Machine Learning: Classification/Supervised Learning

Overview

A large number of data analytical procedures are used for the purposes of *prediction*. The general form of a *prediction problem* is as follows:

> *Given a set of points* $X = \{\bar{x_1}, \ldots, \bar{x_n}\}$ *and the values some function* f *takes on these points:* $Y = \{y_1 \ldots, y_n\}$*, where* $(\forall i \in [1, \ldots, n])(y_i = f(\bar{x}_i))$*, predict the values of* $f()$ *on any input* $\bar{x}$*.*

Uusually, the prediction is to be made in a way that *minimizes* some objective function $error$:

$$\hat{f} = argmin_{\hat{f'}} Error(f, \hat{f}')$$

The $Error()$ can be computed differently depending on circumstances. The space of possible predictions $\{\hat{f'()}\}$ may be limited in some ways as well.

There are a number of special cases for the general *prediction problem*. We identify some of them below:

1. **Regression problems.** The range of $f(\bar{x})$ is the set of real numbers[1].
2. **Classification problems.** Also known as *supervised learning problems.* The range of $f(\bar{x})$ is a finite nominal or ordinal (categorical) set of values, known as *classes* or *categories*.
3. **Clustering problems** otherwise known as *unsupervised learning problems*. The range of $f(\bar{x})$ is an **unknown** finite nominal or ordinal set of values. The observed data consists only of the $\{\bar{x_1}, \ldots \bar{x_n}\}$ set of data points, without

[1]Or simply an infinite, or very large finite set of numbers.

values of $f(\bar{x_i})$ provided. In these problems, it is assumed that $f(\bar{x_i}) = f(\bar{x_j})$ if $d(\bar{x_i}, \bar{x_j})$ is sufficiently small, for some distance function $d()$. A *cluster* is an identified collection of data points $\{\bar{z_1}, \ldots, \bar{z_k}\} \subseteq X$ which all take the same value of $f()$.

4. **Collaborative filtering problems.** In this case, points $\bar{x_i}$ may have missing values in some of the dimensions. Let $A = \{A_1, \ldots, A_N\}$ be the list of all features/dimensions for points $\bar{x_1}, \ldots, \bar{x_n}$. A collaborative filtering problem is specified as follows. Let $Y = A_j$ for some $j \in \{1, \ldots, N\}$. Let $Z_j = \{\bar{x} \in X | x[A_j] = \emptyset\}$, that is, Z_j is the set of all vectors in set X that have an unknown value for attribute A_j. The collaborative filtering problem is to predict for each $\bar{x} \in Z_j$ the value $y = x[A_j]$ given the set X of data points.

Classification Problem. Definitions

Data. Consider a set $A = \{A_1, \ldots, A_n\}$ of attributes, and an additional categorical attribute C, which we call a **class attribute** or **category attribute**.

$dom(C) = \{c_1, \ldots, c_k\}$. We call each value c_i a **class label** or a **category label.**

The **learning dataset** is a relational table D.

Two formats:

1. **Training (data)set**. D has schema $(A_1, \ldots, A_n, C)$, i.e.,

   for each element of the dataset we are given its class label.

2. **Test (data)set**. D has schema $(A_1, \ldots, A_n)$, i.e.,

   the class labels of the records in D are not known.

Classification Problem. Given a (training) dataset D, construct a **classification/prediction function** that correctly predicts the class label for every record in D.

Classification function = **prediction function** = **classification model** = **classifier**.

Supervised learning because training set contains class labels. Thus we can compare (supervise) predictions of our classifier.

Classification is usually performed in two steps:

1. **Step 1. Model fit.** On this step, the incoming training set is analyzed and a classification function is built to fit the training set.

2. **Step 2. Classification (Prediction).** This is the operation of actually producing a prediction $class(\bar{x})$ upon receiving a data point $\bar{x}$ as input. This step uses the function built during the **Model fit** step.

Classification Methodology

Logistic Regression. (But you already know that)

Perceptron. A simple linear or non-linear function that bisects the n-dimensional feature space.

Neural Netowoks. Graphical models that construct a ”separation function” based on the training set data by ”chaining” multiple perceptrons.

Naïve Bayes. Estimation of probability that a record belongs to each class.

Support Vector Machines (SVMs). Linear models for two-class classifiers.

Association Rules. Infer association rules with class label on the right side.

Decision Trees. Build a tree-like classifier. (**key advantage:** human-readable!)

Nearest Neighbor classifiers. Lazy evaluation classifiers that predict class of an input point based on its proximity to points with known category labels.

Linear Discriminant Analysis. Classification by finding a low-dimension hyperplane (e.g., a line) projection of all points onto which gives the best separation.

Simple Ensemble methods. Running multiple independent classifiers and using majority/plurality prediction.

Bagging. Bagging $=$ Bootstrap aggregation is a resampling technique used to construct many classifiers (of the same basic type) on bootstrapped versions of the training sets. The class of a given data point is predicted as majority/plurality class for all constructed individual predictors. (Random Forests is a bagging extension of Decision Trees classifiers).

Boosting. Boosting is an ensemble technique where after a classifier is built for a given training set, the misclassified data points are given higher weight in the training set, and a new classifier is built to account for that. The method constructs a sequence of predictors, each of which is trying to correct for the errors of the previous one. (Adaboost is the classical example of a boosting classifier).

Perceptron

Many classification methods are *naturally defined* for the case when there are only *two categories* in the set of category labels. Such situations are usually called binary classification.

One of the simplest *binary classifiers* is perceptron.

Definition. Let $X = \{\bar{x_1}, \ldots, \bar{x_n}\}$ be a set of data points, where each point $\bar{x_i} = (a_1, \ldots, a_d)$ is a vector of length d. Let $C = \{+1, -1\}$ is the set of category labels, and let $Y = \{y_1, \ldots, y_n\}$, $y_i \in C$ be the category labels $y_i = class(\bar{x_i})$.

A perceptron is binary linear classifier that consists of

1. a linear function
$$f(\bar{x}) = \sum_{j=1}^{d} w_j \cdot a_j$$
for some vector $\mathbf{w} = (w_1, \ldots, w_d)$ of *weights*,

2. a threshold value θ, and

3. a decision procedure:

$$class(\bar{x}) = \begin{cases} +1 & \text{if } f(\bar{x}) > \theta; \\ -1 & \text{if } f(\bar{x}) < \theta; \end{cases}$$

Intuition. The perceptron function $f(\bar{x}) = \mathbf{w} \cdot \bar{x}$ defines a $d - 1$ dimensional hyperplane through the d-dimensional feature space. Points on the *positive* side of f are classified into the *positive class* (the $+1$ class). Points on the negative side of f are classified to the negative class (the -1 class).

Notes.

- For a perceptron to correctly classify the data, the data must be *linearly separable*. A dataset is called *linearly separable* if there exists a hyperplane through its feature space that separates the points in one category from the points in another category.

- If there are multiple hyperplanes that linearly separate the data, the perceptron will converge to *one of them*. The error function for the perceptron is essentially

  $$Error(f(\bar{x})) = \sum_{i=1}^{n} |f(\bar{x}_i) - y_i|,$$

  i.e. the number of incorrectly classified data points. Therefore $Error(f) = 0$ for **any** hyperplane f that linearly separates the dataset, and the perceptron does not differentiate between such hyperplanes.

Training Perceptron

We first present the perceptron training algorithm for $\theta = 0$.

1. Set $\mathbf{w} = (0, \ldots, 0)$.

2. Pick $\eta > 0$, the *learning rate* of the perceptron.

3. For each training example $(\bar{x}, y), \bar{x} \in X$ do:

   (a) $y' = \mathbf{w} \cdot \bar{x}$

   (b) if y' and y have the same sign, do nothing.

   (c) if y' and y have different signs:

   $$w := w + \eta \cdot y \cdot \bar{x}$$

To train the perceptron with an arbitrary value of θ:

- replace the vector $\mathbf{w} = (w_1, \ldots w_d)$ with the vector $\mathbf{w}' = (w_1, \ldots, w_d, \theta)$.

- replace every vector $\bar{x} \in X$, where $x = (a_1, \ldots, a_d)$ with the vector $\bar{x'} = (a_1, \ldots, a_d, -1)$.

- Train the perceptron using the algorithm above on the weights $\mathbf{w}'$ and feature vectors $X' = \{\bar{x'_1}, \ldots, \bar{x'_n}\}$.

Note: If you squint at it, the training process for the perceptron classifier should remind you of something. Hint: where else have you seen the *learning rate* parameter?

Indeed, this algorithm is a special case of *gradient descent/gradient ascent*.

When to stop

The training can stop if:

- All $\bar{x} \in X$ have been correctly classified (i.e., when classification error = 0).
- Failing that, perceptron training can be stopped in one of the following ways:
  - After *M* iterations for some number $M > n$.
  - After the following detection error:

  $$Error' = \frac{1}{2}\sum_{i=1}^{n} |\mathbf{w}\cdot\bar{x}_i\cdot(sign(\mathbf{w}\cdot\bar{x}_i) - y_i)|$$

  stops decreasing.

  (Note: $Error'$ computes the sum of distances from the separating hyperplane of all points that are misclassified. We need the $\frac{1}{2}$ normalizing factor because $|sign(\mathbf{w}\cdot\bar{x}_i) - y_i| = 2$ when the perceptron misclassifies a data point.)

Support Vector Machines

Extending Perceptron Classifiers. There are two ways to extend perceptron classifiers. Each of the two ways corresponds to noticing a significant drawback in how perceptrons operate, and attempting to ”fix” it.

Perceptron weakness $\#1$: binary classification using a single hyperplane. The core problem with a perceptron is that it is a simple binary classifier that separates the classes using a *single* hyperplane. It cannot work well in one of the following cases:

- Separation boundary is non-linear[2].
- There is no clear separability of data points.
- More than one hyperplane is required for true separation of data points (the so called XOR scenario), i.e., the situation, when the function to learn is non-monotonic in all its inputs.

[2]There are some space transformation techniques that can still allow one to use a single perceptron for some non-linearly separable cases.

This weakness is addressed by **Neural Networks**, which are constructed as follows:

- Individual perceptrons are modified into *neurons* by replacing their activation function from a θ-threshold, to some differentiable function (e.g., a hyperbolic tangent function).
- Neurons are organized into networks, but putting the output of the activation function of one neuron as input to another neuron. Typical networks consist of *layers* of neurons, with the input layer having one neuron per input data dimension, and *hidden layer*(s) being given as inputs the outputs of previous layers.
- The output layer may contain multiple neurons, which allows to extend binary classification into multi-class classification.

Perceptron weakness $\#2$: choosing the right hyperplane. This is actually two separate weaknesses:

- inability to deal with linearly inseparable datasets, and
- inability to properly determine the best separating hyperplane.

This weakness is addressed by Support Vector Machines.

Essentially, a perceptron is converted into a Support Vector Machine (SVM) by making the *error function* more complex.

Support Vector Machines with Linear Kernels

Note: The key issue making perceptrons not distinguish between ”good” and ”bad” separating hyperplanes is the *weak* error function. Support Vector Machines (SVMs) change the error function. An SVM attempts to select a hyperplane

$$\mathbf{w} \cdot \bar{x} + b = 0,$$

such that this hyperplane lies *as far as possible* from any point in the training set (while classifying properly as many points as possible).

Idea: Points that are far away from the decision boundary (or from any separating hyperplane) are *easy to classify* and give us more certainty about the class they belong to. Points that are near the decision boundary/separating hyperplane are harder to classify. The further away a separating hyperplane is from the nearest points, the easier it is to classify those nearest points, and the less uncertainty we have about their class.

Support Vectors. Given a separating hyperplane $\mathbf{w} \cdot \bar{x} + b = 0$, the points $\{\bar{z_1}, \ldots, \bar{z_k}\} \subseteq X$ which have the *shortest distance to the hyperplane*, i.e., all the points with the smallest absolute values $e = e_i = \mathbf{w} \cdot \bar{z_i} + b$[3] are called **support vectors** of the hyperplane.

[3]That is, the distance from **any** point $\bar{z_1}, \ldots, \bar{z_k}$ to the hyperplane is the same.

Attempt 1. Given a training set $(X, Y) = \{(\bar{x_i}, y_i)\}, y_i \in \{-1, +1\}$, $\bar{x_i} = (a_1, \ldots, a_d)$ (for some d - number of features/dimensions), determine the vector of weights $\mathbf{w} = (w_1, \ldots, w_d)$ and the intercept b that **maximize the value** γ, such that for all $i = 1, \ldots, n$:

$$y_i \cdot (\mathbf{w} \cdot \bar{x_i} + b) \geq \gamma.$$

Intuitively, we want the largest value of γ, such that for all data points $\bar{x_i} \in X$, where $y_i = +1$, $(\mathbf{w} \cdot \bar{x_i} + b) >= \gamma$, and for all data points $\bar{x_i} \in X$, where $y_i = -1$, $(\mathbf{w} \cdot \bar{x_i} + b) <= -\gamma$.

Problem: too many degrees of freedom. We can always increase all values of $\mathbf{w}$ and b and thus increase γ: if $y_i \cdot (\mathbf{w} \cdot \bar{x_i} + b) \geq \gamma$ then $y_i \cdot (\mathbf{2} \cdot \mathbf{w} \cdot \bar{x_i} + 2b) \geq 2\gamma$.

Changing the formulation of the problem. Consider the training set $(X, Y) = \{(\bar{x_i}, y_i)\}, y_i \in \{-1, +1\}$, $\bar{x_i} = (a_1, \ldots, a_d)$ and a hyperplane

$$h(\bar{x}) = \mathbf{w} \cdot \bar{x} + b = 0$$

Given two points $\bar{v_1} \neq \bar{v_2}$, such that $h(\bar{v_1}) = 0$ and $h(\bar{v_2}) = 0$ (i.e., two points on the hyperplane $h(\bar{x})$, we notice:

$$h(v_1) - h(v_2) = (\mathbf{w} \cdot \bar{v_1} + b) - (\mathbf{w} \cdot \bar{v_2} + b) =$$
$$\mathbf{w} \cdot (\bar{v_1} - \bar{v_2}) + (b - b) = \mathbf{w} \cdot (\bar{v_1} - \bar{v_2}) = 0,$$

from which we observe that vector $\mathbf{w}$ is orthogonal to the hyperplane $h(\bar{x})$.

Now, suppose $\bar{x} = (a_1, \ldots, a_d)$ is an arbitrary d-dimensional point. We can represent this point as the sum

$$\bar{x} = \bar{x_h} + \mathbf{r},$$

where $\bar{x_h}$ is the orthogonal projection of $\bar{x}$ onto the hyperplane $h(\bar{x})$, and $\mathbf{r}$ is the vector $\bar{x} - \bar{x_h}$. Here,

$$\mathbf{r} = r \cdot \frac{\mathbf{w}}{\|\mathbf{w}\|} = r \cdot \frac{\mathbf{w}}{\sqrt{\sum_{j=1}^{d} w_j^2}},$$

where r is the *directed* distance from $\bar{x_h}$ to $\bar{x}$ in terms of the *unit weight vector* $\frac{\mathbf{w}}{\|\mathbf{w}\|}$.

Therefore,

$$h(\bar{x}) = h\left(\bar{x_h} + r\frac{\mathbf{w}}{\|\mathbf{w}\|}\right) = \mathbf{w} \cdot \left(\bar{x_h} + r\frac{\mathbf{w}}{\|\mathbf{w}\|}\right) + b =$$

$$= \mathbf{w} \cdot \bar{x_h} + b + r\frac{\mathbf{w} \cdot \mathbf{w}}{\|\mathbf{w}\|} = h(\bar{x_h}) + r \cdot \frac{\|\mathbf{w}\|^2}{\|\mathbf{w}\|} = r\|\mathbf{w}\|.$$

(remember that since $\bar{x_h}$ is on the hyperplane $h(\bar{x})$, $h(\bar{x_h}) = 0$.)

Therefore, the directed distance from any point $\bar{x}$ to the hyperplane $h(\bar{x})$ is:

$$r = \frac{h(\bar{x})}{\|\mathbf{w}\|}.$$

The *distance* δ from the point $\bar{x_i} in X$ to the hyperplane $h(\bar{x})$ is:

$$\delta_i = y_i \cdot r_i = y \cdot \frac{h(\bar{x_i})}{\|\mathbf{w}\|}.$$

Margin. Let $h(\bar{x})$ be a hyperplane and $(X, Y) = \{(\bar{x_i}, y_i)\}$ be a training set of size n. The **margin** δ^* of $h(\bar{x})$ is defined as

$$\delta^* = \min_i \left(y_i \frac{h(\bar{x_i})}{\|\mathbf{w}\|} \right) = \min_i(\delta_i),$$

where

$$\delta_i = y_i \frac{\mathbf{w} \cdot \bar{x_i} + b}{\|\mathbf{w}\|},$$

i.e., the **margin** of $h(\bar{x})$ is the **smallest relative distance** (in terms of the length of vector $\mathbf{w}$) from some training set point to the hyperplane.

Canonical planes. To address the issue with **Attempt 1**, we fix the scale of the hyperplane $h(\bar{x})$. Since the hyperplane equation can be multiplied by any scalar $s \neq 0$ and preserve the equality:

$$h(\bar{x}) = 0 \Rightarrow s \cdot h(\bar{x}) = s \cdot \mathbf{w} \cdot \bar{x} + sb = (\mathbf{s} \cdot \mathbf{w}) \cdot x + (sb) = 0,$$

we can limit ourselves to only considering the hyperplanes where s takes a restricted value. Specifically, **we want the absolute distance from the support vectors to the hyperplane** to be 1:

$$sy^*(\mathbf{w} \cdot \bar{x^*} + b) = 1,$$

for any support vector $\bar{x^*} \in X$ such that $y^* \frac{\mathbf{w} \cdot \bar{x} + b}{\|\mathbf{w}\|} = \delta^*$. This means that

$$s = \frac{1}{y^* h(x^*)}.$$

We will limit our search for $\mathbf{w}$ and b, i.e., for hyperplanes $h(\bar{x} = \mathbf{w} \cdot \bar{x} + b$ to those instances where the absolute distances to support vectors are equal to 1, that is:

$$\delta^* = \frac{y^* h(\bar{x^*}}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}.$$

We call such hyperplanes **canonical**.

Support Vector Machines: Linearly Separable Case

Optimization Problem for Linearly Separable SVMs. Given the training set $(X, Y) = \{(\bar{x_i}, y_i)\}, i = 1, \ldots, n$, where $\bar{x_i} \in \mathbb{R}^d$, and $y_i \in \{+1, -1\}$, find a *canonical hyperplane*

$$h(\bar{x}) = \mathbf{w} \cdot \bar{x} + b,$$

that maximizes the margin δ^*:

$$h^* = \arg\max_h(\delta^*_h) = \arg\max_{\mathbf{w},b}\left(\frac{1}{\|\mathbf{w}\|}\right).$$

Note: Maximizing δ^* is the same as minimizing $\|\mathbf{w}\|$, which, in turn is the same as minimizing $\|\mathbf{w}\|^2 = \mathbf{w}\cdot\mathbf{w}$. Therefore, we can represent our problem as the following **optimization problem**:

Objective Function:

$$\min_{\mathbf{w},b}\left(\frac{\|\mathbf{w}\|^2}{2}\right)$$

Subject to constraints:

$$y_i(\mathbf{w}\cdot\bar{x}_i + b) \geq 1, \forall \bar{x}_i \in X$$

Soft Margin SVMs: Linear and Non-Separable Cases

If either

- Our training set contains points that make it linearly inseparable, or
- We suspect that some data points we will be asked to classify later will fall inside the margins of the support vector hyperplane,

we can correct our objective function to account for that. Essentially, in this case, we need to change the definition of support vectors. In a linearly non-separable cases, **support vectors are**:

- all support vectors from the linearly separable problem (i.e., all points that lie on the right side of the classification hyperplane at the distance 1 from it);
- all points from each class that lie on the wrong side of the hyperplane.

Because we added another group of *support vectors*, we can no longer use

$$y_i(\mathbf{w}\cdot\bar{x}_i + b) = 1$$

as the condition that determines which points are *support vectors*.

Instead, we introduce *slack variables* $\xi_i \geq 0$ to capture the fact that some support vectors may be closer or further away from the hyperplane:

$$y_i(\mathbf{w}\cdot\bar{x}_i + b) \geq 1 - \xi_i$$

We consider three cases:

1. $\xi_i = 0$. This turns the distance inequality into

   $$y_i(\mathbf{w}\cdot\bar{x}_i + b) \geq 1.$$

   This means that $\bar{x}_i$ is either our old support vector, or any point that lies on the correct side of the hyperplane *further away from the hyperplane* than the "old" support vectors.

2. $0 < \xi_i < 1$. In this case $1 - \xi_i > 0$, and therefore, $\bar{x_i}$ lies on the *correct side* of the hyperplane, but *it is closer to the hyperplane* than the ”old-style” support vectors.

3. $\xi_i ge 1$. In this case, the point $\bar{x_i}$ either lies on the hyperplane (in which case we cannot really classify it) or is on the other side of the hyperplane, in which case it will be misclassified.

Hinge Loss. We define the *soft margin* SVM classifier as a canonical hyperplane $h(\bar{x})$ subject to the following conditions:

Objective Function:

$$\min_{\mathbf{w},b,\{\xi_i\}} \left(\frac{\|\mathbf{w}\|^2}{2} + C\sum_{i=1}^{n} \xi_i \right)$$

Subject to constraints:

$$y_i(\mathbf{w} \cdot \bar{x_i} + b) \geq 1 - \xi_i, \forall \bar{x_i} \in X$$

The function

$$f(d) = \max(0, 1-d)$$

is called the hinge loss function. This function represents the penalty assessed by an SVM construct for a point that is at the directed distance d from a given hyperplane.

- If the point is at the directed distance of 1 or more, it is on the ”correct” side of the hyperplane and therefore, there is no penalty.
- If the point is at the distance between 0 and 1, it lies in the ”neutral zone” between the hyperplane and the support vector from the same class as the point. We give this point a small penalty, because we do not like to see points closer to the hyperplane than the support vector.
- If the point is on the other side of the hyperplane, we penalize it by the distance from this point to the hyperplane that is parallel to ours, but that passes through the support vectors from the point’s class. We do this because for us the penalty is not how far the point is from the hyperplane, but how far it is from ”its” support vectors.

Training SVM Classifiers

Step 1. Get rid of the intercept. This can be accomplished by replacing all vectors $\bar{x} = (a_1, \ldots, a_d) \in X$ with the vectors $\bar{x'} = (a_1, \ldots, a_d, 1)$, and replacing the vector of weights $\mathbf{w} = (w_1, \ldots, w_d)$ with the vector $\mathbf{w'} = (w_1, \ldots, w_d, b)$. (This is a standard procedure that we have seen multiple times already).

Without loss of generality, we assume that all vectors $\mathbf{w}$ and $\bar{x}$ mentioned below have gone through this transformation.

Step 2. Pick the problem to optimize. There are two SVM problems that can be solved: *primal* and *dual*.

The dual problem is solved using Stochastic Gradient Descent, and it is the more commonly used technique. We discuss it in a separate handout.

The primal problem is one of the optimization problems described above (for linearly separable or soft-margin cases). It can also be solved using Stochastic Gradient Descent, but it requires some care.

Step 3. Optimizing the primal problem. Minimize objective function:[4]

$$J(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}(\xi_i)$$

subject to linear constraints

$$y_i(\mathbf{w}\cdot\bar{x_i}) \geq 1-\xi_i;\ \xi_i \geq 0 \text{ for all } i = 1,\ldots,n$$

Let us eliminate ξ_is from the objective function.

$\xi_i \geq 1 - y_i(\mathbf{w}\cdot\bar{x_i})$, and
$\xi_i \geq 0$

imply

$\xi_i = \max(0, 1-y_i(\mathbf{w}\cdot\bar{x_i})$ (i.e., ξ_is are computed via hinge loss function!)

We can now substitute ξ_is for the hinge loss expression in the objective function:

$$J(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\max(0, 1-y_i(\mathbf{w}\cdot\bar{x_i})$$

When $y_i(\mathbf{w}\cdot\bar{x_i} \geq 1$, the hinge loss is 0, and the penalty is not assessed, therefore, we only need to assess the penalty when $y_i(\mathbf{w}\cdot\bar{x_i} < 1)$:

$$J(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{y_i(\mathbf{w}\cdot\bar{x_i})<1}(1-y_i(\mathbf{w}\cdot\bar{x_i}))$$

To solve this problem using stochastic gradient descent, we need to compute partial derivatives. This requires some care, because the hinge loss function is not differentiable at $x = 1$. We need to write the partial derivative out as follows:

$$\frac{\partial\frac{1}{2}\|\mathbf{w}\|^2}{\partial w_j} = \frac{\partial\frac{1}{2}\sum_{i=1}dw_i^2}{\partial w_j} = w_j$$

Let $\bar{x} = (a_1,\ldots,a_d)$ For the hinge loss function $L(\bar{x}, y) = \max(0, 1-y(\mathbf{w}\cdot\bar{x}))$, the partial derivative can be built as follows:

[4]In general, there are two types of objective functions to optimize, the *hinge loss* function, which uses the hinge-loss penalties ξ_i, and the *quadratic loss* function, which uses the squares of hinge-loss penalties. We concentrate on the hinge loss optimization here.

$$\frac{\partial L}{\partial w_j} = \begin{cases} 0 & \text{if } y_i \mathbf{w} \cdot \bar{x} \geq 1 \\ -ya_j & \text{otherwise} \end{cases}$$

The overall partial derivatives are:

$$\frac{\partial J}{\partial w_j} = w_j + C \sum_{y_i \mathbf{w} \bar{x}_i < 1} (-y_i \cdot a_{ij})$$

For Stochastic Gradient Descent we must pick

- C: the misclassification penalty multiplier. Large values of C minimize the number of misclassifications, but also cause the margin δ^* to be small. Smaller values of C will yield more misclassifications, but will also allow for the margin δ^* to be larger. Larger margins means most points are relatively far away from the hyperplane, therefore most points are easier to classify.
- η: the learning rate.
- initial values for $\mathbf{w}$, including the intercept (bias) b.

Run Gradient Descent:

- Compute $\frac{\partial J(\mathbf{w})}{\partial w_j}$
- adjust $w_j \longleftarrow w_j - \eta \frac{\partial J(\mathbf{w})}{\partial w_j}$

Alternatively, when n is very large, perform Stochastic Gradient Descent.

k-Nearest Neighbors Classification (kNN)

k-Nearest Neighbors Classifiers are among some of the simplest classification techniques. They are, **however** surprisingly rather accurate and robust and produce good results for a wide range of datasets.

kNN classifiers follow the prinicplpe of **lazy evaluation** and do not construct the data model until a question is asked.

The principle of **lazy evaluation** is to *postpone any data analysis until an actual question has been asked.*

In case of supervised learning, **lazy evaluation** means **not building a classifier** in advance of reading data from the test data set.

k-Nearest Neighbors Classification algorithm (kNN). kNN is a simple, but surprisingly robust **lazy evaluation** algorithm. The idea behind kNN is as follows:

- The input of the algorithm is a training set $D_{training}$, an instance d that needs to be classified and an integer $k > 1$.
- The algorithm computes the *distance* between d and every item $d' \in D$.

- The algorithm selects k **most similar** or **closest** to d records from D: $d_1, \ldots, d_k$, $d_i \in D$.
- The algorithm assigns to d the class of the plurality of items from the list $d_1, \ldots, d_k$.

Distance/similarity measures. The distance (or similarity) between two records can be measured in a number of different ways.

Note: Similarity measures increase as the similarity between two objects increases. **Distance measures** decrease as the similarity between two objects increases.

1. **Eucledian distance**. If D has continuous attributes, each $d \in D$ is essentially a point in N-dimensional space (or an N-dimensional vector). Eucledian distance:

$$d(d_1, d_2) = \sqrt{\sum_{i=1}^{n} (d_1[A_i] - d_2[A_i])^2},$$

   works well in this case.

2. **Manhattan distance**. If D has ordinal, but not necessarily continuous attributes, Manhattan distance may work a bit better:

$$d(d_1, d_2) = \sum_{i}^{n} |d_1[A_i] - d_2[A_i]|.$$

3. **Cosine similarity**. Cosine distance between two vectors is the cosince of the angle between them. Cosine similarity ignores the amplitude of the vectors, and measures only the difference in their *direction*:

$$sim(d_1, d_2) = \cos(d_1, d_2) = \frac{d_1 \cdot d_2}{||d_1|| \cdot ||d_2||} = \frac{\sum_{i=1}^{n} d_1[A_i] \cdot d_2[A_i]}{\sqrt{\sum_{i=1}^{n} d_1[A_i]^2} \cdot \sqrt{\sum_{i=1}^{n} d_2[A_i]^2}}.$$

   If d_1 and d_2 are *colinear* (have the same direction), $sim(d_1, d_2) = 1$. If d_1 and d_2 are *orthogonal*, $sim(d_1, d_2) = 0$.

Classifier Evaluation

Accuracy Measures

Notation. Let T be a classifier constructed by **any** supervised learning algorithm given a **training set** D.

Let D' be a **test set**, ***drawn from the same data/distribution as*** D.

Let $t \in D'$. As $T(t)$ we denote the **class label** supplied for t by the classifier T.

As $class(t)$ we denote the **actual** class label of t.

As D_{true} we denote the *set of all test cases for which our classifier provides correct prediction*:

$$D_{true} = \{t \in D' | T(t) = class(t)\}$$

As D_{error} we denote the *set of all test cases for which our classifier provides incorrect prediction*:

$$D_{error} = \{t \in D' | T(t) \neq class(t)\}$$

Accuracy. The **accuracy** of the classifier T is:

$$accuracy(T) = \frac{|D_{true}|}{|D'|}.$$

Error rate. The **error rate** of the classifier T is:

$$errorRate(T) = 1 - accuracy(T) = \frac{|D_{error}|}{|D|}.$$

Accuracy Measures for Binary Classification

Binary Classifiers. Many classifiers are **binary**: i.e., the class variable C has only two values. A classifiaction problem with $dom(C) = \{c_1, \ldots c_k\}$, $k > 2$ can be transformed into k classification problems with class variables $C_1, C_2, \ldots, C_k$, such that, $dom(C_i) = \{0, 1\}$. $C_i = 1$ means $C = c_i$.

Classification Errors. Consider a binary classification problem with the class variable C, $dom(C) = \{0, 1\}$, where $C = 1$ is interpreted as *"record belongs to class C"* and $C = 0$ is interpreted as *"record does not belong to class C*.

Let T be a classifier for C. Let D' be a test dataset. Given $t \in D$, we can observe four possibilities:

1. **True Positive:** $T(t) = class(t) = 1$;
2. **True Negative:** $T(t) = class(t) = 0$;
3. **False Positive:** $T(t) = 1; class(t) = 0$;
4. **False Negative:** $T(t) = 0; class(t) = 1$;

There are **two types of errors of classification**:

1. **Type I error:** a.k.a. **error of commission** a.k.a. **false positive**: classifier incorrectly classifies a tuple as belonging to class C.

2. **Type II error:** a.k.a. **error of omission** a.k.a. **false negative**: classifier incorrectly classifies a tuple as NOT belongingto class C.

Notation. Conisder the following notation:

1. D_{TP} : set of all **true positives** in D'; $TP = |D_{TP}|$;

2. D_{TN} : set of all **true negatives** in D'; $TN = |D_{TN}|$;

3. D_{FP} : set of all **false positives** in D'; $FP = |D_{FP}|$;

4. D_{FN} : set of all **false negatives** in D'; $FN = |D_{FN}|$;

Confusion Matrix. The information about the accuracy of a **binary classifier** is usually arranged in a form of **confusion matrix**:

	Classified Positive	Classified Negative
Actual positive	TP	FN
Actual negative	FP	TN

Precision. **Precision** of the classifier is the percentage of the correctly *positively* classified records in the set of all positively classified records:

$$precision(T) = \frac{TP}{TP + FP}.$$

Precision measures *how accurately the classifier selects positive examples*, it reaches 100% when the classifier *admits no false positives*.

Recall. **Recall** of the classifier is the percentage of all correctly *positively* classified records in the set of all actual positive records:

$$recall(T) = \frac{TP}{TP + FN}.$$

Recall measures *how successful the classifier is in correctly identifying all positive records.* It reaches 100% when the classifier *admits no false negatives.*

Note: **Precision** and **recall** make sense **only** when combined together.

It is easy to build a classifier with 100% precision: *T(t) = 0* for all $t \in D'$ guarantees that. **But this classifier will have recall of 0**. It is easy to build a classifier with 100% recall: *T(t) = 1* for all $t \in D'$ guarantees that. **But this classifier will have small precision.**

PF. The **PF** measure is defined as:

$$PF(T) = \frac{FP}{FP + TN}.$$

PF measures the *misclassification rate*: the percentage of records **not** in class C that was **incorrectly classified**.

F-measure. The **F-measure** is the harmonic mean of precision and recall:

$$F(T) = \frac{2}{\frac{1}{precision(T)} + \frac{1}{recall(T)}} = \frac{2 \cdot precision(T) \cdot recall(T)}{precicion(T) + recall(T)}.$$

F-measure combines precision and recall into a single number by balancing them against each other.

In some situations, one of the two measures (precision or recall) is more important than the other. **F-measure** can be skewed to favor each. The F_2**-measure** below assumes recall is twice as valuable as precision. The $F_{0.5}$**-measure** below assumes precision is twice as valuable as recall.

$$F_2(T) = \frac{5 \cdot precision(T) \cdot recall(T)}{4 * precision(T) + recall(T)}.$$

$$F_{0.5}(T) = \frac{1.25 \cdot precision(T) \cdot recall(T)}{0.25 * precision(T) + recall(T)}.$$

The formula for F_β, where β represents the relative importance of recall over precision is:

$$F_\beta(T) = \frac{(1+\beta^2) \cdot precision(T) \cdot recall(T)}{\beta^2 * precision(T) + recall(T)}.$$

Evaluation Techniques

In a typical situation, you are given a **training set** D, and are asked to produce a classifier for it.

If all records from D are used to create a classifier, there will be no way to INDEPENDENTLY test its accuracy.

Holdout set. Divide D into two sets: $D = D_{train} \cup D_{test}$; $D_{train} \cap D_{test} = \emptyset$.

D_{test} is called the **holdout set.**

Create a classifier T using D_{train} as the training set. **Test** T using D_{test}.

Holodout set selection:

- **Random sampling.** Select a fraction x. Randomly sample $x\%$ of records from D, put them in D_{test}.

  Commonly, you use around 90% of D as the training set, reserving the remaining 10% for the holdout set.

- **Time slices.** If D consists of ”old” data and ”new” data, then, the training set can include all of the ”old” data, while the holdout set can include the ”new” data. (e.g., in situations where new records appear every day).

Multiple random sampling. This technique is used when D is small.

- Select some number M of repetitions.
- Perform M random samplings of a **holdout set** from D. Run classifier construction on the remaining set D_{train}. Compute the accuracy of the classifier for the current sample.
- Compute the final accuracy as the mean accuracy over all samples.

Multiple random sampling allows us to avoid *flukes* (or, at least, to downgrade their effects).

Cross-Validation. This is a variant of **multiple random sampling** that uses only one random assignment of records, but performs multiple classifications.

- Select n – the number of *slices* of data in D.
- Using *random sampling* split D into n *slices* of equal (or almost equal) size.
- Peform n classification procedures. On step i, use slice D_i as the **holdout set**, while using all other $n-1$ slices as the **training set**.

Note: Standard cross-validations used in practice are **10-fold**, **5-fold** and **leave-one-out** cross-validations.

References

[1] Jure Leskovec, Anand Rajaraman, Jeffrey D. Ullman, *Mining of Massive Datasets*, 2nd Edition, Cambridge University Press, 2014.

[2] Mohammed J. Zaki, Wagner Meira Jr., *Data Mining and Analysis: Fundamental Concepts and Algorithms*, Cambridge University Press, 2014.