Lecture 2: Machine learning I

• We now embark on our journey into machine learning with the simplest yet most practical tool: linear predictors, which cover both classification and regression and are examples of reflex models. After getting some geometric intuition for linear predictors, we will turn to learning the weights of a linear predictor by formulating an optimization problem based on the loss minimization framework. Finally, we will discuss stochastic gradient descent, an efficient algorithm for optimizing (that is, minimizing) the loss that’s tailored for machine learning which is much faster than gradient descent.

Course plan

Reflex
Search problems
Markov decision processes
Adversarial games
States
Constraint satisfaction problems
Bayesian networks
Variables
Logic
"Low-level intelligence" "High-level intelligence"

Machine learning

Roadmap

Linear predictors
Loss minimization
Stochastic gradient descent

Question

How many parameters (real numbers) can be learned by machine learning algorithms using today’s computers?

- thousands
- millions
- billions
- trillions

cs21.stanford.edu/q
Application: spam classification

Input: \( x = \text{email message} \)

Output: \( y \in \{\text{spam, not-spam}\} \)

Objective: obtain a \textbf{predictor} \( f \)

\( x \xrightarrow{f} y \)

Types of prediction tasks

Binary classification (e.g., email \( \Rightarrow \) spam/not spam):

\( x \xrightarrow{f} y \in \{-1, +1\} \)

Regression (e.g., location, year \( \Rightarrow \) housing price):

\( x \xrightarrow{f} y \in \mathbb{R} \)

Types of prediction tasks

\textbf{Multiclass classification}: \( y \) is a category

\begin{center}
\begin{tikzpicture}
\node (image) {la casa blu};
\node[below=of image, xshift=-1cm, yshift=0.5cm] (f) {f};
\node (output) {the blue house};
\draw[<-] (image) -- (f);
\draw[->] (f) -- (output);
\end{tikzpicture}
\end{center}

\textbf{Ranking}: \( y \) is a permutation

\begin{center}
\begin{tikzpicture}
\node (input) {1 2 3 4};
\node[below=of input, xshift=-1cm, yshift=0.5cm] (f) {f};
\node (output) {2 3 4 1};
\draw[<-] (input) -- (f);
\draw[->] (f) -- (output);
\end{tikzpicture}
\end{center}

\textbf{Structured prediction}: \( y \) is an object which is built from parts

\begin{center}
\begin{tikzpicture}
\node (input) {la casa blu};
\node[below=of input, xshift=-1cm, yshift=0.5cm] (f) {f};
\node (output) {the blue house};
\draw[<-] (input) -- (f);
\draw[->] (f) -- (output);
\end{tikzpicture}
\end{center}

Question

Give an example of a prediction task (e.g., image \( \Rightarrow \) face/not face).
Data

**Example**: specifies that $y$ is the ground-truth output for $x$.

$$(x, y)$$

**Training data**: list of examples

$$D_{\text{train}} = [$$

- ("...10m dollars...", +1),
- ("...CS221...", -1),
$$]$$

Framework

Learning is about taking the training data $D_{\text{train}}$ and producing a predictor $f$, which is a function that takes inputs $x$ and tries to map them to $y = f(x)$. One thing to keep in mind is that we want the predictor to approximately work even for examples that we have not seen in $D_{\text{train}}$. The problem of generalization, which we will discuss two lectures from now, forces us to design $f$ in a principled, mathematical way.

Learning **is** about the training data $D_{\text{train}}$, and producing a predictor $f$, which is a function that takes inputs $x$ and tries to map them to $y = f(x)$. One thing to keep in mind is that we want the predictor to approximately work even for examples that we have not seen in $D_{\text{train}}$. The problem of generalization, which we will discuss two lectures from now, forces us to design $f$ in a principled, mathematical way.

- **Learning** is about taking the training data $D_{\text{train}}$ and producing a predictor $f$, which is a function that takes inputs $x$ and tries to map them to $y = f(x)$. One thing to keep in mind is that we want the predictor to approximately work even for examples that we have not seen in $D_{\text{train}}$. The problem of generalization, which we will discuss two lectures from now, forces us to design $f$ in a principled, mathematical way.
- **We will first focus on examining what $f$ is, independent of how the learning works. Then we will come back to learning $f$ based on data.**

Feature extraction

**Example task**: predict $y$, whether a string $x$ is an email address.

**Question**: what properties of $x$ **might be** relevant for predicting $y$?

**Feature extractor**: Given input $x$, output a set of (feature name, feature value) pairs.

```plaintext
feature extractor

"abc@gmail.com"  arbitrary!

length > 10 : 1
fracOfAlpha : 0.85
contains_@ : 1
endsWith_.com : 1
endsWith_.org : 0
```
Feature vector notation

Mathematically, feature vector doesn’t need feature names:

\[
\begin{bmatrix}
\text{length}>10 & \vdots & 1 \\
\text{fracOfAlpha} & \vdots & 0.85 \\
\text{contains @} & \vdots & 0 \ 1 \\
\text{endsWith .com} & \vdots & 2.2 \\
\text{endsWith .org} & \vdots & 1.4 \\
\end{bmatrix}
\]

Think of \( \phi(x) \in \mathbb{R}^d \) as a point in a high-dimensional space.

Definition: feature vector

For an input \( x \), its feature vector is:

\[ \phi(x) = [\phi_1(x), \ldots, \phi_d(x)]. \]

Think of \( \phi(x) \in \mathbb{R}^d \) as a point in a high-dimensional space.

Weight vector

Weight vector: for each feature \( j \), have real number \( w_j \) representing contribution of feature to prediction

\[
\begin{bmatrix}
\text{length}>10 & \vdots & -1.2 \\
\text{fracOfAlpha} & \vdots & 0.6 \\
\text{contains @} & \vdots & 0 \ 1 \\
\text{endsWith .com} & \vdots & 2.2 \\
\text{endsWith .org} & \vdots & 1.4 \\
\end{bmatrix}
\]

Linear predictors

Weight vector \( w \in \mathbb{R}^d \) Feature vector \( \phi(x) \in \mathbb{R}^d \)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Weight ( w_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>length&gt;10</td>
<td>-1.2</td>
</tr>
<tr>
<td>fracOfAlpha</td>
<td>0.6</td>
</tr>
<tr>
<td>contains @}</td>
<td>0</td>
</tr>
<tr>
<td>endsWith .com</td>
<td>2.2</td>
</tr>
<tr>
<td>endsWith .org</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Score: weighted combination of features

\[
w \cdot \phi(x) = \sum_{j=1}^{d} w_j \phi_j(x)
\]

Example: \(-1.2(1) + 0.6(0.85) + 3(1) + 2.2(1) + 1.4(0) = 4.51\)
**Linear predictors**

Weight vector $w \in \mathbb{R}^d$

Feature vector $\phi(x) \in \mathbb{R}^d$

For binary classification:

**Definition: (binary) linear classifier**

$$f_w(x) = \text{sign}(w \cdot \phi(x)) = \begin{cases} 
+1 & \text{if } w \cdot \phi(x) > 0 \\
-1 & \text{if } w \cdot \phi(x) < 0 \\
? & \text{if } w \cdot \phi(x) = 0 
\end{cases}$$

**Geometric intuition**

A binary classifier $f_w$ defines a hyperplane with normal vector $w$.

($\mathbb{R}^2 \Rightarrow$ hyperplane a line; $\mathbb{R}^3 \Rightarrow$ hyperplane a plane)

Example:

$$w = [2, -1]$$

$$\phi(x) \in \{[2, 0], [0, 2], [2, 4]\}$$

[whiteboard]

**Roadmap**

- Linear predictors
- Loss minimization
- Stochastic gradient descent

**Framework**

$$\mathcal{D}_{\text{train}} \rightarrow \text{Learner} \rightarrow f \rightarrow y$$

- We now have gathered enough intuition that we can formally define the predictor $f$. For each weight vector $w$, we write $f_w$ to denote the predictor that depends on $w$ and takes the sign of the score.
- For the next few slides, we will focus on the case of binary classification. Recall that in this setting, we call the predictor a (binary) classifier.
- The case of $f_w(x) = ?$ is a boundary case that isn’t so important. We can just predict $+1$ arbitrarily as a matter of convention.

So far, we have talked about linear predictors as weighted combinations of features. We can get a bit more insight by studying the geometry of the problem.

Let’s visualize the predictor $f_w$ by looking at which points it classifies positive. Specifically, we can draw a ray from the origin to $w$ (in two dimensions).

Points which form an acute angle with $w$ are classified as positive (dot product is positive), and points that form an obtuse angle with $w$ are classified as negative. Points which are orthogonal ($z \in \mathbb{R}^d: w \cdot z = 0$) constitute the decision boundary.

By changing $w$, we change the predictor $f_w$ and thus the decision boundary as well.
So far we have talked about linear predictors $f_w$, which are based on a feature extractor $\phi$ and a weight vector $w$. Now we turn to the problem of estimating (also known as fitting or learning) $w$ from training data.

The loss minimization framework is to cast learning as an optimization problem. Note the theme of separating your problem into a model (optimization problem) and an algorithm (optimization algorithm).

Score and margin

Correct label: $y$

Predicted label: $y' = f_w(x) = \text{sign}(w \cdot \phi(x))$

Example: $w = [2, -1], \phi(x) = [2, 0], y = -1$

**Definition: score**

The score on an example $(x, y)$ is $w \cdot \phi(x)$, how confident we are in predicting $+1$.

**Definition: margin**

The margin on an example $(x, y)$ is $(w \cdot \phi(x))y$, how correct we are.

**Question**

When does a binary classifier err on an example?

- margin less than 0
- margin greater than 0
- score less than 0
- score greater than 0

**Binary classification**

Example: $w = [2, -1], \phi(x) = [2, 0], y = -1$

Recall the binary classifier:

$$f_w(x) = \text{sign}(w \cdot \phi(x))$$

**Definition: zero-one loss**

$$\text{Loss}_{0,1}(x, y, w) = 1[f_w(x) \neq y] = 1[(w \cdot \phi(x))y \leq 0]$$

**Loss functions**

A loss function $\text{Loss}(x, y, w)$ quantifies how unhappy you would be if you used $w$ to make a prediction on $x$ when the correct output is $y$. It is the object we want to minimize.

Before we talk about what loss functions look like and how to learn $w$, we introduce another important concept, the notion of a margin. Suppose the correct label is $y \in \{-1, +1\}$. The margin of an input $x$ is $w \cdot \phi(x)y$, which measures how correct the prediction that $w$ makes is. The larger the margin, the better, and non-positive margins correspond to classification errors.

Note that if we look at the actual prediction $f_w(x)$, we can only ascertain whether the prediction was right or not. By looking at the score and the margin, we can get a more nuanced view onto the behavior of the classifier.

Geometrically, if $|w| = 1$, then the margin of an input $x$ is exactly the distance from its feature vector $\phi(x)$ to the decision boundary.
Now let us define our first loss function, the zero-one loss. This corresponds exactly to our familiar notion of whether our predictor made a mistake or not. We can also write the loss in terms of the margin.

We can plot the loss as a function of the margin. From the graph, it is clear that the loss is 1 when the margin is negative and 0 when it is positive.

Now let’s turn for a moment to regression, where the output $y$ is a real number rather than $\{-1, +1\}$. Here, the zero-one loss doesn’t make sense, because it’s unlikely that we’re going to predict $y$ exactly.

Let’s instead define the residual to measure how close the prediction $f_w(x)$ is to the correct $y$. The residual will play the analogous role of the margin for classification and will let us craft an appropriate loss function.

**Definition: residual**
The residual is $(w \cdot \phi(x)) - y$, the amount by which prediction $f_w(x) = w \cdot \phi(x)$ overshoots the target $y$.

**Definition: squared loss**

$$\text{Loss}_{squared}(x, y, w) = (f_w(x) - y)^2$$

Example:

$w = [2, -1], \phi(x) = [2, 0], y = -1$

$\text{Loss}_{squared}(x, y, w) = 25$
Regression loss functions

\[
\begin{align*}
\text{Loss}(x, y, w) &= \min_{w \in \mathbb{R}^d} \text{TrainLoss}(w) \\
\text{TrainLoss}(w) &= \frac{1}{|D_{\text{train}}|} \sum_{(x, y) \in D_{\text{train}}} \text{Loss}(x, y, w)
\end{align*}
\]

Key idea: minimize training loss

Which regression loss to use?

- A popular and convenient loss function to use in linear regression is the squared loss, which penalizes the residual of the prediction quadratically. If the predictor is off by a residual of 10, then the loss will be 100.
- An alternative to the squared loss is the absolute deviation loss, which simply takes the absolute value of the residual.

\[
\begin{align*}
\text{Loss}_{\text{squared}}(x, y, w) &= (w \cdot \phi(x) - y)^2 \\
\text{Loss}_{\text{absdev}}(x, y, w) &= |w \cdot \phi(x) - y|
\end{align*}
\]

Example: \(D_{\text{train}} = \{(1, 0), (1, 2), (1, 1000)\}\)

For least squares (L_2) regression:

- \(w\) that minimizes training loss is mean \(y\)
- Mean: tries to accommodate every example, popular

For least absolute deviation (L_1) regression:

- \(w\) that minimizes training loss is median \(y\)
- Median: more robust to outliers

Note that on one example, both the squared and absolute deviation loss functions have the same minimum, so we cannot really appreciate the differences here. However, we are learning \(w\) based on a whole training set \(D_{\text{train}}\), not just one example. We typically minimize the training loss (also known as the training error or empirical risk), which is the average loss over all the training examples.

Importantly, such an optimization problem requires making tradeoffs across all the examples (in general, we won’t be able to set \(w\) to a single value that makes every example have low loss).

Now the question of which loss we should use becomes more interesting.

- For example, consider the case where all the inputs are \(\phi(x) = 1\). Essentially the problem becomes one of predicting a single value \(y^*\) which is the least offensive towards all the examples.
- If our loss function is the squared loss, then the optimal value is the mean \(y^* = \frac{1}{|D_{\text{train}}|} \sum_{(x, y) \in D_{\text{train}}} y\). If our loss function is the absolute deviation loss, then the optimal value is the median.
- The median is more robust to outliers: you can move the farthest away point arbitrarily farther out without affecting the median. This makes sense given that the squared loss penalizes large residuals a lot more.
- In summary, this is an example of where the choice of the loss function has a qualitative impact on the weights learned, and we can study these differences in terms of the objective function without thinking about optimization algorithms.
Roadmap

Linear predictors
Loss minimization
Stochastic gradient descent

Optimization problem

Objective: \( \min_{w \in \mathbb{R}^d} \text{TrainLoss}(w) \)

How to optimize?

**Definition: gradient**

The gradient \( \nabla_w \text{TrainLoss}(w) \) is the direction that increases the loss the most.

**Algorithm: gradient descent**

Initialize \( w = [0, \ldots, 0] \)
For \( t = 1, \ldots, T \):
\[
    w \leftarrow w - \eta \nabla_w \text{TrainLoss}(w)
\]

- Having defined a bunch of different objective functions that correspond to training loss, we would now like to optimize them — that is, obtain an algorithm that outputs the \( w \) where the objective function achieves the minimum value.

- A general approach is to use **iterative optimization**, which essentially starts at some starting point \( w \) (say, all zeros), and tries to tweak \( w \) so that the objective function value decreases.

- To do this, we will rely on the gradient of the function, which tells us which direction to move in to decrease the objective the most. The gradient is a valuable piece of information, especially since we will often be optimizing in high dimensions (\( d \) on the order of thousands).

- This iterative optimization procedure is called **gradient descent**. Gradient descent has two hyperparameters, the step size \( \eta \) (which specifies how aggressively we want to pursue a direction) and the number of iterations \( T \). Let’s not worry about how to set them, but you can think of \( T = 100 \) and \( \eta = 0.1 \) for now.
Least squares regression

Objective function:

\[
\text{TrainLoss}(w) = \frac{1}{|D_{\text{train}}|} \sum_{(x,y) \in D_{\text{train}}} (w \cdot \phi(x) - y)^2
\]

Gradient (use chain rule):

\[
\nabla_w \text{TrainLoss}(w) = \frac{1}{|D_{\text{train}}|} \sum_{(x,y) \in D_{\text{train}}} 2(w \cdot \phi(x) - y) \phi(x)
\]

[live solution]

Gradient descent is slow

\[
\text{TrainLoss}(w) = \frac{1}{|D_{\text{train}}|} \sum_{(x,y) \in D_{\text{train}}} \text{Loss}(x, y, w)
\]

Gradient descent:

\[w \leftarrow w - \eta \nabla_w \text{TrainLoss}(w)\]

Problem: each iteration requires going over all training examples — expensive when you have lots of data!

Stochastic gradient descent

\[
\text{TrainLoss}(w) = \frac{1}{|D_{\text{train}}|} \sum_{(x,y) \in D_{\text{train}}} \text{Loss}(x, y, w)
\]

Gradient descent (GD):

\[w \leftarrow w - \eta \nabla_w \text{TrainLoss}(w)\]

Stochastic gradient descent (SGD):

For each \((x, y) \in D_{\text{train}}\):

\[w \leftarrow w - \eta \nabla_w \text{Loss}(x, y, w)\]

Key idea: stochastic updates

It’s not about quality, it’s about quantity.

- All that’s left to do before we can use gradient descent is to compute the gradient of our objective function. TrainLoss can usually be done by hand; combinations of the product and chain rule suffice in most cases for the functions we care about.
- Note that the gradient often has a nice interpretation. For squared loss, it is the residual (prediction - target) times the feature vector \(\phi(x)\).
- Note that for linear predictors, the gradient is always something times \(\phi(x)\) because \(w\) only affects the loss through \(w \cdot \phi(x)\).
- We can now apply gradient descent on any of our objective functions that we defined before and have a working algorithm. But it is not necessarily the best algorithm.
- One problem (but not the only problem) with gradient descent is that it is slow. Those of you familiar with optimization will recognize that methods like Newton’s method can give faster convergence, but that’s not the type of slowness I’m talking about here.
- Rather, it is the slowness that arises in large-scale machine learning applications. Recall that the training loss is a sum over the training data. If we have one million training examples (which is, by today’s standards, only a modest number), then each gradient computation requires going through those one million examples, and this must happen before we can make any progress. Can we make progress before seeing all the data?
- The answer is stochastic gradient descent (SGD). Rather than looping through all the training examples to compute a single gradient and making one update, SGD loops through the examples \((x, y)\) and updates the weights \(w\) based on each example. Each update is not as good as we’re only looking at one example rather than all the examples, but we can make many more updates this way.
- In practice, we often find that just performing one pass over the training examples with SGD, touching each example once, often performs comparably to taking ten passes over the data with GD.
- There are other variants of SGD. You can randomize the order in which you loop over the training data in each iteration, which is useful. Think about what would happen if you have all the positive examples first and the negative examples after that.
Step size

\[ \mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_w \text{Loss}(x, y, \mathbf{w}) \]

**Question:** what should \( \eta \) be?

\[ \begin{array}{c|c}
0 & 1 \\
\hline
\text{conservative, more stable} & \text{aggressive, faster} \\
\end{array} \]

**Strategies:**

- Constant: \( \eta = 0.1 \)
- Decreasing: \( \eta = \frac{1}{\sqrt{\text{# updates made so far}}} \)

One remaining issue is choosing the step size, which in practice is actually quite important, as we have seen. Generally, larger step sizes are like driving fast. You can get faster convergence, but you might also get very unstable results and crash and burn. On the other hand, with smaller step sizes, you get more stability, but you might get to your destination more slowly.

A suggested form for the step size is to set the initial step size to 1 and let the step size decrease as the inverse of the square root of the number of updates we’ve taken so far. There are some nice theoretical results showing that SGD is guaranteed to converge in this case (provided all your gradients have bounded length).

Recall that we have the zero-one loss for classification. But the main problem with zero-one loss is that it’s hard to optimize (in fact, it’s provably NP hard in the worst case). And in particular, we cannot apply gradient-based optimization to it, because the gradient is zero (almost) everywhere.

Zero-one loss

\[ \text{Loss}_{0-1}(x, y, \mathbf{w}) = 1[(\mathbf{w} \cdot \phi(x))y \leq 0] \]

**Problems:**

- Gradient of \( \text{Loss}_{0-1} \) is 0 everywhere, SGD not applicable
- \( \text{Loss}_{0-1} \) is insensitive to how badly model messed up
**Support vector machines**

\[ \text{Loss}_{\text{hinge}}(x, y, w) = \max\{1 - (w \cdot \phi(x))y, 0\} \]

- **Intuition:** hinge loss upper bounds 0-1 loss, has non-trivial gradient
- **Try to increase margin if less than 1**

**Logistic regression**

\[ \text{Loss}_{\text{logistic}}(x, y, w) = \log(1 + e^{-(w \cdot \phi(x))y}) \]

- **Intuition:** Try to increase margin even when it already exceeds 1

---

**A gradient exercise**

**Problem: Gradient of hinge loss**

Compute the gradient of

\[ \text{Loss}_{\text{hinge}}(x, y, w) = \max\{1 - (w \cdot \phi(x))y, 0\} \]

[whiteboard]

---

**Logistic regression**

Another popular loss function used in machine learning is the **logistic loss**. The main property of the logistic loss is not how correct you are predicting, you will have non-zero loss, and so there is still an incentive (although a diminishing one) to push the margin even larger. This means that you’ll update on every single example.

There are some connections between logistic regression and probabilistic models, which we will get to later.
Summary so far

Classification | Linear regression
--- | ---
Predictor $f_w$ | sign(score)
Relate to correct $y$ | margin (score $y$)
Loss functions | hinge, zero-one, logistic
Algorithm | SGD

Let's generalize from binary classification to multiclass classification. For concreteness, let us assume there are three labels. For each label $y$, we have a weight vector $w_y$, from which we define a label-specific score $w_y \cdot \phi(x)$. To make a prediction, we just take the label with the highest score.

To learn $w$, we need a loss function. Let us try to generalize the hinge loss to the multiclass setting. Recall that the hinge loss is $\text{Loss}_{\text{hinge}}(x, y, w) = \max(1 - \text{margin}(y), 0)$. So we just need to define the notion of the margin. Naturally, the margin should be the amount by which the correct score exceeds the others: $\text{margin} = w_y \cdot \phi(x) - \max_{y' \neq y} [w_{y'} \cdot \phi(x)]$.

Now, we just plug in this expression and do some algebra to get: $\text{Loss}_{\text{multiclass}}(x, y, w) = \max_{y \in \{R, G, B\}} [w_y \cdot \phi(x) - w_{y'} \cdot \phi(x) + 1[y' \neq y]]$.

The loss can be interpreted as the amount by which any competitor label $y'$'s score exceeds the true label $y$'s score when the competitor is given a 1-point handicap. The handicap encourages the true label $y$'s score to be at least 1 more than any competitor label $y'$'s score.

Next lecture

Linear predictors:

$$f_w(x) \text{ based on score } w \cdot \phi(x)$$

Which feature vector $\phi(x)$ to use?

Loss minimization:

$$\min_w \text{TrainLoss}(w)$$

How do we generalize beyond the training set?