CS 1810 Spring 2026 Section 6
Transformers and Decision Trees

1 Transformers

1.1 Why Transformers?

Many relevant machine learning problems involve data that comes in sequences: a sentence is a sequence of words, a time series is a sequence of measurements, and a DNA strand is a sequence of nucleotides. In all of these settings, the prediction at one location may depend on information that appears much earlier or much later in the sequence. For example, in the sentence

"The hippo didn't cross the street because it was tired,"

understanding the word "it" requires linking it to "the hippo," not merely to the immediately preceding word. These are often called long-range dependencies. Earlier sequence models such as recurrent neural networks process a sequence one step at a time, updating a hidden state according to a rule of the form

$$h_t = f(h_{t-1}, x_t)$$

where x_t is the input at position (or time) t in the sequence and h_t is the hidden state. This creates two difficulties. First, information from token x_i must travel through many intermediate states before it can influence token x_j if positions i and j are far apart, so long-range dependencies are hard to learn. Second, because the computation is inherently sequential, training cannot fully exploit parallel hardware.

Transformers address both problems by replacing recurrence with an operation called *self-attention*. Instead of forcing information to flow through a chain, self-attention lets each token directly look at all other tokens and decide which ones matter for building its representation.

1.2 A First Geometric Picture of Attention

Before writing formulas, it helps to form the right picture in mind. Imagine that each token in a sentence is represented by a point or vector in a high-dimensional feature space. The purpose of a transformer layer is to update each token's representation so that it incorporates relevant context.

Geometrically, attention does two things. First, for each token, it determines a set of *weights* telling us which other tokens are relevant. Think about this as representing "who should I listen to?" Second, it forms a *weighted average* of information coming from those relevant tokens. Think about this as representing "what information should I pull in?"

So the output representation of a token is not created from scratch. Rather, it is obtained by moving the token's representation in the direction of a context-dependent average of other token features. One can think of attention as a learned, adaptive smoothing operation on a cloud of token vectors, where the smoothing is local in *semantic relevance*.

1.3 Self-Attention

Suppose the input sequence has length n, and each token is represented by a vector in $\mathbb{R}^d$. We can consider these token vectors as the rows of a matrix

$$X \in \mathbb{R}^{n \times d}$$

where the ith row x_i is the current representation of token i. Self-attention creates a new matrix $Z \in \mathbb{R}^{n \times d_v}$, of which the rows are context-enriched token representations. The central question is: how should token i decide which other tokens to use? The transformer learns three different linear views of the input:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V.$$

Here Q, K, and V are called the *query*, *key*, and *value* matrices.

What are queries, keys, and values really doing?

- A *query* is a representation of what a token is currently looking for. If the token is ambiguous, incomplete, or depends on something earlier in the sentence, its query should point toward the kind of information it needs.
- A *key* is a representation of what each token can offer to others. It is the feature vector against which queries are matched.
- A *value* is the actual information that gets passed along once a match is made.

This separation is important. A token might be good at indicating that it is relevant through its key, while separately carrying useful content in its value. In other words, keys are used for matching, values are used for averaging, and queries decide what to match against.

Geometric interpretation of Q and K. For an arbitrary token i, the query vector q_i defines a direction in feature space that corresponds to the kind of contextual information token i wants. For an arbitrary token j, the key vector k_j defines a direction that corresponds to the kind of information token j provides. Their dot product $q_i^\top k_j$ is large when those directions are aligned. Thus, the query-key dot product measures how relevant token j is to token i.

Collecting these similarities across all pairs gives the score matrix

$$S = \frac{1}{\sqrt{d_k}} QK^\top$$

where the entry S_{ij} measures how strongly token i should attend to token j. We then apply the softmax function row by row to obtain

$$A_{ij} = \frac{e^{S_{ij}}}{\sum_{\ell=1}^{n} e^{S_{i\ell}}}.$$

where this matrix A is called the attention matrix. Note that each row sums to 1, so the ith row is a probability distribution over the tokens in the sequence.

Finally, we form the output

$$Z = AV$$

and equivalently, the output at token i is

$$z_i = \sum_{j=1}^{n} A_{ij} v_j$$

This formula helps clarify the geometry behind attention: the output z_i is a convex combination of the value vectors. Thus, token i moves toward the weighted center of the values it attends to. The weights themselves depend on learned similarities between queries and keys.

1.4 Why the Scaling Factor?

The score matrix uses

$$S = \frac{1}{\sqrt{d_k}} QK^\top$$

rather than just $QK^\top$. This scaling is needed for stability. To see why this is true, suppose that the entries of q_i and k_j are roughly mean-zero with variance on the order of 1. Then the dot product

$$q_i^\top k_j = \sum_{m=1}^{d_k} q_{im} k_{jm}$$

is a sum of d_k terms, so its variance grows on the order of d_k. As the input dimension increases, the dot products become larger in magnitude. Large logits make the softmax very peaked, and a very peaked softmax yields tiny gradients almost everywhere. Dividing by a factor of $\sqrt{d_k}$ keeps the typical scale of the logits approximately constant as the dimension changes. This prevents premature softmax saturation and makes training much more stable.

1.5 Attention as Learned Similarity

Another useful way to think about attention is to compare it with our familiar kernel methods, where we defined a similarity function $k(x, x')$ that says how much data points should influence each other. Attention does something related, but more flexibly: the score $q_i^\top k_j$ acts like a learned similarity between tokens i and j. Unlike a fixed kernel, this similarity is itself trainable because the projections W_Q and W_K are learned from data. This means the model can learn what kinds of relationships matter. Thus, self-attention may be viewed as a learned, data-dependent similarity rule followed by a weighted averaging of value vectors.

1.6 Multi-Head Attention

A single attention map can only express one notion of relevance at a time. However, language contains many kinds of relationships simultaneously. When designing a transformer, we therefore may want one head to track nearby modifiers, another to link pronouns to nouns, another to detect clause boundaries, etc. For this reason, transformers use *multi-head attention*. Instead of creating one set of queries, keys, and values, the model creates several:

$$Q^{(h)} = XW_Q^{(h)}, \qquad K^{(h)} = XW_K^{(h)}, \qquad V^{(h)} = XW_V^{(h)}$$

for heads $h = 1, \ldots, H$. Each head computes its own attention output

$$Z^{(h)} = \mathrm{softmax}\left(\frac{Q^{(h)}(K^{(h)})^\top}{\sqrt{d_k}}\right) V^{(h)}$$

which are then concatenated and linearly mixed to yield

$$\mathrm{MHA}(X) = [Z^{(1)}; \ldots; Z^{(H)}]W_O$$

Geometrically, each head looks at the token cloud through a different set of learned coordinates. Different heads can thus discover different geometric structures in the same sequence. The key intuition to remember is that multi-head attention gives the model several parallel notions of relevance and then recombines them.

1.7 The Transformer Block

A transformer layer is more complicated than attention. It alternates two kinds of operations: token mixing and feature mixing. The standard block may be written as

$$H = X + \mathrm{MHA}(\mathrm{LN}(X))$$
$$Y = H + \mathrm{FFN}(\mathrm{LN}(H))$$

where LN denotes layer normalization and FFN denotes a feed-forward network applied independently to each token. The feed-forward network has the form

$$\mathrm{FFN}(h) = W_2\sigma(W_1 h + b_1) + b_2$$

In this architecture, the role of attention is to mix information *across tokens*, and the role of the feed-forward network is to mix information *within the coordinates of a single token vector*. Intuition: attention decides what context to gather; the feed-forward network decides how to transform the gathered context into more useful nonlinear features.

The residual connections are also very important; these are the same types of residual connections we have covered earlier in the course. They ensure that each block only needs to learn a correction to the current representation rather than a completely new representation from scratch. Recall that this helps optimization and stabilizes deep networks.

1.8 Layer Normalization

Layer normalization is applied separately to each token vector. If

$$h_i \in \mathbb{R}^d$$

is the feature vector for token i, then its mean and variance across coordinates are

$$\mu_i = \frac{1}{d}\sum_{k=1}^{d} h_{ik}, \qquad \sigma_i^2 = \frac{1}{d}\sum_{k=1}^{d}(h_{ik} - \mu_i)^2$$

Layer normalization produces

$$\mathrm{LN}(h_i) = \gamma \odot \frac{h_i - \mu_i \mathbf{1}}{\sqrt{\sigma_i^2 + \varepsilon}} + \beta$$

where $\gamma, \beta \in \mathbb{R}^d$ are learned parameters, and its purpose is to keep feature scales under control. Without normalization, some coordinates may become much larger than others, and this may cause unstable optimization.

1.9 Input Embeddings and Position

Transformers operate on vectors, but in daily life, we think about language as discrete symbols (like words). Therefore, we need to get from these discrete symbols to vectors. If the vocabulary size is V and the embedding dimension is d, then an embedding table

$$E \in \mathbb{R}^{V \times d}$$

assigns each token ID a vector. If x_t is the token at position t, its embedding is

$$e(x_t) = E[x_t]$$

However, self-attention alone is permutation-invariant, i.e., if we reorder the rows of X, then the mechanism has no built-in sense of which token came first. Since word order matters, we must inject positional information. We often add a position vector p_t to the token embedding:

$$h_t^{(0)} = e(x_t) + p_t$$

One classical choice for the positional encoding is called the sinusoidal positional encoding.

$$p_t[2k] = \sin\left(\frac{t}{10000^{2k/d}}\right), \qquad p_t[2k+1] = \cos\left(\frac{t}{10000^{2k/d}}\right)$$

The sinusoidal features give each position a distinct geometric signature across many frequencies. Nearby positions have related encodings while distant positions remain distinguishable. This lets the network reason about relative offsets.

1.10 Summary

It is helpful to summarize the geometry of a transformer block in one sentence:

> Each layer first lets every token move toward a context-dependent weighted average of other token features, and then applies a nonlinear coordinate transformation to each token individually.

Repeated many times over the course of a model, this process turns crude local token embeddings into rich contextual representations.

1.11 Exercises

Exercise 1. Show that every row of the attention matrix A sums to 1.

Solution: By definition,

$$A_{ij} = \frac{e^{S_{ij}}}{\sum_{\ell=1}^{n} e^{S_{i\ell}}}.$$

Summing over j gives

$$\sum_{j=1}^{n} A_{ij} = \frac{\sum_{j=1}^{n} e^{S_{ij}}}{\sum_{\ell=1}^{n} e^{S_{i\ell}}} = 1.$$

Thus each row is a probability distribution over keys.

Exercise 2. Suppose all key vectors are identical. What does attention do?

Solution: If all keys are identical, then for a fixed query q_i, the scores $q_i^\top k_j$ are the same for all j. Therefore the softmax over the row is uniform:

$$A_{ij} = \frac{1}{n}.$$

Hence

$$z_i = \sum_{j=1}^{n} \frac{1}{n} v_j = \frac{1}{n} \sum_{j=1}^{n} v_j.$$

So token i simply averages all value vectors.

Exercise 3. Why can two different heads in the same layer specialize in different linguistic tasks?

Solution: Each head has its own projection matrices $W_Q^{(h)}, W_K^{(h)}, W_V^{(h)}$, so each head defines its own similarity geometry and its own value space. One head can therefore learn to compare tokens in a way that detects short-range syntactic patterns, while another learns a geometry suited to longer-range semantic dependencies such as coreference.

2 Autoencoders and Representation Learning

2.1 Goal

A major theme in machine learning is that raw input coordinates are often not the right coordinates for a task. Two images of the same digit may be far apart in pixel space, while images of different digits may be close in some regions. What we would like instead is a feature map

$$x \mapsto z = \phi(x)$$

that organizes the data in a more useful geometry.

A good representation space should have several properties. First, similar inputs should map to nearby points. Second, semantically meaningful classes should become easier to separate. Third, the learned features should transfer to downstream tasks. A central idea in unsupervised learning is that such representations can be learned even without labels, by exploiting structure in the inputs themselves.

2.2 Basic Setup for Autoencoders

An autoencoder is a fundamental architecture for representation learning. It consists of an encoder

$$f_\theta : \mathbb{R}^D \to \mathbb{R}^m$$

and a decoder

$$g_\phi : \mathbb{R}^m \to \mathbb{R}^D$$

Given an input $x \in \mathbb{R}^D$, the encoder produces a latent code

$$z = f_\theta(x)$$

and the decoder reconstructs the input

$$\hat{x} = g_\phi(z)$$

The standard training objective is the reconstruction loss

$$\mathcal{L}(\theta, \phi) = \frac{1}{N} \sum_{i=1}^{N} \|x^{(i)} - g_\phi(f_\theta(x^{(i)}))\|_2^2$$

The model is therefore asked to compress the input into a latent vector and then decompress it as accurately as possible.

2.3 Geometric Intuition for Autoencoders

Autoencoders and transformers have substantially different geometries. A transformer updates many token vectors by allowing them to interact. In contrast, an autoencoder takes a single input point and tries to map it through a narrow latent space before reconstructing it.

If the latent dimension m is smaller than the input dimension D, then the encoder cannot preserve every direction in the original space. By definition, it must instead discover a lower-dimensional structure on which the data approximately lies. Geometrically, the encoder tries to flatten the data cloud onto a latent manifold, and the decoder tries to inflate that latent manifold back into the input space. If the data actually has a low-dimensional structure, then this can be very effective. For example, a family of handwritten digits may occupy only a small subset of all possible pixel configurations, so a low-dimensional latent code can capture the essential degrees of freedom: stroke thickness, slant, curvature, and so on.

2.4 Importance of the Bottleneck

The most important design feature of an autoencoder is the bottleneck. If the latent vector z has a dimension much smaller than the input, then exact memorization becomes impossible and the model is forced to keep only the most informative aspects of the input. This is the source of representation learning. A low-dimensional bottleneck encourages the model to discard noise and redundancies and to preserve only the structure needed for good reconstruction. However, if the latent dimension is too large and the encoder-decoder pair is very expressive, the model may just learn the identity map. In that case, reconstruction can be excellent while the representation itself is uninteresting.

2.5 A Linear Autoencoder and PCA

Consider the following derivation. Suppose both encoder and decoder are linear:

$$z = Wx, \qquad \hat{x} = Uz,$$

so that

$$\hat{x} = UWx.$$

Assume further that the latent dimension is $m < D$ and that we minimize squared reconstruction error over centered data. Then the optimal rank-m linear autoencoder spans the same subspace as the top m principal components of the data. In other words, a linear autoencoder with MSE loss recovers PCA. This is geometrically natural: PCA finds the m-dimensional subspace onto which orthogonal projection minimizes reconstruction error, and the linear autoencoder learns the same best low-rank approximation. Thus, nonlinear autoencoders may be viewed as a nonlinear generalization of PCA: instead of learning a best linear subspace, they learn a nonlinear latent coordinate system adapted to the data manifold.

2.6 Sparse Autoencoders

Beyond using low dimensionality, we could constrain the latent code by allowing a larger latent dimension and encouraging most coordinates of z to be zero. This leads to the sparse autoencoder objective

$$\mathcal{L}(\theta, \phi) = \frac{1}{N} \sum_{i=1}^{N} \|x^{(i)} - g_\phi(f_\theta(x^{(i)}))\|_2^2 + \lambda \|f_\theta(x^{(i)})\|_1$$

The ℓ_1 penalty promotes sparsity, so each example activates only a small subset of latent features. Geometrically, the model represents each data point using a small collection of basis directions, which often leads to more interpretable or more selective features.

2.7 Denoising Autoencoders

A standard autoencoder learns to reconstruct the input from itself. But if the model is powerful, this may still encourage copying too much irrelevant detail. A denoising autoencoder changes the underlying task: instead of feeding in the original input, we first corrupt it to obtain $\tilde{x}$ and then ask the model to reconstruct the clean input x:

$$z = f_\theta(\tilde{x}), \qquad \hat{x} = g_\phi(z)$$

The loss becomes

$$\mathcal{L}(\theta, \phi) = \frac{1}{N} \sum_{i=1}^{N} \|x^{(i)} - g_\phi(f_\theta(\tilde{x}^{(i)}))\|_2^2.$$

Importantly, this changes the geometry of the learned map. The encoder can no longer rely on fragile pixel-level detail, because such detail may be corrupted. Instead, it must map nearby noisy variants of the same input toward a stable latent representation.

This makes the latent space smoother and more robust. We can think of the denoising autoencoder as learning to contract neighborhoods of noisy observations toward the underlying data manifold.

2.8 Benefits and Drawbacks

Autoencoders encourage compression, smoothness, and structure discovery, which we generally consider to be valuable properties. However, reconstruction is not always the right objective for downstream tasks.

For example, if the task is image classification, we may not really want to reconstruct exact grass texture or lighting variations, even though those details could help reduce the pixel-level error. An autoencoder may therefore devote some of its representational capacity to features that matter for reconstruction but not for classification.

This limitation motivates later self-supervised methods in which the objective is chosen to focus more directly on semantic information.

2.9 Summary

We emphasize that a transformer and an autoencoder learn representations in very different ways.

A transformer helps us represent sequences by repeatedly mixing token information following some learned contextual relevance.

An autoencoder helps us represent points by forcing the data through a compressed latent space and then reconstructing the original input.

Therefore, we can say that the transformer's geometry is about *contextual interaction* while the autoencoder's geometry is about *compression onto a lower-dimensional structure*.

2.10 Exercises

Exercise 4. Suppose the encoder and decoder are both linear and the latent dimension is $m = 1$. What kind of geometric object is the model trying to learn?

Solution: A linear encoder-decoder with one-dimensional latent space tries to learn a single line through the origin onto which the data can be projected and then reconstructed. This is exactly the top principal component direction in PCA.

Exercise 5. Why does a very wide latent space make representation learning less meaningful?

Solution: With a wide latent space and expressive networks, the model can approximate the identity map. Then the latent code need not compress or abstract the data; it can simply preserve almost all information directly, so the learned representation may contain little useful structure beyond memorization.

Exercise 6. Why does denoising encourage smoothness in the latent representation?

Solution: Different corrupted versions of the same clean input must all reconstruct to the same target x so the encoder is encouraged to map those nearby noisy variants to similar latent codes. This makes the latent representation stable under small perturbations and hence smoother.

3 Decision Trees

3.1 Motivation and Setup

Decision trees are a class of non-parametric supervised learning models that represent predictions via a sequence of hierarchical decisions. They are particularly well-suited for *tabular data* with:

- heterogeneous feature types (numeric + categorical),
- nonlinear relationships,
- feature interactions.

This is because trees naturally encode thresholding behavior and interactions without feature engineering, as we will soon see.

Key idea: Learn a function by recursively partitioning the input space into regions and assigning a simple prediction within each region.

3.2 Definition of a Decision Tree

We begin by discussing the basic structure of a decision tree. It consists of:

- **Internal nodes:** tests on features (i.e., $x_j \leq t$ for numeric features, or $x_j \in S$ for categorical features)
- **Edges:** outcomes of tests (e.g., yes/no)
- **Leaves:** predictions, i.e., a scalar $\hat{y}$ if the task is regression, or a distribution $\hat{p}(y \mid x)$ on classes if the task is classification (possibly a class label itself directly)

Prediction rule: To evaluate $\hat{f}(x)$, follow a single path from root to leaf (so that the inference cost is proportional to tree depth).

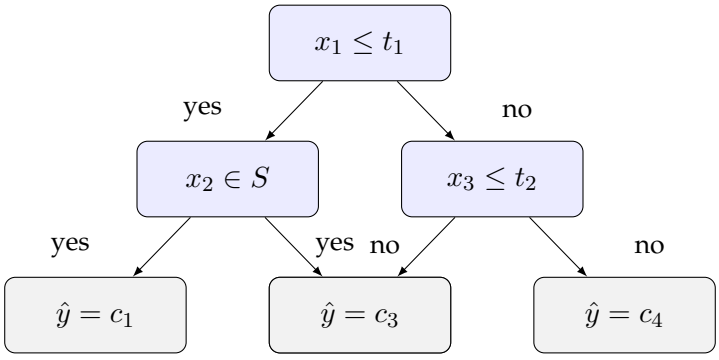


Figure 1: Example of a decision tree. Here we assume $x_1, \ldots, x_d$ are the features, and y is the target.

Trees as Feature Space Partitions. A decision tree induces a partition of the input space into regions $\{R_m\}_{m=1}^M$, where each leaf corresponds to a region defined by a conjunction of splits. Importantly, the prediction is *piecewise constant* on the regions, i.e., the tree predicts $\hat{f}(x)$ as

$$\hat{f}(x) = c_m \quad \text{if } x \in R_m,$$

if the task is regression, or similarly

$$\hat{p}(y = k \mid x) = \pi_{m,k} \quad \text{if } x \in R_m$$

if the task is classification.

3.3 Learning Decision Trees: CART Algorithm

So, we next discuss how we learn decision trees. We will focus on one algorithm, CART, and for now we will develop our understanding more conceptually. At the end of this (sub)section we will discuss briefly on implementation details (which are nontrivial and sometimes requires non-obvious decisions).

CART: Greedy Recursive Partitioning. The standard algorithm is *CART* (Classification and Regression Trees) training, which is a greedy and recursive algorithm. The algorithm is as follows:

1. At a node, consider candidate splits.
2. Choose the split that maximizes improvement in fit, e.g., that best minimizes some impurity metric.
3. Recurse on left and right subsets.
4. Stop when a stopping condition is met.

Importantly, this is a greedy algorithm, which is a good heuristic but not in general globally optimal. Next, we discuss further how this CART algorithm looks for both classification trees and regression trees.

Classification trees: purity metrics, and split criterion on impurity reduction. As we saw above, greedy approaches for learning decision trees requires some metrics to optimize at each recursive iteration. For the case of regression, we might consider minimizing an MSE. For the case of classification, we consider *purity metrics*. Formally, let $\mathbf{p} = (p_1, \ldots, p_K)$ be class proportions at a node. Then, two common impurity measures are:

- Entropy, which may be intuited as the uncertainty in label:

$$H(p) = -\sum_{k=1}^{K} p_k \log p_k.$$

- Gini coefficient, which may be intuited as the probability of misclassification under random labeling sampled according to $\mathbf{p}$:

$$G(p) = 1 - \sum_{k=1}^{K} p_k^2.$$

In both cases, these are 0 for pure nodes and maximized when classes are uniform (which means that these purity metrics are to be minimized). Using these metrics, we define the impurity reduction for a candidate split s producing left and right nodes as

$$I_{\text{after}}(s) = \frac{n_L}{n} I\left(p_L\right) + \frac{n_R}{n} I\left(p_R\right),$$
$$\Delta I(s) = I(\mathbf{p}) - I_{\text{after}}\left(s\right),$$

where here $\mathbf{p}$ was the initial class, proportions, n was the number of samples at the node before the split, and I is the impurity metric of our choice. (In the case of entropy, the $\Delta I(s)$ is called information gain.)

Now, as specified in the CART algorithm, we will choose the split maximizing $\Delta I(s)$; exactly how do we find what is this optimal split requires some engineering that we will briefly discuss later, but the key idea is just that we prefer splits that make children more pure.

Regression trees. For regression, we minimize squared error within each region. In particular, at a node R, the optimal prediction is the mean

$$c^* = \frac{1}{|R|} \sum_{i:x_i \in R} y_i,$$

and so split quality is measured by SSE (sum of squared error) reduction

$$\Delta\text{SSE} = \text{SSE}_{\text{parent}} - \left(\text{SSE}_L + \text{SSE}_R\right).$$

If you have not seen SSE before, not that it is equivalently the MSE except that we no longer take the mean over the number of samples.

Practicalities: efficient search splitting. In the above discussions, we have omitted some fine-grained implementation details. For example, even if we know what is the optimization criterion, how actually do we determine the optimal split at each iteration? For numeric features, we might:

- Sort data by feature x_j.
- Only consider thresholds between consecutive values.
- Sweep through thresholds, updating statistics incrementally.

For categorical features, exhaustive search over subsets is exponential, so practical implementations use heuristics which we will not discuss here.

Exercise: deriving a small decision tree. You are given a small dataset of a runner's habits based on the weather. Your goal is to build a decision tree to predict if the runner will go out (Yes) or stay in (No) based on two features: Outlook and Humidity.

Day	Outlook	Humidity	Ran? (Target)
D1	Sunny	High	No
D2	Sunny	High	No
D3	Overcast	High	Yes
D4	Rain	Normal	Yes
D5	Rain	Normal	Yes
D6	Overcast	Normal	Yes

Please do the following.

1. Calculate the Entropy of the entire starting dataset.
2. Determine which feature (Outlook or Humidity) provides the highest Information Gain to serve as the root node.
3. Draw the final decision tree.

Solution: Let S denote the full dataset. There are 6 examples total:

$$|S| = 6, \qquad |\{\text{Yes}\}| = 4, \qquad |\{\text{No}\}| = 2.$$

Thus

$$p_{\text{Yes}} = \frac{4}{6} = \frac{2}{3}, \qquad p_{\text{No}} = \frac{2}{6} = \frac{1}{3}.$$

1. The entropy of the data is simply

$$H(S) = -\sum_{c \in \{\text{Yes,No}\}} p_c \log_2 p_c = -\frac{2}{3}\log_2\left(\frac{2}{3}\right) - \frac{1}{3}\log_2\left(\frac{1}{3}\right) \approx 0.918.$$

2. Next, we compute the information gain for each feature. Recall that for a split on feature X,

$$\Delta\,\mathrm{I}(S, X) = H(S) - \sum_{v \in \mathcal{V}(X)} \frac{|S_v|}{|S|} H(S_v),$$

where S_v is the subset with feature value v.

So, the split on Outlook yields the three branches

$$S_{\text{Sunny}} = \{D1, D2\}, \qquad S_{\text{Overcast}} = \{D3, D6\}, \qquad S_{\text{Rain}} = \{D4, D5\},$$

with class labels

- Sunny: (No, No) so
$$H(S_{\text{Sunny}}) = 0.$$
- Overcast: (Yes, Yes) so
$$H(S_{\text{Overcast}}) = 0.$$
- Rain: (Yes, Yes) so
$$H(S_{\text{Rain}}) = 0.$$

Therefore the weighted entropy after splitting on Outlook is

$$\frac{2}{6} \cdot 0 + \frac{2}{6} \cdot 0 + \frac{2}{6} \cdot 0 = 0.$$

Hence

$$\mathrm{IG}(S, \text{Outlook}) = H(S) - 0 = 0.918.$$

Similarly, the split on Humidity yields two branches

$$S_{\text{High}} = \{D1, D2, D3\}, \qquad S_{\text{Normal}} = \{D4, D5, D6\},$$

with class labels

- High: (No, No, Yes), so

$$H(S_{\text{High}}) = -\frac{1}{3}\log_2\left(\frac{1}{3}\right) - \frac{2}{3}\log_2\left(\frac{2}{3}\right) \approx 0.918.$$

- Normal: (Yes, Yes, Yes), so

$$H(S_{\text{Normal}}) = 0.$$

Therefore the weighted entropy after splitting on Humidity is

$$\frac{3}{6} \cdot 0.918 + \frac{3}{6} \cdot 0 = 0.459.$$

So

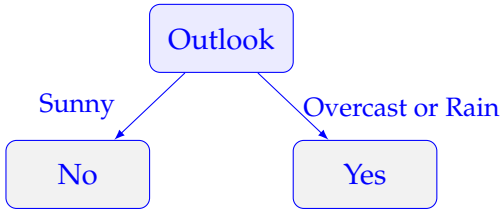
$$\Delta\,\mathrm{G}(S, \text{Humidity}) = 0.918 - 0.459 = 0.459.$$

Comparing the information gains, we see that the best root node is $\boxed{\text{OUTLOOK}}$.

3. Since each Outlook branch is already pure, the tree stops immediately after the root split:

$$\text{Sunny} \to \text{No}, \qquad \text{Overcast} \to \text{Yes}, \qquad \text{Rain} \to \text{Yes}.$$

As a tree diagram this is



Equivalently, the learned decision rule is

$$\hat{y} = \begin{cases} \text{No}, & \text{if Outlook = Sunny}, \\ \text{Yes}, & \text{if Outlook = Overcast}, \\ \text{Yes}, & \text{if Outlook = Rain}. \end{cases}$$

3.4 Complexity in Tree Models.

Unlike parametric models with a fixed number of parameters, trees are non-parametric and have data-dependent capacity. In particular, without constraints (on the training), trees can grow until leaves are 100% pure or accurate (i.e., overfitting to training error 0). In general, more splits yields more regions, which means lower bias but also higher variance. So, the key takeaway in this (sub)section is that trees require explicit complexity control to limit the number and shape of regions $\{R_m\}_{m=1}^{M}$.

Stopping criteria. One way to implement control is specifying some explicit *stopping criteria*, which acts as regularization. We might set limits on

- maximum depth D;
- minimum samples per split;
- minimum samples per leaf;
- minimum impurity decrease ΔI_{min}; and
- maximum number of leaves.

Cost-complexity pruning. Another way we can implement control is *cost-complexity pruning*, where we grow a large tree and then prune it. We try to minimize over trees as

$$\min_{T} \mathcal{L}(T) + \alpha|T|,$$

where for a tree T we let $|T|$ be the number of leaves, and α is a complexity penalty parameter. The intuition is that we should prune splits that don't significantly improve validation or generalization; the advantages of pruning are of course that we reduce overfitting and variance, and also that we simplify rules (possibly helping interpretability). In practice we pick α by validation/cross-validation, and often we pre-prune (i.e., limit tree growth while building it, rather than growing a full tree and pruning it afterward).

Remarking on the interpretability of trees. We note that each leaf in a tree corresponds to a human-readable rule, which is nice for interpretability and naturally captures feature interactions via nested splits. Of course, the limitation is that deep trees are hard to interpret, and that trees can be unstable (in that relatively small data changes sometimes leads to different trees). The practical compromise is that we use shallow trees for interpretability, and ensembles for accuracy.

3.5 Ensembling Trees

Especially given that trees can be unstable, we naturally may wish to combine multiple models to improve accuracy. In general, we might obtain B models and make predictions as, say,

$$\hat{f}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}_b(x).$$

(In regression we might average predictions, and in classification we might take the majority vote.) The main question is how we obtain the B models to ensemble together. There are two main recipes (used for trees, but also used for other types of predictors):

- bagging, where we train many models in parallel on resampled data; and
- boosting, where we train sequentially to correct mistakes.

Bagging and random forests. In bagging, we do the following:

1. Sample bootstrap datasets, each consisting of M points sampled *with replacement* from the training data.
2. Train a tree on each.
3. To run inference on new data points, average predictions from all the trees.

The effect is that we reduce the variance of our predictions, since trees can be unstable as aforementioned.

We can further extend the bagging model into the random forest model, by adding randomness at split time: in particular, at each node, we choose random subset of features of size m and search splits only among those features. The goal is to de-correlate trees, which is desirable since if all trees make the same errors, then averaging will not help. The typical rule of thumb is $m \approx \sqrt{d}$ for classification and $m \approx d/3$ for regression, where d is the total number of features. In practice, RF is often a strong and stable baseline on tabular data.

Boosting. In boosting, we instead refine our model iteratively to gradually decrease bias. That is, we fit the model in stages $\hat{f}_0, \ldots, \hat{f}_T$ iteratively, where at each stage t we define the residuals for each data point as

$$r_i^{(t)} = y_i - \hat{f}_{t-1}(\mathbf{x}_i),$$

train an h_t to predict the residuals by

$$h_t \approx \arg\min_h \sum_{i=1}^{n} (r_i^{(t)} - h(\mathbf{x}_i))^2,$$

and define the refined model

$$\hat{f}_t(\mathbf{x}) = \hat{f}_{t-1}(\mathbf{x}) - \eta h_t(\mathbf{x}).$$

Here η is a shrinkage parameter, where small η requires more trees but can generalize better.

Overall, the key comparison between these two approaches is that:

- bagging and random forests average over learners to decrease variance (more robust, low-tuning, less variance reduction); while
- boosting develops a strong learner by iterative refining the model using weak learners, often having higher peak accuracy since we keep on decreasing bias, but also requiring more tuning sensitivity since boosting models can increase variance if overfit. Regularization approaches to control this include adjusting η and the number of trees in the boosted model, as well as the usual methods to control model complexity in decision trees.