PRELAB 3

- We will cover 3 core methods used in supervised learning:
  1. k-nearest neighbors
  2. Decision Trees
  3. Logistic Regression

  } use scikit-learn
- We'll cover MODEL COMPLEXITY & how it can lead to OVERFITTING
  - ↳ model failure mode that causes model generalization performance to be poor. [AKA. our model adapts to the training data so well that the model doesn't generalize well]
- Learn to control OVERFITTING by adjusting our HYPERPARAMETERS
  - ↳ Algorithm's specific inputs that control how the model is built

TRAINING WITH THE GOAL OF GENERALIZATION

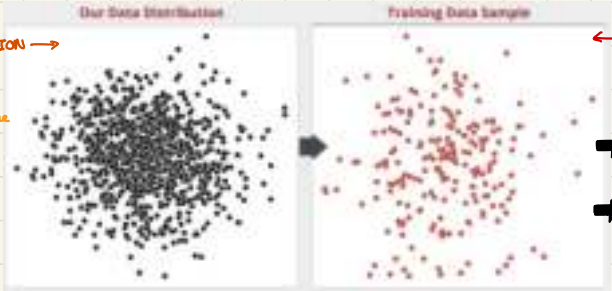
TRAINING PROCESS GOAL: create an accurate model that generalizes well to new data.

↳ GENERAL GOAL WITH SUPERVISE LEARNING:

① MINIMIZE LOSS/MEASURE HOW MANY MISTAKES YOUR MODEL MADE [OPTIMIZE THE LOSS FUNCTION]

- Loss is the way of measuring the difference between a training example's label and the label that the model predicts.
- GOAL: produce a model in which there's little to no prediction errors & ∴ zero to no loss
- The measure is often referred as TRAINING LOSS / TRAINING ERROR
- Depending on the loss function, it can be a MINIMIZING or MAXIMIZATION problem.
  - ↳ We don't want to fully optimize the loss function on the data we have or observed.
  - ↳ We want optimize the loss on data that can come from the same data generating process that created the data we have. = EXPECTED LOSS

THEORETICAL DATA DISTRIBUTION →
- can be generated from a data generating process.
  - ↳ any process that creates the data points you see.



← OBSERVED DATA SAMPLE
- Data comes from the same distribution as the left, but we assume it's finite.

TRAINING → GOOD PERFORMANCE: LOW TRAINING LOSS

OR...

FITTED MODEL PREDICTING → POOR PERFORMANCE: HIGH LOSS

- REAL GOAL: Optimize the loss on NEW DATA.

② AVOID ONE COMMON PROBLEM THAT MAY OCCUR WHEN TRAINING YOUR MODEL

- GOAL: zero to no loss

Less complex algorithms: building blocks to better understand learning algorithms
1. K-NEAREST NEIGHBORS
2. DECISION TREES
3. LOGISTIC REGRESSION

More complex algorithms: practical experience on the algorithms
1. RANDOM FOREST
2. GRADIENT BOOSTING
3. NEURAL NETWORK

- In SUPERVISED LEARNING we want to accurately predict the label in previously unseen data.
- The idea of generalization can be expressed mathematically.
- Tool we use: LOSS FUNCTION
  - ↳ mathematical expression that measures how good our models predict the labels.
  - ↳ ex. On average how often is your prediction exactly equal to the label?

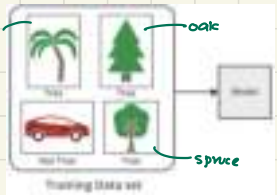
OVERFITTING & UNDERFITTING

2 failures that can occur when training your model:

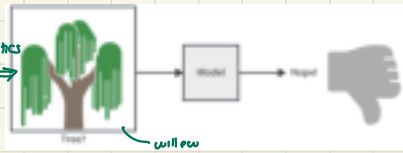
performs well on training data but generalizes poorly in new data.

① OVERFITTING: The model learns idiosyncrasies that are particular to ONLY the training dataset, it'll fail to generalize to new data.
- ↳ Model learned relationships among features and labels that are too specific to the training data and ∴ cannot be used to accurately infer anything about unseen data.
- ↳ occurs when a model is TOO COMPLEX which allows it to fit extremely well against all detailed information that is thrown at it.
- ↳ Model becomes to fixated on the training data that it may not be able to make accurate predictions on unseen data.
- ↳ Model makes decision based on patterns which exist only in the training data & not in the broader data distribution.

EX) You're training your model to recognize trees. You start by giving your model a bunch of images of various trees during training.
GOAL: model to recognize green leaves & brown trunks so that it can recognize characteristics & identify that it's a tree.

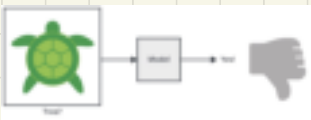


⇒ If model is OVERLY COMPLEX, it'll learn characteristics S.A. leaf shape, trunk shape, and the height-to-width ratio of these trees.



② UNDERFITTING: never performs well on data from the beginning.
↳ high training error ∴ can't generalize well to new data.
↳ Model is TOO SIMPLE
↳ Has not captured relevant details & relationships among features and labels necessary to make proper predictions.
↳ ∴ cannot make predictions on new data & will perform poorly.

EX)



TRAINING AND TEST DATA SETS

HOW TO AVOID OVERFITTING:

↳ To evaluate your model's ability to generalize well to new data: test your model on data that your model has not seen before
↳ TECHNIQUES: split your OG data into 2 partitions:
① TRAINING SET: partition to train a model
② TEST SET: partition to test the model
↳ You'll be making your test set from your initial data set, you'll have both the features & labels ∴ you have all the predicted values in advance.
↳ Test MODEL on FEATURES
↳ Compare model's PREDICTION with the LABELS

EX) CLASSIFICATION PROBLEM: you'll be able to see what fraction of the email examples were accurately classified as spam emails
EX) REGRESSION PROBLEM: see the difference between the predicted & actual values of housing prices.

PITFALLS IN USING THE TEST OVER & OVER AGAIN:

- After evaluating model's performance, you may find that you must improve your model. BUT you can't tweak your model, then train and test using the same test data set.
- Continuing this process will inadvertently change the model to do well on test data set & lead to overfit your model on the test set.
- Testing multiple times may incorrectly lead you to believe that your model will perform well on unseen data.
- Since you're testing the data set ONCE, you can use validation data set to improve your model before testing.

∴ DATA SET SHOULD INITIALLY SPLIT INTO 3 PARTITIONS:

1. TRAINING SET: partition to train a model
2. VALIDATION SET: partition to validate the model's performance.
3. TEST SET: partition to test model

- Continue this process until you improve your model to obtain accurate assessment of the model's performance.
- This will allow for unbiased evaluation of the model's performance.

EVALUATION TECHNIQUES: different ways to partition your data set, different ways to analyze your model to determine loss & accuracy during both the training & validation phase.

1. LOSS FUNCTIONS ← counts number of predictions that are correct & incorrect
2. EVALUATION METRICS ← train model on training set
   evaluate the model on the test set using simple evaluation metric
   counts number of predictions that are correct & incorrect

HYPERPARAMETERS

During the VALIDATION PHASE you can:

- Evaluate your model's performance
- Make necessary changes
- Retrain the model

In SUPERVISED LEARNING during the VALIDATION PHASE we can adjust the model's HYPERPARAMETER.

HYPERPARAMETER: a configuration that's external to the model itself.

↳ knobs that we tweak in ML algorithms. View them as the settings of an algorithm that can be adjusted to optimize performance.

↳ Choose prior to commencing training.
↳ May be adjusted during the validation phase
↳ HYPERPARAMETERS declare the mechanics of the model s.a. its complexity, and how the model is trained s.a. how fast it learns.
↳ Often:
- used to adapt a model to a particular setting
- specified by the practioner
- set using heuristics
- Tuned for a given predictive modeling problem

HYPERPARAMETER OPTIMIZATION: tuning the hyperparameters of the model to discover the model that results in the most accurate predictions

To find optimal hyperparameters you may use common approaches: ① Copy values used on other models
② Search for the best values by trial and error

EXAMPLES OF MODEL HYPERPARAMETERS:
- Size of neighborhood in KNN (K)
- Depth of tree in decision trees (max depth)
- Learning rate / step size in gradient descent

NOTE: Refer to Scikit-learn document as we'll be utilizing it

INTRODUCTION TO K-NEAREST NEIGHBORS (KNN)

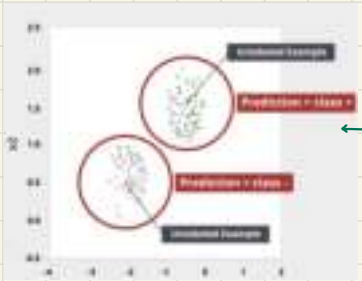
- It's an example of INSTANCE BASED LEARNING
- KNN is represented by the training data.
- Predictions are made at prediction time & involves just a query on the data training examples.

EX)

- Each point represents a different example w/ features X1 and X2
- Each example has a class label which is represented with the color
- RECALL: all learning algorithms work by attempting to find reasonable separating line between the 2 colors.
  ↳ With KNN start with the point we're predicting ← RED DOT
  ↳ When we look for the K closest points using some distance formula we choose
  ↳ Once we find the K closest points are separating boundary is essentially a circle around our point, radius = distance to the farthest point in that set of K-neighbors
  ↳ Once we have our neighborhood & set of points within it, we make a prediction by looking at the class labels of the neighbors.

← NOTICE: most of the green clusters right
some overlap between the two class

- In REGRESSION or CLASSIFICATION we're trying to predict a class label probability

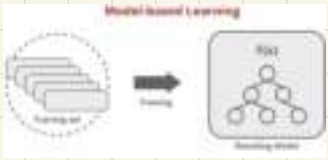


- Each example will get its own custom neighborhood and the prediction will be based only on the points in that specific neighborhood.

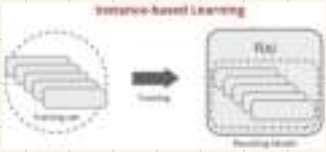
← Label values tend to be concentrated in different regions of the feature space.

- The more the labels cluster the better performing our algorithms will likely be.
- CONCEPT: the examples that are near each other in the feature space are likely to have the same label.

OVERVIEW OF KNN



MODEL BASED LEARNING: Most supervised learning models are represented by some underlying data structure derived from training examples s.a. the tree structure that's produced by the decision tree supervised learning algorithm.



INSTANCE BASED LEARNING: Stores training examples in memory and utilizes those examples in memory and utilize those examples on demand to make predictions for a new unseen example.
[NO SUCH TRAINING, make data available at prediction time]

↳ KNN → stores training in its memory
→ When new unseen examples comes in KNN looks for K most similar examples based on some distance function & makes prediction based on the labels of the examples.

EX)

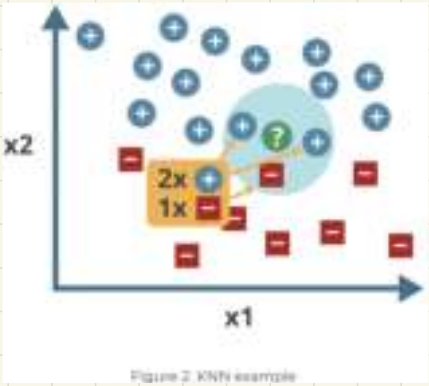


Figure 2: KNN example

- RED and BLUE represent training examples containing 2 features: x1 and x2
- GREEN represents unlabeled examples that come in

TO CLASSIFY IT:
① Choose a size for K (K = number of nearest neighbor)
② Look for the most common class label in this group

EX) 2 BLUE (most common marker)
1 RED
∴ GREEN should be classified as BLUE

- K is also known as the **HYPERPARAMETER** and represents the number of neighboring examples that will be used to make a prediction.
  ↳ Number is usually determined based on domain knowledge, heuristics, or experimentation. RECOMMENDED
  ↳ Too LOW or Too HIGH = poor performance

- concept of notion of closeness/similarity
  ↳ DISTANCE FUNCTIONS:
  - EUCLIDEAN DISTANCE
  - MANHATTAN DISTANCE
  - MAHALANOBIS DISTANCE

  ↳ In our EX) we used EUCLIDEAN DISTANCE as our similarity measure.

DISTANCE FUNCTIONS

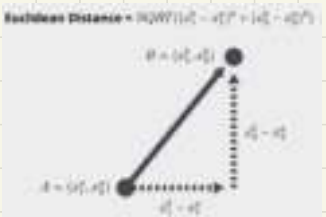
Most important mathamatical elements are used to evaluate the idea of nearness.

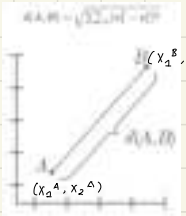
DISTANCE FUNCTION: defined between 2 points:

- ↳ Easy to illustrate 2 dimensions.
- ↳ **DIMENSION:** number of features in your data
- ↳ **DISTANCE:** length of a straight line connecting 2 points.

DISTANCE FUNCTIONS:

① **EUCLIDEAN DISTANCE:** Straighest distance between two points





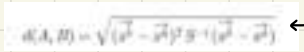
- Formula is generalized to multiple dimensions
- x_i^a is the i-th feature of a given example A
- x_i^b is the i-th feature of another example B you're comparing against

- Euclidean doesn't account covariance and scale among features.

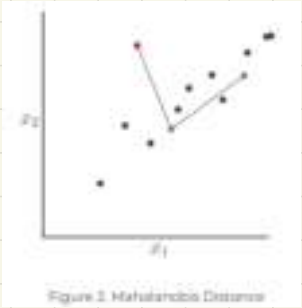
⇓

If you want to: MAHALANOBIS DISTANCE

② MAHALANOBIS DISTANCE: improved version of Euclidean & it accounts for correlation among features and scale.



← S= covariance matrix
↳ if S= identity matrix then there's no correlation among any of the features ⇒ reduced to Euclidean distance



← Ex) Even though the Euclidean distance between red/blue & green/blue are the same, green is considered closer to blue in Mahalanobis b/c it lies along the first principal component of the dataset.

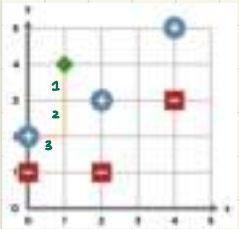
FIRST PRINCIPAL COMPONENT = direction of the highest covariance (AKA. direction where data is most stretched out).

③ **MANHATTAN DISTANCE:** Aka. taxicab metrics due to its likeliness to the distance traveled in the city with the grid layout

PUZZLE: FIND A DECISION BOUNDARY

- KNN decision boundary is influenced by the number of nearest neighbors & position of points from each class.
  - ↳ In case of binary classification using 1NN the decision boundary may look like this:

- Choice of DISTANCE MATRIX also influences the determination of the decision boundary.
  - ↳ using Euclidean, the shortest distance between 2 points is along the line that connect them.
  - ↳ using taxicab/manhattan distance you can only travel vertical & horizontal axis
    - Ex) Determine taxicab distance between (1,4) and positive (0,2)

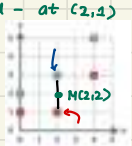


∴ Total distance = 3

DETERMINING BOUNDARY BY HAND

Finding the boundary for 1NN by hand is to find 2 points of opposite class that you suspect are near the boundary.

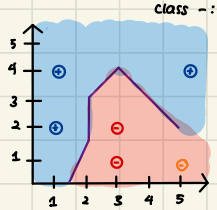
Ex) The + at (2,3) and − at (2,1)
∴ midpoint is (2,2)



ACTIVITY INSTRUCTIONS

Suppose you have: class +: (1,2), (1,4), (5,4)
class −: (3,1), (3,2)

You have to draw decision boundary & you chose 1NN & use EUD metric. You're confined to a [0,5]x[0,5] grid.



Q1: Find the decision boundary using 1NN.
↳ Try and plot the distance.

Q2: How would you test data point (5,1) be classified, given your decision boundary?
↳ New point would be classified as −1

KNN OPTIMIZATION

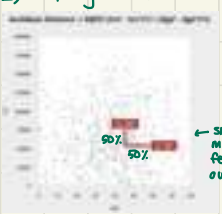
Every supervised learning algorithm has input parameters AKA. MODEL HYPERPARAMETER
↳ controls the complexity of the model
↳ Defines how flexible the model is in terms of its ability to fit small variations in the data meaning it

HYPERPARAMETER OPTIMIZATION: process of finding the best complexity that enables us to generalize the best.
↳ Different complexity parameters:

① Neighbor Count (K): The # of nearest neighborhors to use in prediction
↳ ADVISE: always test out different options for K and to find a value that's best for your data.

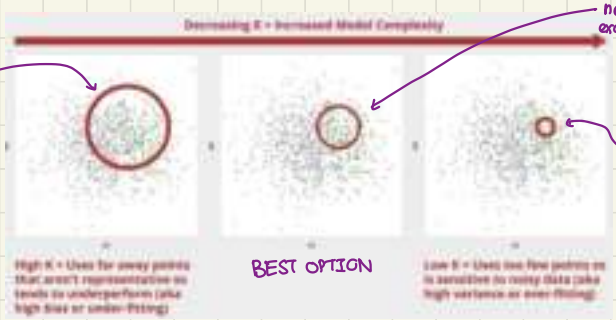
② Distance function: the functional form of the distance metric & the weights used on each nearest neighbor.

③ Normalization: the process to scale each feature so that they have similar magnitude
↳ Preprocessing step ∴ won't specify this when calling the scikit-learn KNN class
↳ Generally improves performance with this algorithm
↳ Ex) Computing distance between 2 points: Average age = 40 Average income = 50,000



← size of arrow is misleading as each feature is on their own scales.

- In EUD these changes are considered equivalent.
  ∴ features with highest scales ⟹ receive higher implicit weight in KNN distance functions will be more sensitive to these features. ⟸ undesired



circle is so wide that we're mixing in too much of the different classes

Leads to UNDERFITTING

nearby points are represented of the example being predicted

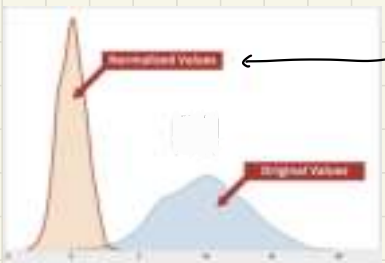
large enough to be not too sensitive to just a few points.

points are more likely to have the same labels

BUT prediction relies heavily on a few points.

Leads to OVERFITTING

HOW NORMALIZATION AFFECTS THE SHAPE OF DISTRIBUTION:



new distribution is shifted ∴ mean = 0
lower variance (depicted by width of bell curves)

When we normalize we shift & rescale on each features so that the distributions look like the orange plot; no matter its starting point.

This process won't hurt predictive performance b/c the normalized values still have the same correlation with the label.

TECHNIQUES FOR IMPROVING KNN

① HYPERPARAMETER OPTIMIZATION

- Hyperparameters are the configurable aspects of your model that can be adjusted to control model complexity and improve a model's performance.
- The KNN model has exactly 1 hyperparameter which is k: the size of the neighborhood you pick to infer the label.
- Typically choose ODD #s to avoid ties esp. with binary classification problems.

What happens if you change K?

1NN

- Picks up on little movements in the data set.
- You can see islands of misclassified & mislabeled examples.
- If you suspect your data set is nosy (aka. some points are mislabeled in my data), the 1NN might be too sensitive to accurately generalize new data.

3NN

- now you're averaging local neighborhoods & the little islands get washed out as they're outvoted by the other examples.
- complexity of boundary increases & becomes smoother with 9NN

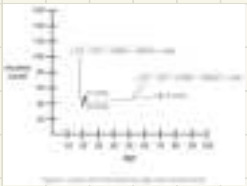
9NN

- Decision boundary becomes smoother and less susceptible to noise as you increase k.
- 9NN begins to misidentify groups of data points which will become more likely as you increase the number of neighbors used.

② NORMALIZATION TECHNIQUES:

↳ Some ML models are sensitive to the range of features (s.a. KNN)

↳ EX)



You'd typically expect red and green dots to be considered close to each other than blue & green.

However based on un-normalized values, our distance function determines that blue & green are actually closer than red & green.

↳ Normalization transforms the values of features into the same scale.
This allows distance function to give equal weight to each feature.

↳ Most common normalization technique:

① STANDARDIZATION (AKA. STANDARD SCALER)

You transform values with a feature to have a mean = 0 and standard deviation = 1.

- Works well with features that follow normal distribution & are less sensitive to outlier.

② MIN-MAX NORMALIZATION (AKA. min-max scaler)

You transform the values within a feature to be between certain min/max range.

- Min-max normalization typically works well with features that don't follow normal distribution & are more sensitive to outliers.

THE CURSE OF DIMENSIONALITY

- High dimension data contains large amount of features [# of dimensions = # of features]
- ↑ dimensions = ↑ features = datapoints (examples) become more unique & less similar = to a point of finding close neighbors to predict the level of test data is no longer feasible. = CURSE OF DIMENSIONALITY

ENHACING KNN

① RESOLVING TIES:

↳ Ex. you have k=4 you could retrieve two neighbors of one class and 2 of another.

To avoid this pick odd #s of k

HOWEVER your neighborhood vote may result in a tie.

↳ common approach to resolve ties: fall back onto the majority label within the k-2 closest neighbors.

② CHOOSING DISTANCE FUNCTIONS:

↳ How you choose your distance between 2 points is critical to the accuracy & performance of your classifier.

③ DATA STRUCTURE FOR SPEED UP:

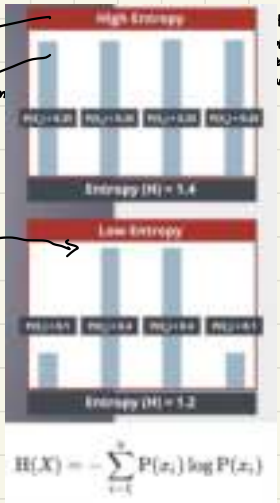
↳ Speed up KNN by using data structures s.a. k-d trees or ball trees.

INFORMATION THEORY OVERVIEW

- Decision trees has the idea of building blocks which is very important.
- Full tree is built using a recursive algorithm
- Root of decision trees are some powerful formulas that come from the field of study called INFORMATION THEORY.

THE 2 FORMULAS:

① **ENTROPY:** measure of dispersion or uncertainty of discrete random variable.



highest entropy you can reach

uniform distribution over the 4 values

more mass in middle 2 values

High entropy = high uncertainty (ie where a particular example tells us as good as random guessing.

- better off getting the middle 2 values.

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log P(x_i)$$

- negative sign means it'll be a positive number
- higher number = higher entropy

Each discrete value would have an associated probability P = bar height

Use entropy to compute information gain.

② **INFORMATION GAIN:** way to measure how much average entropy changes after we segment our data.

Entropy before we segment | Average entropy across the new segments

$$IG = H(Y|Parent) - \sum_{c \in C}^{n} p(c)H(Y|c)$$

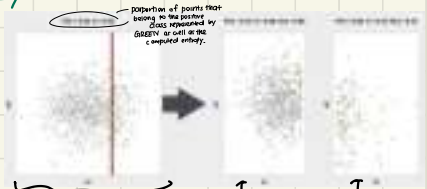
H = entropy

c = child

represents a separate segment of the data.

Let's assume we split the data into 2 or more partitions, with each partition being called a child of the OG data.

Ex) Scatter plots that are related to binary classification



FULL DATA = parent | child of parent | child of parent

Split into 2 parts to produce the plots on the right

INFORMATION THEORY FORMALIZED

Information theory is utilized in the implementation of decision tree based classifiers.

ENTROPY: measure of uncertainty in a random variable.

- value of random value is predictable = low entropy
- value of random value is unpredictable = high entropy
- Range of entropy: 0 - 1
- Mathematical formula:

$$H(y) = -\sum_{i=1}^{n} P(y_i) \log_2 P(y_i)$$

Ex) coin that has 30% chance of flipping H & 70% chance of T.

$H(y) = -(0.30 \cdot \log(0.30) + 0.70 \cdot \log(0.70)) = 0.88$

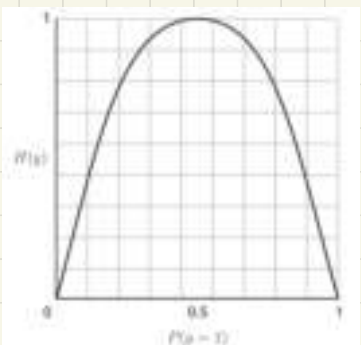
For fair coin:

$H(y) = -(0.50 \cdot \log(0.50) + 0.50 \cdot \log(0.50)) = 1$

Fair coin that always flips H:

$H(y) = -(1 \cdot \log(1)) = 0$

If you plot your observations:



To compute IG we:

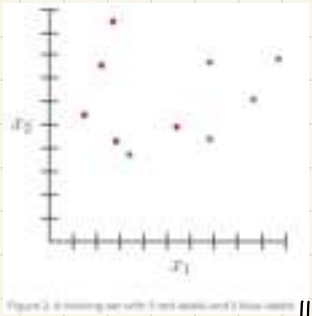
- compute entropy of each child
- Take a weighted average where weight is the proportion of points in the child.
- Difference between the parent entropy and the weighted average is IG.

IDEALLY:

- we want the average Y value in a child partition to be close to 0 or 1 as possible
  - When this happens we say the node has more PURITY.

NOTE: Increase in certainty is the increase in IG.

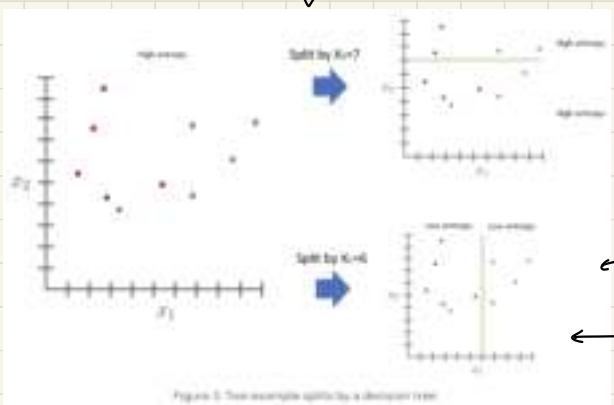
DECISION TREE INTUITION:



- Same number of BLUE & RED
- Without considering the position (feature values) of the dots, all the information we have is that ½ is RED ½ is BLUE
- With new unlabeled point is introduced, our best chance of predicting its color can no better than 50%.
- Intuition behind training a decision tree is to have an algorithm that can automatically looks at the points and subsequently grouping these points into regions where the dots are primarily RED and BLUE.

∴ reducing the overall entropy of the system.

⇓ To separate points into regions, a decision tree will draw splits based on selected feature value



OPTION 1:

where each region still has the same # of BLUE & RED dots.

OPTION 2:

most red are on left
most red on right

← BETTER OPTION as it reduces entropy by large amount

Amount of entropy reduced is the information gain.

That's what a decision tree use to determine the best feature & value to split on.

INFORMATION GAIN: Tells us how much reduction in entropy has taken place once some condition has been met.

↳ The condition in decision tree is the SPLIT.

↳ Formula for IG:

$$IG = H(y|parent) - \sum_{c \in C} p(y|c)\,H(y|c)$$

- $H(y|parent)$: entropy prior to split
- $p(y|c)$: the number of data points in a child region divided by the number of total data prior to a split.
- $H(y|c)$: entropy of the data points within a child region created by the split.

↳ Specified IG formula for binary tree:

- L = left node in binary tree
- R = right node in binary tree
- L & R = 2 child created from a split

$$IG = H(y|parent) - p(y|c_L)\,H(y|c_L) - p(y|c_R)\,H(y|c_R)$$

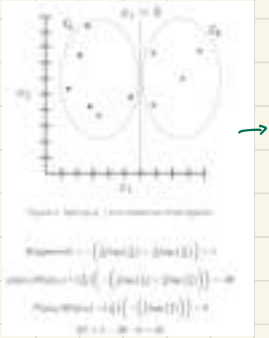
entropy of each region: L & R

$H(y|c_L)$: L region; $H(y|c_R)$: R region

$p(y|c_L)$ & $p(y|c_R)$: The number of data point in each child region divided the number of data points prior to the split.

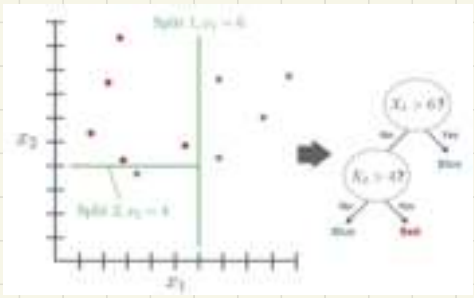
↳ Range of IG: 0-1

EX)



IG = 0.61 = fairly high #

- With new, unlabeled data, you need to look at x_1 value.
  - ↳ $x_1 > 6$ ∴ BLUE
  - ↳ $x_1 < 6$ ∴ RED



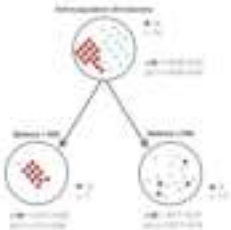
QUIZ: ENTROPY SCENARIOS

Question 1

0 / 1 pts

Calculate the entropy for the parent node and see how much uncertainty exists by splitting on "Balance".

The **blue circles** represent people from the "**payoff**" class and the **red circles** are people from the "**default**" class. Splitting the parent node on the "Balance" attribute gives us two child nodes: "< $50K" or "≥ $50K". Review the graphic below to see how the data is split.



What is the entropy of the root node if we split on "Balance"?

- 0.39

$$H(\text{Balance} < \$50K) = -\frac{12}{13}\log_2\left(\frac{12}{13}\right) - \frac{1}{13}\log_2\left(\frac{1}{13}\right) \approx 0.39$$

$$H(\text{Balance} \geq \$50K) = -\frac{4}{17}\log_2\left(\frac{4}{17}\right) - \frac{13}{17}\log_2\left(\frac{13}{17}\right) \approx 0.79$$

$$H(\text{divide on Balance}) = \frac{13}{30} \cdot 0.39 + \frac{17}{30} \cdot 0.79 = 0.62$$

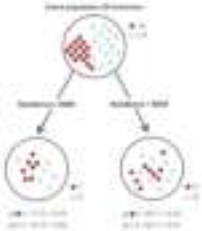
Before splitting, the entropy of the tree is 0.99. After splitting, the entropy of the tree is **0.62**. So, by splitting on "Balance" the entropy is reduced by (0.99 - 0.62) = 0.37.

Question 2

0 / 1 pts

Calculate the entropy for the parent node and see how much uncertainty exists by splitting on "Residence".

The **blue circles** represent people from the "**payoff**" class and the **red circles** are people from the "**default**" class. Splitting the tree on "Residence" gives you two child nodes: "Own" and "Rent". Review the graphic below to see how the data is split.



If we split on "Residence", what happens to the entropy of root?

$$H(\text{Residence} = \text{"Own"}) = -\frac{7}{13}\log_2\left(\frac{7}{13}\right) - \frac{6}{13}\log_2\left(\frac{6}{13}\right) = 0.99$$

$$H(\text{Residence} = \text{"Rent"}) = -\frac{9}{17}\log_2\left(\frac{9}{17}\right) - \frac{8}{17}\log_2\left(\frac{8}{17}\right) = 0.99$$

$$H(\text{divide on Residence}) = \frac{13}{30} \cdot 0.99 + \frac{17}{30} \cdot 0.79 = 0.99$$

Before splitting, the entropy of the tree is 0.99. After splitting, the entropy of the tree is still 0.99. So, by splitting on "Residence" the entropy is not reduced at all.

Question 3

0 / 1 pts

Which feature — "Balance" or "Residence" — provides the lowest entropy split?

- Residence
- Balance

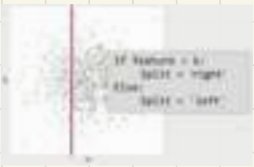
BUILDING A DECISION TREE

Fundamental steps of decision trees: split of the data that maximizes our information gain.

↳ Start with data representing a BINARY CLASSIFICATION problem with 2 features.

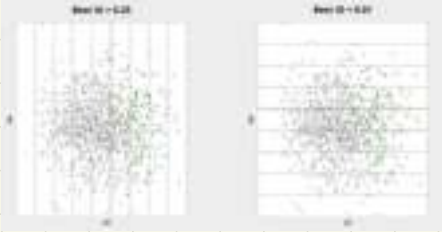
↳ GOAL: split this data into many partitions as necessary to get the best cumulative IG.

Start with the basic structure of a partition:



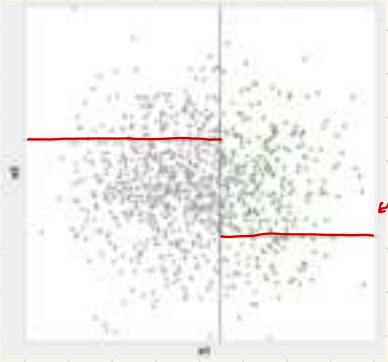
- At each step we'll need to split the data based on a single feature.
- To build a good predictive model we want both features and values of k that lead to the best possible increase in IG.

↳ BASE STEP: determine which feature to split on & what's the best split value k.



- BRUTE FORCE APPROACH: loop through each feature and a range of split values for each of the features.
  - ↳ For each feature, we'll find the split value that maximizes the information gained for that specific feature. [REPRESENTED IN RED LINE]
  - ↳ loops over both features and possible split values to find the combination that yields the highest IG.

NEXT STEP: Loop through each partitions and apply the same exact algorithm to each partition.



- First split: using feature X1
- within each new partitions, we split on feature X 2.

NOTICE: each partition splits at a different value of the feature

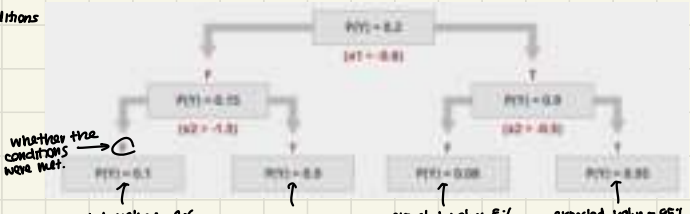
↳ REASON: Each partition steps are INDEPENDENT of each other.

Will continue this splitting process until some termination criteria is met.

↑ input parameters we given to scikit-learn when we initialize the class

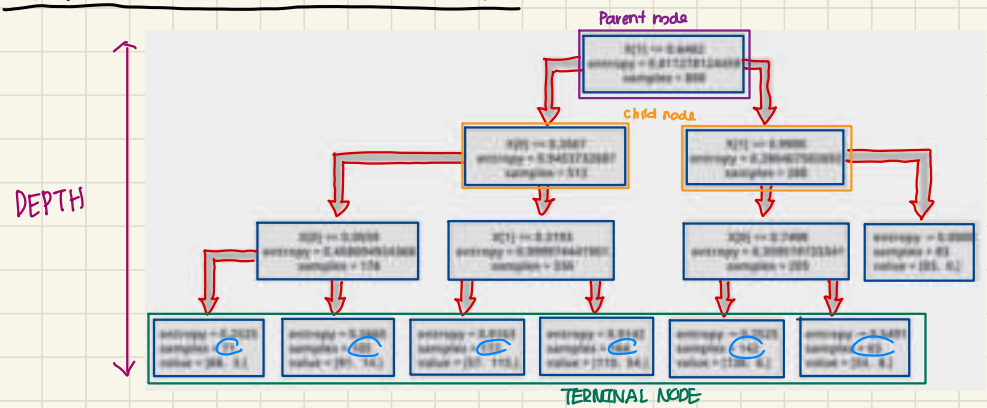
The process we illustrated so far can be represented as a tree: [DECISION TREE]

This represents the conditions we use to split.



- P(Y) = average value of the label in each node.
- When making predictions:
  - ↳ start from the top, we evaluate the new example against the splitting logic.
  - ↳ proceed down the tree until we reach a splitting node.

MAKING PREDICTIONS USING THE DECISION TREE



COMPONENTS OF THE TREE:

NODES: individual elements
DIRECTED: indicates the flow of direction
PARENT NODE: any starting node
CHILD NODE: endpoint of an arrow
TERMINAL NODE: end of the tree / leaves of the tree ← CONTAINS information we use for PREDICTION
DEPTH: how many splits there are down a single path. ← number of logical comparisons we have to make
# of nodes grow exponentially with the depth
LEAF SAMPLE SIZE: examples that make it to that leaf.
LABEL DISTRIBUTION: distribution of labels in that leaf

DIFFERENCE BTW CLASSIFICATION & REGRESSION IN THE ALGORITHM:

- Function used to define the split.
  - ↳ The distribution of the labels in the leaf node is what determines a prediction
- CLASSIFICATION:
  - ↳ We can take the most common class label as the prediction OR take the average value to get that probability
- REGRESSION:
  - ↳ Take average label within leaf node as the prediction.

DECISION TREES USE SCIKIT LEARN API

```
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier(max_depth = 10,
                               min_samples_split = 120,
                               min_samples_leaf = 64
                               )

model.fit(df[X], df[y])

class_probabilities = model.predict_proba(df[X])[:,1]
class_predictions = model.predict(df[X])
```

Tells us how large we can grow the tree

relate to the number of elements in the nodes

won't split a node unless we have that many examples in the splitting node

won't split unless each of the child node gets at least that number of examples

	Class_Prob	Class_Pred
0	0.640625	1
1	0.828571	1
2	0.793814	1
3	0.472222	0
4	0.793814	1
5	0.793814	1

for prediction

We can either predict the class: DIRECTLY or predict the probability of being in each class.

} choice will be problem dependent.

CHEAT SHEET

Decision Trees

Algorithm Name	Decision trees
Description	A framework that solves classification and regression problems.
Applicability	• Classification and regression problems • Makes no assumptions regarding feature representation and works with both continuous-valued and categorical features
Assumptions	Similar inputs have similar labels.
Underlying Mathematical Principles	Entropy and information gain are the criteria used to select features and split values at each non-leaf node.
Hyperparameters	max_depth (often alternatively maximum number of nodes), maximum samples per leaf, splitting criterion (information gain)
Additional Details	The classifier is represented by a binary tree.
Example	Predict whether an individual will default on a loan based on credit score, age, and loan amount.

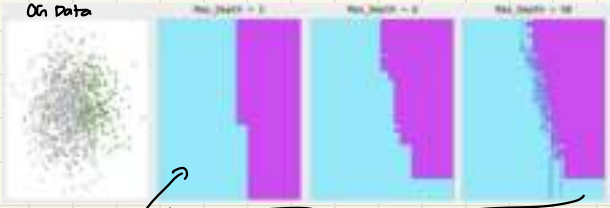
OPTIMIZING A TREE

- Decision tree's complexity is related to the size of the tree
- Each node is a separate & distinct partition of the feature space.
- Tuning the size of the tree leads to the best generalization
- Bigger tree results == more complexity == more likely to overfit == model will have model estimation variance.
- less complexity == higher chance of underfit == may have model estimation bias
- High variance: sensitive to slight variations of the data.
- Model with higher bias tends to not fit the data well.
- Right complexity is the balance of HIGH & LOW complexity.

max_depth
higher values lead to larger trees and higher complexity
→ controls how much splits we can make

min_samples_split
higher values lead to smaller trees and lower complexity
→ controls the minimum number of samples and a parent node to split it.

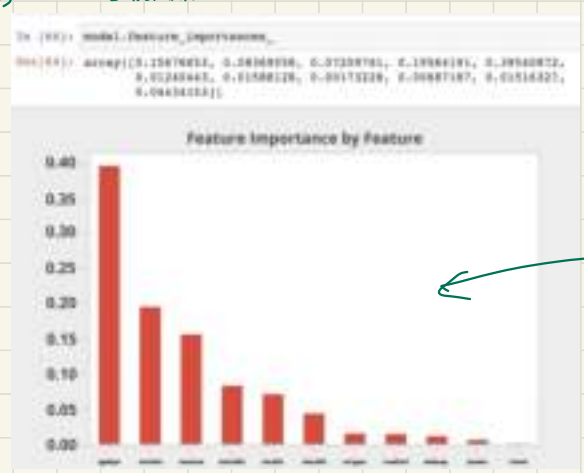
min_samples_leaf
higher values lead to smaller trees and lower complexity
→ dictates how many samples can be in the leaf node.



- Colors represent how new examples will be predicted.
- Visualization shows the typical complexity trade off.
- We want to see smooth & curved lines without many alternating colors.

- The decision tree automatically computes an attribute called FEATURE IMPORTANCE.
  ↳ cumulative IG that features contribute to the learning process.

EX) FEATURE IMPORTANCE



We can determine which feature on the right hand side have little to no predictive value.

have little to no predictive value.

MODEL COMPLEXITY REVISITED

To revisit, there are the two equally problematic cases which can arise when training a model: underfitting and overfitting. If the model isn't able to distinguish important aspects of the training data, it will fail to classify new data accurately. On the other hand, if the model learns the idiosyncrasies that are particular to only the training dataset, it will fail to generalize to new data.

Underfitting: The model is too simple. The model has not captured the relevant relationships and details among features and labels that are necessary to make proper predictions. In this case, both the training error and the test error will be high and the model will not be able to make predictions about new, unseen data.

Overfitting: The model is too complex. The model has learned relationships and details among features and labels that are too specific to the training data only. The model therefore cannot be used to accurately infer anything about new, unseen data. Although training error may be low, test error will be high, as the model can only make decisions based on patterns which exist only in the training set and not in unseen data.

An overfit model is too complex; it has captured the idiosyncrasies of the training data and will not generalize to unseen data.

An underfit model is too simple; it has not captured the predictive nuances present in the data and will perform poorly on new data.

Finding the ideal hyperparameters helps control the model complexity, which, in turn, reduces model estimation errors due to bias and variance.

Bias-variance trade-off

Finding the ideal hyperparameters helps control the model complexity, which in term reduces model estimation errors due to bias and variance.

- **Bias**: Model bias expresses the error that the model makes (how *different is the* prediction from the training data). Bias error arises when a model is too simple and is underfitting.
- **Variance**: Model variance expresses how consistent the predictions of a model are on different data. High variance is a sign that the model is too complex and is overfitting to the particular data set it is trained on.
- The ideal trade-off between bias and variance lies in finding the right hyperparameters, leading to a model with balanced complexity.

Hyperparameter tuning to manage complexity

To ensure your model is not susceptible to overfitting or underfitting, you want to tune the model's hyperparameters to some optimal values. As a refresher, hyperparameters are tune-able properties to control the behavior of a model. Most models have some kind of hyperparameter to control a model's complexity. For a decision tree, it can be the maximum allowable depth of the tree. Note in the graph below how changing the tree depth of a decision tree can affect underfitting and overfitting.

A machine learning engineer's job is to experiment with various hyperparameters values to ensure that the models developed are not overfitted or underfitted.