Supervised Learning

Cornell CS 4/5780

Spring 2023

Intro

The goal in supervised learning is to make *predictions from data*. For example, one popular application of supervised learning is email spam filtering. Here, an email (the data instance) needs to be classified as *spam* or *not-spam*. Following the approach of traditional computer science, one might be tempted to write a carefully designed program that follows some rules to decide if an email is spam or not. Although such a program might work reasonably well for a while, it has significant drawbacks. As email spam changes it would have to be rewritten. Spammers could attempt to reverse engineer the software and design messages that circumvent it. And even if it is successful, it could probably not easily be applied to different languages. Machine Learning uses a different approach to generate a program that can make predictions from data. Instead of programming it by hand it is *learned* from past data. This process works if we have data instances for which we know exactly what the right prediction would have been. For example past data might be user-annotated as spam or not-spam. A machine learning algorithm can utilize such data to learn a program, a *classifier*, to predict the correct *label* of each annotated data instance. Other successful applications of machine learning include web-search ranking (predict which web-page the user will click on based on his/her search query), placing of online advertisements (predict the expected revenue of an ad, when placed on a homepage, which is seen by a specific user), visual object recognition (predict which object is in an image - *e.g.* a camera mounted on a self-driving car), face-detection (predict if an image patch contains a human face or not).

Setup

Let us formalize the supervised machine learning setup. Our training data comes in pairs of inputs $(\mathbf{x}, y)$, where $\mathbf{x} \in \mathcal{R}^d$ is the input instance and y its label. The entire training data is denoted as

$$D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\} \subseteq \mathcal{R}^d \times \mathcal{C}$$

where:

- $\mathcal{R}^d$ is the d-dimensional feature space
- $\mathbf{x}_i$ is the input vector of the i^{th} sample
- y_i is the label of the i^{th} sample
- $\mathcal{C}$ is the label space

The data points $(\mathbf{x}_i, y_i)$ are drawn from some (unknown) distribution $\mathcal{P}(X, Y)$. Ultimately we would like to learn a function h such that for a new pair $(\mathbf{x}, y) \sim \mathcal{P}$, we have $h(\mathbf{x}) = y$ with high probability (or $h(\mathbf{x}) \approx y$). We will get to this later. For now let us go through some examples of X and Y.

Examples of Label Spaces

There are multiple scenarios for the label space $\mathcal{C}$:

Binary classification	$\mathcal{C} = \{0, 1\}$ or $\mathcal{C} = \{-1, +1\}$.	Eg. spam filtering. An email is either spam $(+1)$, or not (-1).
Multi-class classification	$\mathcal{C} = \{1, 2, \cdots, K\}$ $(K \geq 2)$.	Eg. face classification. A person can be exactly one of K identities (e.g., 1="Barack Obama", 2="George W. Bush", etc.).
Regression	$\mathcal{C} = \mathbb{R}$.	Eg. predict future temperature or the height of a person.

Examples of feature vectors

We call $\mathbf{x}_i$ a feature vector. Each one of its d dimensions is a features describing the $i-$th sample. Let us look at some examples:

- Patient Data in a hospital. $\mathbf{x}_i = (x_i^1, x_i^2, \cdots, x_i^d)$, where $x_i^1 = 0$ or 1, may refer to the patient i's gender, x_i^2 could be the height of patient i in cm, and x_i^3 may be his/her in years, etc. In this case, $d \leq 100$ and the feature vector is dense, i.e., the number of nonzero coordinates in $\mathbf{x}_i$ is large relative to d.

- Text document in bag-of-words format. $\mathbf{x}_i = (x_i^1, x_i^2, \cdots, x_i^d)$, where x_i^α is the number of occurrences of the α^{th} word in a dictionary in document i (often referred to as *term frequencies*). In this case, $d \sim 100,000 - 10M$ and the feature vector is sparse, i.e., $\mathbf{x}_i$ consists of mostly zeros. A common way to avoid the use of a dictionary is to use feature hashing instead to directly hash any string to a dimension index (the advantage is that no dictionary is needed, but a minor disadvantage can be that multiple words are hashed into the same dimension.) A popular improvement over bag-of-words features is TF-IDF, which down-scales common words and highlights rare words.

- Images. Here, the features typically represent pixel values. $\mathbf{x}_i = (x_i^1, x_i^2, \cdots, x_i^{3k})$, where x_i^{3j-2}, x_i^{3j-1}, and x_i^{3j} refer to the red, green, and blue values of the jth pixel in the image. In this case, $d \sim 100,000 - 10M$ and the feature vector is dense. A 7MP camera results in $7\text{M} \times 3 = 21\text{M}$ features.

Hypothesis classes and No Free Lunch

Before we can find a function h, we must specify what type of function it is that we are looking for. It could be an artificial neural network, a decision tree or many other types of classifiers. We call the set of possible functions the hypothesis class. By specifying the hypothesis class, we are encoding important assumptions about the type of problem we are trying to learn. The No Free Lunch Theorem states that every successful ML algorithm must make assumptions. This also means that there is no single ML algorithm that works for every setting.

Loss Functions

There are typically two steps involved in learning a hypothesis function $h()$. First, we select the type of machine learning algorithm that we think is appropriate for this particular learning problem. This defines the hypothesis class $\mathcal{H}$, i.e. the set of functions we can possibly learn. The second step is to find the best function within this class, $h \in \mathcal{H}$. This second step is the actual learning process and often, but not always, involves an optimization problem. Essentially, we try to find a function h within the hypothesis class that makes the fewest mistakes within our training data. (If there is not a single function we typically try to choose the "simplest" by some notion of simplicity - but we will cover this in more detail in a later class.) How can we find the best function? For this we need some way to evaluate what it means for one function to be better than another. This is where the loss function (aka risk function) comes in. A loss function evaluates a hypothesis $h \in \mathcal{H}$ on our training data and tells us how bad it is. The higher the loss, the worse it is - a loss of zero means it makes perfect predictions. It is common practice to normalize the loss by the total number of training samples, n, so that the output can be interpreted as the average loss per sample (and is independent of n).

Examples:

Zero-one loss:

The simplest loss function is the zero-one loss. It literally counts how many mistakes an hypothesis function h makes on the training set. For every single example it suffers a loss of 1 if it is mispredicted, and 0 otherwise. The normalized zero-one loss returns the fraction of misclassified training samples, also often referred to as the training error. The zero-one loss is often used to evaluate classifiers in multi-class/binary classification settings but rarely useful to guide optimization procedures because the function is non-differentiable and non-continuous. Formally, the zero-one loss can be stated has:

$$\mathcal{L}_{0/1}(h)=\frac{1}{n}\sum_{i=1}^{n}\delta_{h(\mathbf{x}_i)\neq y_i},\ \text{where } \delta_{h(\mathbf{x}_i)\neq y_i}=\begin{cases}1, & \text{if } h(\mathbf{x}_i)\neq y_i\\ 0, & \text{o.w.}\end{cases}$$

This loss function returns the error rate on this data set D. For every example that the classifier misclassifies (i.e. gets wrong) a loss of 1 is suffered, whereas correctly classified samples lead to 0 loss.

Squared loss:

The squared loss function is typically used in regression settings. It iterates over all training samples and suffers the loss $\left(h(\mathbf{x}_i)-y_i\right)^2$. The squaring has two effects: 1., the loss suffered is always nonnegative; 2., the loss suffered grows quadratically with the absolute mispredicted amount. The latter property encourages no predictions to be really far off (or the penalty would be so large that a different hypothesis function is likely better suited). On the flipside, if a prediction is very close to be correct, the square will be tiny and little attention will be given to that example to obtain zero error. For example, if $|h(\mathbf{x}_i)-y_i|=0.001$ the squared loss will be even smaller, 0.000001, and will likely never be fully corrected. If, given an input $\mathbf{x}$, the label y is probabilistic according to some distribution $P(y|\mathbf{x})$ then the optimal prediction to minimize the squared loss is to predict the expected value, i.e. $h(\mathbf{x})=\mathbf{E}_{P(y|\mathbf{x})}[y]$. Formally the squared loss is:

$$\mathcal{L}_{sq}(h)=\frac{1}{n}\sum_{i=1}^{n}(h(\mathbf{x}_i)-y_i)^2.$$

Absolute loss:

Similar to the squared loss, the absolute loss function is also typically used in regression settings. It suffers the penalties $|h(\mathbf{x}_i) - y_i|$. Because the suffered loss grows linearly with the mispredictions it is more suitable for noisy data (when some mispredictions are unavoidable and shouldn't dominate the loss). If, given an input $\mathbf{x}$, the label y is probabilistic according to some distribution $P(y|\mathbf{x})$ then the optimal prediction to minimize the absolute loss is to predict the median value, i.e. $h(\mathbf{x}) = \text{MEDIAN}_{P(y|\mathbf{x})}[y]$. Formally, the absolute loss can be stated as:

$$\mathcal{L}_{abs}(h) = \frac{1}{n}\sum_{i=1}^{n}|h(\mathbf{x}_i) - y_i|.$$

Generalization:

Given a loss function, we can then attempt to find the function h that minimizes the loss:

$$h = \text{argmin}_{h\in\mathcal{H}}\mathcal{L}(h)$$

A big part of machine learning focuses on the question, how to do this minimization efficiently.

If you find a function $h(\cdot)$ with low loss on your data D, how do you know whether it will still get examples right that are not in D?

Bad example: "memorizer" $h(\cdot)$

$$h(x) = \begin{cases} y_i, & \text{if } \exists(\mathbf{x}_i, y_i) \in D, \text{ s.t., } \mathbf{x} = \mathbf{x}_i, \\ 0, & \text{o.w.} \end{cases}$$

For this $h(\cdot)$, we get 0% error on the training data D, but does horribly with samples not in D, i.e., there's the overfitting issue with this function.

Train / Test splits

To resolve the overfitting issue, we usually split D into three subsets: D_{TR} as the training data, D_{VA}, as the validation data, and D_{TE}, as the test data. Usually, they are split into a proportion of 80%, 10%, and 10%. Then, we choose $h(\cdot)$ based on D_{TR}, and evaluate $h(\cdot)$ on D_{TE}.

Quiz: Why do we need D_{VA}?
D_{VA} is used to check whether the $h(\cdot)$ obtained from D_{TR} suffers from the overfitting issue. $h(\cdot)$ will need to be validated on D_{VA}, if the loss is too large, $h(\cdot)$ will get revised based on D_{TR}, and validated again on D_{VA}. This process will keep going back and forth until it gives low loss on D_{VA}. Here's a trade-off between the sizes of D_{TR} and D_{VA}: the training results will be better for a larger D_{TR}, but the validation will be more reliable (less noisy) if D_{VA} is larger.

How to Split the Data?

You have to be very careful when you split the data in Train,Validation,Test. The test set must simulate a real test scenario, i.e. you want to simulate the setting that you will encounter in real life. For example, if you want to train an email spam filter, you train a system on past data to predict if future email is spam. Here it is important to split train / test temporally - so that you strictly predict the future from the past. If there is no such thing as a temporal component, it is often best to split uniformly at random. Definitely never split alphabetically, or by feature values.

By time, if the data is temporally collected.
In general, if the data has a temporal component, we must split it by time.

Uniformly at random, if (and, in general, only if) the data is $i.i.d.$.

The test error (or testing loss) approximates the true generalization error/loss.

Putting everything together:

We train our classifier by minimizing the training loss:

$$\text{Learning: } h^*(\cdot) = \text{argmin}_{h(\cdot)\in\mathcal{H}} \frac{1}{|D_{\text{TR}}|} \sum_{(\mathbf{x},y)\in D_{\text{TR}}} \ell(\mathbf{x}, y|h(\cdot)),$$

where $\mathcal{H}$ is the hypothetical class (i.e., the set of all possible classifiers $h(\cdot)$). In other words, we are trying to find a hypothesis h which would have performed well on the past/known data.

We evaluate our classifier on the testing loss:

$$\text{Evaluation: } \epsilon_{\text{TE}} = \frac{1}{|D_{TE}|} \sum_{(\mathbf{x},y)\in D_{\text{TE}}} \ell(\mathbf{x}, y|h^*(\cdot)).$$

If the samples are drawn i.i.d. from the same distribution $\mathcal{P}$, then the testing loss is an unbiased estimator of the true **generalization loss**:

$$\text{Generalization: } \epsilon = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{P}}[\ell(\mathbf{x}, y|h^*(\cdot))].$$

Quiz: Why does $\epsilon_{\text{TE}} \to \epsilon$ as $|D_{\text{TE}}| \to +\infty$? This is due to the weak law of large numbers, which says that the empirical average of data drawn from a distribution converges to its mean.

No free lunch. Every ML algorithm has to make assumptions on which hypothesis class $\mathcal{H}$ should you choose? This choice depends on the data, and encodes your assumptions about the data set/distribution $\mathcal{P}$. Clearly, there's no one perfect $\mathcal{H}$ for all problems.

Example. Assume that $(\mathbf{x}_1, y_1) = (1, 1)$, $(\mathbf{x}_2, y_2) = (2, 2)$, $(\mathbf{x}_3, y_3) = (3, 3)$, $(\mathbf{x}_4, y_4) = (4, 4)$, and $(\mathbf{x}_5, y_5) = (5, 5)$.

Question: what is the value of y if $\mathbf{x} = 2.5$? Well, it is utterly impossible to know the answer without assumptions. The most common assumption of ML algorithms is that the function to be approximated is locally smooth.