Machine Learning

January 23, 2017

Contents

1 Introduction

Predictive (supervised) machine learning: the goal is to learn the mapping from inputs $\vec{x}$ to outputs y, given a labeled set on input-output pairs $\mathcal{D} = \{(\vec{X}_i, y_i)\}_{i=1}^N$. $\mathcal{D}$ is called the training set and N is the number of training examples.

$\vec{x}$ is a D-dimensional vector which is referred to as either features, attributes, or covariates. Most methods assume that y, the output or response variable is categorical or nominal from some finite set, $y_i \in \{1, ..., C\}$ or a real-valued scalar. when y is categorical the problem is termed as classification or pattern recognition. when y is a real-valued scalar it is called regression. A special case is ordinal regression, in which the output space has some natural ordering.

Descriptive (unsupervised) machine learning: In this case the training set consists of only features, $\mathcal{D} = \{\vec{x}_i\}_{i=1}^N$. The goal is to discover knowledge hidden in the training set.

Reinforcement machine learning

Parametric models have the advantage of being faster to use, but they make stronger assumptions about the nature of the data distributions. **Non-parametric models** are more flexible but often the efficiency declines for large data sets.

Miss-classification Rate Denoting our classifier by $f(\vec{x})$

$$\text{erf}(f, \mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{I}(f(\vec{x}_i) \neq y_i)$$

1.1 Supervised Learning

function approximation: assume $y = f(\vec{x})$ for some unknown mapping. our goal is to learn to estimate the function for a given training set. One can then make predictions using this learned function on a new training set, $\hat{y} = f(\vec{x})$. We want to generalize our predictions.

To handle ambiguous cases, it is desirable to predict a probability of classification. we denote the probability distribution over the possible labels, given the input vector $\vec{x}$ and training set $\mathcal{D}$ by $p(y|\vec{x}, \mathcal{D})$. In general, the probability distribution over C possible labels is a vector of lenght C itself. This probability distribution is conditional on the set of inputs and our training set. The classification probability distribution is implicitly conditional on the form of the model used to make the predictions. When choosing between different M models this will be made obvious by writing as $p(y|\vec{x}, \mathcal{D}, M)$.

$$\hat{y} = \hat{f}(\vec{x}) = argmax_{c=1}^{C} p(y|\vec{x}, \mathcal{D})$$

This best guess is the mode. This type of labeling is termed MPA (maximum a posteriori) estimate.

1.1.1 KNN (supervised non-parametric classifier)

Look at K points in the training set that are nearest to the test input $\vec{x}$, and count how many members of each each class are in a given set. Returns the

empirical fraction as an estimate.

$$p(y = c|\vec{x}, \mathcal{D}, K) = \frac{1}{K} \sum_{i \in N_K(\vec{x}, \mathcal{D})} \mathcal{I}(y_i = c)$$

where $N_K(\vec{x}, \mathcal{D})$ are the indices of the K nearest points to $\vec{x}$ in $\mathcal{D}$, and $\mathcal{I}$ is the indicator function:

$$\mathcal{I}(x) = \begin{cases} 1 & \text{if x is true} \\ 0 & \text{if x is false} \end{cases}$$

1.1.2 Regression (supervised parametric)

Linear Regression

$$p(y|\vec{x}, \vec{\theta}) = \mathcal{N}(y|\mu(\vec{x}), \sigma^2(\vec{x}))$$

The model is a conditional probability density.

In the simplest case, we assume μ is a linear function of $\vec{x}$, $\mu = \vec{w}^T\vec{x}$, with fixed noise, $\sigma^2(\vec{x}) = \sigma^2$. In this case, the parameters of the model are $\vec{\theta} = (\vec{x}, \sigma^2)$.

For example, if our input data is one dimensional. The expected response could be represented as:

$$\mu(\vec{x}) = w_0 + w_1 x = \vec{w}^T\vec{x}$$

where w_0 is the bias term. Our input vector would then be $\vec{x} = (1, x)$.

Linear regression can be made to model non-linear relations by replacing $\vec{x}$ with some non-linear function of the inputs $\phi(\vec{x})$. That is,

$$p(y|\vec{x}, \vec{\theta}) = \mathcal{N}(y|\vec{w}^T\phi(\vec{x}), \sigma^2(\vec{x}))$$

This is known as the basis function expansion.

Logistic Regression Generalizing linear regression for binary classification we can replace the Gaussian distribution for y with a Bernoulli distribution,

$$p(y|\vec{x}, \vec{w}) = \text{Ber}(y|\mu(\vec{x}))$$

where $\mu(\vec{x}) = \mathcal{E}[y|\vec{x}] = p(y = 1|\vec{x})$. Furthermore, we compute a linear combination inputs as before but let's add the constraint that $0 \leq \mu(\vec{x}) \leq 1$ by defining

$$\mu(\vec{x}) = \text{sigm}(\vec{w}^T\vec{x})$$

where

$$\text{sigm} \equiv \frac{1}{1 + e^{-\eta}} = \frac{e^{\eta}}{e^{\eta} + 1}$$

1.2 Unsupervised Learning

Density estimation: we want to build models of the form $p(\vec{x}_i|\vec{\theta})$. Notice that supervised learning is conditional density estimation, whereas unsupervised learning is unconditional density estimation. Also, since $\vec{x}_i$ is a vector of features, we need to create a multivariate probability model.

Clustering

Graphs

Discovering Latent Factors: although the data may appear high dimensional there may only be a small number of degrees of variability corresponding to latent factors.

Imputation in matrix completion

2 Notation

Each sample corresponds to a row in a feature matrix $\vec{X}$, $\vec{X} \in \mathbb{R}^{\text{samples} \times \text{features}}$.

We use the superscript i to refer to the ith training sample, and j to refer to the jth dimension of the dataset. We will refer to vectors by $\vec{x}$ where $\vec{x} \in \mathbb{R}^{n \times 1}$ and $\vec{X}$ to denote a matrix $\vec{X} \in \mathbb{R}^{n \times m}$.

3 Perceptron

The proposed system depends on probabilistic rather than deterministic principles for its operation, and it gains its reliability from the properties of statistical measurements obtained from large populations of elements.

All of the equivalent forms compose a transposition set, which we will call T.

The penalty that we pay for the use of statistical principles in the design of the system is a probability that we may get a wrong response in a particular case - i.e.g, a wrong response that is inherent in the nature of the system, rather than due to the malfunction of its components.

A perceptron has three main components: S (sensory), A (association), and R (response) systems. The A-units are characterized by a fixed parameter θ, the threshold value which corresponds to the algebraic sum of input pulses necessary to evoke an output, and the stochastic variable $\vec{v}$, the output value. The R-units are activated when the mean or net value of the signals received from the A-system exceeds a critical level θ_r.

Rosenblat proposed an algorithm that would automatically learn the optimal weight coefficients that are then multiplied with the input features in order to make the decision of whether the neuron fires or not (binary classification).

positive class as 1, negative class as -1. activation function $\phi(z)$ that takes a linear combination of certain input values $\vec{x}$ and a corresponding weight factor $\vec{w}$. z is the net input $z = w_1 + x_1 + ... + w_m x_m$.

$$\vec{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}; \vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \tag{1}$$

If the activation of a particular sample $\vec{x}^{(i)}$, that is, the output of $\phi(z)$ is greater than a defined threshold θ we predict class 1 and class -1, otherwise.

For the perceptron algorithm, the activation function is a simple step function

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

We can bring θ to the left side of the equation and define $w_o = -\theta$ and $x_0 = 1$. So the net input becomes $z = w_0 x_0 + ... + w_m x_m = \vec{w}^T \vec{x}$ and

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Rosenblat's perceptron rule:

1. Initialize the weights to zero or small random numbers.
2. For each sample $\vec{x}^{(i)}$ perform the following steps:
   (a) Compute the output value $\hat{y}$.
   (b) Update the weights.

The output value $\hat{y}$ is the class label predicted by the unit step function and the simultaneous update of each weight $\vec{w}_j$ is done by

$$w_j = w_j + \Delta w_j = w_j + \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

Here η is the learning rate, $\eta \in [0, 1]$. Note that all the weight are updated simultaneously, so we do not recompute $\hat{y}^{(i)}$ before all Δw_j have been updated.

The convergence of our perceptron is only guaranteed for linearly separable classes. Otherwise we should implement a maximum number of epochs.

1. Initialize weights.
2. Calculate expectation value based on activation function.
3. Update the weights.
4. Repeat steps 2 and 3.

3.1 Implementation

```
import numpy as np

class Perceptron(object):
    """Perceptron Classifier.
    by Sebastian Raschka
    (https://github.com/rasbt/python-machine-learning-book)

    Parameters:
    ------------
    eta : float
        Learning rate (or bias) [0,1]
    n_iter: int
        Iterations.

    Attributes
    ----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclasifications in every epoch.
    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter
```

```
def fit(self, X, y):
    """Fit training data

    Parameters
    ----------
    X : {array-like}, shape = [n_samples, n_features]
        Training vectors.
    y : array-like, shape = [n_samples]

    Returns
    -------
    slef : object
    """
    self.w_ = np.zeros(1 + X.shape[1])
    self.errors_ = []

    for _ in range(self.n_iter):
        errors = 0
        for xi, target in zip(X,y):
            errors += self._update_weights(xi, target)
        self.errors_.append(errors)
    return self

def _update_weights(self, xi, target):
    """Apply ADALINE learning rule to update weights"""
    output = self.activation(xi)
    error = ( target - output )
    self.w_[1:] += self.eta * error * xi
    self.w_[0] += self.eta * error
    errors = int(error!=0.0)
    return errors

def net_input(self,X):
    """Calculate net input"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def activation(self,X):
    """Compute Linear Activation"""
    return np.where(self.net_input(X)>=0.0, 1,-1)

def predict(self, X):
    """Return class label"""
    return np.where(self.net_input(X)>=0.0, 1,-1)
```

4 ADAptive LInear NEurons (ADALINE)

The Adaline algorithm is particularly interesting because it illustrates the key concept of defining and minimizing cost functions.

The key difference between Adaline and the perceptron is that the weights

are updated based on a linear activation function rather than a unit step function.

In Adaline the linear activation function is simply the identity function of the net input, $\phi(z) = z = \vec{w}^T\vec{x} = \sum_i w_i x_i$.

While the linear activation function is used to learn the weights, a quantizer can be used to predict the class labels.

4.1 Minimizing cost functions with gradient descent

The cost function must be optimized during the learning process. In the case od Adaline the cost function J is the sum of squared errors

$$J(\vec{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi(z^{(i)})\right)^2$$

A nice property of our cost function is that it is convex so we can use gradient descent to find the weights that minimize our cost function to classify the samples. We take a step away from the gradient of our cost function in order to update the weights.

so,

$$\begin{aligned}\vec{w}_j &= \vec{w}_j + \Delta\vec{w}_j = \vec{w}_j - \eta\nabla_j J(\vec{w}) \\ &= \vec{w}_j + \eta\left(y^{(i)} - \phi(z^{(i)})\right)x_j^{(i)}\end{aligned}$$

Although very similar to the perceptron, the cost function is a number and not a integer. Furthermore, the weight update is based on all samples in the training set, instead of updating the weights incrementally after each sample.

4.2 Implementation

```
import numpy as np

class AdalineGD(object):
    """ADAptive LInear NEuron classifier via gradient descent
    by Sebastian Raschka
    (https://github.com/rasbt/python-machine-learning-book)

    Parameters:
    ------------
    eta : float
        Learning rate (or bias) [0,1]
    n_iter: int
        Iterations.

    Attributes
    ------------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclasifications in every epoch.
```

```
    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Fit training data

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors.
        y : array-like, shape = [n_samples]

        Returns
        -------
        slef : object
        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for _ in range(self.n_iter):
            self.cost_.append(self._update_weights(X,y))
        return self

    def _update_weights(self, X, y):
        """Apply ADALINE learning rule to update weights"""
        output = self.activation(X)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
        cost = 0.5 * (errors**2).sum()
        return cost

    def net_input(self,X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self,X):
        """Compute Linear Activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label"""
        return np.where(self.activation(X)>=0.0, 1,-1)
```

5 Stochastic Gradient Descent

Instead of updating the weights based on the sum of the accumulated errors we update the weights incrementally for each training sample. It typically reaches

convergence much faster because of the more frequent weight updates. Since each gradient is calculated based on a single example, the error surface is much noisier than in gradient descent which can also help escaping shallow minima more rapidly.

To obtain accurate results in stochastic gradient descent it is important to feed it data at random. To achieve this we will shuffle the training set for every epoch to prevent cycles.

The fixed learning rate will be replaced by an adaptive one that decreases over time,

$$\eta = \frac{c_1}{\text{time} + c_2}$$

By using an addpative learning rate we can improve the annealing and get an answer closer to the global minimum.

5.1 Implementation

```
import numpy as np
from numpy.random import seed

class AdalineSGD(object):
    """ADAptive LInear NEuron classifier via stochastic gradient descent
    by Sebastian Raschka
    (https://github.com/rasbt/python-machine-learning-book)

    Parameters:
    ------------
    eta : float
        Learning rate (or bias) [0,1]
    n_iter: int
        Iterations.

    Attributes
    ------------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclasifications in every epoch.
    shuffle : bool (default: True)
        If True, shuffles training data set every epoch to prevent cycles.
    random_state: int (default: None)
        Set random state for shuffling and initializing the weights.
    """
    def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.shuffle = shuffle
        self.w_initialized = False
        if random_state:
            seed(random_state)
```

```
def fit(self, X, y):
    """Fit training data

    Parameters
    ----------
    X : {array-like}, shape = [n_samples, n_features]
        Training vectors.
    y : array-like, shape = [n_samples]

    Returns
    -------
    slef : object
    """
    self._initialize_weights(X.shape[1])
    self.cost_ = []

    for _ in range(self.n_iter):
        if self.shuffle:
            X, y = self._shuffle(X, y)
        cost = []
        for xi, target in zip(X,y):
            cost.append(self._update_weights(xi,target))
        avg_cost = sum(cost) / len(y)
        self.cost_.append(avg_cost)
    return self

def _initialize_weights(self, m):
    """Initialize weights to zero"""
    self.w_ = np.zeros(1+m)
    self.w_initialized = True

def _shuffle(self, X, y):
    """Shuffle training set"""
    r = np.random.permutation(len(y))
    return X[r], y[r]

def _update_weights(self, xi, target):
    """Apply ADALINE learning rule to update weights"""
    output = self.activation(xi)
    error = (target - output)
    self.w_[1:] += self.eta * xi.T.dot(error)
    self.w_[0] += self.eta * error
    cost = 0.5 * error**2
    return cost

def net_input(self,X):
    """Calculate net input"""
    return np.dot(X, self.w_[1:]) + self.w_[0]
```

```
def activation(self,X):
    """Compute Linear Activation"""
    return self.net_input(X)

def predict(self, X):
    """Return class label"""
    return np.where(self.activation(X)>=0.0, 1,-1)

def partial_fit(self, X, y):
    """Fit training data without reinitializing the weights"""
    if not self.w_initialized:
        self._initialize_weights(X.shape[1])
    if y.ravel().shape[0] > 1:
        for xi, target in zip(xi, target):
            self._update_weights(xi,target)
    else:
        self._update_weights(X, y)
    return self
```

6 Logistic Regression

Logit function is the logarithm of the odds ratio

$$\text{logit}(p) = \log \frac{p(y)}{1 - p(y)}$$

The logit function takes values in the range of zero to one and maps it onto the reals, which can be used to express a linear relation between feature values and the log-ods:

$$\text{logit}(p(y = 1|\vec{x}; \vec{w})) = \log \frac{p(y|\vec{x}; \vec{w})}{1 - p(y|\vec{x}; \vec{w})} = \sum_i w_i x_i = \vec{w}^T \vec{x} = z$$

Looking at the logarithm of the probability that a sample belongs to a class given its features. We are interested in the inverse problem, that is, given a set of features, how can we arrive at the probability that a given sample belongs to a particular class.

So,

$$p(y|\vec{x}; \vec{w}) = \frac{1}{1 + e^z} = \phi(z)$$

In this case the output of our activation function corresponds to the probability of belonging to class y given its features which are parametrized by the set of weights $\vec{w}$.

By looking at the sigmoid function, it is easy to see that

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

or equivalently,

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

6.1 Cost function for logistic regression

The likelihood L that we want to maximize when building a logistic regression model, assuming that the elements in our set are independent of each other, is

$$L(\vec{w}) = p(\vec{y}|\vec{x};\vec{w}) = \prod_i^n p(y^{(i)}|x^{(i)};\vec{w}) = \prod_i^n \left(\phi(z^{(i)})\right)^{y^{(i)}} \left(1-\phi(z^{(i)})\right)^{1-y^{(i)}}$$

Like always, let's try instead working with

$$l(\vec{w}) = \log L(\vec{w}) = \sum_i^n \left[y^{(i)} \log\left(\phi(z^{(i)})\right) + \left(1-y^{(i)}\right)\left(1-\phi(z^{(i)})\right)\right]$$

Using the logarithm of the likelihood makes it less likely to come across numerical underflow.

As in Adeline, let's optimized the log-likelihood via gradient descent by writing

$$J(\vec{w}) = -\sum_i^n \left[y^{(i)} \log\left(\phi(z^{(i)})\right) + \left(1-y^{(i)}\right)\left(1-\phi(z^{(i)})\right)\right]$$

$$\frac{\partial l(\vec{w})}{\partial w_i} = \left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\frac{\partial \phi(z)}{\partial w_i}$$

$$= \frac{\partial l(\vec{w})}{\partial w_i} = \left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\frac{\partial \phi(z)}{\partial z}\frac{\partial z}{\partial w_i}$$

Notice that

$$\frac{\partial \phi(z)}{\partial z} = \frac{e^{-z}}{\left(1+e^{-z}\right)^2} = \phi(z)\left(1-\phi(z)\right)$$

So finally,

$$\frac{\partial l(\vec{w})}{\partial w_i} = \left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\phi(z)\left(1-\phi(z)\right)\frac{\partial z}{\partial w_i}$$

$$= \left(y\left(1-\phi(z)\right) - (1-y)\phi(z)\right)x_i$$

$$= \left(y-\phi(z)\right)x_i$$

and by an act of the heavens we have arrived at the same update rule as for the adaline classifier.

$$\vec{w}_j = \vec{w}_j + \Delta\vec{w}_j = \vec{w}_j - \eta\nabla_j J(\vec{w})$$

$$= \vec{w}_j + \eta\sum_i \left(y^{(i)} - \phi(z^{(i)})\right)x_j^{(i)}$$

6.2 Implementation

```
import numpy as np

class LogisticRegressionGD(object):
    """Logistic Regression classifier via gradient descent
    by Sebastian Raschka
    (https://github.com/rasbt/python-machine-learning-book)

    Parameters:
    ------------
    eta : float
        Learning rate (or bias) [0,1]
    n_iter: int
        Iterations.

    Attributes
    ----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclasifications in every epoch.
    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Fit training data

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors.
        y : array-like, shape = [n_samples]

        Returns
        -------
        slef : object
        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for _ in range(self.n_iter):
            self.cost_.append(self._update_weights(X,y))
        return self

    def _update_weights(self, X, y):
        """Apply max. likelihood learning rule to update weights"""
        output = self.activation(X)
```

```
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
        cost = -y.dot(np.log(output)) - (1-y).dot(np.log(1-output))
        return cost

    def net_input(self,X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self,X):
        """Compute Sigmoid Activation"""
        z = self.net_input(X)
        sigmoid = 1.0 / (1.0 + np.exp(-z))
        return sigmoid

    def predict(self, X):
        """Return class label"""
        return np.where(self.activation(X)>=0.5, 1,0)
```

This document is supposed to be a companion to:

- Machine Learning A Probabilistic Perspective. https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020.
- Python Machine Learning. https://www.amazon.com/Python-Machine-Learning-Sebastian-Raschka-ebook/dp/B00YSILNL0.
- CS229 Machine Learning. http://cs229.stanford.edu/.