Chapter 1 Introduction

Machine Learning = Computer improves based on Experience

Data Mining = Make useful information of large database

Applications

e.g. Face Detection : Pixel matrix square traverses the whole image, telling whether it is a face or not.

More e.g. Hand-written Digit Detection, Genomes Data ...

Overview of Supervised Training

In supervised learning, training set contains an unordered set of examples (**X** ,**Y**):

X is input vector (represents reason/factor), called features/attributes; **Y** is output vector (represents result/consequence), called target/label.

The goal of training/learning is to produce a (mapping) function that takes X in and generates most expected Y, even for as-yet-unseen examples.

Different ML models are just different types of the **target (mapping) function**. (e.g. classifier/ regressor)

The set of possible output rules (mapping functions) is called **hypothesis space**.

Common Approach

- Define a hypothesis space ==> Choose functions that are thought to be suitable for mapping. (e.g. Classification/ Regression)
- Define error ==> Choose a measurement of deviation (e.g. Number of incorrect/ Squared Errors)
- Develop an algorithm from hypothesis space that minimize the error of training set
- Feed the training set for learning and examine the performance with testing set

Chapter 2 Probabilistic

Assumption

Data examples are drawn randomly, independently from the same but unknown distribution (a.k.a. IID).

Under this kind of assumption, all data example (**X** ,**Y**) ~ *P(x,y)* (obey distribution *P*).

Bayes' Theorem

$$p(x=\vec{X},y=\vec{Y})=p(y=\vec{Y}|x=\vec{X})p(x=\vec{X})=p(x=\vec{X}|y=\vec{Y})p(y=\vec{Y})$$

$$Prior\ Probablity\ =p(y=\vec{Y})$$
$$Conditional\ Probablity\ =\ p(x=\vec{X}|y=\vec{Y})$$

Explanation

- The joint probability is calculated by prior probability (of y/result occurring) *p(y=Y)* times conditional probability (of x/reason occurring given the condition of y/result) *p(x=X|y=Y)*. The posterior probability has the same form of conditional, and is interpreted as when result y happens, how possible this is caused by reason x.
- Prior probability: An event will happen before you taken any new evidence into account. That is probability before take reasons into account, just calculating the probability of result from collected data.
- Posterior probability: An event will happen after all evidence or background information has been taken into account. When given the result of an event, we can narrow the range down of data needed for search.

Example

According to previous data of games played,

P(x=[teamwork, :],y=wining)=0.4, P(x=[no teamwork, :],y=wining)=0.2,

P(x=[teamwork, :],y=losing)=0.1, P(x=[no teamwork, :],y=losing)=0.3,

P(x=[: , good weather], y=wining)=0.3, P(x=[: , bad weather],y=wining)=0.3,

P(x=[: , good weather], y=losing)=0.2, P(x=[: , bad weather],y=losing)=0.2,

P(x=[:,:]) are calculated by previous collected statistical data. We can also know prior probability P(y). (P(x) can also be called prior probability by definition, it depends on what we are interested in).

Now a new game is played and lost, I want to know what caused our loss.

Each feature of vector X can be considered separately when calculating posterior probability.

First, we consider the first feature "teamwork":

P(x=[teamwork, :]|y=losing) = 0.1 / 0.4 = 0.25, P(x=[no teamwork, :]|y=losing) =0.3 / 0.4 = 0.75

Then, the second feature "weather":

P(x=[: , good weather]|y=losing) = 0.2 / 0.4 = 0.5, P(x=[: , bad weather]|y=losing) = 0.2 / 0.4 = 0.5

Then other features one by one if applicable.

Naive Bayes Classifier

Bayes-Optimal Classification

To minimize the chance of misclassification, Bayes-optimal rule is used. We always classify **X** by the category y=**Y** that has the biggest probability. That is:

$$f(x) = argmax_{\vec{c}} p(y = \vec{c}|\vec{X})$$

Function argmax() returns y=**c** that makes p(y|**X**) reach max.

Bayes' Rule

In the following example, P(x,y) is obtained by counting previous data, and the target is P(y|x).

Example from lecture slide:

$$\hat{P}(x_2,y)\quad \begin{matrix}\text{Rte-113} & \text{Rte-75}\\ \end{matrix}$$

$$\begin{matrix}\text{cloudy}\\ \text{raining}\\ \text{sunny}\end{matrix}\begin{bmatrix}0.1 & 0.2\\ 0.1 & 0.1\\ 0.4 & 0.1\end{bmatrix}$$

$$\hat{P}(x_2)\quad \begin{matrix}\text{cloudy}\\ \text{raining}\\ \text{sunny}\end{matrix}\begin{bmatrix}0.3\\ 0.2\\ 0.5\end{bmatrix}$$

$$\hat{P}(y|x_2)\quad \begin{matrix}\text{Rte-113} & \text{Rte-75}\\ \end{matrix}$$

$$\begin{matrix}\text{cloudy}\\ \text{raining}\\ \text{sunny}\end{matrix}\begin{bmatrix}0.33 & 0.67\\ 0.50 & 0.50\\ 0.80 & 0.20\end{bmatrix}$$

$$\hat{P}(y)\quad \begin{matrix}\text{Rte-113} & \text{Rte-75}\\ \end{matrix}$$

$$\begin{bmatrix}0.6 & 0.4\end{bmatrix}$$

$$P(A{=}a) = \textstyle\sum_b P(A{=}a, B{=}b)$$

$$\hat{P}(x_2|y)\quad \begin{matrix}\text{Rte-113} & \text{Rte-75}\\ \end{matrix}$$

$$\begin{matrix}\text{cloudy}\\ \text{raining}\\ \text{sunny}\end{matrix}\begin{bmatrix}0.17 & 0.50\\ 0.17 & 0.25\\ 0.67 & 0.25\end{bmatrix}$$

$$P(A{=}a, B{=}b) = P(A{=}a)P(B{=}b \mid A{=}a)$$
$$= P(B{=}b)P(A{=}a \mid B{=}b)$$

$$P(A{=}a \mid B{=}b) = \frac{P(A{=}a,B{=}b)}{P(B{=}b)}$$

$$P(A{=}a \mid B{=}b) = \frac{P(B{=}b|A{=}a)P(A{=}a)}{P(B{=}b)} \quad \text{(Bayes' Theorem)}$$

$$\text{Example: } \hat{P}(\text{Rte-113} \mid \text{sunny}) = \frac{\hat{P}(\text{sunny}|\text{Rte-113})\hat{P}(\text{Rte-113})}{\hat{P}(\text{sunny})} = \frac{0.67\times 0.6}{0.5} = 0.8$$

Approximate Bayes-Optimal Classifier

Data of all feature-value combinations are not always available. (e.g. the possible value of features are m,n,p, then at least m*n*p data sample should be obtained, or there exists combination of feature values that not in record).

Estimated probability is used to solve this, for we cannot get the exact probabilities (even enough samples, the distribution we calculate is only proxy of real one), we then want to know estimated ones.

$$f(x) = argmax_{\vec{c}}\hat{p}(y = \vec{c}|\vec{X})$$
$$\text{" } \hat{} \text{ "} \quad means\ estimation$$

(Approximate) Naïve Bayes (Optimal) Classifier

Naive Bayes assumes that all features of **X** are independent, so the probability of a certain given vector is calculated by the multiplication of probability of each features of the vector.

$$\begin{aligned} BayesClassifier(x) &= argmax_y p(y = \vec{Y}|\vec{X}) \\ &= argmax_y \hat{p}(y = \vec{Y}|\vec{X}) \quad (Approximate: Use\ Estimated\ p) \\ &= argmax_y(\frac{\hat{p}(y = \vec{Y})\hat{p}(\vec{X}|y = \vec{Y})}{\hat{p}(\vec{X})}) \quad (Bayes' Rule) \\ &= argmax_y(\frac{\hat{p}(y = \vec{Y})\prod_{i=1}^{n}\hat{p}(\vec{X}_i|y = \vec{Y})}{\hat{p}(\vec{X})}) \quad (Naive's\ Assumption) \\ &= argmax_y(\hat{p}(y = \vec{Y})\prod_{i=1}^{n}\hat{p}(\vec{X}_i|y = \vec{Y})) \quad (Argmax\ \&Probabilistic\ Assumption) \end{aligned}$$

Example

Add a new feature to previous example, we know :

P(x=[teamwork, : , :]|y=losing) = 0.1 / 0.4 = 0.25, P(x=[no teamwork, : , :]|y=losing) =0.3 / 0.4 = 0.75

Implementation

Here is an example code of naive Bayes classifier that runs efficiently.

Dataset Clarification

X is in shape (m,n) (m data points, n features), *X*[i , j] is an integer value which represents in i-th collected article, how often the feature word j appears. Feature j corresponds to the j-th most common word in articles, and it's categorical rather than numerical (integer 1 does not mean that a word appear once, but means it is in some "not so frequent" range).

Y is in shape (m,1) (m data points, 1 output), *Y*[i] is an integer value which represents in i-th collected article, what is the writer's attitude. 0 is negative, 1 is positive.

Training Function

```
import numpy as np

maxfeatval = X.max().max()
# Gets the max feature value in X, so that all  *X*[i , j] are all in range
(0, maxfeatval)

priorp = np.array([(Y<=0).sum()+1,(Y>0).sum()+1])
priorp = priorp/priorp.sum()
# (Y<=0) and (Y>0) are true/false arrays that satisfy the condition. +1 is
avoiding 0 values.
# Get the prior probability by dividing total amount of article

Yind = np.hstack((Y[:,np.newaxis]<=0,Y[:,np.newaxis]>0))
# [hstack] : concatenation along the second axis
# in this case, hstack is putting the two true/false array side by side so
that each item in Yind means this is negative([T F]) or postive([F T]) review

fcounts = X[:,:,np.newaxis]==((np.arange(maxfeatval+1))
[np.newaxis,np.newaxis,:])
# X is 2-dimensional, first D is article number, second D is feature value,
```

```
    # X[:,:,np.newaxis] -> add void third D
    # np.arange(maxfeatval+1) is 1-dinmensional, first D is all possible feature
    values
    # ((np.arange(maxfeatval+1))[np.newaxis,np.newaxis,:]) -> add void second and
    third D
    # "==" sign in the middle -> compare the two matrix gets a new 3D matrix (with
    boardcasting)
    # fcounts is 3-dimensional. fcounts[x,y,z] is a true/false value, which means
    "on article x's feature y, the value of that feature is z" is t/f

    fcounts = fcounts[:,:,:,np.newaxis]*Yind[:,np.newaxis,np.newaxis,:]
    # this is really intuitive that dimensions of fcounts are article, feat,
    featval(t/f) and void; dimensions of Yind are article, void, void and
    positive(t/f)
    # after boardcasting, there are two fcounts calculates pos/neg separately
    # fcounts[x,y,z,0] (a t/f value) means that "passage x feat y featval z is
    negative" is true or false
    # fcounts[x,y,z,1] (a t/f value) means that "passage x feat y featval z is
    positive" is true or false

    fcounts = fcounts.sum(axis=0)
    # sum by x axis (article), sum are presented an the last dimension. This step
    cancels the article axis, thus result:
    # fcounts[y,z,0] means count of feature y (of all passages)'s value z (how
    many negtive are true),
    # fcounts[y,z,1] means count of feature y (of all passages)'s value z (how
    many positive are true)
    fcounts = fcounts+1 # Laplace smoothing

    fcounts = fcounts/(fcounts.sum(axis=1)[:,np.newaxis,:])
    # consider (fcounts.sum(axis=1) first
    # --> M = [fcounts.sum(axis=1)], sums are presented in the last dimension
    #     this step gets total count in every feature (pos/neg separatly)
    #     M[y,0] means count of how many feat y (of all featval) are negative
    #     M[y,1] means count of how many feat y (of all featval) are positive
    # then fcounts.sum(axis=1)[:,np.newaxis,:] part:
    # --> Add dummy axis to the middle M[:,newaxis,:] for boardcasting
    #     summing up in last step cancels the sceond dimenson, now adding a void
    one
    # finally fcounts/(fcounts.sum(axis=1)[:,np.newaxis,:])
    # --> The final division divides the <feat-featval-count_of_each_featval> (n-
    by-f-by-2 matrix) by <feat-(void)-count_of_each_feat> (n-by-1-by-2 matrix)
```

```
    #     Boardcasting: value automatically duplicated f times along the middle
    axis
    #     result = featval-count_of_each_featval/count_of_each_feat = conditional
    probability
    #     (because calculations presents in the last dimension)
    # fcounts[y,z,0] frequency of feat y that has featval z and it is negative
    # fcounts[y,z,1] frequency of feat y that has featval z and it is positive

    model = (priorp,fcounts)
    # the training result
```

Understanding High Dimension Matrix in Numpy

High dimensional matrices are common in the code of numpy, usually with abstract "new axis".

If regard a N-dimension matrix as N-tuple+1-Value Table Notation, it might be easier to understand. N-tuple corresponds to N heads of table and 1 value is the type of real data in a table

According to this notation, the data flow in training functions can be analyzed again.

```
    Yind = np.hstack((Y[:,np.newaxis]<=0,Y[:,np.newaxis]>0))
    # Yind[article #, neg/pos](t/f for neg/pos)

    fcounts = X[:,:,np.newaxis]==((np.arange(maxfeatval+1))
    [np.newaxis,np.newaxis,:])
    # X[:,:,np.newaxis] -> [article #, feature, void](feature value)
    # ((np.arange(maxfeatval+1))[np.newaxis,np.newaxis,:]) -> [void, void, featval
    range](feature value)
    # "==" sign in the middle ->  [article #, feature, feature val range]
    (boardcasting feature val==feature val in range)
    # result[x,y,z] is a true/false value, which means "on article x's feature y,
    the value of that feature is z" is t/f

    fcounts = fcounts[:,:,:,np.newaxis]*Yind[:,np.newaxis,np.newaxis,:]
    # fcounts[:,:,:,np.newaxis] -> [article #, feature, feature val, void](t/f)
    # Yind[:,np.newaxis,np.newaxis,:] -> [article #, void, void, neg/pos](t/f for
    neg/pos)
    # Result -> [article #, feature, feature val, neg/pos](t/f for feature val *
    t/f for neg/pos), it gets the feature val for neg/pos attitude.
```

```
    # fcounts[x,y,z,0] (a t/f value) means that "passage x feat y featval z is
    negative" is true or false
    # fcounts[x,y,z,1] (a t/f value) means that "passage x feat y featval z is
    positive" is true or false

    fcounts = fcounts.sum(axis=0)
    # [sum(article #), feature, feature val, neg/pos] -> [feature, feature val,
    neg/pos](int)
    # fcounts[y,z,0] means count of feature y (of all passages)'s value z (how
    many negtive are true),
    # fcounts[y,z,1] means count of feature y (of all passages)'s value z (how
    many positive are true)
    fcounts = fcounts+1 # Laplace smoothing

    fcounts = fcounts/(fcounts.sum(axis=1)[:,np.newaxis,:])
    # fcounts.sum(axis=1) -> [feature, sum(feature val), neg/pos] -> [feature,
    neg/pos](int)
    #     M[y,0] means count of how many feat y (of all featval) are negative
    #     M[y,1] means count of how many feat y (of all featval) are positive
    #  fcounts.sum(axis=1)[:,np.newaxis,:] -> [feature, void, neg/pos](int)
    #  fcounts/(fcounts.sum(axis=1)[:,np.newaxis,:])
    #  -> [feature, feature val, neg/pos](int) / [feature, void, neg/pos](int)
    #  -> <feat-featval-count_of_each_featval> (n-by-f-by-2 matrix) / <feat-
    (void)-count_of_each_feat> (n-by-1-by-2 matrix)
    # fcounts[y,z,0] frequency of feat y that has featval z and it is negative
    # fcounts[y,z,1] frequency of feat y that has featval z and it is positive
```

Predicting Function

With the new notation tool, the predicting function code is:

```
    X = testX

    (priorp,condp) = model
    # condp [feature, featval, neg/pos](conditional probability)

    maxfeatval = condp.shape[1]
    # shape of (featval) -> featval range

    # Use log to calculate, make * and / to become + and -
    logpriorp = np.log(priorp)
```

```
    # priorp [neg/pos](probability), so logpriorp [neg/pos](log(probability))
    logcondp = np.log(condp)[np.newaxis,:,:,:]
    # logcondp [void, feature, featval, neg/pos](log(probability))

    Xext = X[:,:,np.newaxis]==((np.arange(maxfeatval))[np.newaxis,np.newaxis,:])
    # X[:,:,np.newaxis] -> [article #, feature, void](feature value)
    # (np.arange(maxfeatval))[np.newaxis,np.newaxis,:] -> [void, void, featval
    range](feature value)
    # Xext -> [article #, feature, featval range](t/f for equal feature value)

    logpr = (logcondp*Xext[:,:,:,np.newaxis]).sum(axis=2).sum(axis=1)
    # logcondp*Xext[:,:,:,np.newaxis]
    # -> [void, feat, featval, positive](lg(condp)) * [article, feat, featval,
    void](t/f) = [article, feat, featval, positive](t/f* lg(condp))
    # t/f* lg(condp) ==> if false, then 0; if true, then lg(condp)
    # add all featurevals' t/f* lg(condp) then sum all features in an article
    # result is [passage, pos/neg](sum_lg_condp)

    logpr = logpr+logpriorp
    # logprior [pos/neg](priorprob)
    # add prior p to all (boardcasting)

    return logpr.argmax(axis=1)
```

Broadcasting

(Copied from numpy official documentation: https://numpy.org/devdocs/user/theory.broadcasting.html)

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes.

The Broadcasting Rule: **In order to broadcast, the size of the trailing axes for both arrays in an operation must either be the same size or one of them must be one.**

The most simple example is [1,1,1] * 2 = [1*2, 1*2, 1*2], actually it is [1,1,1] * [2,2,2] = [1*2, 1*2, 1*2], [2] is broadcast to [2,2,2] for making compatible shape. Other examples see the link to official documentation.

Chapter 3 Linear Regression

Review on Common Approach

The common approach provides a framework of ML: define a hypothesis space, define error, develop an algorithm and train. The Naive Bayes classifier use probabilistic hypothesis and use rate of misclassification as error. Training algorithm is based on Bayes' rule and predict by maximize the probability.

As for linear regression, linear function is used as hypothesis, squared loss is defined as error and the training algorithm is developed for giving argument weights while minimizing error.

Functions

The commonly used linear regression function is:

$$\hat{y}_{\cdot}(\vec{X}) = \sum_{j=0}^{n} \omega_j \vec{X}_{\cdot,j}$$

$$\hat{y}_i : \ Estimated\ value\ of\ y\ at\ data\ point\ i$$

$$\omega_j : \ Weight\ of feature\ j,\ \omega_0 = 1$$

$$\vec{X}_{i,j} : The\ value\ of\ data\ point\ i's\ feature\ j$$

In the equation, **X**(. , j) means that taking all data points into account, which is equal to the whole range (or all possible value) of **X**(i , j). This can be regarded the "coordinate axis" in visualization.

The squared loss function is:

$$L = \frac{1}{m} \sum_{i=1}^{m} (y_i - \sum_{j=0}^{n} \omega_j \vec{X}_{i,j})^2$$

$$m : \ Amount\ of\ data\ point$$

$$y_i : \ Real\ value\ of\ y\ at\ data\ point\ i$$

The learning goal is to find w(j) that makes L minimal.

Minimization of Squared Loss

When **X**(i, j) points are given, variables of loss function are w(j). For w(j) is linear, L(w(j)) is squared, which means L reaches minimal when:

$$\frac{\partial L}{\partial \omega_i} = 0$$

Back to the essence of linear algebra, if w(j) stands for dimension axis, then X(i , j) is coordinate.

$$\begin{aligned}
Define\ \vec{Y} = [y_1, y_2, \ldots, y_n]^T,\ \vec{\omega} = [\omega_0, \omega_1, \ldots, \omega_n]^T \\
L &= \frac{1}{m}\sum_{i=1}^{m}(y_i - \sum_{j=0}^{n}\omega_j \vec{X}_{i,j})^2 \\
&= \frac{1}{m}(\vec{Y} - \vec{X}\vec{\omega})^T(\vec{Y} - \vec{X}\vec{\omega}) \\
\frac{\partial L}{\partial \vec{\omega}} &= \frac{\partial L}{\partial(\vec{Y} - \vec{X}\vec{\omega})}\frac{\partial(\vec{Y} - \vec{X}\vec{\omega})}{\partial \vec{\omega}} \\
&= \frac{1}{m} \times 2(\vec{Y} - \vec{X}\vec{\omega})^T \times (-\vec{X}) \\
&= -\frac{2}{m}(\vec{Y}^T\vec{X} - \vec{\omega}^T\vec{X}^T\vec{X}) \\
When \frac{\partial L}{\partial \vec{\omega}} = \vec{0} = \vec{0}^T,\ L\ reaches\ minimal, \\
\vec{0}^T &= -\frac{2}{m}(\vec{Y}^T\vec{X} - \vec{\omega}^T\vec{X}^T\vec{X}) \\
\vec{0} &= \vec{X}^T\vec{Y} - \vec{X}^T\vec{X}\vec{\omega} \\
\vec{\omega} &= (\vec{X}^T\vec{X})^{-1}\vec{X}^T\vec{Y}
\end{aligned}$$

*Matrix Calculus and exactly the same example: https://blog.csdn.net/nomadlx53/article/details/50849941

$$Note:\ \frac{\partial \vec{A}^T\vec{A}}{\partial \vec{A}} = 2\vec{A}^T\vec{I}$$

When calculating by computer, solve a linear equation (Aw=B) is more stable than reverse and multiplication (w=rev(A)B), however, when calculating by hand, matrix is not the most efficient way, use data points and directly solve formula derivative equals 0 is the easiest.

Feature Mapping for Non-Linear Regression

In linear regression, the vector **X**(i , j) is the value of data point i's feature j. **X**(. , j) looks like [x(.,0), x(.,1), ..., x(.,n)], which are all linear to x. When it comes to non-linear cases, new features should be used. Feature mapping function takes the raw attributes and creates mapped features **Φ**(**X**(. , j)) = [Φ(x(.,0)), Φ(x(.,1)), ... , Φ(x(.,n))].

$$f(x) = \vec{\omega}^T\vec{X} \longrightarrow f(x) = \vec{\omega}^T\phi(\vec{X})$$

Example

$$\left.\begin{aligned} \vec{X} = [1, x_1, x_3, x_2, x_4, x_5] \\ x_3 = x_1^2 \\ x_4 = x_1 x_2 \\ x_5 = sin(x_1) \end{aligned}\right\} \longrightarrow \vec{\phi} = [1, x_1, x_1^2, x_2, x_1 x_2, sin(x_1)]$$

The vector can be regarded as a linear coordinate defined by **Φ**.

Follow the same method of solving linear equation, the estimated weights are:

$$\vec{\omega} = (\vec{X}^T \vec{X})^{-1} \vec{X}^T \vec{Y} \longrightarrow \vec{\omega} = (\vec{\phi}^T \vec{\phi})^{-1} \vec{\phi}^T \vec{Y}$$

Overfitting and Regularization

Always use higher order polynomial and more features as possible can result in an extremely small error (close to 0) for training set, but there is no guarantee that it will work for testing set or any new example. When the regression model becomes more and more complex, the training error decreases but the testing error will increase after some threshold before which it decreases. The phenomenon that results produces too closely or exactly to a particular limited set of data points, and may therefore fail to fit additional data or predict future observations reliably due to the overly complex model is called overfitting.

To discourage learning algorithm from using too many features, regularization method is introduced. Named similar to L1 (absolute deviation) and L2 (squared error) loss function, L1 and L2 regularization represents for absolute and squared magnitude.

$$Lasso\ Regression: \ \lambda \sum_{j=1}^{n} |\omega_j| \ (L1\ Regularization)$$

$$Ridge\ Regression: \ \lambda \sum_{j=1}^{n} |\omega_j^2| \ (L2\ Regularization)$$

Regularization increases with model complexity, because more features bring more weights. Lambda controls the "strictness" of regularization. Small lambda causes overfitting (high variance), overly huge lambda causes underfitting (high bias). There is trade-off between bias and variance that depends on the value of lambda.

- High variance: Possible surfaces are far from the average one, but the average surface is near the correct one (low bias). (Happens when lambda too small)
- High bias: Possible surfaces are near the average one (low variance), but the average surface is far from the correct one. (Happens when lambda too big)

$$L = \frac{1}{m}\sum_{i=1}^{m}(y_i - \sum_{j=0}^{n}\omega_j\vec{X}_{i,j})^2 \longrightarrow L = \frac{1}{m}\sum_{i=1}^{m}(y_i - \sum_{j=0}^{n}\omega_j\vec{X}_{i,j})^2 + \frac{\lambda}{m}\sum_{i=1}^{n}\omega_j^2$$

$$\vec{\omega} = (\vec{X}^T\vec{X})^{-1}\vec{X}^T\vec{Y} \longrightarrow \vec{\omega} = (\vec{X}^T\vec{X} + \lambda\vec{I})^{-1}\vec{X}^T\vec{Y}$$

So, how to pick lambda?

Cross Validation

The answer is to train lambda like we train weight. We split the training set into two parts, the new training set and validation set. Training set is used to train parameters (weight w(j)), validation set is used to train hyper-parameter (lambda) (while validating parameter). The process of training hyper-parameter is to try each different lambda, check performance on validate and pick the (lambda) version that does the best on validation set.

N-fold Cross Validation

N-fold Cross Validation does cross validation for N times using N different train/validation splits, store every lambda trial results each time, and pick the best performed lambda (that has best average of N times). The most common method is to divide the whole training set into N bins, use Bins(i) as validation set at i-th time and the rest as training set. When N equals the amount of data points M, it is called Leave-one-out Cross Validation (**LOO CV**) which means use only one observation as validation data at each run, all the remaining observations as training data.

(Regularized) (Least Squares) (Non-)Linear Regression Algorithm (with Cross Validation)

Algorithm Development:

$$Pick\ function\ from\ hypothesis\ space: \begin{cases} f(x) = \vec{\omega}^T\vec{X} \quad (Linear) \\ f(x) = \vec{\omega}^T\phi(\vec{X}) \quad (Non-Linear) \end{cases}$$

$$Pick\ loss/cost\ function: l(y,f) = (y-f)^2 \quad (Squared\ Loss)$$

$$Pick\ regularizer: R(\omega) = \lambda\vec{\omega}^T\vec{\omega} \quad (L2, Ridge)$$

$$Total\ loss\ function: L = \frac{1}{m}\sum_{i=1}^{m} l(y_i, f(x_i)) + \frac{1}{m}R(\omega)$$

$$Minimize\ solution: \vec{\omega} = (\vec{X}^T\vec{X} + \lambda\vec{I})^{-1}\vec{X}^T\vec{Y}$$

Algorithm Use:

```
    ////////////// Algorithm: LLS with XValid ////////////////////////
```

```
    // Pick a set of lambda
    lambda_candidate = [...]
    // Divide data into Training, Validation and Testing set
    for lambda in lambda_candidate:
      weights = train_model(train_set,lambda)
      avg_loss = loss_function(validate_set,weights,lambda)
      if avg_loss < current_min_loss:
        current_min_loss = avg_loss
        current_best_arg = (weights,lambda)

    //Optional: retrain
    //retrain_weights =
    train_model(train_set+validate_set,current_best_arg.lambda)
    //current_best_arg.weights = retrain_weights

    test_model(test_set,current_best_arg.weights,current_best_arg.lambda)
```

Chapter 4 Binary Classification

Perceptron Learning Algorithm

Perceptron algorithm is for binary classifiers. A binary classifier is a function which can decide whether or not an input, represented by a vector of numbers, belongs to some specific class. It is a type of linear classifier. The value of f(x) is 0(negative) or 1(positive) that represents **x** as either or not a instance of a class. The perceptron learning algorithm does not terminate if the learning set is not linearly separable. Perceptron algorithm is the base of neuron network, and is called single-layer perceptron to distinguish from more complicated multilayer perceptron.

This algorithm pick a random initial vector **w** and draw a (hyper)plane in the space with $\mathbf{w}^{\top}\mathbf{X}$, then adjust **w** by one point data each time.

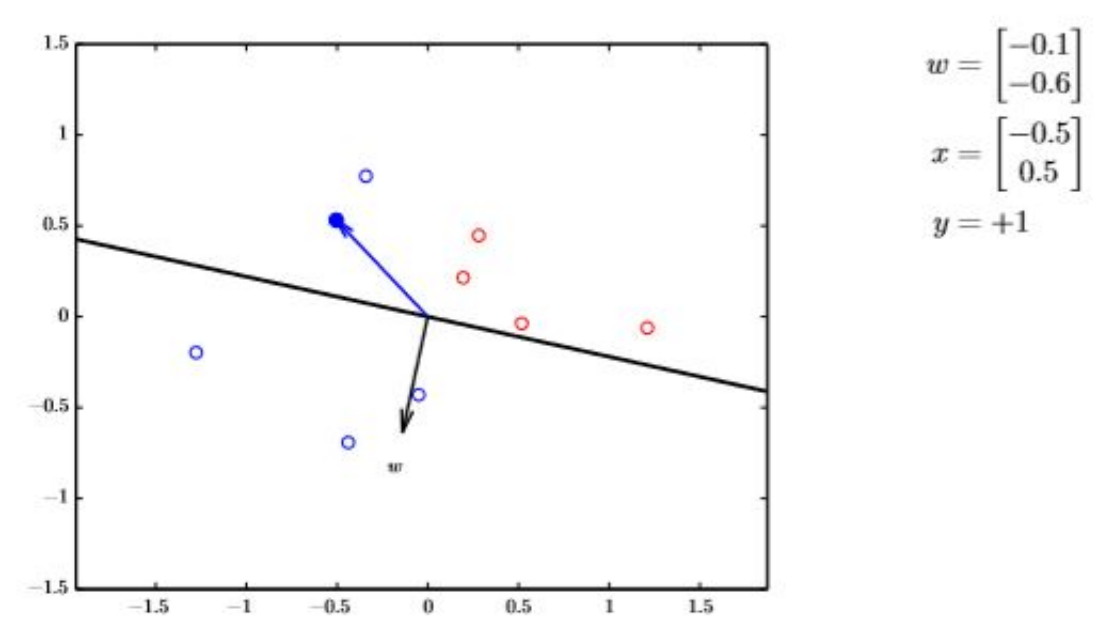
```
    ////////////////////Perceptron Learning Algorithm//////////////////
    // pick a random initial vector w
    w = vector.random()
    // pick a random positive scaler η
    eta = int.random(0,)
```

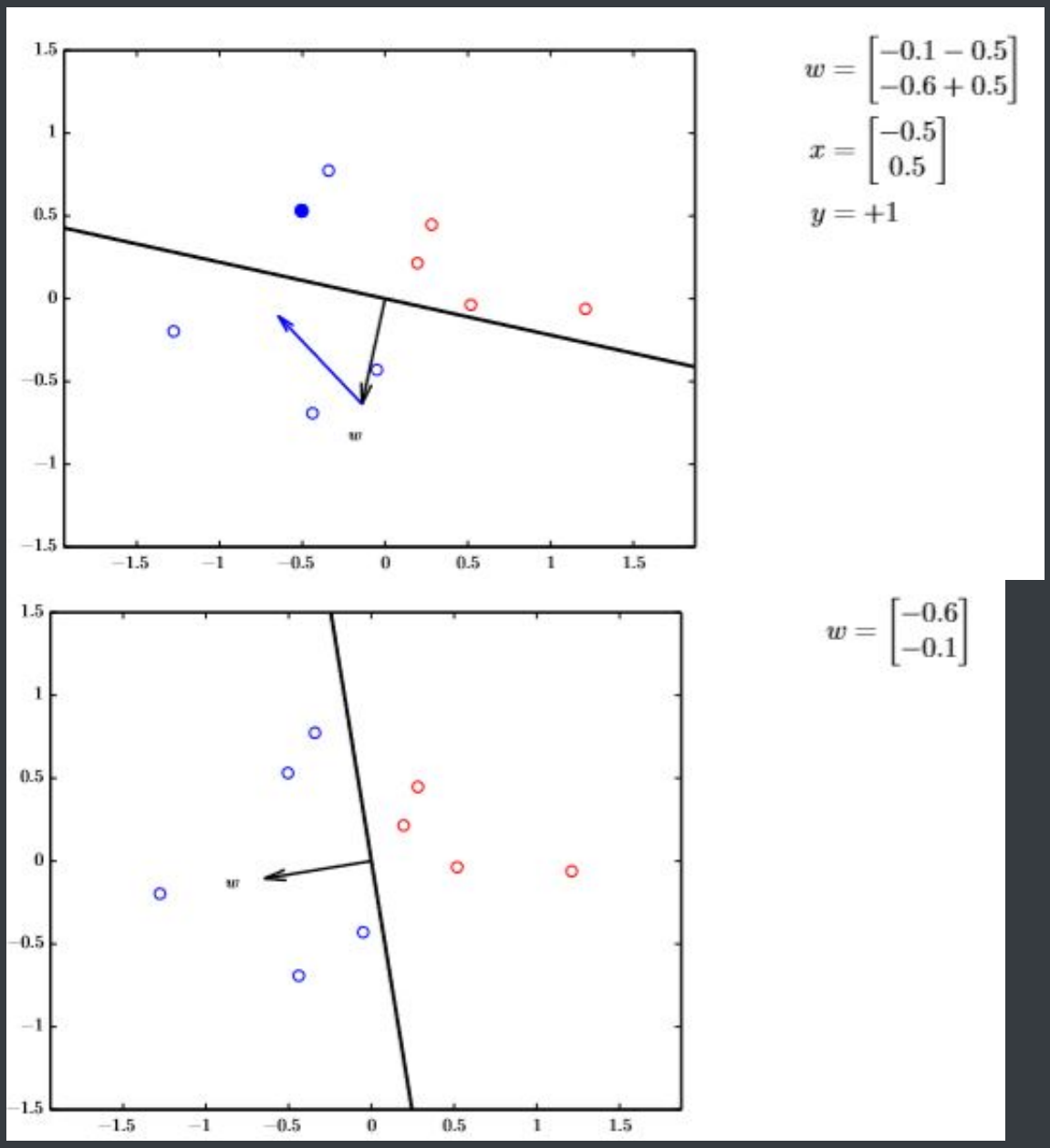
```
while w keeps change:
  for point_i in data_points:
    p = w.transpose * point_i.x // Compute prediction result given by current
model
    if p * point_i.y < 0: //If current model does not match actual
      w = w + eta * point_i.y * point_i.x
      // Change the direction vector of the (hyper)plane

// When w no longer changes, the loop stops
// If the data points are not linear separable, the loop will not stop
```

Example

These are images visualizing the adjusting process. The algorithm update direction vector of plane by adding the direction vector of point (times scaler η).





Binary Classification Loss Functions

In regression, the loss function are defined in terms of (y-f), like |y-f| (L1) and $(y-f)^2$ (L2). In binary classification the loss is in terms of yf, which means whether (actual) y and (predicted) f can match.

$$l(y, f) = max(0, -yf) \quad (1)\ consider\ 'distance'$$
$$l(y, f) = \begin{cases} 0, yf > 0 \\ 1, yf \leq 0 \end{cases} \quad (2)\ only\ consider\ sign$$

In binary classification, if y and f matches, then there is no loss, if y and f cannot match, it should take the penalty. Equation (1) uses the "distance" of misclassified results from actual one, equation (2) tells whether sign of y and f matches.

The perceptron learning algorithm uses equation (1) by trying to minimize:

$$L = \frac{1}{m}\sum_{i=1}^{m} l(y_i, f_i) = \frac{1}{m}\sum_{i=1}^{m} max(0, y_i \vec{\omega}^T \vec{x}_i)$$

(Binary) Logistic Regression

Logistic regression is a statistical model that uses a logistic function to model a binary dependent variable, commonly with two possible values labeled 0 and 1. In regression analysis, logistic regression is estimating the parameters of a logistic model (a form of binary regression).

Example

A set of data in which student's GRE score and whether the student is admitted to a certain university is collected. A high GRE score cannot guarantee the admission but a higher score is logically helps levitate the possibility of getting admitted. In this case we cannot use linear regression because you might see people who have 320 are partly admitted while others not. What we can really get is a probability of admission with some score. This is when logistic regression comes in.

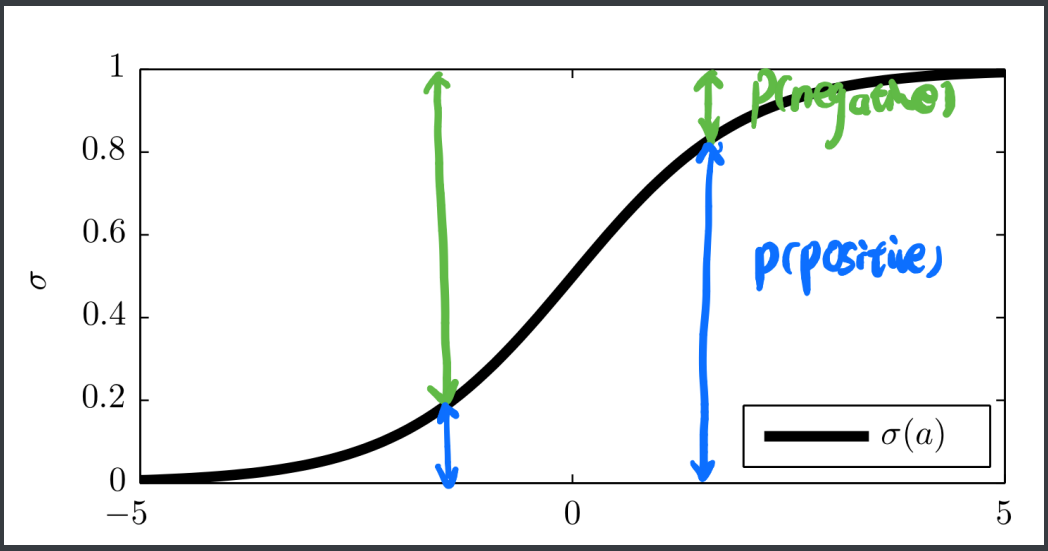
To get the probability (0 to 1), the y range should be remapped from (-inf, +inf) to (0, 1) using sigmoid function:

$$\sigma(\alpha) = \frac{1}{1 + e^{-\alpha}}$$

he derivative of sigmoid happens to be:

$$\sigma'(\alpha) = \sigma(\alpha)(1 - \sigma(\alpha))$$

This function kind of "bend" the decision (hyper)plane. Define the probability as distance between function and line y=1(y=0):



In math form:

$$\left.\begin{aligned} p(y=+1|x) = \sigma(f(x)) = \sigma(\alpha) = \frac{1}{1+e^{-\omega^T x}} \\ p(y=-1|x) = \sigma(f(x)) = 1-\sigma(\alpha) = \frac{e^{-\omega^T x}}{1+e^{-\omega^T x}} = \frac{1}{1+e^{\omega^T x}} = \sigma(-f(x)) \end{aligned}\right\} p(y|x) = \sigma(yf(x))$$

Output classifier for training is

$$f(x) = \omega^T x$$

and the training goal is to maximize the probability of getting correct class y when x is given by trying **w** :

$$argmax_\omega \prod_{i=1}^{m} p(y_i|x_i) \longrightarrow argmax_\omega \sum_{i=1}^{m} ln(p(y_i|x_i)) \longrightarrow argmin_\omega \sum_{i=1}^{m} -ln(p(y_i|x_i))$$

$$L = \frac{1}{m}\sum_{i=1}^{m} -ln(p(y_i|x_i)) = \frac{1}{m}\sum_{i=1}^{m} -ln\sigma(y_i f(x_i)) = \frac{1}{m}\sum_{i=1}^{m} -ln\sigma(y_i \omega^T x_i)$$

When derivative equals 0, L function reaches minimum:

$$-\nabla_w L = \frac{1}{m}\sum_{i=1}^{m}(1-\sigma(y_i\omega^T x_i))y_i x_i$$

Logistic regression solved problems with perceptron's :

- Cannot distinguish between different decision surface
- Does not work on data that is not separable
- W^TX is treated the same as 2W^TX, an extra free parameter
- Difficult for multi-class

Gradient Descent

When derivative equal to 0 cannot be solved analytically, numeric optimization workhorse is used: gradient descent. Gradient descent starts from a random point, returns a local minimal by continuously "going downhill". Suppose L(x) is the function to minimize:

```
//////////////////Gradient Descent Algorithm ///////////////////
W = point.random() //start with a random point
// eta is a positive scaler (learning rate)
while W is not the local minimum:
  grad = L_partial_derivative_to_x(W) //gradient is the steepest ascend
direction
  W += eta * (-grad) //grad points to uphill, so -grad is the fastest secend
direction
```

In calculus, gradient is defined by partial derivative:

$$\nabla_w L = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_n}]^T$$

There is a sly condition (while W is not the local minimum) above. Actually, whether W gets to the local minimum can be told by difference between new and old function output, commonly, if |**η*grad**|< E = 10^{-6}, W is near the local minimum, the algorithm stops here.

Stochastic Gradient Descent

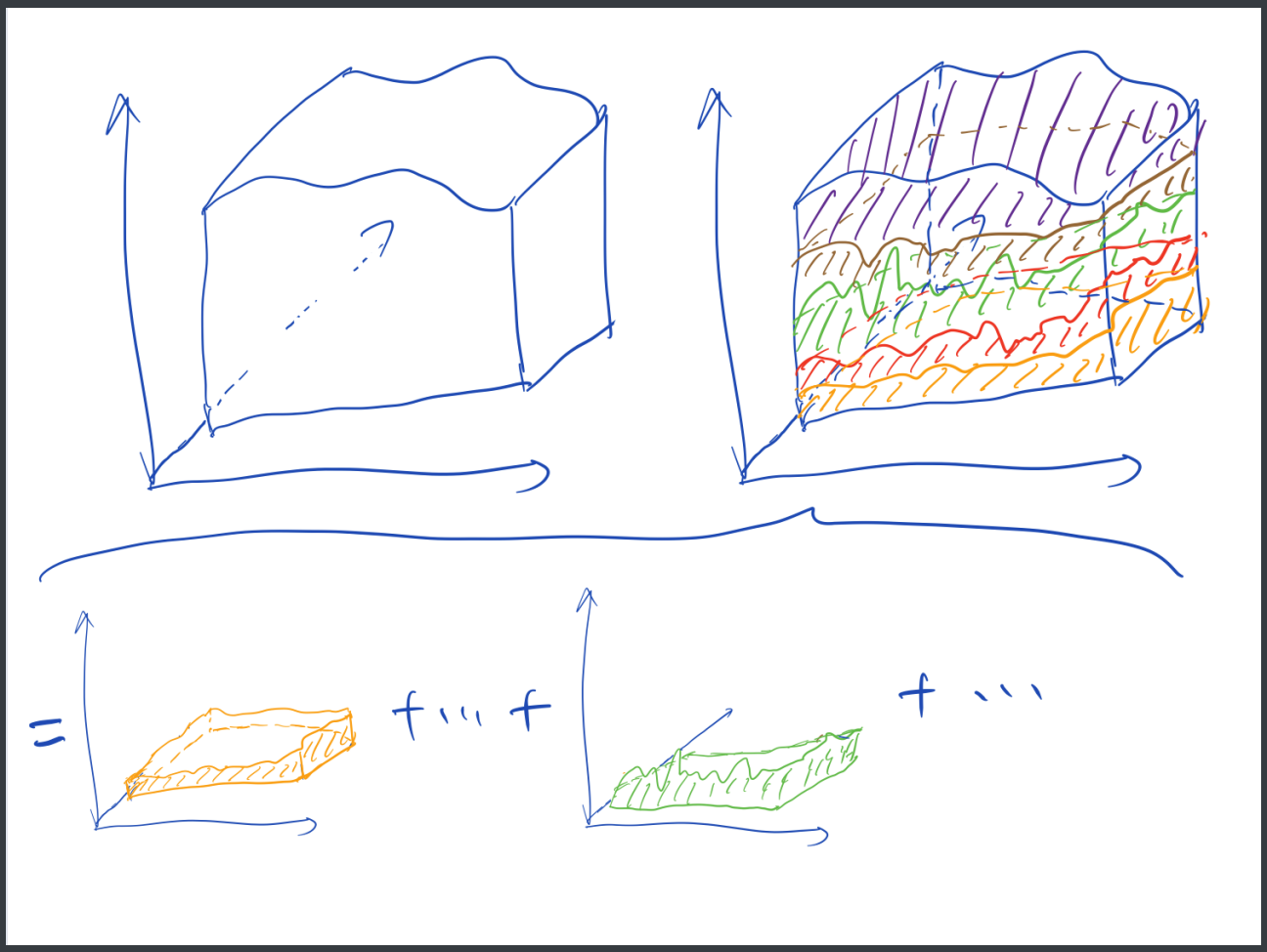
When data capacity becomes big, the gradient descent consumes too much time. It is when Stochastic Gradient Descent comes in handy. SGD does not use total loss function L but use loss of each data point l(i) instead.

If L is in sum form, then:

$$\nabla_w L = \nabla_w (\frac{1}{m} \sum_i l_i) = \frac{1}{m} \sum_i \nabla_w l_i$$

```
//////////////////Stochastic Gradient Descent Algorithm ///////////////////
W = point.random() //start with a random point
// eta (learning rate) is much smaller (1/m) than in GD
while W is not the local minimum:
  for data_i in data_points:
    grad_i = l_i_partial_derivative_to_x(W) //Use i-th derivative item of the
sum
    W += eta * (-grad_i) //Update W instantly instead of waiting for all
points
//Converge faster
```

Explanation



The total loss function (blue) is the sum of loss items (yellow+red+green+...).

GD: (1)Calculate all M loss item's derivative and sum up

(2)Let the newest point go down the "blue hill" with a full step

(3)Check the height difference, then repeat(1)-(2) or stop

SGD: (1)Calculate one of all M loss item's derivative (yellow)

(2)Let the newest point go down the "yellow hill" with a 1/m step

(3)Repeat (1)-(2) for all M loss items

(calculate for red, green,... and go down corresponding "red hill", "green hill",...)

(4)Check stop condition, then repeat(1)-(3) or stop

Example: Logistic Regression with Gradient Descent

```
//////////////////////Logistic Regression w/ Gradient
Descent//////////////////
w = Vector.random()
while w is not a local minimum:
  g = 0
  for i in range(m): # for each data points in dataset
    grad = -(1-sigmoid(y(i)*w.T*x(i)))*y(i)*x(i)
    g = g + 1/m * (-grad)
  w += eta * g
```

```
//////////////////////Logistic Regression w/ SGD//////////////////
w = Vector.random()
while w is not a local minimum:
  for i in range(m): # for each data points in dataset
    grad = -(1-sigmoid(y(i)*w.T*x(i)))*y(i)*x(i)
    w +=  1/m * eta * (-grad)
```

Chapter 5 K-Nearest Neighbor

In a short sentence, k-NN algorithm is letting the nearest K points of the test point vote for a class. If apply k-NN to all the points in space, decision surface that classify different classes can be drawn.

Distance between points can be easily measured by Euclidean or Manhattan method.

$$Euclidean\ (L2\ Metric) : d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

$$Manhattan\ (L1\ Metric) : d(p,q) = \sum_{i=1}^{n}|p_i - q_i|$$

The metric can also be string/graph editing distance if necessary. Editing distance is the minimum number of edits needed to change one string/graph into another (insert symbol, delete symbol, ...).

The laziest training method is doing nothing and putting all work into testing phase, this causes computational concerns. Condensing can prune the dataset for faster prediction.

K-NN is performs well when space dimension is low, computational power is enough and decision surface is irregular.

1-NN vs Bayes Optimal

Bayes optimal gives the argmax of y, which means the probability of correct, P(y|x), is the biggest among all, so the error probability is the smallest. Thus, the error rate can be calculated by:

$$error_{bayes-optimal}(x) = min_y(1 - P(y|x))$$

Suppose the amount of data examples approaches infinite, that means infinite points are around given point x with zero distance. Picking any point is possible, so the error rate is the expectation of every point's error rate.

$$error_{1-NN}(x) = \sum_y P(y|x)(1 - P(y|x))$$

$$P(y|x): \ Probability\ of\ picking\ y$$
$$(1 - P(y|x)): \ Error\ rate\ when\ y\ is\ picked$$
$$Sum\ for\ expectaion$$

$$error_{bayes-optimal}(x) \leq error_{1-NN}(x) \leq 2error_{bayes-optimal}(x)$$

k-NN vs Bayes Optimal

First, for k-NN algorithm, k must be a function of m (number of examples) to get consistency.

If $\lim_{m\to\infty} k(m) = \infty$ and $\lim_{m\to\infty} k(m)/m = 0$, then k-NN error rate converges to the Bayes-optimal.

If $m<\infty$, then no conclusion can be drawn.

Pick Hyper-parameter K

Using the same way in linear regression, K here is like hyper-parameter lambda for regularization. Cross validation is also suitable for K.

Distance metric can also be selected by cross validation.

Attribute Scaling

There are many problems to be solved when use the concept of "distance".

Example(Significant Larger)

Suppose you had a dataset (N examples, 2 features). Feature 1 has values between 0 and 1, while the other feature has values that range from -1000000 to 1000000. When taking the euclidean distance, the values of the feature 1 become uninformative and feature 2 is solely relied on.

Example(Irrelevant Attribute)

Suppose you had a dataset (N examples, 2 features). Feature 1 is logically and numerically related to classification, feature 2 contains randomly generated noise. Clearly, feature 2 is not useful at all, but when calculating distance, it is still counted.

Scaling is not as simple as picking same unit (and picking same unit is not logically just), for those features have no similarity or there are discrete features (binary/ ordinal/ categorical), distance should be calculated or not? How? For irrelevant attributes, scale should be 0 because it is not related to classifying.

Normalization

Scale points into range 0 and 1 with the old-fashioned **range** normalization:

$$x_{new} = \frac{x_{old} - min(X)}{max(X) - min(X)}$$

or **z-**normalization:

$$x_{new} = \frac{x_{old} - mean(X)}{std(X)}$$

$$std(X) = \sqrt{\frac{1}{m}\sum_i (x_i - mean(X))^2}$$

Condensing

The time complexity is O(m), linear to the amount of data point. For large dataset, it costs much time. Condensing is one of the ways for pruning dataset.

```
condense_set = {}
condense_set.insert(point.random(data_set)) //pick a random point to start
c_count = 1 //processed point
while c_count < data_set.count: //while condensing not done
  c_count +=1
  if(classify(m = point.random(condense_set))!= data_set(m).y):
  //if classify by condensed set is not correct
    condense_set.insert(m)
```

Chapter 6 Neural Network

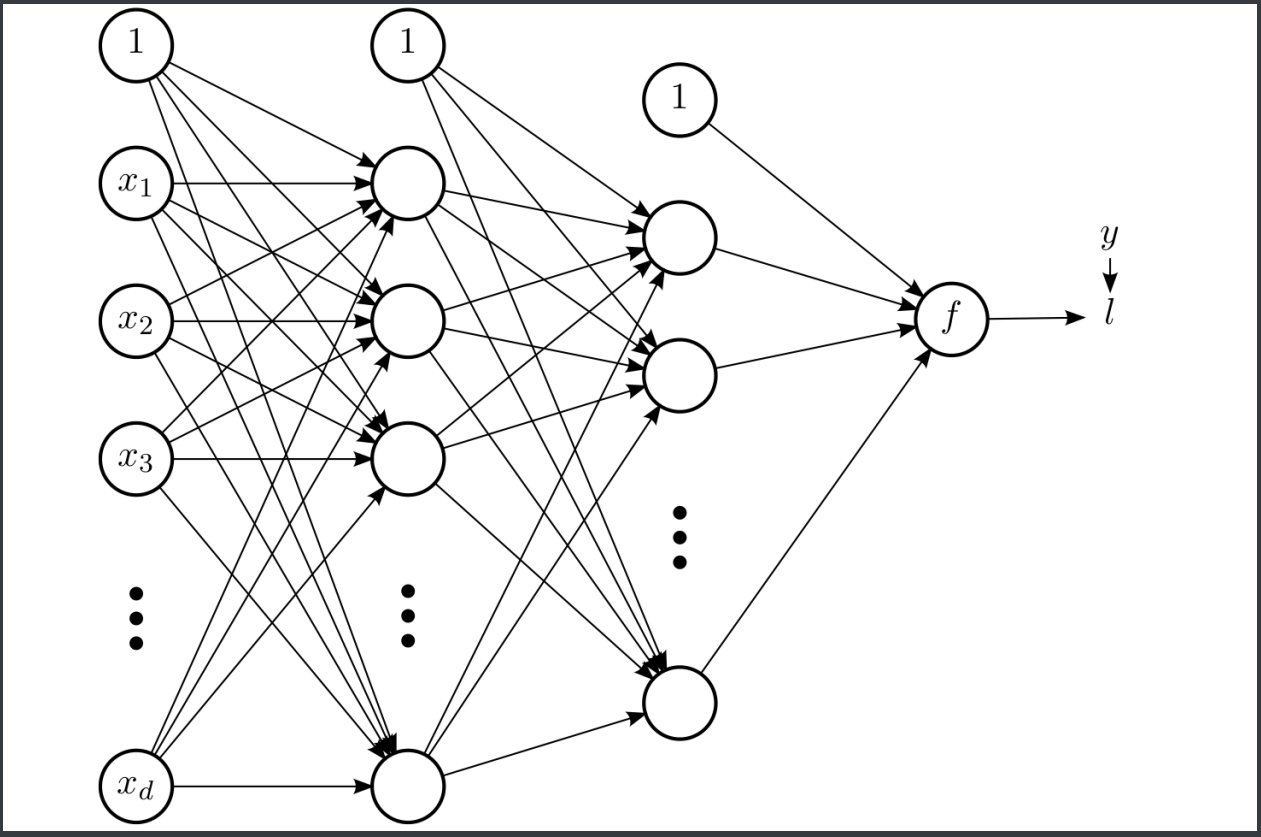
In (Non-) Linear Regression, mapped features $\mathbf{\Phi}(\mathbf{X}(. , j)) = [\Phi(x(.,0)), \Phi(x(.,1)), ... , \Phi(x(.,n))]$ are used instead of raw linear feature [x(.,0), x(.,1), ..., x(.,n)] . However, finding these mapped features are not that easy when the patterns are hard to percept. Neural network is used for solving the problem of selecting these mapped features.

Structure of a Network

The neural network is composed of input layer, hidden layers and output layer. The hidden layers simulates the selection of mapped features and each neurons (except for the bias base '1') connect to all neurons of next layer.

This structure is inspired by the real biological neural network. The principle is: the neurons fire together wire together. The connection, in this model, is weight. Higher weight means higher connectivity. Output layer can have multiple neurons, the following shows only one for the use of binary classification.

Historically, 1 hidden layer with enough units can approximate any function, however, deep learning have many.



Algorithm Development

There are some terminologies needed to be explained before the construction of algorithm.

Forward Propagation

A neuron takes in the sum of last layer's neuron output times weights and put it into an activation function to produce a value between 0 and 1. This is how the input be computed to get the output.

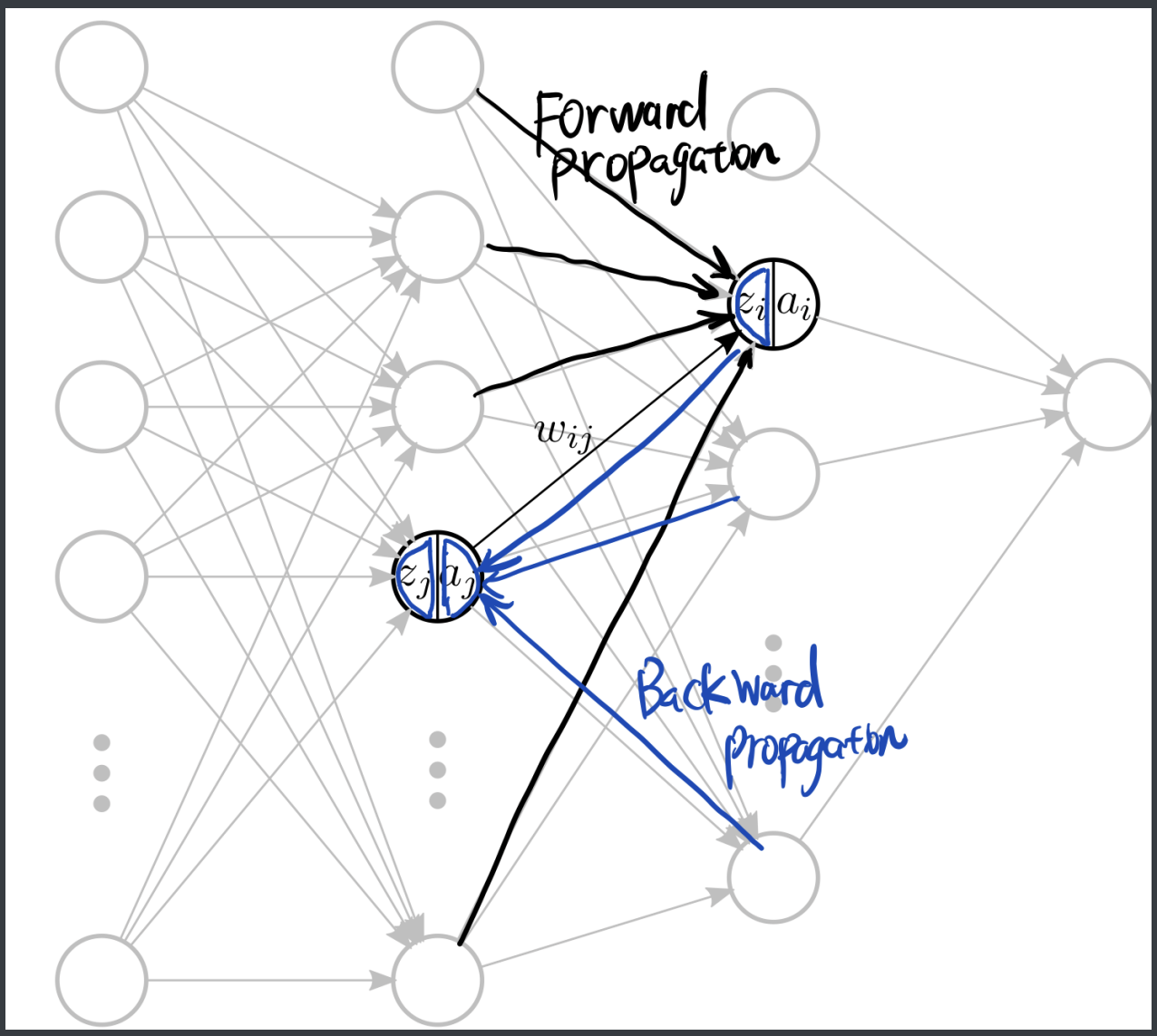
$$W_{ij}^{(L)}\text{: The weight from neuron j in layer L-1 to neuro i in layer L (!)}$$
$$\text{g(x) is (non-linearity) activation function, like sigmoid, ReLU, ...}$$
$$\text{(Pre-activation) } z_i^{(L)} = \sum_j (\omega_{ij}^{(L-1)} a_j^{(L-1)})$$
$$\text{(Activation) } a_i^{(L)} = g(z_i^{(L)})$$

Loss Function

The loss is defined as cross entropy for binary classification and squared error for regression or other classification.

$$l(f, y) = -(ylog(f) + (1 - y)log(1 - f))$$
$$y \in \{0, 1\}$$

Backward Propagation



The only thing we can adjust in the model is weight matrix. The weight matrices are like pipelines and vectors flow through them. What is needed is how weights affects the total loss so that we can make adjustments to weights using the distributed loss of next layer. Once the partial derivative of weight is calculated, use gradient descent or SGD to minimize the loss.

$$\frac{\partial l}{\partial w_{ij}^{(L)}} = \frac{\partial l}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial w_{ij}^{(L)}} \text{ (Chain Rule)}$$
$$= \frac{\partial l}{\partial z_i^{(L)}} \frac{\partial \sum_{j'} (w_{ij'}^{(L)} a_{j'}^{(L-1)})}{\partial w_{ij}^{(L)}}$$
$$= \frac{\partial l}{\partial z_i^{(L)}} a_j^{(L-1)}$$

In order to get unknown item in the above equation is the derivative of z(i) in layer L to loss. Apply chain rule again so that it can be obtained recursively.

$$[\frac{\partial l}{\partial z_j^{(L-1)}}] = \sum_{i' \ in \ layer \ L} (\frac{\partial l}{\partial z_{i'}^{(L)}} \frac{\partial z_{i'}^{(L)}}{\partial a_j^{(L-1)}} \frac{\partial a_j^{(L-1)}}{\partial z_j^{(L-1)}})$$
$$= \sum_{i' \ in \ layer \ L} (\frac{\partial l}{\partial z_{i'}^{(L)}} \frac{\partial \sum_{j'} (w_{i'j'}^{(L)} a_{j'}^{(L-1)})}{\partial a_j^{(L-1)}} g'(z_j^{(L-1)}))$$
$$= g'(z_j^{(L-1)}) \sum_{i' \ in \ layer \ L} ([\frac{\partial l}{\partial z_{i'}^{(L)}}] w_{i'j}^{(L)})$$
$$\delta_{(L-1)} = \delta_{(L)} \cdot g'(z^{(L-1)}) \times \omega^{(L)}, \delta_{(F)} = f - y$$

Non-Linearity (Activation) Function

An activation function takes a pre-activation value and produce activation value. The most commonly used was sigmoid and now is ReLU.

$$\sigma(x) = \frac{1}{1+e^{-x}}, \sigma'(x) = \sigma(x)(1-\sigma(x))$$
$$ReLU(x) = max(0, x)$$

Regularization

Add L2 regularization to loss function:

$$L = \frac{1}{m} \sum_{i=1}^{m} l(f, y) + \frac{\lambda}{m} (\sum_{ijk} \omega_{ij}^{(k)})^2$$

Hyper-parameter lambda is called weight decay.

Thus, the gradient (per data point) is changed:

$$\nabla_l \omega = \frac{\partial l}{\partial w_{ij}^{(L)}} + \frac{\partial \lambda (\sum_{ijk} \omega_{ij}^{(k)})^2}{\partial w_{ij}^{(L)}} = \frac{\partial l}{\partial w_{ij}^{(L)}} + 2\lambda\omega$$

Algorithm

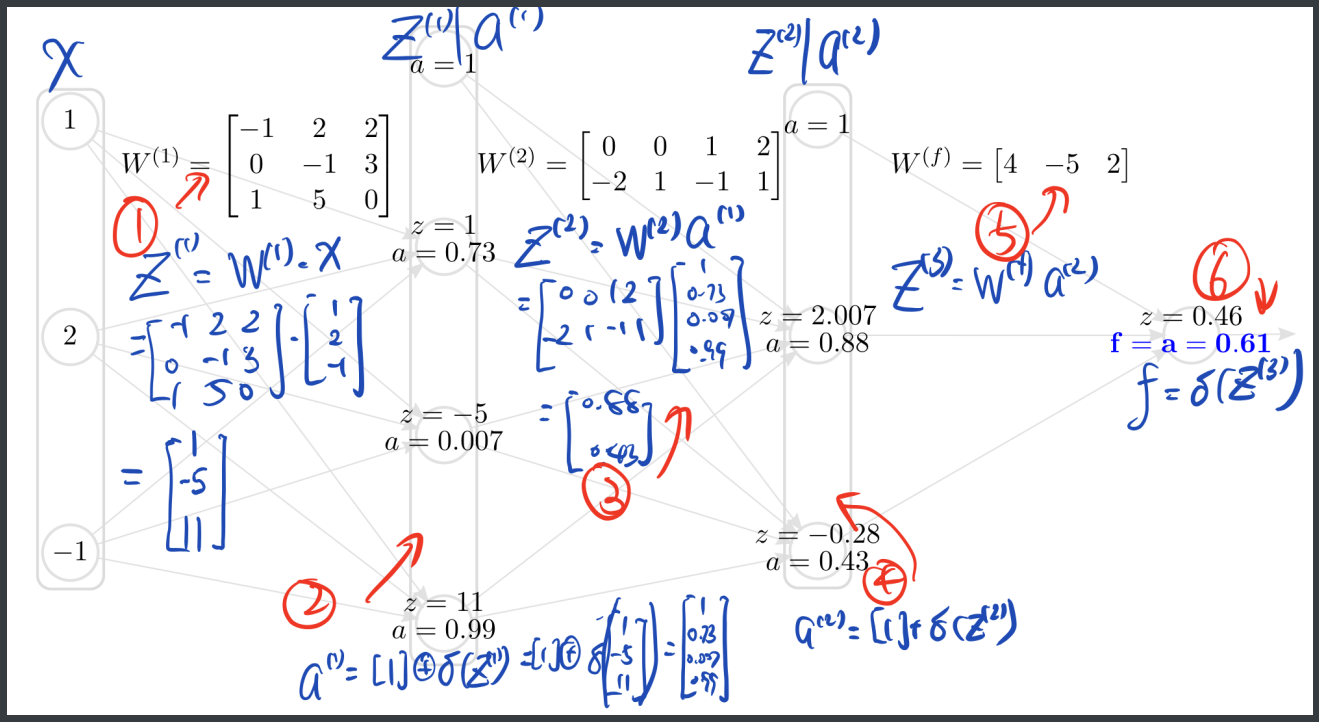
```
While not Converged:
  grad = 0
  for data_point in X.batch:
      f = forward_propagation(data_point)
        delta = backward_propagation(f - Y.batch)
        grad += delta * a.T
    W += -eta*grad
```

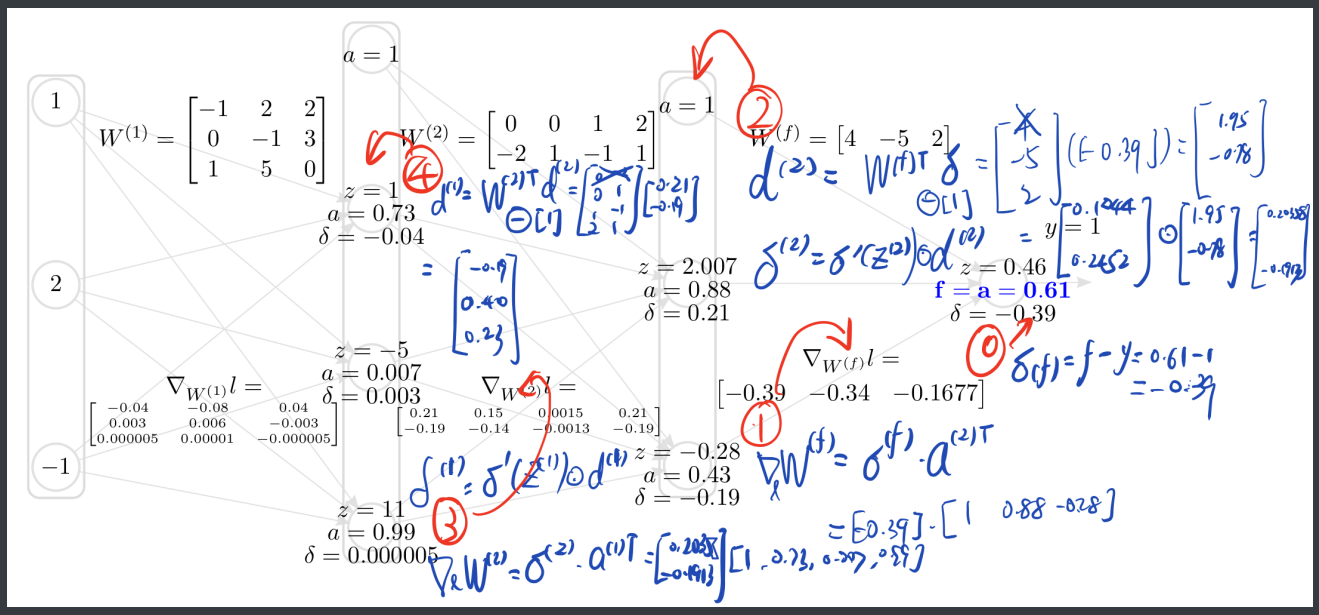
Data batch can be the whole set or part of it (GD) or single point (SGD).

When many saddle points are wanted, use random restart.

Example and Implementation

Example of Calculation





Implementation

Chapter 7 Decision Tree

The decision tree is the classification model in which nodes represent tests and branches are the test results, an input is classified at a leaf node.

Only binary classification (output Y is 0 or 1) is considered to make examples simple as possible.

Greedy Decision Tree Building

- Select binary test of current X that best separates current Y to make the current node
- Recursively run greedy building on separated X (Xt, Xf) and Y (Yt, Yf) to make left and right sub-tree (for left sub-tree, new current X=Xt, current Y=Yt, for right one, new current X=Xf, Y=Yf)
- When no test separates data or test separates no data, return single leaf

What is the criteria of "Best separates"?

Data point are either labeled (Y) with positive+ or negative- in binary classification. When a data point reaches a test, it will either be tested as true or false. The expected situation is a test can put all positive in one category (all in true or all in false). Any deviation is calculated as error.

Misclassification rate is defined as:

$$min(N_-^t, N_+^t) + min(N_-^f, N_+^f) = N^t min(p_-^t, p_+^t) + N^f min(p_-^f, p_+^f)$$

where N(t,-) means number of negative data point are tested true, N(t,+) means number of positive data point are tested true, N(f,-) means number of negative data point are tested false, N(f,+) means number of positive data point are tested false; p(t,-) means portion of negative data point are tested true to all that tested true (N(t,-) divided by N(t)), p(t,+) means portion of positive data point are tested true to all that tested true (N(t,+) divided by N(t)), p(f,-) means portion of negative data point are tested false to all that tested false (N(f,-) divided by N(f)), p(f,+) means portion of positive data point are tested false to all that tested false (N(t,+) divided by N(f)).

When a test separates data well, p+ and p- has a great difference (extreme: p+=1,p-=0), so error is low; When a test cannot separate data, p+ and p- has little difference (extreme: p+=p-=0.5), so error is high. (**Higher Score = Worse, Lower Score = Better**)

There are other **scores** too:

$$N^t(p_-^t p_+^t) + N^f(p_-^f p_+^f) \text{ (Gini Index)}$$
$$N^t(-p_-^t lnp_-^t - p_+^t lnp_+^t) + N^f(-p_-^f lnp_-^f - p_+^f lnp_+^f) \text{ (Cross Entropy)}$$

Therefore,

- Select binary test of current X that best separates current Y to make the current node

Can be translated as:

- Select binary test of current X that has the lowest score

What if X is not categorical but continuous?

Make thresholds to transform continuous to categorical. If X can be any real number, and in observed data, X = 1.5, 2.5, 3.5 ... then use test X<2, X<3, X<4, ... to create categories.

When to stop?

Review the above description:

- When no test separates data or test separates no data, return single leaf

, which easily leads to overfitting. New stop criterion is needed to avoid that.

There are two main kinds of pruning: pre-pruning and post-pruning. Pre-pruning is done while building the tree, it makes building process stops early; post-pruning is done after the whole tree is built and trim nodes that useless.

REP(Reduced Error Pruning) post-pruning algorithm is: prune if doing so does not change or improves pruning set error. Pruning set is like validation set, it is used for pruning the tree. If a sub-tree is pruned, Y value is chosen from training data (or store "If tree stops here, what Y would be" when training).

Reduced Error Pruning on Decision Tree

```
function prune_tree(T):
    if T.left is leaf and T.right is leaf:
        if T.error > T.left.error + T.right.error:
            T.error = T.left.error + T.right.error
        else :
            replace(T)
    else :
        prune_tree(T.left)
        prune_tree(T.right)
```

Note:

- Error is number of misclassification on pruning/validation set. Error of each node does not consider the condition in that node (if stop here, what error would be)
- replace(T) means use the pre-stored Y value ("If tree stops here, what Y would be" from training process) to replace the whole tree T.
- Recursively calling the function make it actually starts from leaf.

Convert Decision Tree to Rule Set

After learning a decision tree, from which a rule set can be converted, make each path from root to leaf as a rule and generate the original rule set.

Then use the following algorithm to prune a rule set:

Rule Set Pruning Algorithm

For each rule in the rule set, exclude one feature of the rule each time and compare the accuracy (on pruning/validation set) of new rule and rule, if new accuracy is not lower (either get higher accuracy or less feature in single rule) then use the new one to replace old one. After this, sort the pruned rule set by accuracy (from high to low, because the rule set is considered in order, if multiple rules match, only the first will be applied).

```
    for rule in rule_set:
        updated = 1
        while len(rule.features)>0 and updated==1:
        # use condition len>0 because the extreme situation is all in same rule
            updated = 0
            for feat in rule.features:
                new_rule = rule.features.exclude(feat)
                if new_rule.accuracy >= rule.accuracy :
                    rule = new_rule # replace old with new, this changes rule_set
    directly
                    updated = 1
    rule_set.sort(key=accuracy,order=high_to_low)
```

Learn a Rule Set Directly

The sections above introduce the process of building and pruning a decision tree, converting tree to rule set and pruning a rule set. However, a rule set can be learned directly.

```
    function learn_one_rule(data):
        rule = []
        best_rule = rule
        while new feature c can be added to rule:
            if rule.performance < (rule.append(c)).performance:
                rule = rule.append(c)
            if rule.perforamce > best_rule.performance :
                best_rule = rule

    function learn_rule_set(data):
        rule_set = []
        while (data):
            new_rule = learn_one_rule(data)
            rule_set.append(new_rule)
            data -= example_correctly_classified_by_new_rule
        rule_set.rescore(all_data)
        rule_set.sort(key=perforamce,order=high_to_low)
```

Note:

- This greedy algorithm learn a rule set from training data, it replaces the (build tree--prune tree--convert tree) process

- By contrast to pruning algorithm, learn one rule function is "add a feature if it improves performance", pruning is "delete a feature if it not degrading the performance"
- Learn rule set function always use current data when learning, which is all_data minus data that correctly classified. However, the re-score of the final step uses all_data
- Performance can be misclassification rate, Gini index or cross entropy just like "score"
- Some conditions like "a rule match no more than 2 are not considered " and "same performance matching more examples win" could be set

Chapter 8 Unsupervised Learning

In the above chapters, supervised learning are discussed, which includes regression (linear, non-linear), classification(Bayes, perceptron, logistic, k-NN) and models do both (neural network, decision tree). The main difference between supervised and unsupervised learning is whether a learning goal is set. Either regression or classification has a goal feature Y, and learn to minimize the error rate of getting Y. In the context of unsupervised learning, there is no specific Y to learn. The only goal is to find the patterns in data.

Association Analysis

Data collected and the pattern of interest is how the data associates. This kind of association is **not causal** (there is no clear reason-result relation), and **not symmetrical** (X->Y cannot result in Y->X).

Example

In a grocery store, the sale records of customer are kept in database. The owner wants to know customers bought certain products would most probably buy what else. Association analysis is used on the records to find out what things are usually bought together. (Famous "beer-diaper" case)

Algorithm Development

Support and Confidence

$$\text{Support: } S(X \to Y) = \frac{\#(X \cup Y)}{\#\{\}} = p(X, Y)$$

$$\text{Confidence: } \alpha(X \to Y) = \frac{\#(X \cup Y)}{\#X} = p(Y|X)$$

The goal is to find rules that has at least S-min support and A-min confidence.

Naive Association Rule Learning

```
T = Generate_all_rules(Data)
rule_set = {}
for rule in T:
    if s(rule)>s_min and a(rule)>a_min:
        rule_set.append(rule)
```

This pseudo-code is just the translation of our goal, and there are multiple way of improving the time cost.

Improvements

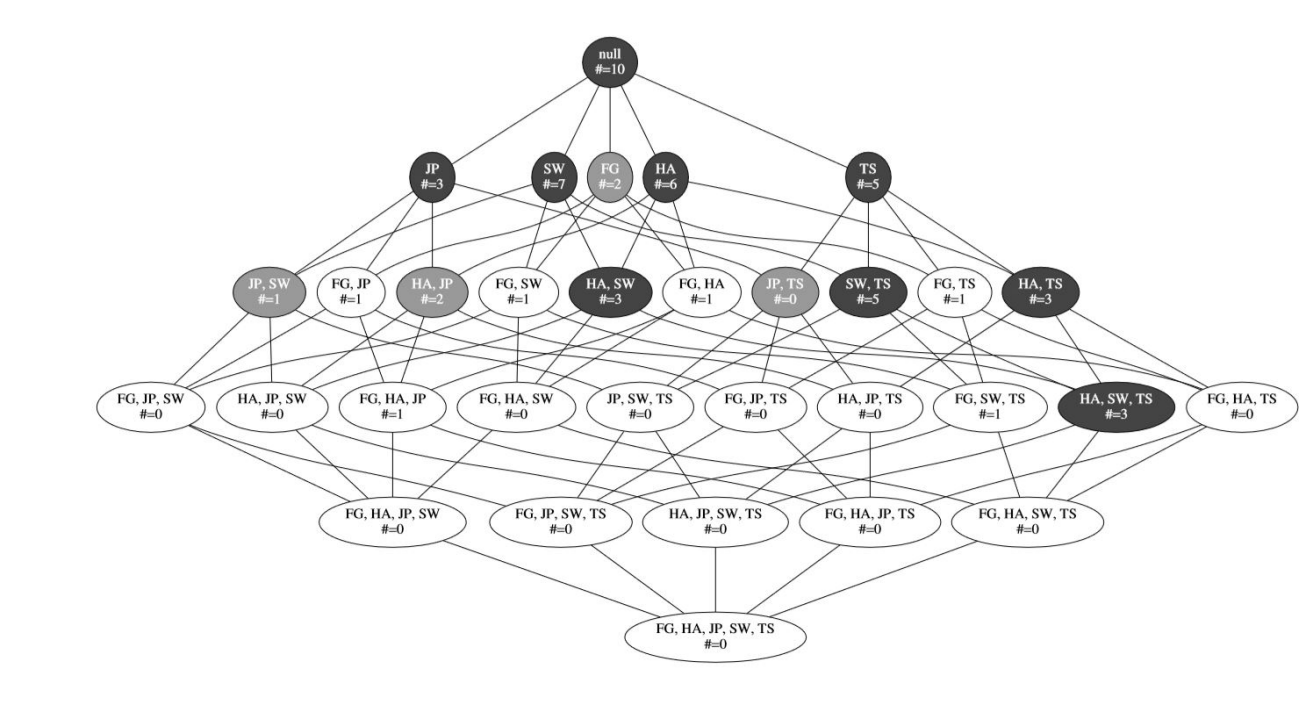
- Support can be calculated when a subset is generated, instead of generate all rules (far more than 2^n, actually $3^n - 2^{n+1} + 1$, see appendix), generate subsets that has at least s_min support.
- Rules comes from subsets, consider part of a subset as X and part as Y, rules are obtained then.

```
L = Generate_all_LARGE_subsets(Data,s_min)
// large subsets means the support=#(subset)/#(all) is more than s_min
for S in L: // S=X+Y, Y=S-X
    for X in S.subsets: // rule now is X->(S-X)
        if #(S)/#(X) >= a_min:
            rule_set.append(X->(S-X))
```

Apriori Algorithm

This algorithm is to replace the *Generate_all_LARGE_subsets* function above. It uses lattice as visualization and generates children of large subset nodes only.

```
function Apriori(Dara, s_min):
    i = 1
    black_nodes(i) = {{i}|i in I and s({i})>s_min} // single item, level 1 in
lattice
    while black_nodes(i).count>0:
        grey_nodes = Apriori_gen(black_nodes(i))
        //generate grey nodes from balck nodes of level i only
        black_nodes(i += 1) = {} //contains all black nodes, level updated to
i+1
        for grey_node in grey_nodes:
            if(s(grey_node)>=s_min):
                L(i).append(grey_node)
```



Clustering

Clustering is defined as partitioning data points into clusters, which are disjoint sets, similar data points are in the same set and different points in different sets. There are two main methods of doing general clustering: agglomerative and divisive. Agglomerative start with every point in its own group, and then merge; Divisive starts with every point on one group and the divide. Clustering is used for finding noise / outliers in data or divide into groups.

k-Means

```
centriods = generate_start_points(k)
while not done:
  for point in data_points:
    i = get_nearest_centroid(centriods)
    point.cluster = i
  for cluster_i in range(cluster_k):
      centroid[i] = average(cluster_i)
```

There are some issues regarding this algorithm:

- Distance metrics: use Euclidean or Manhattan distance
- Start point: choose one randomly, and pick the rest that maximally far from all previous.
- Selection of k : either it is defined or validation can be used to train hyper-parameter

Hierarchal

First, define dissimilarity, which can simply be Euclidean distance, and compute dissimilarity matrix.

Then do the following :

```
while not done:
  (x,y) = argmin(dissimilarity_matrix)
  combine(x,y)
  dissimilarity_matrix.update()
```

There are 3 methods of calculating the distance between clusters:

$$C_1 = \{M_1, M_2, \ldots, M_i\}, C_2 = \{N_1, N_2, \ldots, N_j\}$$

- Single linkage:

$$d(C_1, C_2) = min(d(M_1, N_1), d(M_1, N_2), \ldots, d(M_2, N_1), d(M_2, N_2), \ldots, d(M_i, N_j))$$

- Complete linkage:

$$d(C_1, C_2) = max(d(M_1, N_1), d(M_1, N_2), \ldots, d(M_2, N_1), d(M_2, N_2), \ldots, d(M_i, N_j))$$

- Average linkage:

$$d(C_1, C_2) = \frac{d(M_1, N_1) + d(M_1, N_2) + \ldots + d(M_2, N_1) + d(M_2, N_2) + \ldots + d(M_i, N_j)}{i + j}$$

Chapter 9 Other Techniques

Principle Component Analysis

Principle component is defined as the vector, on direction of which has the most variation for data points. PCA is used for finding these vectors representing principle. This process helps reduce the dimension of data resulting in faster manipulation. (Dimension Reduction & Space Reconstruction)

Assume v is the expected principle component and x is a data point. As the definition says, along the direction of principle, data are most varied, so the distance of data points to v is the minimal. Find v to minimize the projection error:

$$\sum_{i=1}^{m}(x_i^T x_i - (v^T x_i)^2) = \sum_{i=1}^{m}(x_i^T x_i) - \sum_{i=1}^{m}(v^T x_i)^2$$

same as maximizing:

$$\sum_{i=1}^{m}(v^T x_i)^2 = \sum_{i=1}^{m}(v^T x_i)(v^T x_i)^T = v^T(\sum_{i=1}^{m}(v^T x_i))v$$

Use Lagrange multiplier and gradient:

$$Av = \lambda v$$
$$A = \sum_{i=1}^{m}(v^T x_i)$$

which means v is the **eigenvector** of A. Since maximum is needed, v should be eigenvector that has the biggest eigenvalue.

Debugging

Variance-Bias Diagnostics

The common problem can be either high variance and high bias.

High variance: overfitting, error rate on testing set is much higher than training set but both close to target

High bias: underfitting, error rate of testing and training are close but both far from target

Get more training data can reduce variance (learn more, lower variance), reduce the set of feature fix the problem of overfitting (high variance). Increase set of feature, fixes underfitting (high bias).

Optimization Diagnostics

Example: Use SVM instead of LR on the same problem, then plug the learning result of SVM into LR

$$\text{if } Loss_{LR}(w_{LR}) > Loss_{LR}(w_{SVM})$$

then problem is with optimization, because SVM loss is lower. Else the problem is with objective function (loss function)

Error Analysis

Replace each component in a system with 'ground true' (always correct) sequentially, and check the accuracy. Find the biggest increment in accuracy, and that is the component needs to be improved.

Example

Work flow: Comp1 -- Comp2 -- Comp3, overall accuracy = 70%

first, replace Comp1 with ground true (GT -- Comp2 -- Comp3), now overall accuracy = 80%, error in Comp1 = 80% - 70% = 10%

next, replace Comp2 with ground true (GT -- Comp3), now overall accuracy = 85%; error in Comp2 = 85% - 80% = 5%

then, replace Comp3 with ground true (GT), now overall accuracy = 100% (of course); error in Comp3 = 100% - 85% = 15%

Therefore, need to improve Comp3 first.

Ablative Analysis

Delete one component each time and see the drop in accuracy. The component that has the largest decrement is the most helpful one.

Example

Work flow: Comp1 -- Comp2 -- Comp3 -- other overall accuracy = 95%

first, take out Comp1 (Comp2 -- Comp3 -- other), now overall accuracy = 90%, help of Comp1 = 95% - 90% = 5%

next, take out Comp2 (Comp3 -- other), now overall accuracy = 80%; help of Comp2 = 90% - 80% = 10%

then, take out Comp3 (other), now overall accuracy = 78%; help of Comp3 = 80% - 78% = 2%

Therefore, Comp2 provides the most help to the system.

Appendix

Python Function Reference

np.hstack()

np.vstack()

{np_array}.sum()

np.newaxis

{np_array}.argmax()

np.arange()

np.eye()

{np_array}[condition of {np_array}]

np.zeros()

np.logspace()

np.array_split()

np.repeat()

np.linalg.norm()

np.column_stack()

np.linalg.solve()

{np_array}.dot()

np.sign()

Chapter 8, Association Analysis, Amount of Rule

For S that has #S ($1 \leq \#S \leq n$)items, there are $C_n^{\#S}$ to choose from, number of real subsets of S is $2^{\#S} - 2$, which equals the number of possible rules. (Association rules are like ($X \rightarrow S - X$),X is real subset of S)

The number of total possible rule is $A(n) = \sum_{\#S=1}^{n}(C_n^{\#S}(2^{\#S} - 2))$, then
$A(n+1) = \sum_{\#S=1}^{n+1}(C_n^{\#S}(2^{\#S} - 2))$.

$$A(n+1) - A(n) = \sum_{\#S=1}^{n}[(2^{\#S} - 2)(C_{n+1}^{\#S} - C_n^{\#S})] + C_{n+1}^{n+1}(2^{n+1} - 2)$$

For combinations, we have $C_{n+1}^{m} - C_n^{m} = C_n^{m-1}$

$$
\begin{aligned}
\therefore A(n+1) - A(n) &= \sum_{\#S=1}^{n} [(2^{\#S} - 2) C_n^{\#S-1}] + (2^{n+1} - 2) \\
&= \sum_{\#S=1}^{n} (2^{\#S} C_n^{\#S-1}) - 2 \sum_{\#S=1}^{n} C_n^{\#S-1} + 2^{n+1} - 2 \\
&= \sum_{\#S=1}^{n} (2^{\#S} C_n^{\#S-1}) - 2(2^n - 1) + 2^{n+1} - 2 \\
&= \sum_{\#S=1}^{n} (2^{\#S} C_n^{\#S-1}) \\
2A(n) &= 2 \sum_{\#S=1}^{n} (C_n^{\#S} (2^{\#S} - 2)) \\
&= 2 \sum_{\#S=1}^{n} (2^{\#S} C_n^{\#S}) - 2(2^{n+1} - 2) \\
&= \sum_{\#S=1}^{n} (2^{\#S+1} C_n^{\#S}) - (2^{n+2} - 4) \\
2A(n) + (2^{n+2} - 4) &= \sum_{\#S=2}^{n+1} (2^{\#S} C_n^{\#S-1}) \\
&= \sum_{\#S=1}^{n} (2^{\#S} C_n^{\#S-1}) + 2^{n+1} C_n^n - 2C_n^1 \\
&= A(n+1) - A(n) + (2^{n+1} - 2)
\end{aligned}
$$

Therefore, we can know the recursive formula of A(n):

$$
\begin{aligned}
2A(n) + (2^{n+2} - 4) &= A(n+1) - A(n) + (2^{n+1} - 2) \\
A(n+1) &= 3A(n) + 2^{n+1} - 2
\end{aligned}
$$

Start with $A(1) = 0$ and we can finally get A(n):

$$
\begin{aligned}
&A(1) = 0 \\
&A(2) = 3A(1) + (2^2 - 2) \\
&A(3) = 3A(2) + (2^3 - 2) = 9A(1) + 3(2^2 - 2) + (2^3 - 2) \\
&\quad \cdots \\
&A(k) = 3A(k-1) + (2^k - 2) = 3^{k-1} A(1) + 3^(k-2)(2^2 - 2) + \ldots + 3^(k-j)(2^j - 2) + \ldots + (2^k - 2) \\
&\quad \cdots \\
&A(n) = 3^{n-1} A(1) + 3^(n-2)(2^2 - 2) + \ldots + 3^(n-j)(2^j - 2) + \ldots + (2^n - 2) \\
&\qquad = \sum_{j=1}^{n} (3^{n-j}(2^j - 2)) \\
&\qquad = 3^n - 2^{n+1} + 1
\end{aligned}
$$