Machine Learning

Kaige Yang

Contents

Report Draft.

Abstract

This report contains machine learning algorithms, theoretical concepts. The majority of the report follows books including *Machine Learning in Action*, **Hands on Machine Learning with Scikit-learning and TensorFlow**, **Python machine learning**, **Math for Machine Learning** and **Machine Learning and Pattern Recongnization**.

1 Foundations

The quality of fit:
The bias-variance trade-off
The expected loss can be divided into variance, bias and noise.

$$expected\,loss = (bias)^2 + variance + noise \tag{1}$$

bias represents the extent to which the average prediction over all data sets differs from the desired regression function.
variance measures the extent to which the solution for individual datasets vary around their average and hence this measures the extent which the function $f(x)$ is sensitive to the particular choice of dataset.
There is a trade-off between bias and variance with very fexible models have low bias and high variance and relatively rigid models having high bias and low variance.

No free lunch theorem
If you make no assumption about the data, then there is no reason to prefer one model over any other. There is no model that is a prior guaranteed to work between.

In practice, we make some reasonable assumptions about the data and you evaluate only a few reasonable models.

Curse of dimensionality
It describes the phenomenon where the feature space becomes increasing sparse for an increasing number of dimensions of a fixed-size training data-set. Intuitively, we can think of even the closest neighbors being too far away in a high-dimensional space to give a good estimate.

Model selection
Learning curve
cross-validation

2 Classification

Performance measures Evaluating a classifier is often significantly trickier than evaluating a regressor. There are many performance measures available.

Accuracy is generally not the preferred performance measure for classifier, especially when you are dealing with skewed datasets (when some classes are much more frequent than others).

Confusion matrix is a much better way to evaluate the performance of a classifier. The general idea is to count the number of times instances of class A are classified as B.

Each row in a confusion matrix represents an actual class, while each column represents a predicted classed.

$$\begin{array}{cc} & \begin{array}{cc} n & p \end{array} \\ \begin{array}{c} n \\ p \end{array} & \begin{bmatrix} TN & FP \\ FN & TP \end{bmatrix} \end{array} \tag{2}$$

This is the confusion matrix for binary classification. A perfect classifier would have only true positive (TP) and true negative (TN).

The confusion matrix gives you a lot information, but sometimes you may prefer a more concise metric. An interesting one to look at is the accuracy of the positive predictions, this is called the

precision.

$$precision = \frac{TP}{TP + FP} \tag{3}$$

A trivial way to have perfect precision is to make a one single positive prediction and ensure it is correct. This would not be very useful since the classifer would ignore all but one positive instance. So precision is typically used along with another metric named recall, also called sensitivity or true positive rate. This is the ration of positive instances that are correctly detected by the classifier.

$$recall = \frac{TP}{TP + FN} \tag{4}$$

It is often convenient to combine the precision and recall into a single metric called F_1 score, which is the harmonic mean of precision and recall.

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2\frac{precision\ X\ recall}{precision + recall} = \frac{TP}{TP + \frac{FN+FP}{2}} \tag{5}$$

F_1 score favors classifiers that have similar precision and recall. This is not always what you want. For example, If you train a classifier to detect many good videos that are safe for kids. you would prefer high precision and low recall. This is because you prefer to reject many good videos by keep only safe one, instead of having a high recall but lets a few bad videos show up. On the other hand, suppose you train a classifier to detect shoplifters on surveillance images: it is probably fine if your classifier has only 30% precision as long as it has 99% recall (the security guards will get a few false alters, but almost all shoplifters will get caught).

There is a precision/recall trade-off: increasing precision reduces recall and vice versa.

How to control precision and recall?
In binary classification problem, for each instance, the model computes a score based on a decision function and if that score is greater than a threshold, it assigns the instance to the positive class, or else it assigns it to the negative class. Increasing threshold increase precision and reduces recall. On the other hand, lowering the threshold increases recall and reduces precision.

How to decide which threshold to user?
First get the score of each instance, changing the threshold and calculate the precision recall. Finally, plot the precision and recall curve versus the threshold.

Note: A high-precision classifier is not very useful if its recall is too low.

ROC Curve: receiver operating characteristic (ROC) which plots the true positive rate (TPR) (recall) versus false positive rate (FPR) under various threshold. FPR is the ration of negative instances that are incorrectly classified as positive, which is $1 - TNR$ where TNR is the ration of negative instances that are correctly classified as negative. TNR is also called specificity. Hence, ROC curve plots sensitivity (recall) versus 1- specificity.

There is a trade-off: the higher the recall (TNR), the more false positive (FPR).

A good classifier stays as far as away from the diagonal line. The diagonal line represents the ROC curve of a purely random classifier. One way to compare classifiers is to measure the area under the curve (AUC). A perfect classifier will have a ROC AUC equals 1.

Note: Which one to user ROC or precision/recall curve.?
KG: As a rule of thumb, you should prefer the PR curve whenever the positive class is rare or when you care more about the false positive than the false negative.

Multiclass Classification
Multiclass classifiers can distinguish between more than two classes.

Binary classifier can be used for multiclass classification problem via OneVSOne or OneVSAll.

The confusion matrix of multiclass can be visualized as heatmap. The color can help analysing the error.

Multilabel Classfication: In this case, you want to output multiple classed for each instance. For example, recognizes several people on the same picture.

There are many ways to evaluate the multilabel classifiers. One approach is measure the F_1 score of each individual label, then simple compute the average score.

Multioutput classification: It is simply a generalization of multilabel classification where each label can be multicalss.

3 Logistic regression

Logistic regression estimate the probability that an instance belongs to a particular class. If the estimated probability is great than 50%, then the model predicts that the instance belongs to that class, or else it predicts that it does not. This is a binary classifier.

The decision function os logistic regression is

$$\hat{p} = \sigma(\theta^T x) \tag{6}$$

where $\sigma(\cdot)$ is the sigmoid function $[0, 1]$.

$$\sigma(t) = \frac{1}{1 + \exp(-t)} \tag{7}$$

The decision is made by $y = 0, if \hat{p} < 0.5$ and $y = 1, if \hat{p} \geq 0.5$.

When training logistic regression, the idea is to set the parameter such that returns high probability for positive instance and low probability for negative instance.

The objective function is

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{p}_i) + (1 - y_i)\log(1 - \hat{p}_i)] \tag{8}$$

No close-form solution exits, but the objective function is convex.

The gradient is

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(\sigma(\theta^T x_i) - y_i)x_{ij} \tag{9}$$

Logistic regression can be regularized by $l1$ or $l2$ norm.

Logistic regression can be generalized for multiclass problem. This is called softmax regression or multinomail logistic regression. In this case the parameters is a matrix $\Theta \in \mathbb{R}^{k \times d}$. For an input x, the score of class k is

$$s_k(x) = \theta_k^T x \tag{10}$$

The probability of x be class k is

$$\hat{p}_k = \sigma(s(x))_k = \frac{\exp(s_k(x))}{\sum_{j=1}^{K}\exp(s_j(x))} \tag{11}$$

The prediction is

$$\hat{y} = \arg\max_k \hat{p}_k \tag{12}$$

The objective function is cross-entropy

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^i \log(\hat{p}_k^i) \tag{13}$$

The gradient is

$$\bigtriangledown_{\theta_k} J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}(\hat{p}_k^i - y_k^i)x_i \tag{14}$$

Note: softmax regression is not clear in terms of softmax score.

Cross-entropy between two probability distribution p and q is defined as

$$H(p, q) = -\sum_x p(x)\log(q(x)) \tag{15}$$

Note: Kullback-Leibler divergence?

4 Linear Discriminant Analysis

5 Linear Regression

In regression, we aim to find a function f that maps input x to a corresponding function values $f(x)$. We assume we are given a set of training inputs x_n and corresponding noisy observations $y_n = f(x_n) + \eta$. The task is to infer the function f that generates the data and generalizes well to new inputs.

Problem formulation
The functional relationship between x and y is given as

$$y = f(x) + \eta \tag{16}$$

where $\eta \sim \mathcal{N}(0, \sigma^2)$ is independent, identically distributed (i.i.d).

In linear regression, we consider the special case that the parameters θ appear linearly in out model. The likelihood function is

$$p(y|x, \theta) = \mathcal{N}(y|x^T\theta, \sigma^2) \tag{17}$$

The likelihood is the probability density function of y evaluated at $x^T\theta$.

Assume we are given a training set $\mathcal{D} = \{(x_1, y_1), ..., (x_N, y_N)\}$. Denote $\mathcal{X}$ as the input set and $\mathcal{Y}$ as the output set. Assume training instances are independent with each other. The likelihood function is

$$\begin{aligned} p(\mathcal{Y}|\mathcal{X}, \theta) &= p(y_1, ..., y_N|x_1, ..., x_N, \theta) \\ &= \Pi_{n=1}^N p(y_n|x_n, \theta) \\ &= \Pi_{n=1}^N \mathcal{N}(y_n|x_n^T\theta, \sigma^2) \end{aligned} \tag{18}$$

Maximum Likelihood
Maximizing the likelihood means maximizing the predictive distribution of training set given the model parameters. The objective function is

$$\theta_{ML} = \arg\max_\theta p(\mathcal{Y}|\mathcal{X}, \theta) \tag{19}$$

To find the parameters θ_{ML} we can use gradient ascent or close-form solution.

Note: In practice, instead of maximizing the likelihood directly, we apply the log-transformation to the likelihood function and minimize the negative log-likelihood. The reasons that: a) log-transformation does not suffer from numerical underflow. b), the differentiation rules will be simpler.

$$-\log(p(\mathcal{Y}|\mathcal{X}, \theta) = -\log \Pi_{n=1}^N p(y_n|x_n, \theta) = -\sum_{n=1}^N \log p(y_n|x_n, \theta) \tag{20}$$

As the noise is assumed to be Gaussian, the likelihood is also Gaussian.

$$\log p(y_n|x_n, \theta) = -\frac{1}{2\sigma^2}(y_n - x_n^T\theta)^2 + const \tag{21}$$

Then, we have

$$\mathcal{L}(\theta) = -\log(p(\mathcal{Y}|\mathcal{X}, \theta) = \frac{1}{2\sigma^2}\sum_{n=1}^N (y_n - x_n^T\theta)^2 = \frac{1}{2\sigma^2}(\mathbf{y} - \mathbf{X}\theta)^T(\mathbf{y} - \mathbf{X}\theta) = \frac{1}{2\sigma^2}||\mathbf{y} - \mathbf{X}\theta||_2^2 \tag{22}$$

The gradient over θ is

$$\frac{d\mathcal{L}}{d\theta} = \frac{1}{\sigma^2}(-\mathbf{y}^T\mathbf{X} + \theta^T\mathbf{X}^T\mathbf{X}) \tag{23}$$

The solution is

$$\theta_{ML} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{24}$$

If linear features are not sufficiently expressive, we can apply nonlinear transformation $\phi(x)$ to input x.

$$p(y|x, \theta) = \mathcal{N}(y|\phi(x)^T\theta, \sigma^2) \tag{25}$$

Following the same steps, the solution is

$$\theta_{ML} = (\Phi^T\Phi)^{-1}\Phi^T\mathbf{y} \tag{26}$$

Thus far, the noise variance is assumed to be knonw, we can also estimate the variance by treating it as unknown parameter

$$\begin{aligned}\log p(\mathcal{Y}|\mathcal{X},\theta,\sigma^2) &= \sum_{n=1}^{N}\log\mathcal{N}(y_n|\phi(x_n)^T\theta,\sigma^2)\\ &= -\frac{N}{2}\log\sigma^2 - \frac{1}{2\sigma^2}\sum_{n=1}^{N}(y_n-\phi(x_n)^T\theta)^2 + const\\ &= -\frac{N}{2}\log\sigma^2 - \frac{1}{2\sigma^2}s + const\end{aligned} \tag{27}$$

The derivative is

$$\frac{\partial\log p(\mathcal{Y}|\mathcal{X},\theta,\sigma^2)}{\partial\sigma^2} = -\frac{N}{2\sigma^2} + \frac{1}{2\sigma^2}s \tag{28}$$

The solution is

$$\sigma^2_{ML} = \frac{s}{N} = \frac{1}{N}\sum_{n=1}^{N}(y_n-\phi(x_n)^T\theta)^2 \tag{29}$$

Maximum a posterior MAP
We can place a prior distribution $p(\theta)$ on the parameters. We can maximize the posterior distribution $p(\theta|\mathcal{X},\mathcal{Y})$. The posterior over the parameters θ is obtained by applying Bayes' Theorem

$$p(\theta,\mathcal{X},\mathcal{Y}) = \frac{p(\mathcal{Y}|\mathcal{X},\theta)p(\theta)}{p(\mathcal{Y}|\mathcal{X}} \tag{30}$$

The log-posterior is

$$\log p(\theta|\mathcal{X},\mathcal{Y}) = \log p(\mathcal{Y}|\mathcal{X},\theta) + \log p(\theta) + const \tag{31}$$

The objective function is

$$\theta_{MAP} = \arg\min_{\theta} -\log p(\mathcal{Y}|\mathcal{X},\theta) - \log p(\theta) \tag{32}$$

With the Gaussian prior $p(\theta) = \mathcal{N}(0, b^2\mathbf{I})$.

$$-\log p(\theta|\mathcal{X},\mathcal{Y}) = \frac{1}{2\sigma^2}(\mathbf{y}-\Phi\theta)^T(\mathbf{y}-\Phi\theta) + \frac{1}{2b^2}\theta^T\theta \tag{33}$$

The gradient is

$$-\frac{d\log p(\theta|\mathcal{X},\mathcal{Y})}{d\theta} = \frac{1}{\sigma^2}(\theta^T\Phi^T\Phi - \mathbf{y}^T\Phi) + \frac{1}{b^2}\theta^T \tag{34}$$

The solution is

$$\theta_{MAP} = (\Phi^T\Phi + \frac{\sigma^2}{b^2}\mathbf{I})^{-1}\Phi^T\mathbf{y} \tag{35}$$

Ridge regression

$$\theta_{RLS} = \arg\min_{\theta} ||\mathbf{y}-\Phi\theta||_2^2 + \lambda||\theta||_2^2 \tag{36}$$

The solution is

$$\theta_{RLS} = (\Phi^T\Phi + \lambda\mathbf{I})^{-1}\Phi^T\mathbf{y} \tag{37}$$

It can be seen that $\lambda = \frac{\sigma^2}{b^2}$ where the prior $p(\theta) = \mathcal{N}(0, b^2\mathbf{I})$.

Bayesian Linear Regression
Bayesian linear regression pushes the idea of the parameter prior a step further and does not even attempt to compute a point estimate of the parameters, but instead the full posterior distribution over the parameters.

In Bayesian linear regression, the prior is

$$p(\theta) = \mathcal{N}(m_0, S_0) \tag{38}$$

The likelihood is

$$p(y|x, \theta) = \mathcal{N}(y|\phi(x)^T\theta, \sigma^2) \tag{39}$$

The joint probability of observation and parameters is

$$p(y, \theta|x) = p(y|x, \theta)p(\theta) \tag{40}$$

In practice, we are usually not so much interested in the parameter values θ. Instead, we focus often lies in the prediction. In Bayesian setting, we take the parameter distribution and average over all plausible parameters settings. To make predictions at an input x_*, we return the distribution of predictions.

$$p(y_*|x_*) = \int p(y_*|x_*\theta)p(\theta)d\theta = \mathbb{E}_\theta[p(y_*|x_*, \theta)] \tag{41}$$

Note that $p(\theta)$ is the prior distribution of parameters. We choose a conjugate Gaussian prior $p(\theta) = \mathcal{N}(m_0, S_0)$, we obtain the predictive distribution as

$$p(y_*|x_*) = \mathcal{N}(\phi(x_*)^T m_0, \phi(x_*)^T S_0 \phi(x_*) + \sigma^2) \tag{42}$$

Note: How to derive above?

Given a training set $\{\mathcal{X}, \mathcal{Y}\}$, the posterior distribution is

$$p(\theta|\mathcal{X}, \mathcal{Y}) = \frac{p(\mathcal{Y}|\mathcal{X}, \theta)p(\theta)}{p(\mathcal{Y}|\mathcal{X})} \tag{43}$$

and the marginal likelihood (the expected likelihood under the parameter prior).

$$p(\mathcal{Y}|\mathcal{X}) = \int p(\mathcal{Y}|\mathcal{X}, \theta)p(\theta)d\theta = \mathbb{E}_\theta[p(\mathcal{Y}|\mathcal{X}, \theta)] \tag{44}$$

Now, we show what is the marginal likelihood based on parameters prior. We consider he following generative process:

$$\theta \sim \mathcal{N}(m_0, S_0) \tag{45}$$

and

$$y_n|x_n, \theta \sim \mathcal{N}(x_n^T\theta, \sigma^2) \tag{46}$$

The marginal likelihood is given by

$$\begin{aligned} p(\mathcal{Y}|\mathcal{X}) &= \int p(\mathcal{Y}|\mathcal{X}, \theta)p(\theta)d\theta \\ &\int \mathcal{N}(\mathbf{y}|\mathbf{X}\theta, \sigma^2\mathbf{I})\mathcal{N}(\theta|m_0, S_0)d\theta \\ &= \mathcal{N}(\mathbf{y}|\mathbf{X}m_0, \mathbf{X}S_0\mathbf{X}^T + \sigma^2\mathbf{I}) \end{aligned} \tag{47}$$

The final step is derived by estimating the mean and variance.

$$\mathbb{E}[\mathcal{Y}|\mathcal{X}] = \mathbb{E}_{\theta,\eta}[\mathbf{X}\theta + \eta] = \mathbf{X}\mathbb{E}_\theta[\theta] = \mathbf{X}m_0 \tag{48}$$

The variance is

$$\begin{aligned} Cov[\mathcal{Y}|\mathcal{X}] &= Cov_{\theta,\eta}[\mathbf{X}\theta + \eta] \\ &= Cov_\theta[\mathbf{X}\theta] + \sigma^2\mathbf{I} \\ &= \mathbf{X}Cov_\theta[\theta]\mathbf{X}^T + \sigma^2\mathbf{I} = \mathbf{X}S_0\mathbf{X}^T + \sigma^2\mathbf{I} \end{aligned} \tag{49}$$

The parameter posterior is

$$p(\theta|\mathcal{Y}, \mathcal{X}) = \mathcal{N}(\theta|m_N, S_N) \tag{50}$$

where $S_N = (S_0^{-1} + \sigma^{-2}\Phi^T\Phi)^{-1}$ and $m_N = S_n(S_0^{-1}m_0 + \sigma^{-2}\Phi^T\mathbf{y})$.

Proof. Note: How to derive the above? □

The posterior predictive distribution is

$$\begin{aligned} p(y_*|\mathcal{X}, \mathcal{Y}, x_*) &= \int p(y_*|x_*, \theta) p(\theta|\mathcal{X}, \mathcal{Y}) d\theta \\ &= \int \mathcal{N}(y_*|\phi(x_*)^T\theta, \sigma^2)\mathcal{N}(\theta|m_N, S_N)d\theta \\ &= \mathcal{N}(y_*|\phi(x_*)^T m_N, \phi(x_*)^T S_N \phi(x_*) + \sigma^2) \end{aligned} \tag{51}$$

Note that the predictive mean $\phi(x_*)^T m_N$ coincides with the predictions made with the MAP estimate θ_{MAP}.

The Lasso regression adds a regularization term to the cost function, but uses the $l1$ norm of the parameters instead of the $l2$ norm. The objective function is

$$\hat{\theta} = \arg\min_{\theta} ||\mathbf{y} - \mathbf{X}\theta||_2^2 + \lambda||\theta||_1 \tag{52}$$

Lasso regression tends to completely eliminate the parameters of the least important features. Thus, it automatically performs feature selection and outputs an sparse model.

Note: Why?

The lasso cost function is not differentiable at $\theta_i = 0$, but gradient descent still works if you use a subgradient vector g instead when any $\theta_i = 0$.

$$g(\theta, J) = \bigtriangledown_\theta ||\mathbf{y} - \mathbf{X}\theta||_2^2 + \lambda[sign(\theta_1), ..., sign(\theta_n)] \tag{53}$$

where $sign(\theta_i) = -1 if \theta_i < 0, 0 if \theta_i = 0, +1 if \theta_i > 0$.

Elastic Net is a combination of ridge and lasso regression. The objective function is

$$J(\theta) = ||\mathbf{y} - \mathbf{X}\theta||_2^2 + r\lambda||\theta||_1 + \frac{1-r}{2}\lambda||\theta||_2^2 \tag{54}$$

In general, Elastic Net is preferred over lasso since lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

Note: Why would you want to user

- Ridge regression over linear regression.
- Lasso instead ridge regression.
- Elastic Net instead of Lass0

KG:

- A model with some level regularization typically performs better than a model without any regularization, so you should generally prefer ridge regression over plain linear regression.
- Lasso regression tends to generate sparse model. This is a way to perform features selection automatically, which is good if you suspect that only a few features actually matter.
- In general, Elastic Net is preferred over lasso since lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

6 SVM

Support Vector Machine is a very powerful and versatile Machine learning model, capable of performing linear or nonlinear classification, regression and even outerlier detection.

SVM: the idea of SVM is finding a hyper-plane such that two classes are separated but also stays as far as away from the closest as possible. This is called large margin classification.

Note that adding more training instances off the street will not affect the decision boundary at all, it is fully determined by the instances located on the edge of the street. These instances are called the support vector. SVM are sensitive to the feature scales.

Linear SVM classifier prediction

$$\hat{y} = 0 \; if \; w^T x + b < 0 \tag{55}$$

$$\hat{y} = 1 \; if \; w^T x + b \geq 0 \tag{56}$$

The goal is to find the value of w and b that make the margin as wide as possible while avoiding margin violation.

Define the target value $t_i = -1$ if the instance is negative instance and $t_i = 1$ for positive instance. The objective function is

$$\min_{W,b} \frac{1}{2} w^T w, \; s.t. \; t_i(w^T x_i + b) \geq 1, \; for \; i \in [m] \tag{57}$$

Soft SVM
Hard margin classification strictly imposes that all instances be off the street and on the correct side. There are two main issues: 1) it only works if the data is linearly separable. 2), it is sensitive to outliers.

Soft margin classification solves this issue by finding balance between keeping the margin as large as possible and limiting the margin violations.

The objective function is

$$\min_{W,b\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{m} \zeta_i, \; s.t., \; t_i(w^T x_i + b) > 1 - \zeta_i \; \; and \; \zeta_i \geq 0 \; for \; i \in [m] \tag{58}$$

where ζ_i measures how much the i-th instance is allowed to violate the margin.

Dual SVM
Dual SVN makes it possible for kernel trick.

Kernel SVM
When the data is not linearly separable, high order features can be added. However, a low polynomail degree it can not deal with very complex datasets, and with high polynomial degree it creates a huge number of features, making the model training slow and overfitting.

kernel SVM makes it possible to get the same results as if you added many polynomial features, even with very high degree polynomials, without actually having to add them. So there is no combinatorial explosion of the number of features since you do not actually add any features.

SVM regression: Instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM regression tries to fit as many instances as possible on the street while limiting margin violations. The width of the street is controlled by a hyper-parameter ϵ. Adding more instances within the margin does not affect the model's predictions, thus the model is said to be ϵ-insensitive.

To tackle nonlinear regression tasks, kernel SVM model can be used.

Online SVMs: online learning means learning incrementally. This can be achieved by applying gradient descent to minimize the cost function.

$$\min J(w, b) = \min \frac{1}{2} w^T w + C \sum_{i=1}^{m} \max(0, 1 - t_i(w^T x_i + b)) \tag{59}$$

The first term push the model to have a small weight vector w, leading to a larger margin. The second term computes the total of all margin violations. An instance's margin violation is equal to 0 is it is located off the street and on the correct side, or else it is proportional to the distance to the correct side of the street. Minimizing this term ensures that the model makes the margin violations as small and as few as possible.

The hinge loss function

$$\max(0, 1-t) \tag{60}$$

It is not differentiable at $t = 1$, but just like Lasso regression, you can still use gradient descent using any subderivative at $t = 0$.

Note: Why is it important to scale the inputs when using SVMs?

KG: SVMs try to fit the largest possible "street" between the classes, so if the training set is not scaled, the SVM will tend to neglect small features.

7 K-means

Clustering refers to a very broad set of techniques for finding subgroups or clusters in a data set. When we cluster the observations of a data set, we seek to partition them into distinct group so that the observations within each group are quite similar to each other, while observations in different groups are quit different from each other. We must define what it means to be similar or different. Indeed, this is often a domain-specific consideration that must be made based on knowledge of the data being studied.

K-mean clustering seeks to partition the observations into a distinct, non-overlapping pre-specified number of clusters.

The idea behind K-means clustering is that as good clustering is one for which the within-cluster variation is as small as possible. The within-cluster variation is cluster C_k is a measure $W(C_k)$ of the amount by which the observations within a cluster differ from each other. Hence we want to solve the problem

$$\min_{C_1,\ldots,C_k} \sum_{k=1}^{K} W(C_k) \tag{61}$$

In words, this formula says that we want to partition to the observations into K clusters such that the total within-cluster variation, summed over all K clusters, is as small as possible.

A most common choice involves squared euclidean distance. That is, we define

$$W(C_k) = \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2 \tag{62}$$

where $|C_k|$ denotes the number of observations in the $k-th$ cluster. In other words, the within-cluster variation for the $k-th$ cluster is the sum of all of pairwise squared euclidean distance between the observations in the $k-th$ cluster, divided by the total number of observations in the $k-th$ cluster.

The objective function of K-means clustering is

$$\min_{C_1,\ldots,C_K} \sum_{k=1}^{K} \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2 \tag{63}$$

This is in fact a very difficult problem to solve precisely, since there are almost K^n ways to partition n observations into K clusters. We would like to find an algorithm to solve that is a method to partition the observations into K clusters such that the objective is minimized.

The K-means algorithm finds a local rather than a global optimum, the results obtained will depend on the initial random cluster assignment of each observation. For this reason, it is importance to run the algorithm multiple times from different random initial configurations. Then one selects the best solution, the objective is smallest.

KG: K-means from the perspective of EM.

In hierarchical clustering, we do not know in advance how many clusters we want. In fact, we end up with a tree-like visual representation of the observations called a dendrogram that allows us to view at once the clustering obtained for each possible number of clusters from 1 to n.

One potential disadvantage of K-means clustering is that it requires us to pre-specify the number of clusters K. Hierarchical clustering is an alternative approach which does not require that we commit to a particular choice of K. Hierarchical clustering has an added advantage over K-means clustering in that it results in an attractive tree-based representation of the observations, called a dendrogram.

Algorithm 1: K-means clustering

- Randomly assign a number, from 1 to K, to each of the observations. These serves as initial cluster assignments for the observations.
- Iterate until the cluster assignments stop changing:
  - For each of the K clusters, compute the cluster centroid. The $k-th$ cluster centroid is the vector of the p feature means for the observations in the k-th cluster.
  - Assign each observation to the cluster whose centroid is closest where closest is defined using euclidean distance.

8 DBSCAN

Clustering algorithms looking for similarities and dissimilarties among data points.

Clustering algorithms can be divided in three sub-categories

- Partition-based clustering: k-means, k-median
- Hierarchical clustering: Agglomerative
- Density-based clustering: DBSCAN, GMM

DBSCAN stands for density-based spatial clustering of applications with noise. It is able to find arbitrary shaped clusters and clusters with noise.

The main idea is that a point belongs to a cluster if it is close to many points from that cluster.

There are two key parameters:

- ϵ: the distance that specifics the neighborhood. Two points are considered as neighbor if the distance is less than ϵ.
- n: Minimum points of data points to define a cluster.

Based on the above parameters, points can be categorized as three classes:

- core point: a point is a core point if there are at least n pints around it within ϵ distance.
- Border point: A point is border point if it can be reachable from a core point and no more than n points around it within ϵ distance (not a core point).
- Outlier: if it is not a core point or a border point.

How does it work?

- Staring from a random point at its neighborhood defined by ϵ. If the point is a core point, this point and its neighbors form a cluster. For other points in the cluster, if a point is also a core point, it neighbors are added into the cluster. If the starting point is not core point, marked as noise. This may be revisited and be part of a cluster.
- Randomly choose another point among the points that are not visiting in the previous step. Repeat the procedure.
- This process is finished when all points are visited.

DBSCAN is able to find high density regions and seperate them from low density regions.

Pros and cons

- Does not requires pre-define number of cluster
- performs well with arbitrary shapes clusters
- Robust to outlier and able to detect outlier.

- The distance metric is not easy to choose.
- If clusters are very different in terms of in-cluster densities, DBSCAN is not well suited to define clusters.

9 Gaussian Mixture Model

GMM is an algorithm for density estimation which gives the generative model for the data, from which we can generate more data by sampling the model.

Mixture models can be used to describe a distribution $p(x)$ by a convex combination of K simple distributions.

$$p(x) = \sum_{k=1}^{K} \pi_k p_k(x) \; s.t., \; 0 \leq \pi_k \leq 1, \; \sum_{k=1}^{K} \pi_k = 1 \tag{64}$$

where the components p_k are members of a family of basic distribution and π_k are mixture weights.

Mixture models are more expressive than the corresponding base distributions because they allow for multimodal data representations. They can describe datasets with multiple clusters.

In GMM, the basic distributions are Gaussian. For a given dataset, we sim to maximize the likelihood of the model parameters to train the GMM. But, we will not find a closed-form maximum likelihood solution. Instead, we can solve iteratively via EM.

A GMM is a sensity model where we combine a finite number of K Gaussian distribution $\mathcal{N}(x|\mu_k, \Sigma_k)$

$$p(x|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mu_k, \Sigma_k) \; s.t., \; 0 \leq \pi_k \leq 1, \; \sum_{k=1}^{K} \pi_k = 1 \tag{65}$$

The parameters $\theta = \{\pi_k, \mu_k, \Sigma_k : k \in [K]\}$ is defined as the collection of all parameters of the model.

Assume we are given a dataset $\mathcal{X} = \{x_1, ..., x_N\}$ are drawn i.i.d from an unknown distribution $p(x, \theta)$. The goal is to find the parameter θ by maximum likelihood.

The likelihood is

$$p(\mathcal{X}|\theta) = \Pi_{i=1}^{N} p(x_n|\theta) \tag{66}$$

and

$$p(x_n|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k) \tag{67}$$

The log-likelihood is

$$\log p(\mathcal{X}|\theta) = \sum_{n=1}^{N} \log p(x_n|\theta) = \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k) \tag{68}$$

Then the objective function is

$$\theta_{ML} = \arg\max_{\theta} \mathcal{L} = \arg\max_{\theta} \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k) \tag{69}$$

The gradient is

$$\frac{\partial L}{\partial \theta} = \sum_{n=1}^{N} \frac{\partial \log p(x_n|theta)}{\partial \theta} = \sum_{n=1}^{N} \frac{1}{p(x_n|\theta)} \frac{\partial \log p(x_n|theta)}{\partial \theta} \tag{70}$$

where

$$\frac{1}{p(x_n|\theta} = \frac{1}{\sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)} \tag{71}$$

We define the quantity: responsibility

$$r_{nk} = \frac{\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_n|\mu_j, \Sigma_k)} \tag{72}$$

Note that $r_n = \{r_n 1, ..., r_n k\}$ is a probability vector where $\sum_{j=1}^{K} r_n j = 1$ and $r_n j > 0$. We can think r_n as a soft assignment of x_n for K mixture components. Therefore, the responsibility r_{nk} from represents the probability that x_n has been generated by the k-th mixture component.

The responsibility r_{nk} is proportional to the likelihood

$$p(x_n|\pi_k, \mu_k, \Sigma_k) = \pi_k \mathcal{N}(\mu_k, \Sigma_k) \tag{73}$$

We will update one model parameter at one time.

$$\mu_k \leftarrow \frac{\sum_{n=1}^{N} r_{nk} x_n}{\sum_{n=1}^{N} r_{nk}} \tag{74}$$

Proof. ..... □

$$\Sigma_k \leftarrow \frac{1}{N_k} \sum_{n=1}^{N} r_{nk}(x_n - \mu_k)(x_n - \mu_k)^T \tag{75}$$

where $N_k = \sum_{n=1}^{N} r_{nk}$.

$$\pi_k \leftarrow \frac{N_k}{N} \tag{76}$$

Note: What are the algorithms for clustering?

10 EM

The expectation maximization algorithm is a general iterative scheme for learning parameters (maximum likelihood or MAP) in mixture model and more generally latent-variable models.

In case of GMM, we choose initial values for π_k, μ_k and Σ_k and alternate until convergence between

- E: evaluate the responsibilities r_{nk}.
- M: Use r_{nk} to estimate the parameters π_k, μ_k and Σ_k.

11 KNN

The idea behind KNN is simple:

- Choose the number of k and a distance metric.
- Fins the k nearest neighbors of the sample that we want to classify.
- Assign the class label by majority vote.

12 Naive Bayes

NB is a popular classification method, in which we form a joint model of a D-dimensional attribute (input) vector x and the corresponding class label c.

$$p(x, c) = p(x)p(x|c) = p(x)\Pi_i^N p(x_i|c) \tag{77}$$

For a new input x_*

$$p(c|x_*) = \frac{p(x_*|c)p(c)}{p(x_*)} = \frac{p(x_*|c)p(c)}{\sum_c p(x_*|c)p(c)} \tag{78}$$

13 Dimensionality Reduction

In this direction, we discuss the curse of dimensionality and present the two main approached to dimensionality reduction: Projection ad Manifold Learning.

The fact that high dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other. This means new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions, since they will based on much larger extrapolations. In short,the more dimensions the training set has, the greater the risk of overfitting. This can be solved by more data-sets, but in practice, the number of training instances required to research a given density grows exponentially with the number of dimensions.

Projection: In real-applications,all training instances actually lie within a much lower dimensional subspace of the high-dimensional space.

Manifold learning: Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie. It relies on the manifold assumption that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold.

If you reduce the dimensionality of training set before training a model, it will definitely speed up training, but it may not always lead to a better or simpler solution. It all depends on the dataset.

PAC: Principal component analysis: it first identifies the hyperplane that lies closest to the data and then it projects the data onto it.

It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely loss less information than the other projections. Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original data-set and its projection onto that axis. This is the simple idea behind PCA.

How does PCA work?
PCA identifies the axis that accounts for the largest amount of variance in the training set. It also finds the second axis, orthogonal to the first one, that accounts for the largest amount for the remaining variance, and so on as many axes as the number of dimensions in the dataset.

The i-th principal component is the unit vector that defines the i-th axis. The direction of the principal components is not stable: if the training set is perturbed slightly, the direction may point to the opposite direction, however, they will generally still lie on the same axes.

How to find PCs?
The singular value decomposition (SVD) that can decompose the training set matrix X into dot product of three matrix $U\Sigma V^T$ where V^T contains all the principal components that we are looking for.

Note: PCA assumes that the dataset is centered around the origin. Therefore, the dataset is needed to be centered before apply PCA.

The next step is to project the X into the subspace defined by d principal components W_d (the first d columns of V^T).

$$\tilde{X} = XW_d \tag{79}$$

Another very useful piece of information is the explained variance ration of each principal component. It indicates the proportion of the dataset's variance that lies along the axis of each principal component.

Note: What is SVD?

The right number of dimensions: It is generally preferable to choose the number of dimension that add up to a sufficiently large portion of the variance (95%). Unless, you are reducing dimensionality for data visualization.

KG: The original dataset can be reconstructed by lossing information.

$$\bar{X} = \tilde{X}W_d^T \tag{80}$$

One problem with the preceding PCA is that it requires the whole training set to fit in memory in order for SVD algorithm to run. Incremental PCA algorithms have been developed: you can split

the training set into mini-batches and deeds an incremental PCA algorithm one mini-batch at a time. This is useful for large training sets, and also to apply PCA online.

Randomize PCA can quickly find an approximation of the first d principal components.

Note: PCA applies linear projections The kernel trick can be applied to PCA making it possible to perform complex nonlinear projections for dimensionality reduction. This is called Kernel PCA. It is often good at preserving clusters of instances after projections.

KG: How to select the kernel? depends on the following machine learning task or the reconstruction error. However reconsturction is not easy in kernel PCA, as teh kernel feature space is infinite-dimensional, we cannot compute the reconstructed points and therefore we cannot compute the true reconstructed error. It is possible to find a point in the original space that would map close to the reconstructed point. This is called reconstruction pre-image. Once you have this pre-image, you can measure its squared distance to the original instance.

LLE Locally Linear Embedding is an very powerful nonlinear dimensionality reduction technique. It first measure how each training instance linearly relates to its cloest neighbors and then looking for a low-dimensional representation of the training set where these local relationships are best preserved.

How does it work?
First, for each instance x, the algorithm identifies its k closest neighbors, then tries to reconstruct x as linear function of these neighbors. It finds the weight w_{ij} such that the squared distance between x_i and $\sum_{j=1}^{k} w_{ij}x_j$ is as small as possible. The objective function of the first step is

$$\hat{W} = \arg\min_{W} \sum_{i=1}^{m} ||x_i - \sum_{j=1}^{k} w_{ij}x_j||_2^2, \quad s.t. \quad \sum_{j=1}^{k} w_{ij} = 1, \ \forall i \in [m] \tag{81}$$

After this step, the weight matrix $\hat{W}$ encodes the local linear relationships between the training instances.

The second step is to map the training instances into d-dimensional space while preserves these local relationships as much as possible. If z_i is the image of x_i, the objective function is

$$\hat{Z} = \arg\min_{Z} \sum_{i=1}^{m} ||x_i - \sum_{j=1}^{k} \hat{w}_{ij} z_j||_2^2 \tag{82}$$

Note: PCA is a linear algorithm. It will not be able to interpret complex polynomial relationship between features. On the other hand, t-SNE is based on probability distributions with random walk on neighborhood graphs to find the structure within the data.

t-Distributed Stochastic Neighbor Embedding (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart.

- Step 1: SNE (stochastic neighbor embedding) starts to converting the high-dimensional euclidean distance between instances into conditional probability.

$$p(j|i) = \frac{\exp(-||x_i - x_j||_2^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||_2^2/2\sigma_i^2)} \tag{83}$$

- For the low-dimensional counterparts y_i, it is possible to compute the similar conditional probability

$$q(j|i) = \frac{\exp(-||y_i - y_j||_2^2)}{\sum_{k \neq i} \exp(-||y_i - y_k||_2^2)} \tag{84}$$

- SNE attempts to minimize the difference of these two conditional probability.

t-SNE is an extension of SNE: To measure the minimization of sum of difference of conditional probability SNE minimizes the sum of KL-divergence overall data points using a gradient descent method. It is very difficult (computationally inefficient) to optimize this cost function.

So t-SNE also tries to minimize the sum of the difference in conditional probabilities. But it does that by using the symmetric version of the SNE cost function, with simple gradients. Also, t-SNE employs

a heavy-tailed distribution in the low-dimensional space to alleviate both the crowding problem (the area of the two-dimensional map that is available to accommodate moderately distant data points will not be nearly large enough compared with the area available to accommodate nearby data points) and the optimization problems of SNE. t-SNE performs a binary search for the value of σ_i.

Linear Discriminant Analysis (LDA) is actually a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyper-plane onto which to project the data.

14 Decision Trees

Decision trees are versatile Machine Learning algorithms that can perform both classification and regression tasks and even multioutput tasks. They are very powerful algorithms, capable of fitting complex datasets.

Note: Decision trees requires very little data preparation. In particular, they do not require feature scaling or centering at all.

The Gini impurity of node i

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2 \tag{85}$$

where $p_{i,k}$ is the ration of class k instances among the training instances in the i-th node.

Alternatively, the impurity can be measured by entropy

$$H_i = - \sum_{k=1, p_{i,k} \neq 0}^{n} p_{ik} log(p_{ik}) \tag{86}$$

Most of the time Gini and entropy lead to similar trees. Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.

A decision tree can estimate the probability that an instance belongs to a particular class k: first it traverses the tree to find the leaf node for this instance, and then it returns the ration of training instances of class k in this node.

Regularization Hyper-parameters
Decision tree is an non-parameteric model. To avoid overfitting the training data, we can control: the maximum depth of the tree, the minimum number of samples a node must have before it can be split, the minimum number of a samples a leaf node must have, the maximum number of leaf nodes, the maximum number of features that are evaluated for splitting at each node.

CART algorithm:
The idea is simple: the algorithm first splits the training set into two subsets using a single feature k and a threshold t_k. It search for (k, t_k) that produces the purest subsets. The cost function is

$$J(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right} \tag{87}$$

where $m_{left/right}$ is the number of instances in the left/right subset. $G_{left/right}$ measures the impurity of the left/right subset.

Once it has split the training set into two, it splits the subsets using the same logic, then the sub-subsets and so on, recursively.

It stops recursing once it reaches the maximum depth or if it cannot find a split that will reduce impurity.

Note: CART algorithm is a greedy algorithm: it greedily searches for an optimum split at the top level, then repeats the process at each level. It does not check whether or not the split will lead to the lowest possible impurity several levels down.

Note: CART algorithm produces only binary trees: non-leaf nodes always have two children. ID3 algorithm can produce Decision trees with nodes that have more than two children.

Decision tree regression:
The main difference is that instead of predicting a class in each node, it predicts a value. For an instance x, you traverse the tree starting at the root, and you eventually reach the leaf node that predicts the value which is the average target value of the training instances associated to this leaf node.

The cost function of CART algorithms is

$$J(k, t_k) = \frac{m_{left}}{m} MSE_{left} + \frac{m_{right}}{m} MSE_{right} \tag{88}$$

where $MSE = \sum_{i \in node} (\hat{y}_{node} - y_i)$ and $\hat{y}_{node} = \frac{1}{m_{node}} \sum_{i \in node} y_i$

Decision trees are sensitive to small variation of dataset. This issue is solved by random forest by averaging predictions over many trees.

15 Random Forests

Random forest is a substantial modification of bagging that builds a large collection of de-correlated trees, and then averages them.

Bagging (bootstrap aggregation) is a technique for reducing the variance of an estimated prediction function. For regression, we simple fit the same regression tree many times to bootstrap sampled versions of the training data, and average the result. For classification, a committee of trees each cast a vote for the predict class.

The essential idea in bagging is to average many noisy but approximately unbiased models, and hen reduce the variance. Trees are idea candidates for bagging, since they can capture complex interaction structure in the data, and if grown sufficiently deep, have relatively low bias. Since trees are notoriously noisy, they benefit greatly from the averaging. Moreover, since each tree generated in bagging is identically distributed, the expectation of an average of B such trees is the same as the expectation of any one of them. This means the bias of bagged trees is the same as that of the individual trees and only hope of improvement is through variance reduction. This is contrast to boosting, where the trees are grown in an adaptive way to remove bias, and hence are not i.d.

The idea of random forests is to improve the variance reduction of bagging by reducing the correlation between the trees, without increasing the variance too much. This is achieved in the tree-growing process through random selection of the input variables.

Specifically, when growing a tree on a bootstrapped data-set. Before each split, select $m \leq p$ of the input variables at random as candidate of splitting. Intuitively, reducing m will reduce the correlation between any pair of trees in the ensemble and hence reduce the variance of the average.

An important feature of the random forests is its use of out-of-bag (OOB) samples: for each observation $z_i = (x_i, y_i)$ construct its random forest predictor by averaging only those trees corresponding to bootstrap samples in which z_i did not appear.

Variable importance of random forest: At each split in each tree, the improvement in the split-criterion is the importance measures attributed to the splitting variable, and is accumulated over all the trees in the forest separately for each variable. In other word, feature importance is measured by looking at how much the tree nodes that use that feature reduce impurity on average. More precisely, it is a weighted average, where each node's weight is equal to the number of training sampled that are associated with it.

16 Ensemble Methods

The goal behind ensemble methods is to combine different classifiers into a meta-classifier that has a better generalization performance than each individual classifier alone.

The most popular ensemble methods that use the majority voting principle which mean that we select the class label that has been predicted by the majority of classifiers.

Depending on the techniques, the ensemble can be built from different classification algorithms, for example, decision trees support vector machines, logistic regression classifiers, and so on.

Alternatively, we can also use the same base classification algorithm fitting different subsets of the training set. One prominent example of this approach would be the random forest algorithm, which combines different decision tree classifiers.

Bagging building an ensemble of classifiers from bootstrap samples.

Bagging is an ensemble learning technique that is closely related to the Majority-Vote-Classifier that we implemented in the previous section. However, instead of using the same training set to fit the individual classifiers in the ensemble, we draw bootstrap samples (random samples with replacement) from the initial training set, which is why bagging is also known as bootstrap aggregating.

In fact, random forests are a special case of bagging where we also use random features subsets to fit the individual decision trees.

When sampling is performed without replacement, it is called pasting. when sampling is performed with replacement, this is called bagging.

Bagging reduces the variance without increasing the bias.

Random forest introduces extra randomness by sampling subset features. It trades higher bias for even lower variance.

Out-of-bag evaluation: with bagging, some instances many be sampled several times for any given predictor while others may not be sampled at all. Since a predictor never sees that oob instances during training, it can be evaluated on these instances, without the need for a separate validation set. You can evaluate the ensemble itself by averaging out the oob evaluation of each predictor.

Boosting
The generally idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many boosting methods available, but the most popular are adaboost and gradient boosting.

AdaBoost
The key concept behind boosting is to focus on training samples that are hard to classify, that is, to let the weak learners subsequently learn from mis-classified training samples to improve the performance of the ensemble. In contrast to bagging, the initial formulation of boosting, the algorithm uses random subset of training samples drawn from the training data-set without replacement. The original boosting procedure is summarized in four steps:

- Draw a random subset of training samples d1 without replacement from the training set D to train a weak learner C1.
- Draw second random training subset d2 without replacement from the training set and add 50 percent of the samples that were previously mis-classified to train a weak learner C2.
- Find the training samples d3 in the training set D on which C1 and C2 disagree to train a third weak learner C3.
- Combine the weak learners C1 , C2 , and C3 via majority voting.

In contrast to the original boosting procedure as described here, Ada-Boost uses the complete training set to train the weak learners where the training samples are re-weighted in each iteration to build a strong classifier that learns from the mistakes of the previous weak learners in the ensemble.

Once all predictors are trained, the ensemble makes predictions very much like bagging, except that predictors have different weights depending on their overall accuracy on the weighted training set.

To make predictions, adaboost simple computes the predictions of all the predictors and weights them using the predictor weights α_i. The predicted class is the one that receives the majority of weighted votes.

Gradient Boosting
Another very popular Boosting algorithm is Gradient Boosting, just like Ada-boosting, which works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking weights at every iteration, this method tries to fit the new predictor to the residual errors made by the previous predictors.

Stacking The last ensemble method is called stacking. It is based on the simple idea: instead of using trivial functions to aggregate the predictions of all predictors in an ensemble, why do not we train a model to perform this aggregation. A set of predictors are trained by bagging/boosting, then a final predictor takes the prediction of preceding predictors and makes the final prediction.

17 XgBoost, LightGBM

LightGBM: is a fast, distributed, high-performance gradient boosting framework based on decision tree algorithm, used for ranking, classification and many other machine learning task.

Since it is based on decision tree algorithm, it splits the tree leaf wise with the best fit whereas other boosting algorithms split the tree depth wise or level wise rather than leaf-wise. So when growing on the same leaf in LightGBM, the leaf-wise algorithm can reduce more loss than the level-wise algorithm and hence results in much better accuracy which can rarely be achieved by any of the existing boosting algorithm,

Advantage of LightGBM:

- Faster training and higher efficiency: LightGBM uses histogram based algorithms where continuous features are transformed into discrete bins which fasten the training procedure.
- Lower memory usage
- Better accuracy: Due to the lead-wise split, it produces much more complex trees. That why better accuracy, but is leads to overfitting. This can be avoid by setting the max-depth parameter.
- Compatibilit with large dataset.

The difference between XGBoost and LightGBM LightGBM uses a novel sample techniques: Gradient-based one-side sampling. It retains instances with large gradients while performing random sampling on instances with small gradient. As we know, instances with small gradients are well trained and those with large gradients are undertrained.

XGBoost uses pre-sorted algorithm & Histogram-based algorithm for computing the best split.

How does it work? A new model is trained based on the sampled instances and its associated residual error. At test time, the prediction sis weighted sum of all predictors.

Note: How to determine the weights?

Categorical features:
XGBoost does not support categorical variables naively. One-hot encoding is commonly used as a standard pre-processing techniques.

LightGBM provides direct support of categories as long as they are integer encoded prior to the training.

CatBoost proposed target encoding for categorical features.

How to regularize models?

How to tune hyper-parameters?

18 Optimization Techniques

18.1 Optimization for Neural Network

Optimization algorithms are the basic engine behind deep learning methods that enable models to learn from data by adapting there parameters. They solve the problem of minimization of an objective function that measure the mistakes made by the model.

The standard neural network training objective is given by

$$h(\theta) = \frac{1}{m} \sum_{i=1}^{m} l(y_i, f(\theta, x_i)) \tag{89}$$

Gradient Descent
Basic gradient descent iteration

$$\theta_{k+1} = \theta_k - \alpha \bigtriangledown h(\theta_k) \tag{90}$$

Intuition: gradient descent is minimizing a local approximation

1st-order Taylor series for $h(\theta)$ around current θ is

$$h(\theta + d) \approx h(\theta) + \bigtriangledown h(\theta)^T d \tag{91}$$

Gradient update computed by minimizing this within a sphere of radius r

$$-\alpha \bigtriangledown h(\theta) = \arg \min_{d:||d|| \leq r} (h(\theta) + \bigtriangledown h(\theta)^T d) \tag{92}$$

where

$$r = \alpha || \bigtriangledown h(\theta)|| \tag{93}$$

Momentum methods
Motivation: The gradient has a tendencu to flip back and forth as we take steps when the learning rate is large.

Key ideas:
Accelerate movement along directions that point consistently down-hill across many consecutive iterations.

How to do this
Treat current solution of θ as a ball rolling along a surface whose height is given by $h(\theta)$, subject to the force of gravity.

Class Momentum:

$$v_{k+1} = \eta_k v_k - \bigtriangledown h(\theta_k), \;\; v_0 = 0 \tag{94}$$

$$\theta_{k+1} = \theta_k + \alpha_k v_{k+1} \tag{95}$$

where α_k is the learning rate and η_k is the momentum constant.

Nesterov's variant:

$$v_{k+1} = \eta_k v_k - \bigtriangledown h(\theta_k + \alpha_k \eta_k v_k), \;\; v_0 = 0 \tag{96}$$

2nd-order methods
The problem of 1st-order methods: the number of steps needed to converge grows with condition number $\kappa = L/\mu$. THis will be very large for some problems (certain deep architectures).

2nd-order methods can improve (or even eliminate) this dependency.

Derivation of Newton's method
Approximate $h(\theta)$ by its 2nd-order Taylor series around current θ.

$$h(\theta + d) \approx h(\theta) + \bigtriangledown h(\theta)^T d + \frac{1}{2} d^T H(\theta) d \tag{97}$$

Minimize this local approximation to obtain

$$d = -H(\theta)^{-1} \bigtriangledown h(\theta) \tag{98}$$

Update Current iterate with this

$$\theta_{k+1} = \theta_k - H(\theta)^{-1} \bigtriangledown h(\theta_k) \tag{99}$$

Gradient descent implicitly minimizes a bad approximation of 2nd-order Taylor series:

$$\begin{aligned} h(\theta + d) &\approx h(\theta) + \bigtriangledown h(\theta)^T d + \frac{1}{2} d^T H(\theta) d \\ &\approx h(\theta) + \bigtriangledown h(\theta)^T d + \frac{1}{2} d^T (LI) d \end{aligned} \tag{100}$$

LI is too pessimistic /conservative an approximation of $H(\theta)$. Treats all directions as having max curvature.

The breakdown of local quadratic approximation and how to deal with it.

- Quadratic approximation of objective is only trustworthy in a local region around current θ.
- Gradient descent (implicitly) approximates the curvature everywhere by its global max.
- Newton's method uses $H(\theta)$, which may become an underestimate in the region we are taking our update step.

Solution: Constrain update d to lie in a trust region R around where approximation remains good enough.

Trust-regions and damping
If we take $R = d : ||d||_2 \leq r$ then computing

$$\arg\min_{d \in R} h(\theta) + \bigtriangledown h(\theta)^T d + \frac{1}{2} d^T H(\theta) d \tag{101}$$

is often equivalent yo

$$-(H(\theta) + \lambda I)^{-1} \bigtriangledown h(\theta) = \arg\min_{d} h(\theta) + \bigtriangledown h(\theta)^T d + \frac{1}{2} d^T (H(\theta) + \lambda I) d \tag{102}$$

λ depends on r in a complicated way, but we can just work with λ directly.

Alternative curevature matrices $H(\theta)$ does not necessarily give the best quadratic approximation for optimisation. Different replacements for $H(\theta)$ could produce: A more global approximation, a more conservative approximation.

- Generalized Gauss-Newton matrix (GGN)
- Fisher information matrix
- Empirical Fisher

The barrier to application of 2nd-order methods for neural network: For network with large number of parameters, computing $H(\theta)$ is infeasible.
We must simplify the curvature matrix's computation, storage and inversion.

This is typically done by approximating the matrix with a simpler form.

Diagonal approximation: including only the diagonal entries of curvature matrix. This is used in `RMS-prop` and `Adam` methods to approximate empirical fisher matrix.

Block-diagonal approximation: take only include certain diagonal blocks

- Weights on connections going into a given unit.
- weights on connections going out of a given unit.
- all the weights for a given layer.

Well-known algorithm: `TONGA`

Kronecker-product approximation:
Block-diagonal approximation of GGN/Fisher where blocks correspond to network layers. Approximate each block as Kronecker product of two small matrices. This is used in `K-FAC`.

Stochastic methods
Motivation: typical objective in machine learning are an average over training cases of case-specific losses

$$h(\theta) = \frac{1}{m} \sum_{i=1}^{m} h_i(\theta) \tag{103}$$

m can be very big and so computing the gradient gets expensive

$$\bigtriangledown h(\theta) = \frac{1}{m} \sum_{i=1}^{m} \bigtriangledown h_i(\theta) \tag{104}$$

Idea: randomly sub-sample a mini-batch of training case S with size $b << m$ and estimate gradient as

$$\tilde{\bigtriangledown} h(\theta) = \frac{1}{b} \sum_{i \in S} \bigtriangledown h_i(\theta) \tag{105}$$

Stochastic gradient descent (SGD) replaces $\bigtriangledown h(\theta)$ with $\tilde{\bigtriangledown} h(\theta)$, giving

$$\theta_{k+1} = \theta_k - \alpha_k \tilde{\bigtriangledown} h(\theta_k) \tag{106}$$

Mini-batch gradients estimates can be used with 2nd-order and momentums methods too. Curvature matrices estimated stochastically using decayed averaging over multiple steps.

18.2 Optimizers used in neural network

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training:

- Applying a good initialization strategy for weights.
- Using a good activation function.
- Using batch normalization.
- Reusing parts of pre-trained network.

Another speed boost comes from using a faster optimizer. In this section we present the most popular ones: Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp and Adam optimization.

Gradient Descent:
Gradient Descent simple updates the weights θ by directly substracting the gradient of the cost function $J(\theta)$ with regards to the weights multiplied by the learning rate η.

$$\theta \leftarrow \theta - \eta \bigtriangledown_\theta J(\theta) \tag{107}$$

It does not care the previous gradient. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares great deal about what previous gradients are. The algorithm introduces a new hyper-parameter β, simply called the momentum, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9. The update rule is

$$m \leftarrow \beta m + \eta \bigtriangledown_\theta J(\theta) \tag{108}$$

$$\theta \leftarrow \theta - m \tag{109}$$

Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it is good to have a bit of friction β in the system: it gets rid of these oscillations and thus speeds up convergence.

Nesterov Accelerated Gradient
The ideas of Nesterov Accelerated Gradient is to measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum. This small tweak works because in general the momentum vector m will be pointing in the right direction (toward to the optimum), so it will be slightly more accurate to use the gradient measured a bit father in that direction rather than is the gradient at the original position. The updated rule is

$$m \leftarrow \beta m + \eta \bigtriangledown_\theta J(\theta + \beta m) \tag{110}$$

$$\theta \leftarrow \theta - m \tag{111}$$

AdaGrad
Gradient descent starts by quickly going down the steepest slope, then slowly goes down the bottom of the valley. It would be nice if the algorithm could detect this early on and correct its direction to point a bit more toward the global optimum. The AdaGrad algorithm achieves this by scaling down the gradient vector along the steepest dimensions

$$s \leftarrow s \bigtriangledown_\theta J(\theta) \otimes \bigtriangledown_\theta J(\theta) \tag{112}$$

$$\theta \leftarrow \theta - \eta \bigtriangledown_\theta J(\theta) \oslash \sqrt{s + \epsilon} \tag{113}$$

where $\otimes$ is element-wise multiplication and $\oslash$ represents element-wise division and ϵ is a smoothing term to avoid division by zero.

The first step accumulated the square of the gradients into the vector s. Each s_i accumulates the squares of the partial derivative of the cost function with regards to parameter θ_i. If the cost function is steep along the i-th dimension, then s_i will get larger and larger at each iteration.

In the second step, the gradient vector is scaled down by a factor $\sqrt{s + \epsilon}$.

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an adaptive learning rate. It helps point the resulting updates more directly toward the global optimum. One additional benefit is that it requires much less tuning of the learning rate hyper-parameter η.

AdaGrad often performs well for simple quadratic problems, but unfortunately it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum.

RMSProp solves the problem of AdaGrad by accumulating only the gradients from the most recent iteration as opposed to all the gradients since the beginning of training. It does so by using exponential decay in the first step.

$$s \leftarrow \beta s + (1 - \beta) \bigtriangledown_\theta J(\theta) \otimes \bigtriangledown_\theta J(\theta) \tag{114}$$

$$\theta \leftarrow \theta - \eta \bigtriangledown_\theta J(\theta) \oslash \sqrt{s + \epsilon} \tag{115}$$

The decay rate β is typically set to 0.9.

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. It also generally performs better than Momentum optimization and Nesterov Accelerated Gradients. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

Adam Optimization
Adam, which stands for adaptive moment estimation, combines the idea of Momentum optimization and RMSProp: just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and just like RMSProp it keeps track of an exponentially decaying average of past squared gradients.

$$m \leftarrow \beta_1 m + (1 - \beta_1) \bigtriangledown_\theta J(\theta) \tag{116}$$

$$s \leftarrow \beta_2 s + (1 - \beta_2) \bigtriangledown_\theta J(\theta) \otimes \bigtriangledown_\theta J(\theta) \tag{117}$$

$$m \leftarrow \frac{m}{1 - \beta_1^t} \tag{118}$$

$$s \leftarrow \frac{s}{1 - \beta_2^t} \tag{119}$$

$$\theta \leftarrow \theta - \eta m \oslash \sqrt{s + \epsilon} \tag{120}$$

where t is the iteration number.

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both Momentum optimization and RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $1 - \beta_1$ times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since m and s are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost m and s at the beginning of training.

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. As earlier, the smoothing term ϵ is usually initialized to a tiny number such as 10^{-8}.

In fact, since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyper-parameter η. You can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

FTRL Optimization

19 Applications

19.1 Linear Regression

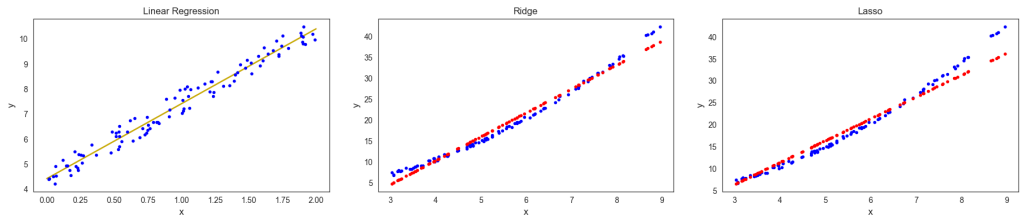

Figure 1

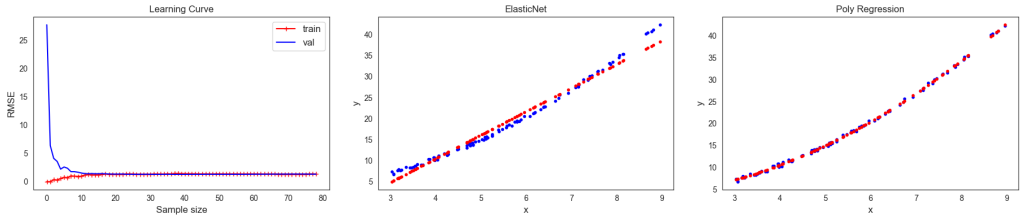

Figure 2

19.2 Logistic regression

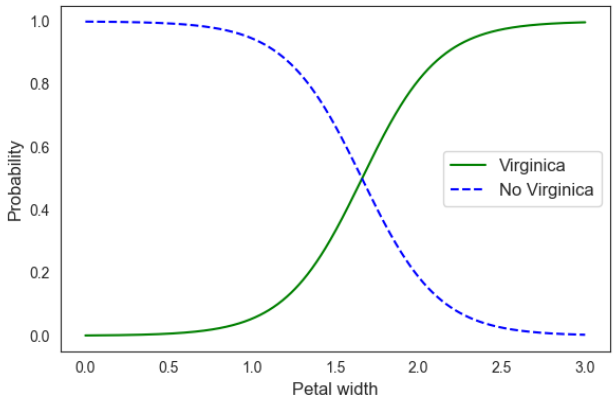

Figure 3

References