5000,10)` to generate your data: 5000 examples, each has 10 dimensions. Write a Matlab (or Octave) program to find the nearest neighbor of every example. You need to compute the distances between one example and *the rest* examples, and find the smallest distance. Record the time used to find all nearest neighbors. For every example, record the index of its nearest neighbor and distance between them. Calculate the distances from scratch (i.e., do not use functions such as `pdist`), and avoid using multiple CPU cores in your computation.

   (c) Use VLFeat functions to finish the same task. Carefully read and understand the documents for `vl_kdtreebuild` and `vl_kdtreequery` in VLFeat. Set `NumTrees` to 1 and `MaxNumComparisons` to 6000, compare

[2] If you do not have access to the Matlab software, you can use the Octave interface as an alternative.

the running time of your program and VLFeat functions. You need to exclude the running time of `vl_kdtreebuild` in the comparison.

(d) These two VLFeat functions provide an ANN method, which seeks approximate nearest neighbors. How can you reveal the error rate of these approximate nearest neighbors? What is the error rate in your experiment?

(e) When you choose different parameters in the VLFeat ANN functions, how do the error rate and running speed change?

(f) When the dataset size changes (e.g., from 5000 to 500 or 50000), how do the error rate and running speed change?

(g) From the VLFeat documentation, find the paper based on which these two functions are implemented. Carefully read the paper and understand the rationale behind these functions.