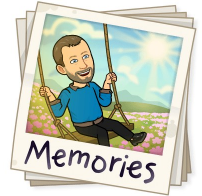


Machine Learning

Textbook: Machine Learning Refined, by Watt, Borhani, and Katsaggelos, 2nd edition:
MLRefined.com

Previous Lecture

- ♦ Neural-network training repeatedly combines a forward pass, loss evaluation, backpropagation, and a parameter update.
- ♦ Mini-batch training applies the finite-sum optimization ideas from stochastic gradient methods to gradients produced by backpropagation.
- ♦ Learning rate controls step size: a small step can reduce loss while a larger step in the same gradient direction can overshoot.
- ♦ Random variance-scaled initialization breaks hidden-unit symmetry and helps keep activations and gradients at useful numerical scales.
- ♦ Weight penalties, validation monitoring, and early stopping help control generalization rather than merely training fit.
- ♦ Backpropagation supplies gradients; the optimizer, batch structure, learning-rate schedule, and stopping rule determine how those gradients are used.



§14.1 thru 14.4 - Decision Trees

Neural networks create nonlinear predictors
by composing learned feature maps.

Decision trees take a very different approach.

They ask a sequence of simple questions:

$$x_j \leq s ?$$

Each answer sends an observation
left or right.

Repeated splits partition the input space
into regions.

Each final region receives a simple prediction.

No gradient descent is required
to choose the splits.

A Decision Tree

A tree contains:

internal nodes

that test a condition;

branches

that represent the possible outcomes;

leaves

that make predictions.

For a numerical feature:

a common internal-node rule is

$$x_j \leq s.$$

If true:

go left.

If false:

go right.

A path from the root to a leaf

is a sequence of if–then conditions.

A Decision Stump

The simplest nontrivial tree

has one split.

This is called a

decision stump.

For one variable:

$$\hat{y}(x) = \begin{cases} c_L, & x \leq s, \\ c_R, & x > s. \end{cases}.$$

The model is nonlinear in the input

even though each region uses

only a constant prediction.

A deeper tree

recursively splits the regions

created by earlier splits.

Regression Trees Are Piecewise Constant

For regression:

every leaf

contains a subset of training responses.

All observations reaching that leaf

receive the same prediction:

$$\hat{y} = c_{\text{leaf}}.$$

Thus a regression tree

is a piecewise-constant function.

The important questions are:

where should we split?

and

what constant should each leaf predict?

The Best Regression Leaf Value

Suppose a leaf contains responses

$$y_1, \dots, y_n.$$

Choose a constant

$$c$$

to minimize squared error:

$$J(c) = \sum_{i=1}^n (y_i - c)^2.$$

Differentiate:

$$J'(c) = 2 \sum_{i=1}^n (c - y_i).$$

Set equal to zero:

$$nc = \sum_i y_i.$$

Therefore

$$c^* = \bar{y}.$$

A least-squares regression leaf predicts
the mean response in that leaf.

Why the Mean Appears Again

This is the same result
from earlier lessons:

a constant model
fit by squared error
chooses the sample mean.

Thus tree fitting separates into:

discrete problem:
choose which observations belong together;

continuous problem:
choose the best constant inside each group.

Once the groups are fixed:
the continuous optimization is solved exactly
by taking means.

A Candidate Regression Split

At a node containing index set
 S ,

consider splitting feature

j
at threshold
 s .

Define

$$S_L = \{p \in S : x_{pj} \leq s\},$$

and

$$S_R = \{p \in S : x_{pj} > s\}.$$

For fixed

$$(j, s):$$

left prediction

$$c_L = \bar{y}_L;$$

right prediction

$$c_R = \bar{y}_R.$$

Regression Split Cost

The total squared error after the split is

$$Q(j, s) = \sum_{p \in S_L} (y_p - \bar{y}_L)^2 + \sum_{p \in S_R} (y_p - \bar{y}_R)^2.$$

Choose

$$(j^*, s^*) = \underset{j, s}{\operatorname{argmin}} \quad Q(j, s).$$

Equivalent viewpoint:

maximize the reduction in squared error.

If

$$Q_{\text{parent}}$$

is the unsplit node SSE:

$$\Delta Q = Q_{\text{parent}} - Q(j, s).$$

Larger

$$\Delta Q$$

means a better split.

Where Do Candidate Thresholds Come From?

For one numerical feature:

sort the distinct observed values.

If consecutive distinct values are

$$x_{(r)}$$

and

$$x_{(r+1)}.$$

a candidate threshold may be placed at

$$s = \frac{x_{(r)} + x_{(r+1)}}{2}.$$

Only thresholds that change
which observations go left or right
need to be considered.

For many features:
perform this search
across candidate thresholds
for every feature.

A Regression Example

Consider:

x	1	2	3	4	5	6
y	1	1.5	2	8	8.5	9

Without a split:

$$\bar{y} = 5.$$

Parent SSE:

$$Q_{\text{parent}} = 74.5.$$

Candidate thresholds:

1.5, 2.5, 3.5, 4.5, 5.5.

We evaluate each one.

Evaluate the Regression Thresholds

Threshold s	$\bar{y}_L$	$\bar{y}_R$	Child SSE
1.5	1	5.8	55.3
2.5	1.25	6.875	32.3125
3.5	1.5	8.5	1.0
4.5	3.125	8.75	32.3125
5.5	4.2	9	55.3

The best threshold is

$$s^* = 3.5.$$

The stump is

$$\hat{y}(x) = \begin{cases} 1.5, & x \leq 3.5, \\ 8.5, & x > 3.5. \end{cases}.$$

Improvement From the Best Split

Before splitting:

$$Q_{\text{parent}} = 74.5.$$

After splitting at

$$s = 3.5:$$

$$Q_{\text{children}} = 1.$$

Therefore:

$$\Delta Q = 74.5 - 1 = 73.5.$$

The split explains

most of the within-node variation.

But the algorithm may continue splitting

the two children.

Every additional split

can further reduce or preserve
training SSE.

Recursive Splitting

After choosing the root split:

repeat the same procedure

inside each child node.

For each current node:

1. search candidate features and thresholds;
2. select the split with best local improvement;
3. create two children;
4. repeat until a stopping rule is reached.

This is

recursive binary splitting.

The procedure is greedy:

choose the best split now, without solving the entire future tree globally.

Greedy Does Not Mean Globally Optimal

A root split that gives

the largest immediate decrease in loss

may not be the root split

of the globally best tree

of a specified final size.

Why?

An initially weaker split

might create child regions

that can be partitioned much better later.

Searching all possible tree structures
is combinatorially difficult.

Decision-tree algorithms therefore use
greedy local optimization
as a practical compromise.

Tree Geometry

A split
 $x_j \leq s$
is axis-aligned.

In two dimensions:

a split on
 x_1
creates a vertical boundary;

a split on
 x_2
creates a horizontal boundary.

Recursive splitting creates
axis-aligned rectangular regions.

Thus tree decision boundaries
can form staircase-like shapes.

This geometry is very different
from hyperplanes,
kernels,
or neural-network surfaces.

Trees Naturally Represent Interactions

Consider the rules:

if
 $x_1 \leq 2$:
predict 0;

else if
 $x_2 \leq 1$:
predict 3;

else:
predict 7.

The effect of
 x_2
matters only when
 $x_1 > 2$.

Thus:
tree paths automatically create feature interactions.

No explicit interaction feature
 x_1x_2
needs to be constructed.

Classification Trees

For classification:
each leaf contains class labels.

Suppose there are
 K
classes.

Within one leaf:
 $\hat{p}_k$
=
fraction of leaf observations
belonging to class
 k .

Predicted class:
 $\hat{y} = \underset{k}{\operatorname{argmax}} \hat{p}_k$.

Thus a classification tree
estimates piecewise-constant class probabilities.

Node Purity

A classification node is
pure
when every observation
belongs to the same class.

A split should ideally create children
that are purer
than the parent.

We therefore need a numerical measure
of class mixture.

Two standard choices:
Gini impurity
and
entropy.

Gini Impurity

If class proportions in a node are
 $p_1, \dots, p_K$:
 $I_G = 1 - \sum_{k=1}^K p_k^2$.

For a pure node:
one
 $p_k = 1$
and all others are zero.

Therefore:
 $I_G = 0.$

For a balanced binary node:
 $p_1 = p_2 = 1/2.$

Then:
$$I_G = 1 - \frac{1}{4} - \frac{1}{4}$$
$$= \frac{1}{2}.$$

Entropy

Another impurity measure:
$$I_H = - \sum_{k=1}^K p_k \log_2 p_k,$$

with convention
 $0 \log 0 = 0.$

Pure node:
 $I_H = 0.$

Balanced binary node:
 $I_H = 1 \text{ bit}.$

For proportions
(3/4, 1/4):
 $I_H \approx 0.8113.$

Gini and entropy differ numerically,
but both reward purer children.

Classification Split Criterion

Suppose a parent node contains
 n
observations.

A candidate split produces:
 n_L
left observations
and
 n_R
right observations.

Weighted child impurity:
$$I_{\text{split}} = \frac{n_L}{n} I_L + \frac{n_R}{n} I_R.$$

Impurity decrease:

$$\Delta I = I_{\text{parent}} - I_{\text{split}}.$$

Choose the split
with the largest impurity decrease
or equivalently
smallest weighted child impurity.

A Classification Example

Consider one feature

$$x = 1, \dots, 8$$

with labels:

x	1	2	3	4	5	6	7	8
y	0	0	0	1	1	1	0	1

Parent proportions:

$$p_0 = p_1 = 1/2.$$

Parent Gini:

$$I_G = 0.5.$$

Consider threshold

$$s = 3.5.$$

Gini Gain for $s = 3.5$

Left node:

labels
(0,0,0).

Thus:

$$I_L = 0.$$

Right node:

labels
(1,1,1,0,1).

Class proportions:

$$p_1 = 4/5,$$

$$p_0 = 1/5.$$

Right Gini:

$$1 - (4/5)^2 - (1/5)^2$$

$$= 0.32.$$

Weighted child Gini:

$$\frac{3}{8}(0) + \frac{5}{8}(0.32)$$

$$= 0.20.$$

Gini gain:

$$0.50 - 0.20 = 0.30.$$

Compare Another Threshold

At

$s = 4.5:$

left labels:

$(0, 0, 0, 1);$

right labels:

$(1, 1, 0, 1).$

Each child has class proportions

$(3/4, 1/4)$

in some order.

Each child Gini:

$0.375.$

Weighted child Gini:

$0.375.$

Gain:

$0.5 - 0.375 = 0.125.$

Thus

$s = 3.5$

is preferred to

$s = 4.5$

by the Gini criterion.

Why Not Split Using Classification Error Alone?

Node misclassification impurity:

$I_{\text{class}} = 1 - \max_k p_k.$

This is natural for final prediction error.

But:

- it can be relatively insensitive
- to changes in class proportions
- that do not change the majority class.

Gini and entropy are usually more sensitive
for choosing splits.

Thus:

the metric used to choose tree structure
need not be identical
to the final reported classification metric.

When Does Tree Growth Stop?

If we keep splitting:
training error usually continues decreasing
or stays unchanged.

Possible stopping controls:

maximum depth;

minimum observations required to split a node;

minimum observations allowed in a leaf;

maximum number of leaves;

minimum impurity decrease.

These are
hyperparameters.

They control model complexity.

A Fully Grown Tree Can Overfit

With enough splits:

a regression tree can create
very small leaves
and nearly memorize training responses.

A classification tree
can often create pure leaves
and achieve very small
or zero training error.

But:
tiny leaves have unstable predictions.

Small changes in the training sample
may produce different splits.

Single decision trees therefore
can have high variance.

Pruning

One strategy:
first grow a relatively large tree,

then remove splits
that do not justify their complexity.

A cost-complexity form is
 $R_\alpha(T) = R(T) + \alpha|T|_{\text{leaves}},$

where:
 $R(T)$
measures training fit;

$|T_{\text{leaves}}|$
is the number of terminal leaves;

$\alpha \geq 0$
penalizes tree size.

Larger
 α
favors smaller trees.

A Small Pruning Comparison

Suppose:

Tree A:
 $R(A) = 10,$
2 leaves;

Tree B:
 $R(B) = 4,$
5 leaves.

Then:
 $R_\alpha(A) = 10 + 2\alpha,$

$R_\alpha(B) = 4 + 5\alpha.$

Tree B is preferred when
 $4 + 5\alpha < 10 + 2\alpha.$

Thus:
 $\alpha < 2.$

At
 $\alpha = 2:$
tie.

For
 $\alpha > 2:$
the smaller Tree A is preferred.

Use Validation to Control Tree Complexity

Possible model-selection choices include:

maximum depth;

minimum leaf size;

pruning parameter

α .

Training loss alone
favors deeper or less-pruned trees.

Therefore:
tree complexity should be selected using validation / cross-validation.

The leakage rules from Lesson 1.30
still apply
to any preprocessing learned from data.

Trees Usually Do Not Need Feature Standardization

An ordinary numerical tree split depends on
the ordering of feature values.

Suppose
 $z = ax + b$
with
 $a > 0$.

Then:
 $x \leq s$
is equivalent to
 $z \leq as + b$.

Thus a positive affine rescaling
preserves the possible partitions.

More generally:
strictly monotone transformations
preserve order
and can be matched by transformed thresholds.

Therefore ordinary axis-aligned trees
usually do not require standardization
for the same reason
distance-based or gradient-based methods often do.

Tree Prediction Is Fast

After the tree has been fit:
predicting one observation means
follow one path
from root to leaf.

At each node:
evaluate one split rule.

A balanced tree of depth
 d
uses roughly
 d

decisions per prediction.

The expensive part is usually
searching many candidate splits
while fitting the tree.

Decision Trees Versus Earlier Models

Model	Representation	Training	Prediction Shape
Linear model	raw / fixed features	convex optimization	hyperplane / linear
Kernel model	implicit fixed features	kernel optimization	nonlinear smooth-ish
Neural network	learned features	gradient-based, nonconvex	flexible composition
Decision tree	recursive regions	greedy split search	piecewise constant

Decision trees are unusual in this course:

their fitting algorithm is fundamentally
combinatorial and greedy,

not derivative-based.

A Regression Stump in Python

```
import numpy as np

x = np.array([1., 2., 3., 4., 5., 6.])

y = np.array([1., 1.5, 2., 8., 8.5, 9.])

thresholds = (
    x[:-1] + x[1:]
) / 2
```

Candidate thresholds:
[1.5 2.5 3.5 4.5 5.5]

Now compute the child SSE
for each threshold.

Search for the Best Regression Split

```
best_sse = np.inf
best_threshold = None

for s in thresholds:

    left = y[x <= s]
    right = y[x > s]

    left_mean = np.mean(left)
    right_mean = np.mean(right)

    sse = np.sum(
```

```

        (left - left_mean)**2
    )

    sse += np.sum(
        (right - right_mean)**2
    )

    if sse < best_sse:
        best_sse = sse
        best_threshold = s

```

Result:

```

best_threshold = 3.5,
best_sse = 1.0.

```

Store the Leaf Predictions

After selecting

```

s = 3.5:
    left_mean = np.mean(
        y[x <= best_threshold]
    )

    right_mean = np.mean(
        y[x > best_threshold]
    )

```

Results:

```

left_mean = 1.5,
right_mean = 8.5.

```

Prediction function:

```

def predict_stump(x_new):
    return np.where(
        x_new <= best_threshold,
        left_mean,
        right_mean
    )

```

The tree is represented

by a threshold
plus two leaf values.

Gini Impurity in Python

For class labels encoded

```

0, 1, ..., K - 1:
def gini(y):
    classes, counts = np.unique(
        y,
        return_counts=True
    )

    p = counts / len(y)

    return 1 - np.sum(p**2)

```

New Python idea:

```

np

```

with

```
return_counts=True
```

returns each observed class
and its frequency.

For a pure node:

```
gini(y)
```

returns

0.

Classification Split Search

For each candidate threshold:

```
parent = gini(y)
```



```
left_y = y[x <= s]
right_y = y[x > s]
```



```
weighted = (
    len(left_y) / len(y)
) * gini(left_y)
```



```
weighted += (
    len(right_y) / len(y)
) * gini(right_y)
```



```
gain = parent - weighted
```

Choose the threshold
with largest
gain.

Then recurse
inside the child nodes
until a stopping rule is reached.

Why Ensembles Are So Effective for Trees

A single deep tree:

can fit complicated interactions;

is easy to interpret locally;

but can be unstable.

Small changes in training data
can change the first few splits,

which changes the entire downstream tree.

This high variance suggests:

fit many trees
and combine them.

Two major approaches:
random forests
and
gradient boosting.

That is the final lesson.

What did we learn?

- ◆ Decision trees recursively partition the input space with simple threshold rules and make constant predictions inside terminal leaves.
- ◆ A least-squares regression leaf predicts the mean response; candidate splits are chosen by minimizing the total child SSE or maximizing its reduction.
- ◆ Classification trees estimate class proportions in leaves and commonly choose splits using weighted Gini impurity or entropy.
- ◆ Tree fitting is greedy recursive split search rather than gradient-based optimization, so locally optimal splits need not produce a globally optimal final tree.
- ◆ Deeper trees reduce training error but can have high variance; depth limits, minimum leaf sizes, pruning, and cross-validation control complexity.
- ◆ Ordinary axis-aligned trees are largely insensitive to monotone rescaling of individual numerical features, but single trees can be unstable enough that ensembles are often substantially stronger.



Next

- A single decision tree is flexible but often unstable and high-variance.
- Random forests reduce variance by averaging many intentionally decorrelated trees.
- Boosting instead builds trees sequentially, with each new tree correcting weaknesses of the current ensemble.
- These two ensemble ideas produce some of the strongest general-purpose models for tabular data.
- **1.36 - Random Forests and Gradient Boosting**