Machine learning Algorithms

蔡苗[1]

2019-04-21

[1]Department of Epidemiology and Biostatistics, College for Public Health and Social Justice, Saint Louis University. Email: miao.cai@slu.edu

感谢我的家人的支持。

Acknowledgement

I want to thank my mentor.

目录

表格

插图

Preface

This book works as a notebook to summarize the algorithms used in Bayesian inference and machine learning.

第一章　Introduction

第二章　Discrete optimization

Most of the concepts are explain in Chapter 4 Numerical Computation from the Deep Learning book https://www.deeplearningbook.org/contents/numerical.html. The other useful resource is CS231m Convolutional Neural Networks[1] for Visual Recognition by Stanford University

The **objective function** allows us to measure how "good" any given solution to the problem is. We seek to maximize or minimize the objective function.

Derivative/gradient based methods keep going "uphill" until they are at the top of the h

2.1 Heuristic and metaheuristic methods

> a **heuristic** is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution.
>
> Wikipedia

Heuristic methods do not guarantee to find the global optimal solution (best solution)! Instead, they seek to find **a best available solution, given the resource spent looking for it**. A **heuristic method** is **geared towards a specific problem**.

[1] http://cs231n.github.io/

a **metaheuristic** is a higher-level procedure or heuristic designed to find, generate, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity. Metaheuristics sample a set of solutions which is too large to be completely sampled. Metaheuristics may make few assumptions about the optimization problem being solved, and so they may be usable for a variety of problems

– Wikipedia

A metaheuristic method is like a heuristic, but generalizable to a broad class of problems.

1. Genetic Algorithms (Holland – 1975)

- Natural selection / genetics based. Popular method.

2. Simulated Annealing (Kirpatrick – 1983)

- Metallurgy annealing, find lowest energy level!

3. Particle Swarm Optimization (Eberhart Kennedy - 1995)

- Based on insect behavior, swarming towards optimal location (food). Less common in discrete spaces. originally proposed for continuous spaces.

4. Tabu Search (Al-Sultan – 1999)

- Search for best neighborhood solution, then find new neighborhood. Prior neighborhoods are forbidden (tabu)

General meta-heuristics traits

- Evaluate many potential optimal solutions.
- Evaluate the fitness of each solution based on a cost (objective) function.
- Use some concept of stochastic (random) movement to generate new solutions from the parameter space.
- Use some set of rules to determine where to move next in the parameter space.

- Declare convergence once some set of criteria has been met. Perhaps no improvement for X iterations.

2.2 Genetic algorithm and simulated Annealing as examples

Genetic algorithm: need to explore large portions of the parameter space at random. Concept of "neighbor" is vague.

A nice shiny app[2]

An GA example: Since a new treatment for Hep C has become available, where is the optimal place to locate limited new Hep C resources, considering where our patients live?

The problem become intractable with large number of locations and resources: How many combinations of patients and clinics can I calculate the full feature space for to find a maximum?

- Exact Solution is NP-Hard
- Calculations = $n^{\sqrt{k}}$
- I conveniently stopped my analysis at 6 sites with ~5k patients, requiring 1,149,712,053 distance calculations (I have a big server)
- The "k-center" problem

Simulated Annealing:

- The concept of a 'neighbor' is strong.
- Can be sensitive to parameter choice, or algorithm gets stuck in global minima!
- Generally, you should try both to see what works best. Hard to guess up front.

[2] https://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/

2.3 Constrains

Hard constraints

- If this constraint is violated, we have invalid solution.
- Labor laws, number of nurses available, etc

Soft Constraints

- These are nice to meet if possible (included in cost function somehow), but if they are not met the solution is still valid.
- Nurse prefers to only work X night shifts per month.
- Leave requests.

第三章　Continuous optimization

Points to learn in continuous optimization:

- Understand first and second derivatives and the role they play in optimizing continuous functions.
- Understand general steps in continuous optimization
- Contrast 1st order versus 2nd order derivative optimization methods
- Extend these thoughts to the distributed computing context

Things to consider for smart steps:

- Initialization value: where should I start?
- Direction of the gradient: what direction should I we step towards?
- Step size: how big of a step should we take?

3.1 First and second derivative methods

First derivative/gradient

- Instantaneous slope of a point (rate of change of the function)
- If we have multiple input variables (multiple x's), then we need to know the gradient in the direction of each of the x's (partial derivatives). The matrix of partial first derivatives is called **the Jacobian**.

Second derivative

- Tells me the 'curvature' of a function.

- Rate of change of the first derivative.
- If we have multiple input variables (multiple x's), then the matrix of partial second derivatives is called **the Hessian**.

A comparison of first and second order derivative methods

- Second order derivative methods are generally more accurate and converge in fewer steps
- Second order derivative methods are more resource intensive
- Sometimes it is easy and cheap to calculate the Hessian... (generalized linear models with canonical link), so why not?
- Sometimes it is expensive though.
- There is a tradeoff here that is context dependent.

Why not always second derivative

- It's expensive and takes more time / resources / memory.
- The Jacobian matrix only requires $O(n)$ storage.
- The Hessian matrix requires $O(n^2)$ storage.
- The size of the matrix grows exponentially with the size of the input data (specifically the number of columns).
- But... it can be more efficient if we take fewer steps, as long as the dataset isn't too big.

There is a tradeoff between the accuracy of the next step we take, and the amount of resources is take to calculate the next step.

3.2 First Order Derivative methods

3.2.1 Gradient descent

Sourced from wikipedia[1]

- Start somewhere (initial values for X)

[1] https://en.wikipedia.org/wiki/Gradient_descent

- Calculate the gradient at that point
- Take a step in the correct direction based on the gradient
- Step size is a function of the gradient (larger gradient means larger step size)
- Repeat.
- Stop algorithm once it converges (within tolerance) to a single point.
- This is **expensive**! Requires a full pass over all training data at every step...

Batch and Mini-batch optimization

- Decomposing optimization into a sum
- Estimating the gradient versus calculating the full gradient
- **Batch gradient methods**:
  - Use the entire training set to calculate the gradient at each step.
- **Stochastic (online) gradient methods**:
  - Only use a single example (row) at a time to estimate the gradient.
- **Minibatch gradient methods**:
  - Use more than one, but less than the full training dataset to estimate the gradient.
  - These are sometimes also called 'stochastic methods'.

3.2.2 Batch Gradient Descent

- Algorithm:
  - Start with some initial values for W
  - Using the full training dataset (batch), calculate the gradient (all partial derivatives for each W)
  - Take a step 'downhill' for each W using the gradient information.
  - Step size is a function of the learning rate (learning rate is a constant here)
  - Stop taking steps once you are at the minimum (convergence)
- You must try different learning rates.
  - Too slow (small steps), and it never converges.
  - Too fast, and it jumps around the minimum and may oscillate further away.

- This can take a long time, since it requires a full data pass every step.

Mini-batch size choice

- The larger the batch, the better the estimate of the gradient.
- Very small batches are not more efficient than slightly larger batches due to computational overhead on parallel frameworks.
- Memory requirements scale with simultaneous batch evaluation.
- Small batches can have a regularizing effect (decrease generalization error).
- First order methods (only use gradient, no Hessian!) can handle smaller minibatch sizes (100+)
- Second order methods (estimating the Hessian) need more data to get reliable estimates of the Hessian, so larger minibatch sizes are needed (10K +)
- MiniBatches must be randomly selected, so each new minibatch is an i.i.d. sample from the training data. (shuffle your data!)

Multiple passes through the data

- On the first pass through a shuffled dataset, we are using each minibatch to compute an unbiased estimate of the generalization error.
- However, most implementations will make multiple passes over the data. This 'resampling' starts to bias the estimate of generalization error.
  - However, training error is decreased on subsequent passes through the data.
  - Each pass over the datasets is considered one epoch
  - Epoch: one complete presentation of the dataset to our algorithm.

Challenges in optimization

- ill- conditioning of the Hessian or gradient matrix
  - Condition number: how much the output value of a function may change for a small change in the input values. Ill-conditioned = high condition number.
- Local minima / saddle points
- Steep cliffs, exploding gradients

Convex versus non-convex optimization

- Convex optimization:
  - Easy! General linear models, etc
  - No local minima.
  - Convergence usually happens.
  - (Hyper) parameter tuning is about getting better results
- Non-convex optimization:
  - Hard.
  - Deep learning usually exhibits this (more on this later in the course).
  - Parameter tuning enables convergence!
  - Stochastic gradient descent is a popular method for this...

3.2.3 Stochastic (online) gradient descent

(`https://en.wikipedia.org/wiki/Stochastic_gradient_descent`)

- Start somewhere (initial values for X)
- Randomly shuffle the data by row.
- For i=1,2,3...n, calculate the gradient **only for the i'th sample** (not the full dataset).
- Take a step in the correct direction based on the gradient
- Step size is a function of the gradient (larger gradient means larger step size)
- Repeat.
- Stop algorithm once it converges (within tolerance) to a single point.
- This is less costly, since you don't need a full pass over the data for every step. But it is less accurate as well...
- Most used optimization method for deep learning (and possibly machine learning in general on larger datasets)
- Instead of using the full dataset at every step, only take a sample of data from the training data to use (a 'mini-batch').

- Scales well to bigger datasets, since mini-batch size can be constant.
- Provides an estimate of the gradient based on this sample of data.
  - Advantage: more and more data has decreasing returns in estimating the gradient, so mini-batch instead of batch makes sense.
  - Disadvantage: There is a new source of variance being introduced – minibatch random sample variance. This source of variance does not decrease as the algorithm get's closer to convergence.This means the final solution is good, but may not be optimal (just in the neighborhood of optimal).

Minibatch sample variance

- This 'jumping around behavior' can be an advantage, as this may escape a local minimum or saddle.
- But the downside is we never reach the minimum, because we keep jumping around it. . .
- One solution is to decrease the learning rate as the algorithm proceeds, enforcing smaller steps as the algorithm proceeds. This is in contrast to keeping the learning rate constant, which we did in Batch gradient descent. This change is called the 'learning rate schedule'.

Learning Rate Schedule

- The learning rate schedule is different depending on the software you use!
  - Can be constant or decreasing by some function.
- Check the docs for sklearn (1.3.6.1) – https://scikit-learn.org/0.15/modules/sgd.html#sgd

3.2.4 Coordinate descent

(https://en.wikipedia.org/wiki/Coordinate_descent

- If we have multiple X values, then we optimize them by only considering a change in a single X value at a time. The step size is based on only changing one X.

- This is useful if it is hard to calculate the gradient for all variables (the Jacobian), but easier to only work on one variable at a time.
- This is the optimal solver for regularized GLM's (elastic net regression).
  - Start somewhere (initial values for X)
  - Choose one of the X's (coordinates), change the value (your step).
  - Calculate the objective function. Next round, change a different X.
  - Repeat.
  - Stop algorithm once it converges (within tolerance) to a single point.

Reference to Regularization Paths for Generalized Linear Models via Coordinate Descent[2]

3.3 Second Order Derivative methods

3.3.1 Iteratively reweighted least squares (IRLS)

- This is the most popular in data science frameworks.
- This is efficient and accurate for generalized linear models (logistic regression, Poisson regression etc. . .)
- The Hessian matrix (second derivative) gives us information about the uncertainty of the solution. This is where our confidence intervals and p-values come from!

3.3.2 Newton-Raphson optimization and Fisher Scoring

- More general cases of IRLS. These are identical when applied to GLM's, so you often see the terms interchanged when talking about GLM's. These are not necessarily identical outside of GLM's.
- Second-order methods are sometimes termed 'Newton methods'

[2] `https://web.stanford.edu/~hastie/Papers/glmnet.pdf`

3.3.3 Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS)

Technically first order since it does not evaluate the Hessian.

- However, it does approximate the Hessian by storing the prior gradient evaluations!
- So we get some idea of the rate of change of the gradient by looking at the trend of the prior gradients.
- This is termed 'quasi-Newton' since we approximate the Hessian without actually incurring the full cost

Orthant-Wise Limited-memory Quasi-Newton (OWL-QN) is an extension to this that effectively optimizes regularized regression (L1 or elastic net). This is implemented in Apache Spark.

3.3.4 L-BFGS Versus IRLS for GLM's

- Both can be implemented in parallel by calculating chunks of rows at a time.
- Consider m rows and n columns…the IRLS algorithm requires an NxN matrix be generated no matter how small we make M by chunking by row.
- So if we have a large number of columns, IRLS can underperform (take too long / too much memory), even in distributed environments.
- L-BFGS is more efficient for a large number of columns. But, it is generally less accurate (Takes more steps).

3.4 Close thoughts

- Choice of optimization method is important!
- Depending on how large your data is, or how complex your objective function is, you may have to try different optimization methods.
- If the optimization method you choose does not give you estimates about the uncertainty of the solution (I.E. confidence intervals and p-values), you may

be able to get that from a direct Hessian calculation once you have declared the optimal solution to be found.

第四章　Machine learning basics

Machine learning is essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around those functions

– Deep Learning Book

- Using data, develop a ‘learning algorithm’ (our model).
- Often the focus is prediction of an outcome, given inputs.
- Finding patterns in the data versus finding generalizable trends in the data.

4.1 What do we need to develop a learning algorithm?

- Data
- Model
- Cost function
- Optimization function

4.2 Classification, regression, and clustering

1. Classification

- Predicting class membership (or probabilities) among distinct classes.
- Death (Yes / No)
- Risk Strata (Low / Medium / High)

2. Regression

- Predicting a continuous summary statistic (like the mean)
- Hospital cost (Mean, median, 90th percentile)

3. Clustering

- Identifying clusters in our data.
- Project data into smaller dimensionality.
- Clustering can be discrete or continuous.

Central challenge to ML: **generalization**.

The algorithm must perform well on new data it has never seen before

- Next years data
- New healthcare system
- New patients

4.3 Under/overfitting and out of sample data

Given the data we have, how good is our model?

- This is really just optimization.
- *Training error* is how well we fit the training data.
- Increased performance here sometimes decreases performance outside of our sample of data (*overfitting*)

In ML, we target generalization error

- *Generalization error* is how well our algorithm fits data outside our sample.
- But we don't have any data outside our sample…
- Can we pretend we do?

4.4 Validation approaches

- If the new data does not come from the same data-generating distribution as the observed data, full stop.
- If we assume the new data comes from the same data-generating distribution, then we can implement validation approaches.
  - Create multiple random samples from the data we have
  - Call one 'training data' and one 'Validation' data.
  - Actually we usually split into training / validation / testing (3 splits).
- Optimization goal:
  - Minimize training error (high error = underfitting)
  - Minimize gap between training and testing error (big gap = overfitting)

Creative Validation Approaches:

- Splitting by clusters.
  - Split by year, region, etc
- Cross validation

Balancing under and overfitting:

- We can balance under and overfitting by making our model more/less complex. The deep learning book calls this **model capacity**
- Increasing model capacity generally allows the model to fit more nuanced relationships.
  - In linear modeling – add more inputs, consider non-linear terms (polynomials), consider interactions...
- What is the downside of increased model capacity?

Model parsimony

> Among competing hypotheses that explain known observations equally well, choose the simplest one.
>
> - Occam's razor (c. 1287-1347)

4.5 Regularization

- Hard code preferences into the model.
  - I prefer Beta's close to or equal to zero (parsimony)
  - However, if I find enough support for a relationship, it can stay.
  - How to I hard code that into my model?
  - What is an example you have learned of this in ML?

  Regularization is any modification we make to a learning algorithm that is intended to reduce its **generalization error not it's training error**.'

4.6 Hyperparameter tuning

- Hyperparameters are 'knobs' we can use to tune an ML algorithm
- We do not learn these from the training data, because…
  - It is too hard/impossible to optimize them directly OR
  - Their intent is to decrease generalization error (not training error), so it is not appropriate to learn them from the training data. Why is this true?
- We often have multiple hyperparameters, and wish to tune across all of them. This is referred to as the 'hyperparameter grid'.

4.7 Binary classification models in scikit-learn

- Logistic regression
- Elastic net logistic regression (regularized)
- Support Vector Machine
- General SGD estimation (sklearn), it can specify different loss functions
- Nearest neighbor majority vote (non-parametric)
- Random forests
  - Fully grown trees (not weak learners). Low bias, high variance per tree

  - Grow many trees (maybe in parallel?) to reduce variance.
- Gradient boosting machine
  - Forest of weak learner trees (high bias, low variance)
  - Correct bias each new sequence.
  - Sequential method!

第五章　Decision trees

Supervised machine learning review

- Given input X, predict target Y
- Come up with model that is a function if inputs (X) and model parameters (θ)
- Training goal – learn the model parameters (θ) that gives the best model
- Must define an objective function to optimize model. This is a function of *training loss* and *regularization*:

$$obj(\theta) = L(\theta) + \Omega(\theta)$$

- Different objective (loss) functions can be used:
  - Linear model - mean square error

$$L(\theta) = \sum (y_i - \hat{y})^2$$

  - Logistic loss

$$L(\theta) = \sum [y_i ln(1 + e^{-y_i}) + (1 - y_i) ln(1 + e^{\hat{y}_i})]$$

5.1 Random forest

- Grow uncorrelated deep trees. **Embassingly parallel -> high performance**
- Each tree definitely overfits the data (but in different ways)

- The large number of trees and stochastic nature of each tree hopefully averages out the differences.
- Not much parameter tuning required here.

This is generally called 'bagging': generate a bag of fully developed trees – the population vote is a good prediction.

5.2 Gradient boosting machine (GBM)

- Forward learning ensemble method.
  - Increasingly refine the model step by step to get good predictions. sequential model and cannot be parallelized.
  - Sequential in nature – next step requires prior step (for each tree).
- Use gradient descent to update predictions based on a learning rate.
- **'Boosting' instead of 'bagging'**.
  - This method creates an ensemble of weak learners.
- As we grow trees, implement regularization to avoid overfitting.

The idea of GBM:

- Fit a simple model
- Analyze the errors (residuals) from that simple model
- Update the model to better predict the residuals
- Analyze the new errors.
- Repeat...
- **Well-tuned GBM is almost always better than RFT.**

GBM Parameter tuning

Primarily about bias-variance tradeoff

- Balance model complexity and predictive power.
- Tree size – the deeper the tree, the more complex the model
  - Max_depth/Max_leaf_nodes – control the size of each tree
- Regularization

  - Learning rate (eta) – step size shrinkage during updates (prevents over-fitting)
  - Lambda – more regularization in xgboost)
- Total number of trees in the ensemble

5.3 xgboost

5.4 Python packages

5.4.1 `dask-ml`

http://ml.dask.org/

- Generalized linear models
  - Linear, logistic, Poisson
  - L1 and L2 regularization
  - Newton optimization method, gradient descent, lbfgs
- GBM's through the xgboost library
- Somewhat immature at this point.
- I prefer to use Dask for bigger than memory data transformations rather than ML.

5.4.2 `Apache Spark ML`

- Seems robust? Many models available
  - GBM, Random Forest, GLM, survival analysis, SVM, multilayer perceptron
- Mllib – 'new' API based on Dataframes
  - 'old' RDD based API is deprecated.
- Some benchmarks indicate Apache Spark ML is very slow compared to it's peers.

- However, Spark is a popular cluster technology, so perhaps your company has a cluster available…
- Python or R API's, in addition to Scala and Java

5.4.3 `h2o`

> 'H2O is an open source, in-memory, distributed, fast, and scalable machine learning and predictive analytics platform that allows you to build machine learning models on big data and provides easy productionalization of those models in an enterprise environment.'
>
> – from the docs

- H2o seems to be a leader in productionalizing models post development.
- GUI Flow browser in addition to Python and R API
- Cox proportional hazards models
- Deep learning models
  - Supports feed forward artificial neural networks
  - Does not support recurrent neural networks (RNN) or convolutional neural networks (CNN)
  - RNN -> time series or 1-d data, CNN considers covariance and relationships among different pixels.
- Generalized linear models
  - Gaussian, Poisson, binomial, multinomial, gamma, ordinal, negative binomial
  - L1/L2/Elastic net – can compute regularization path!
  - IRLS, L_BFGS, coordinate descent, gradient descent
- Distributed random forest
- Generalized boosting machines
- XGBoost

- AutoML
- Some clustering as well
  - K-means
  - Principal component analysis
  - Generalized low rank models

`h2o` checkpoint

- Only available for distributed random forests, generalized boosted machine, and deep learning models (ANN)
- Update past model estimates with new data rather than fitting a new model from scratch.

第六章　Deep learning

6.1 Deep Feed-Forward Neural Network (DNN)

A DNN, aka multilayer perceptrons (MLPs) is composed with multiple non-linear transformation layers, which is used to classify some outcome y. The output of each layer is fed to the next layer as input.

6.2 Recurrent Neural Network (RNN)

In order to handle *sequential or temporal* data of **arbitrary length** and capture temporal information from the data, recurrent neural network (RNN) models are widely used. Unlike feedforward neural network models, RNN models perform the same operation at each time step of the sequence input, and feed the output to the next time step as part of the input. Thus, RNN models are able to memorize what they have seen before and benefit from shared model weights (parameters) for all time steps. In order to capture complex long temporal dependency and avoid vanishing gradient problems, some modified RNN models such as Long ShortTerm Memory (LSTM) and Gated Recurrent Unit (GRU) have been proposed with state-of-the-art performance.

第七章　MCMC and variational inference

In MCMC, we construct ergodic Markov chains whose stationary distribution is the posterior distribution. Instead of using **sampling**, variational inference uses **optimization** to approximate the posterior distribution, which is better suited when applied to large data or complex models.

MCMC and VI are different approaches to solve the sample question. MCMC is more computationally intensive but guarantees producing exact samples from the target density (it is guaranteed to find a global optimal solution[1]). VI takes advantage of methods such as stochastic optimization and distributed optimization, so it is preferred if we want to fit model to a large dataset. VI will almost never find the globally optimal solution, but we will always know if it converged, and we will hav bounds on its accuracy.

7.1 MCMC

7.2 Variational inference

Variational inference (VI) first posits a family of densities, and then find a member of that family that is close to the target density, where the closeness is evaluated

[1] https://ermongroup.github.io/cs228-notes/inference/variational/

by Kullback–Leibler divergence [Blei et al., 2017]. Compared with MCMC, VI methods tends to be faster and easier to use in terms of large data, but it generally **underestimates** the variance of the posterior density due as a consequence of its objective function.

Steps to perform VI:

1. posit a family of approximate densities Q
2. find a member of that family that minimizes the Kullback-Leibler (KL) divergence to the exact posterior
3. approximate the posterior with the optimized member of the family $q^{\star}(.)$.

The KL divergence of two distributions q and p with discrete support is:

$$KL(q||p) = \sum_{x} q(x) \log \frac{q(x)}{p(x)}$$

One of the key ideas behind VI is to choose Q to be *flexible enough* to capture a density close to $p(\theta|x)$, but *simple enough* for efficient optimization.

The aims of modern VI:

- tackling Bayesian inference problems that involve massive data,
- using improved optimization methods for solving equation 1, which is ususally subject to local minima,
- develop generic VI algorithms that apply to a wide class of models
- increase the accuracy of VI

Mean-field VI

coordinate-ascent optimization

stochastic VI

第八章　undecided

第九章　参考文献

David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518): 859–877, 2017.