CSC311 Notes

Introduction to Machine Learning

https://github.com/ICPRplshelp

Last updated January 23, 2023

1 Introduction

Machine learning is when we want to teach computers to do things.

- A computer program is said to learn when performance increases with experience.

The motivation:

- Hard to specify correctness by hand (especially with images)
- The machine learning approach is to program an algorithm to learn from data or experience
- Want system to do better than human programmers
  - Specifying goals is easier than steps, such as chess. The goal is that we want to win, but it's hard to design a good program for this.
- Want to react to changing environment
  - You see, spam gets better, and machine learning helps you beat that with barely any effort
- Privacy and fairness
  - Humans find it hard to be objective.

Stats is better with making good decisions. Machine learning is (everything else) more focused on predictions and autonomy, but they rely on similar concepts.

1.1 Types

- Supervised
  - Dataset with labeled examples (inputs, given outputs) $\rightarrow$ generalize this with a new test set
- Reinforcement
  - Learning by interacting with the world to maximize a scalar reward signal

- Unsupervised
  - No labels; look for interesting patterns

All three rely on providing some sort of learning signal. Some sort of **loss** for supervised and **reward** for reinforcement.

1.2 A Bit of History

ML is a new field, and it really took off in 2010. Today, there are increasing attention to ethical and societal implications. ChatGPT is prominent and is good at answering questions as if they were Wikipedia articles.

1.3 Examples of Machine Learning

- Computer vision
  - Object detection (what are...)
  - Semantic segmentation (paint a picture in a style of ___)
  - Pose/instance estimation (creating rigs from a photo)
  - A lot more
- Speech
  - TTS and speech-to-text
- NLP
- Playing games
- Recommender systems

1.4 Implementation

Usually, you'll be asked to turn math into code. For example, in math, we have:

$$z = Wx + b$$

We have array processing software like NumPy to vectorize these computations. NumPy can parallelize operations and can help make operations way faster.

There are a lot of frameworks which can do a lot of stuff for you. Such as:

- Automatic differentiation (gradients, derivatives in higher dimensions)
  - PyTorch and TensorFlow are optimized for these

However, this course is important, as if your algorithm isn't working, you'll understand what went wrong. Was it your training data, or was it something else?

2 Nearest Neighbors

For much of the course, we'll focus on supervised learning. Hence, we have a training set consisting of inputs and labels. For example:

- If I'm asked to do object recognition
  - I am given images
  - With object category as their labels
- For image captioning
  - I am given images
  - With the caption as the label
- For document classification
  - I am given text
  - With the document category as the label

And so on.

Supervised learning is a bit costly as it does require labor to label them.

2.1 Definitions

- A label is a feature of an input/related.
- The set of class labels is the set of values our labels can take.
- $\vec{x}$? That depends on context. If we want to do some stuff with images, then $\vec{x}$ would be a vector that represents an image (mainly, a list of RGB values, and potentially its dimensions).
  - d is used for input, which all $\vec{x} \in \mathbb{R}^d$
  - Outputs will mostly be one dimension

2.2 Imaging

Computers see images as a big array of numbers. The computer's goal is to output some distribution of the classifications. For example, an image of a cat would output:

- A% cat
- B% something else...

2.3 Representing inputs

We represent inputs as an input vector $\mathbb{R}^d$.

- Vectors are great representation as we can do linear algebra.

2.4 Input Vectors

In supervised learning, there are two tasks we'll focusing on

- Regression (output $\mathbb{R}$)
- Classification (output something from a discrete set)
- In practice, we may return something more complex like an object (JSON)

Notation.

- $\vec{x}$ is something in our training set (such as an image)
- t is a label.

Our training set looks like:

$$\left\{\left(\vec{x}^{(1)}, t^{(1)}\right), \ldots, \left(\vec{x}^{(N)}, t^{(N)}\right)\right\} = D$$

The input matrix looks like this

$$\underset{\text{design/data matrix}}{X} = \underset{\underset{\text{data points}}{N} \times \underset{\text{features}}{d}}{\begin{bmatrix} - & \vec{x}^{(1)} & - \\ - & \vec{x}^{(2)} & - \\ - & \vdots & - \\ - & \vec{x}^{(N)} & - \end{bmatrix}}$$

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

We can also do this with our targets:

$$T = \begin{bmatrix} t^{(1)} \\ t^{(2)} \\ \vdots \\ t^{(N)} \end{bmatrix} \in \mathbb{R}^n$$

For nearest neighbors, we don't really need all this matrix representation… yet.

2.5 The Nearest Neighbors Algorithm

We have a new input $\vec{x}$ we want to classify

The nearest neighbors are a classification algorithm. The idea is to find then nearest input vector to $\vec{x}$ in the training set and copy its label.

We can formalize nearest in terms of Euclidean distance: the magnitude of the difference of the two vectors

$$\begin{aligned}&\left|\left|\mathbf{x}^{(a)}-\mathbf{x}^{(b)}\right|\right|_2\\&=\sqrt{\sum_{j=1}^{d}\left(x_j^{(a)}-x_j^{(b)}\right)^2}\end{aligned}$$

Algorithm. The nearest vector is:

$$\begin{aligned}&\vec{x}^*\\&=\underset{\vec{x}^{(i)}\in\text{training set}}{\text{argmin}}\quad\text{distance}\left(\vec{x}^{(i)},\vec{x}\right)\end{aligned}$$

Output $y=t^*$. This requires $\vec{x}^*$ (image in training set closest to $\vec{x}$; t^* is $\vec{x}$'s label).

For example, finding the vector in the training set CLOSEST to our input vector.

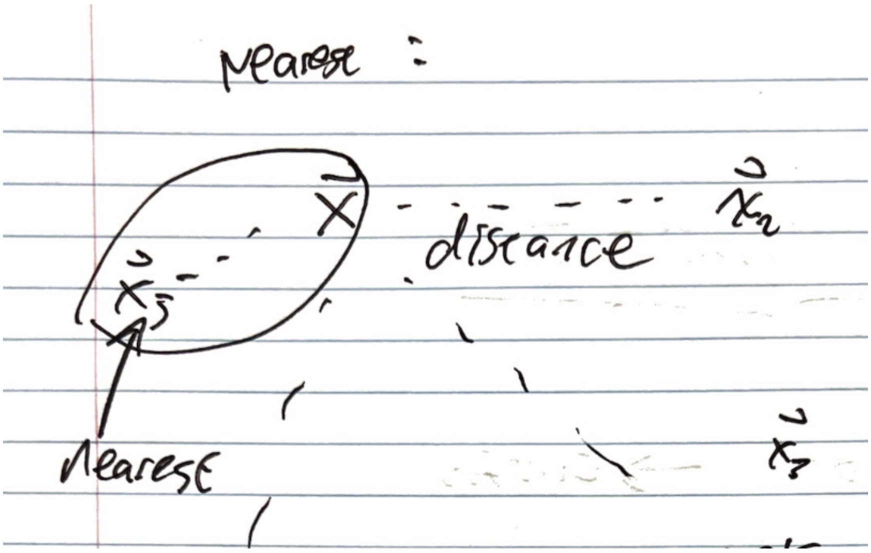

Figure 1: Visualization of the only nearest neighbour to the input vector x

2.6 Voronoi Diagrams

The decision boundaries are when nearest neighbors make a different decision (in practice, we'll never touch the boundary).

Voronoi diagrams can be in over two dimensions. It becomes a lot more difficult in higher dimensions.

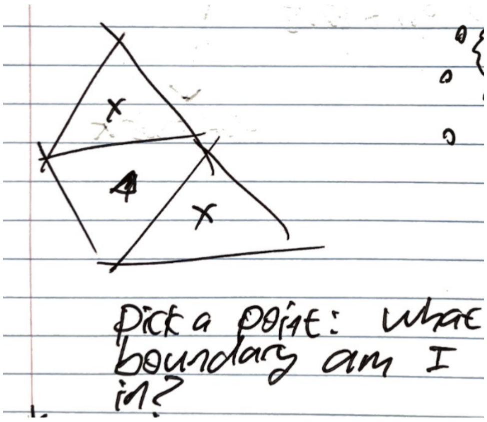

Figure 2: Voronoi Diagram with 3 datapoints and 2 distinct labels

2.7 K Nearest neighbors

Noise in the data could be a problem. In our Voronoi diagram, there might be a single point with a different label in an area filled of points with just the other label.

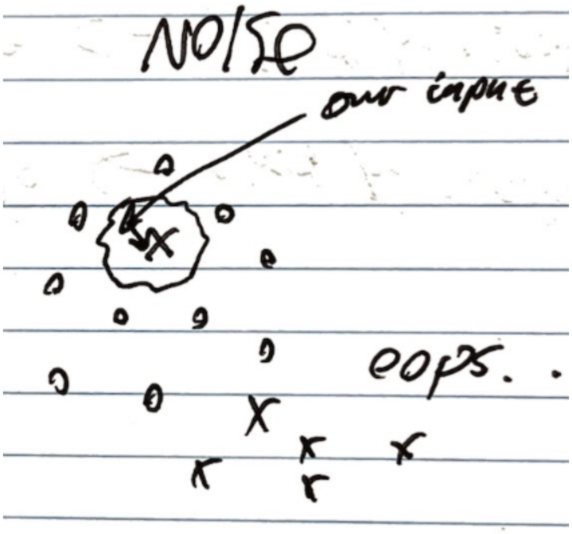

Figure 3: The problem with noise

Solution? Smooth it out by having k-nearest neighbors. Here, **we take up to k nearest neighbors instead of 1 (what we initially did).** k is odd to avoid ties; one example is we take 3 NNs.

Algorithm for kNN:

1. Find k examples $\left\{\left(\vec{x}^{(i)}, t^{(i)}\right), \ldots\right\}$ closest to the test instance $\vec{x}$
2. Classification output is majority class.

$$y^* = \underset{t^{(z)} \in \text{class labels}}{\operatorname{argmax}} \sum_{i=1}^{k} \mathbb{I}\left(t^{(z)} = t^{(i)}\right)$$

When I say class labels, it means the set of possible classifications (e.g., is it a cat, dog, and so on). This only works if the labels are discrete.

This notation is kind of confusing:

- $\mathbb{I}$ is the indicator and behaves like `int()` in python which takes in a Boolean.
- When dealing with `max` functions, break ties however you wish.

I might write this as pseudocode:

```
def kNN(examples: list[tuple[vec, label]]) -> label:
    highest_val = 0
    highest_item = None
    for tz in CLASS_LABELS:  # EVERY label in T
        acc1 = 0
        for ti in examples:
            if ti[1] == tz:
                acc1 += 1
        if acc1 >= highest_val:
            highest_val = acc1
            highest_item = tz
    return highest_item
```

This algorithm above just decides what label occurred the most in our k nearest neighbors.

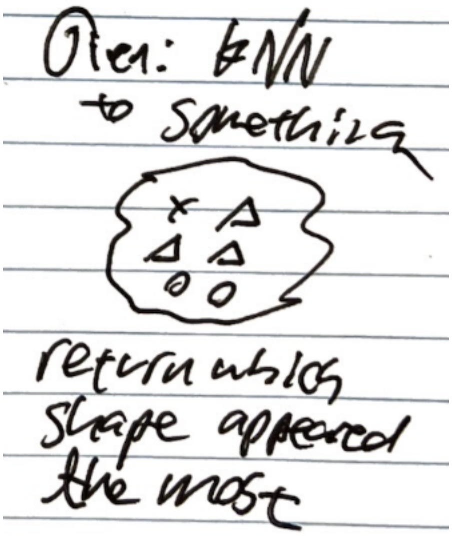

Figure 4: What is that complicated algorithm actually doing?

We can treat kNN as:

- “Averaging” stuff out

How do we choose k?

- There are a lot of ways
- Sometimes it must be personalized from the data
- Might be on the features we are measuring

Some examples:

- $k = N$ ALWAYS picks the majority. Hence if there are way too many cats, we will always predict cats. Dumb idea; don't do it.

Trade-offs?

- Smaller ks helps us capture fine-grained patterns
  - **Very prone to overfitting!**
- Large ks makes stable predictions by averaging out lots of examples
  - Underfits; fails to catch important patterns of the data
- Balancing k
  - Optimal choice depends on data points n
  - Rule of thumb: $k < \sqrt{n}$
  - We chose k with validation sets in practice.

We want our algorithm to generalize to data it hasn't seen before. We can measure the generalization error using a **test set.**

2.8 Training set, Test set

- Training error is 0 if $k = 1$
  - But test set error goes up (due to overfitting)!
- Training error increases as k increases
- Test error is usually U-shaped.

The **bayes error** is the theoretical minimal test error if you have infinite training data.

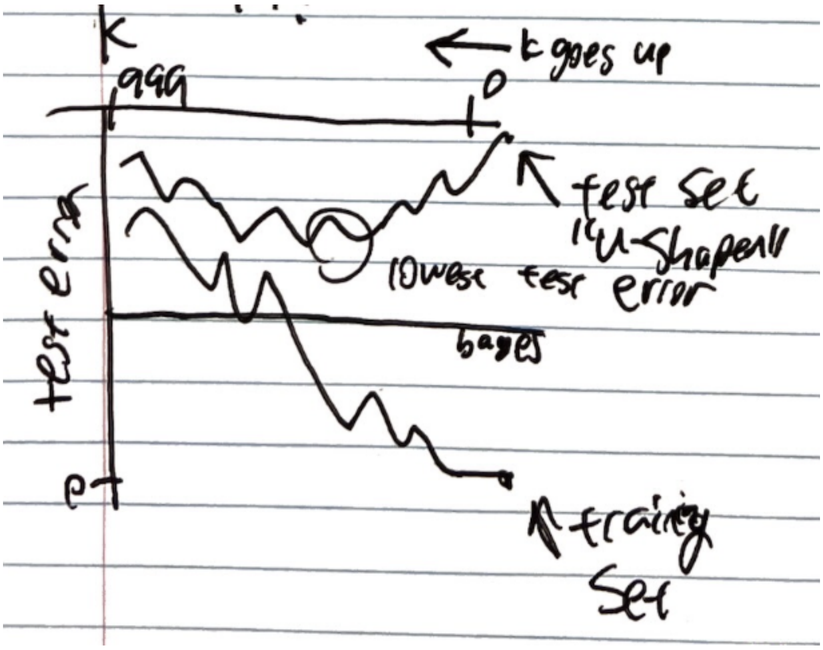

Figure 5: How k impacts the training and test set error.

2.9 Validation and Test Sets

Standard procedure. k is an example of a **hyperparameter**, which means **we must choose this, and we can't fit it in the learning algorithm.**

So, we have:

- Training set (Usually 80%)
- Validation set (used to decide k, like test sets. Allocate it. Usually 20%)
- Test set (do not touch until k is picked; oftentimes new data that we don't have yet).
    - The test set measures the generalization performance of the final configuration

2.10 The Curse of Dimensionality

DIMENSION COUNT IS FEATURE COUNT

As we move to higher dimensions, issues will arise. Low dimensional visualizations are misleading. In high dimensions, most points are far apart, and it takes a lot of points.

If we want any query x to be closer than ε. How many points do we need to guarantee it?

- The volume of a ball is $\mathcal{O}\left(\varepsilon^d\right)$
- The total volume of $[0,\ 1]^d$ is 1.
- Therefore, $\mathcal{O}\left(\left(\frac{1}{\varepsilon}\right)^d\right)$ points are needed to cover the volume.

For example, if $\varepsilon = 0.01$, we need $\mathcal{O}\left(100^d\right)$ to achieve what we want to achieve.

If we want a good classification in higher dimensions, we need a lot of points. In higher dimensions, most points are approximately the same distance.

We do have ways to avoid the problem:

- Our data isn't truly random. For instance, the space of megapixel images is 3 million-dimensional.
- The number of degrees of freedom for an actual photo is way less.
- Nearest neighbors care more about the intrinsic dimension.

2.11 Units and The Problems They CAUSE, Normalization

- Units: they are arbitrary. Imperial units can mess things up. Normalize everything.
  - Large units like km will be seen as insignificant if the other axis is small (pm)
- Normalize each dimension to be zero mean and unit variance.
  - $\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}$ (All variables are in $\mathbb{R}^n$ for any n we can choose, given they're consistent. We use vectors to conveniently express all that computation.)
- Scales MAY be important, so only normalize if you think it won't cause side effects.

2.12 Computational Cost

When making a new algorithm, we have trade-offs:

- Computational cost
- Memory cost

For example, for NN

- No computations for training, but memory intensive as we need to store all data
- Expensive to run: D-dimensional Euclidean distances with N data points are $\mathcal{O}(ND)$
    - Then we need to sort the distance: $\mathcal{O}(N\log(N))$

2.13 NN Wrap Up

- A way to use our training data to help decide labels to new inputs
- All work done in testing. No learning!
- Complexity can be controlled by varying k
- Curse of dimensionality! Becomes a lot more expensive and a lot more data is needed as dimensions go up!

2.14 TLDR

Stages for ML

- Training
    - Update parameters
- Evaluation/Validation
    - To select hyperparameters to avoid overfitting and underfitting

- Test
  - Evaluate the algorithm's performance

Supervised learning:

n input samples with d features: $M_{n \times d}$ where n is data points and d is features.

The entire matrix outputs a real number or a class, if we're dealing with a classification problem.

Nearest neighbors are for binary classification.

Hyperparameters: There are a lot of knobs for the algorithm; as designers, we must determine them. We'll use validation data to choose the right settings (validation is like test, but less formal)

2.15 Classification

Non-parametric classifier: No parameters learned in the training process.

Parametric classifier: Something was learned first. Giving it training data, you learn something from it (e.g., stuff gained from linear regression)

Hyperparameter: Can't be learned (but can be optimized – remember all those optimization problems from calculus and the ∩-shaped curve?).

3 Decision Trees

Recall how they look like.

- Internal nodes test a feature
- Branching is determined by the feature value
- Leaf nodes are outputs

At the bottom of the tree, whatever we end up with will be the decision we make.

So, what are the hyperparameters (specified beforehand before running the training process)?

- The number of nodes
- The maximum depth of the tree

These are different design choices we can make.

3.1 Classification Trees

Discrete output, meaning we return something from a set, i.e., a fruit, color, and so on.

Decision trees are very interpretable. We don't specify the logic of the trees; we specify the objectives.

When making decision trees, if we have a bunch of training data (datapoints $\vec{x} =$ a class filled with values put into each feature), and an output, we can form a decision tree.

Decision trees are universal function approximators. This isn't useful to us and finding the smallest decision tree that correctly classifies a training set is NP complete (too difficult).

So how do we construct a useful decision tree?

3.2 The Greedy Heuristic

Rather than trying to figure out the whole tree, we try to make the best decision on each split. Each step along the way, we make the best possible choice for that metric.

We also introduce **loss**, the metric to measure performance, which we wish to optimize. This is done in some **scalar value,** and we want to minimize it.

What motivates the choice of loss?

- The goal is for each leaf to correctly identify each class

The idea is to use counts as leaves to define probability distributions. Then, we can use entropy. This is one way to evaluate a split.

3.3 Entropy

To describe uncertainty in the random variable. It is:

$$\sum_i p(i) \log\left(\frac{1}{p(i)}\right) = -p\log_2(p) - (1-p)\log_2(1-p)$$

Information theory is concerned about how you can send information.

If an outcome is more certain, there will be a lower entropy. For example, a 90%-coin flip will have a low entropy, whilst a 50%-coin flip will have a higher entropy.

You cannot store the output of a random draw using fewer expected bits than the entropy without losing information. Units of entropy when using log-base 2 are bits; a fair coin flip has 1 bit of entropy.

More generally, the entropy of a discrete random variable Y is:

$$H(Y) = -\sum_{y\in Y} p(y)\log_2\left(p(y)\right)$$

This means you can calculate the entropy for any random variable (maybe discrete, for now).

High entropy:

- Uniform-like distribution

Low entropy:

- Distribution concentrated

You do not need to understand this fully; you just need to be aware of the definition of entropy.

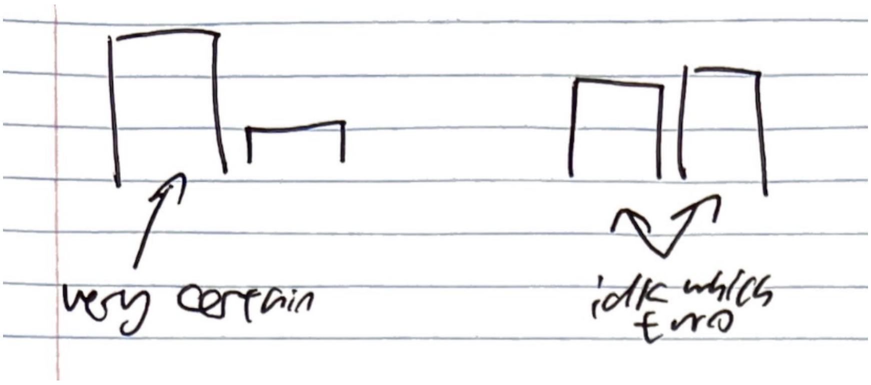

Figure 6: The visual example of entropy

3.3.1 How are we going to use it?

We have a split. We're going to evaluate the entropy after performing a split, and we'll see the expected reduction in our uncertainty about Y after observing X.

- Y is our initial distribution of labels
- X is the split we consider
    - Refers to the event of a split

For example:

- $X = \{\text{Raining}, \text{Not}\}$ and $Y = \{\text{Cloudy}, \text{Not}\}$

And suppose you want to decide whether you want to bring **an umbrella outside:**

	Cloudy	Not cloudy
Raining	$\frac{24}{100}$	$\frac{1}{100}$
Not raining	$\frac{25}{100}$	$\frac{50}{100}$

The entropy is:

$$H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2 p(x, y)$$

3.3.2 Conditional Entropy

$$H\left(Y|X=x\right)=-\sum_{y\in Y}p\left(y|x\right)\log_2\left(p\left(y|x\right)\right)$$

This is the same formula. If we want $P\left(\text{cloudy}|\text{raining}\right)$, we need to use Bayes' rule and the marginal probability formula. We get $\frac{24}{25}$. And $P\left(\text{not cloudy}|\text{raining}\right)=\frac{1}{25}$. Now, plug it into the formula above.

Every time we traverse a label, we are essentially saying that something is given.

The expected conditional entropy is:

$$\begin{aligned}H(X)&=\mathbb{E}_x\left(H\left(Y|x\right)\right)\\&=\sum_{x\in X}p(x)H\left(Y|X=x\right)\\&=-\sum_{x\in X}\sum_{y\in Y}p(x,y)\log_2\left(p\left(y|x\right)\right)\end{aligned}$$

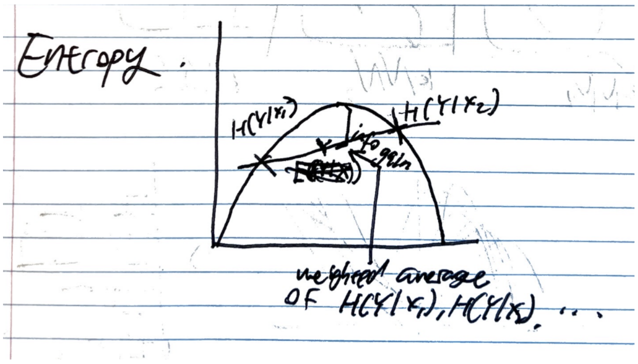

Figure 7: Expected Entropy

3.4 Using Conditional Entropy

Example: $X=\{\text{Raining},\ \text{Not raining}\}$, $Y=\{\text{Cloudy},\ \text{Not}\}$

The entropy of cloudiness given whether it's raining or not, plug it into the entropy formula.

You'll need to know this formula to understand most of the text that follows here.

$$\begin{aligned} H(Y|X) &= \sum_{x \in X} p(x) H(Y|X=x) \\ &= \frac{1}{4} H(\text{CL}|\text{RA}) + \frac{3}{4} H(\text{CL}|\neg\text{RA}) \end{aligned}$$

3.4.1 Properties of Entropy

- $H \geq 0$
- $H(X < Y) = H(X|Y) + H(Y) = H(Y|X) + H(X)$
- X, Y are independent $\Rightarrow H(Y|X) = H(Y)$
- $H(Y|Y) = 0$
    - If we know it, we don't need to know anything else
- $H(Y|X) \leq H(Y)$
    - More information makes entropy go down (never up)

3.5 Information Gain

How much more certain do I get after knowing something>

$$IG(Y|X) = H(Y) - \underbrace{H(Y|X)}_{\text{ALL OF THEM}} \geq 0$$

The difference in entropy before and after

- If knowing X gives us NOTHING about Y, $IG(Y|X) = 0$
    - And $H(Y) = H(Y|X)$

- If completely informative, $IG(Y|X) = H(Y)$
  - So, given knowledge of X, there is no more uncertainty of Y.
  - We get all the information

3.5.1 So, How Do We Decide Splits?

We take splits that max out the information gain. We can consider the information gain from making splits:

If we have 5 reds/circles and 2 blues/triangles, we have an entropy of 0.86

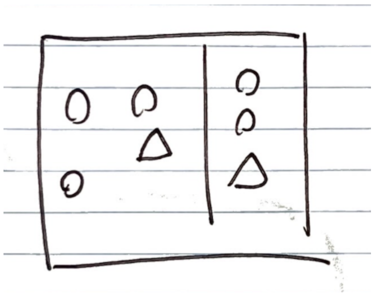

Figure 8: First split example. Not a good split

Then when we split it (FIRST ONE IN THE SLIDES):

$$H(Y|\text{left}) = 0.81 \quad H(Y|\text{right}) = 0.92$$

How do we calculate it? Using 1 blue 3 red, corresponding to a distribution of $\frac{1}{4}, \frac{3}{4}$ (we're dealing with a sum, so the order does not matter).

From this, we can calculate the information gain (RECALL THE CONDITIONAL ENTROPY FORMULA):

$$0.86 - \left(\frac{4}{7} \cdot 0.81 + \underbrace{\frac{3}{7}}_{\text{split ratio}} \cdot 0.92\right)$$

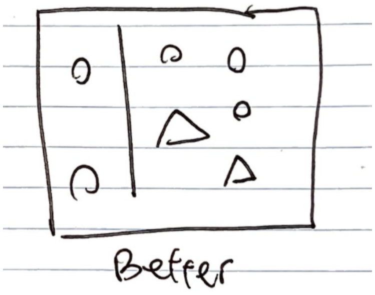

Figure 9: The better split

The BETTER split: the entropy is lower – no uncertainty on the left branch. Hence, we have information gain.

- $H(Y|\text{left}) = 0.86$
- $H(Y|\text{right}) = 0.97$
- $IG(\text{split}) = 0.86 - \left(\frac{2}{7} \cdot 0 + \underbrace{\frac{5}{7}}_{\text{split ratio}} \cdot 0.97\right)$

(FIGURE GOES HERE)

3.6 So, How do we Actually Construct a Decision Tree?

At each level:

- Choose which feature to split
- Where we would split it

Choose based on how much information we would gain.

An algorithm statement:

1. Start from root node, pick feature to split
2. Split examples based on feature value
3. For each group

a. If no examples, return majority from parent

b. If all from the same class, return class

c. Else loop to step 1

Loop ends when all leaves contain only examples in the same class or are empty.

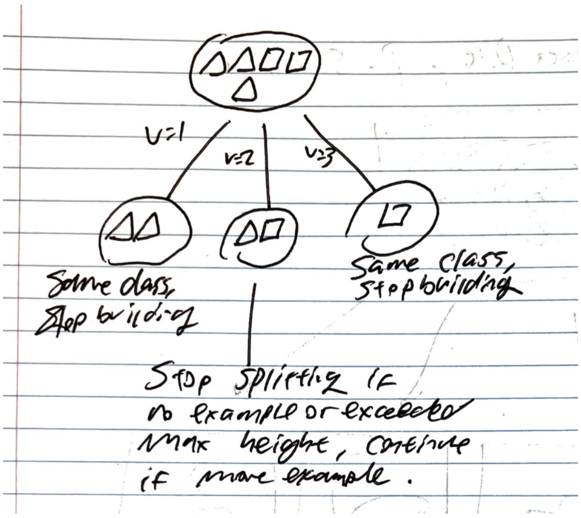

Figure 10: Tree creation

3.7 Creating Trees

What makes a good tree?

- Not too small
  - We need to handle important but subtle distinctions
- Not to big
  - Inefficient and redundant
  - Overfitting
  - Hard to interpret by us

OCCAM'S RAZOR: Find the simplest hypothesis that fits the observations

- We want small trees with informative nodes near the root.

3.8 Problems with Decision Trees

Problems

- Exponentially less data at lower levels
  - To have enough labeled data after 5 splits, we need 2^5 pieces of data… way too much.
- Too big of a tree can overfit
- Greedy algorithms don't necessarily yield the global optimum
  - Greedy approaches don't produce the best trees. Out of all the trees, greedy trees are one of them

Usage

We can handle continuous attributes, but we'll split them based on a threshold, chosen to maximize information gain. They can be used for regression on real-valued outputs (minimize squared error instead of info gain, in this case here).

3.9 kNN vs Decision Trees

A lot would be "it depends."

Decision trees:

- Are simple to deal with for discrete, missing values, or poorly scaled data
- Fast at test time
  - kNN requires us to look at all training examples. For decision trees, we just use the tree and that's it. Extremely fast, running time is number of levels of the tree
- More interpretable

kNN:

- kNN has less hyperparameters (k); decision trees have way more
- kNN is more sensitive to data
- 0 training time (compared to decision trees, which we need to perform computations to build the tree)
- Can incorporate interesting distance/special measures (e.g., shape contexts)

However, in general, decision trees are more preferred over nearest neighbors. kNN are prone to the dimensionality problem.

3.10 Ensembling

If 10 expert make predictions, we'll have better predictions. This is the notion of the wisdom of the crowd – we want a majority vote. The more people that guess, the more likely we'll be accurate.

We want the classifiers to be slightly different. We can combine kNN and decision trees, or decision trees trained on different subsets / hyperparameters.

3.11 Constructing Trees

A greedy heuristic is to choose the split that minimizes entropy. Loss happens is the function most practically most likely squared distance between prediction and true prediction.

The accuracy of a predictive model (85% predictions right?) is an evaluation metric but is not a differentiable continuous function. So, accuracy is something we may not want to focus on.

A way we could define a loss (it can measure how well we're doing), we can let $L(y, t) \to \mathbb{R}$. Where $\mathscr{D}$ is a tuple of a dataset of $\left(\underbrace{X}_{\text{all inputs}}, \underbrace{t}_{\text{all outputs}}\right)$. If X is a matrix of all inputs, t is a vector, where each index corresponds to the ith input.

4 Bias-Variance Decomposition

We can finally define overfitting and underfitting?

4.1 Generalization

We want our model to do well on an unseen test set, drawn from the same distribution from the training data.

- We want to deploy our model in the real world and get it to work on data it has never seen before

Trade-offs?

- If you have an overly simple model, we'll underfit (such as too high of a k for kNN)
- On the other hand, if our model is too complicated, we'll overfit

We can quantify underfitting and overfitting using bias-variance decomposition.

4.2 Set up for Decomposition

A bit tricky to think about. Think it as a thought experiment.

- p_{sample} is a data generated distribution
  - ground truth, so we know the real distribution
  - IRL we don't know this.
- Pick a fixed point $\vec{x}$. We want to get a prediction y at $\vec{x}$.
- Generate multiple algorithms on the training set.
  - Each algorithm or instance takes a random subset from the training set p_{sample}.
- We can view y as a radom variable, where the randomness comes from the **choice of the training set.**

- The classification accuracy can be determined by the distribution of y
  - We can calculate $E(y)$, $V(y)$, and so on.

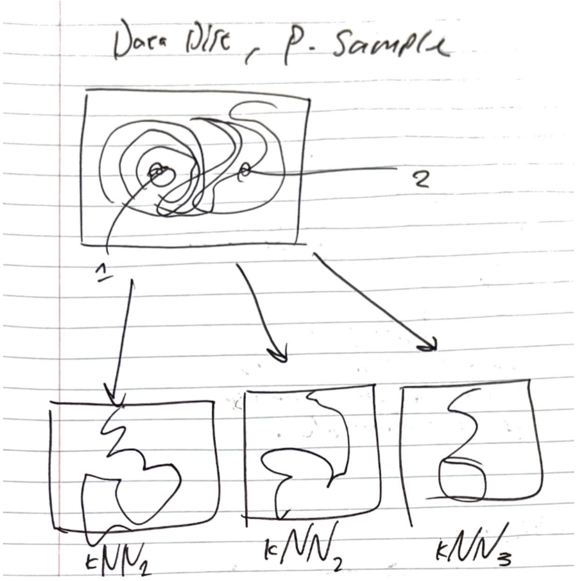

Figure 11: Decomposition, this time into three

We can also use regression: regress multiple times on different samples of the data.

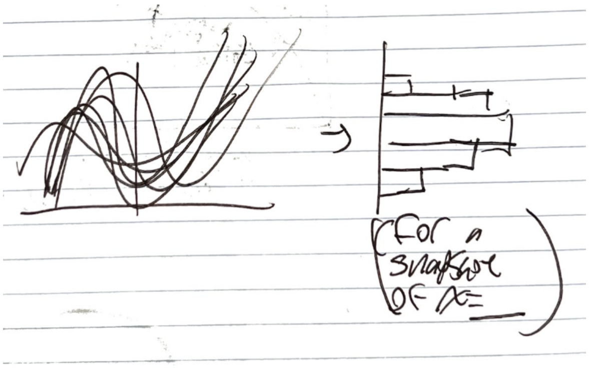

Figure 12: Regressing multiple times

4.3 Setup

- Fix $\vec{x}$

- Repeat
  - Sample random training dataset $\mathscr{D}$ (iid) from p_{sample}
  - Run learning algorithm on $\mathscr{D}$ to get prediction of y at $\vec{x}$
  - Sample the true target from the conditional distribution $p\left(t|\vec{x}\right)$
  - Compute loss $L\left(\underset{\text{training}}{y}, \underset{\text{test}}{t}\right)$. Loss can be interpreted in many ways but for here, it's $(y-t)^2$.

Both y and t are independent, as t is from the test dataset and y is from the training dataset.

We get a distribution over the loss at $\vec{x}$, with expectation $\mathbb{E}\left(L(y, t)|\vec{x}\right)$. Read this as expected loss with $\mathbf{x}$.

For each query point $\vec{x}$, expected loss is different. We want to look for y that minimizes the expectation.

4.4 Choosing a Prediction

Consider that we know the ground truth. We want to minimize

$$\begin{aligned}\underset{\text{for a choice of}}{\text{Loss}(y)} &= \mathbb{E}\left[(y-t)^2|\vec{x}\right] \\ &= E\left[y^2-2yt+t^2|\vec{x}\right] \\ &= E\left[y^2|\vec{x}\right]-E\left[2yt|\vec{x}\right]+E\left[t^2|\vec{x}\right]\end{aligned}$$

We can treat y as a constant as it does not depend on x:

$$
\begin{aligned}
&= y^2 - 2yE\left[t|\vec{x}\right] + E\left[t^2|\vec{x}\right] \\
&= y^2 - 2yE\left[t|\vec{x}\right] + V\left(t|\vec{x}\right) + E\left(t|\vec{x}\right)^2 \\
&= \underbrace{\left(y - E\left(t|\vec{x}\right)\right)^2}_{y \text{ controls this}} + \underbrace{V\left(t|\vec{x}\right)}_{\substack{\geq 0, \text{ Bayes error} \\ \text{unavoidable noise}}}
\end{aligned}
$$

The best choice of y, is $y_* = E\left(t|\vec{x}\right)$, if we knew p_{sample}.

EXPECTED LOSS (mean squared error) ON DATA POINT $\vec{x}$:

$$
E\left[(y-t)^2|\vec{x}\right] = (y - y_*)^2 + V\left(t|\vec{x}\right)
$$

4.5 Bias, Variance, Bayes Error

If we treat y as a random variable, we can decompose the loss of it further:

$$
\begin{aligned}
&E\left[(y-t)^2\right] \\
&= \underbrace{(y_* - E(y))^2}_{\text{bias}} \\
&+ \underbrace{V(y)}_{\text{variance}} + \underbrace{V(t)}_{\text{bayes error}}
\end{aligned}
$$

The following (think of the target analogy you've learned in chemistry and physics):

- **Bias**: how far we are from the optimal point. **Opposite to accuracy**
  - Corresponds to underfitting. High bias models are underfitting models. Models with too much bias could possibly oversimplify the problem and consistently produce predictions that are way off what is true.
  - Bias is supposed to go down the more complex our model.
- **Variance**: the amount of variability in the prediction. **Opposite to precision**
  - Corresponds to overfitting.

  - Averaging helps (removes noise)
  - A model with high variance tends to change a lot even due to small fluctuations in the training data.
  - Variance may go up if our model becomes too complex.
- Bayes error: Noise

Each algorithm has a bias term and the variance term. There is a trade-off. More powerful classifiers are more prone to overfitting.

4.6 Bagging (Bootstrapping)

We sample multiple datasets from the dataset. That is, sampling with replacement. If we have a dataset of n data points, we are going to sample n of them with replacement. That is our new dataset. You'll get points that are repeated, but the idea of bagging is that even though points are repeated, we get classifiers that work, and we average all of them together.

The point is that we are going to miss some points in our original dataset and resample some points multiple time.

Then, we train a model on each of our bootstrap datasets.

Bayes' error does not change at all when you bootstrap, and bias won't change because average predictions still have the same expectation, but the term you can control is the variance (one over the inherit properties of variance: $V(cx) = c^2V(x)$).

Decision trees can be trained quickly, so people can start off with 100 bootstrap samples. There is no default formula for this.

$$y_{\text{bagged}} = \mathbb{I}\left(\frac{1}{m}\sum_{i=1}^{m} y_i > 0.5\right)$$

4.7 Bagging Properties

A bagged classifiers can be stronger than the average model. However, if m datasets are not independent, we don't get the full $\frac{1}{m}$ variance reduction. The terms are not independent. We're trying to decorrelate the trees more, to make the trees more independent. We can use random forests.

4.8 Random Forests

Restricts splitting to only a subset of features. This is the best black-box ML algorithm: From $d = 100$ features, and randomly sample 10 features.

- Works well with no tuning
- Helps reduce features as the trees become less correlated
- Tends to work relatively well

4.9 Limitations

Bagging reduces overfitting. But

- Does not reduce bias
- Use random forests to get rid of dependence
- Weighting members may not be the best. We may want to re-weigh the datapoints, if we do particularly bad on them.

5 Linear Algebra

A powerful set of tools to concisely depict data, parameters, and measure different quantities. We have some elements:

- Scalar: a number. a

- Vector: 1-D array, $\vec{a}$
- Matrix: 2-D array, A
- Tensor: 3-D or above array. We're never going to use this

5.1 Norms

Measures how large a vector is.

$$l_p\text{-norm } ||x||_p = \left(\sum_{\substack{i,\text{ each} \\ \text{element of } x}} |x_i|^p \right)^{\frac{1}{p}}$$

In this course, we'll mostly use the l_2-norm (the Euclidean distance). The l_1 norm is the Manhattan norm (taxicab distance): $\sum_i |x_i|$, and the l_∞ norm is the max-norm: $\max_i |x_i|$ ($\max_i$ means choose x_i for which it is the greatest).

p can be unrelated to the dimension of your vector. Some norms are better/worse fits to characterize data.

5.2 Projections

To interpret linear models, such as linear and logistic regression, is to project datapoints to certain linear subspaces.

The formula for projection of $\vec{a}$ onto $\vec{b}$ is given by $\text{proj}_{\vec{b}}\vec{a}$:

$$\frac{\vec{a}^T\vec{b}}{||b||_2} \cdot \vec{b}$$

6 Linear Regression

Used to predict scalar-valued target. We have a linear function of the inputs.

Supervised learning setup:

Input $\vec{x} \in \mathcal{X}$ and target $t \in \mathcal{T}$

We have data $\mathcal{D} = \left\{ \left(\vec{x}^{(i)}, t^{(i)} \right) \text{ for } i = 1, 2, \ldots, N \right\}$

We're focusing on the linear model, features $\vec{x} = (x_1, \ldots, x_D) \in \mathbb{R}^D$

Each of the features are scalar. We can see that our functions are

$$w_1 x_1 + w_2 x_2 + \cdots + b$$

We have the idea called the bias: this is the y-intercept, and this is the default prediction. We want the prediction to be close to the target as possible.

Parameters are weights $\vec{w}$ and the bias b (intercept).

6.1 Loss Function

Measures how bad the algorithm's prediction is vs. the ground truth. The most common loss function is the squared error:

$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

$y - t$ is the residual; $\frac{1}{2}$ is for convivence if you take the derivative of the loss function.

COST FUNCTION: empirical or average loss

$$\begin{aligned} &\frac{1}{2N} \sum_{i=1}^{N} \left(y^{(i)} - t^{(i)} \right)^2 \\ &= \frac{1}{2N} \sum_{i=1}^{N} \left(\vec{w}^T x^{(i)} + b - t^{(i)} \right)^2 \end{aligned}$$

Squared error helps penalize higher errors more.

6.2 Vectorization

To express computations to a computer, write things in vector form rather than using summation notation or for loops. For example, you can take the sum of a vector by taking the dot product with a vector filled with 1s.

$$y = \vec{w}^T \vec{x} + b$$

Which can be done with `np.dot(w, x)+ b`

For linear regression, you can add another feature that is always 1, so you don't need the b term in your linear regression term.

6.2.1 Rationale

It is way more efficient to vectorize. When writing code that goes in production, use a library that does the complicated operations.

Sometimes, you can make derivations using vectors right away. Or you can write things using for-loops first then change to vectors afterwards.

6.3 Making Dataset Predictions

$$X\vec{w} + b1 = y$$

This is our design matrix. We can calculate the squared error loss across the whole dataset:

$$\mathcal{J} = \frac{1}{2N} ||\vec{y} - \vec{t}||_2^2$$

Where N is the size of the dataset. $\frac{1}{2N}$ is the normalizer to get an averaged loss.

Rather than adding the bias separate, we can augment the matrices.

6.4 Optimization

We want to minimize the cost function $\mathcal{J}(\vec{w})$. The minimum of a smooth function occurs at a critical point. This works perfectly for linear regression, as you don't need to worry about hitting the wrong critical point. You may experience that problem for neutral networks.

We can try to set the gradient to 0 and try to solve for that, or we can use gradient descent. This is iterative. We compute the gradient with respect to our current parameters, we update them, and we repeat.

$$\nabla_{\vec{w}}\mathcal{J} = \frac{\partial \mathcal{J}}{\partial \vec{w}} = \begin{bmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_{D+1}} \end{bmatrix}$$

Vector of partial derivatives, on each weight. y is a function involving w.

6.4.1 Minimization example

$$\mathcal{J}(\vec{w}) = \frac{1}{2}\left|\left| \underset{n\times d}{X} \underset{d\times 1}{\vec{w}} - \underset{n\times 1}{\vec{t}} \right|\right|_2^2$$

The cost function is a function of the data. Because you can't change what data you are given, we're only interested on taking the gradient with respect to the weights.

$$\begin{aligned} &= \frac{1}{2}\left(X\vec{w}-\vec{t}\right)^T\left(X\vec{w}-\vec{t}\right) \\ &= \frac{1}{2}\left(\vec{w}^TX^T - t^T\right)\left(X\vec{w}-\vec{t}\right) \\ \mathcal{J}(\vec{w}) &= \frac{1}{2}\left(\vec{w}^TX^TX\vec{w} - 2t^TX\vec{w} + t^Tt\right) \end{aligned}$$

We want to look at $\nabla_{\vec{w}} \mathscr{J}\left(\vec{w}\right)$. We know that $\vec{w}^T A\vec{w} \overset{\rightarrow}{\nabla} 2A\vec{w}$, and $c^T w \overset{\rightarrow}{\nabla} c$

$$\begin{aligned}\nabla_{\vec{w}} \mathscr{J}\left(\vec{w}\right) &= \frac{1}{2}\left(2X^T X\vec{w} - 2X^T t\right)\\ &= X^T X\vec{w} - X^T t\end{aligned}$$

To solve this, we set this expression to 0:

$$X^T X\vec{w} - X^T t = \vec{0}$$

And look for $\vec{w}^*$ (optimal weights) that satisfies the equation:

$$\vec{w}^* = \underbrace{\left(X^T X\right)^{-1} X^T t}_{\text{pseudoinverse}}$$

The solution might not exist as not all matrices are invertible. This calls for gradient descent

6.5 Gradient Descent

This will always get you to a critical point. You drop a ball on the function, and hopefully the ball lands somewhere. The gradient is the direction of steepest ascent, so to go down we head in the opposite direction of it. When you take a step, the weights will have changed.

This works well for linear regression, but maybe not other fields.

We'll use this:

$$\nabla_{\vec{w}} \mathscr{J}\left(\vec{w}\right) = X^T\left(X\vec{w} - \vec{t}\right)$$

What is $X\vec{w} - \vec{t}$? The difference between the prediction and the true value. If our predictions are extremely off from our true value, the gradient is going to be large. If our gradient is close to 0, then our prediction is close to the true value.

The following update always decreases the cost function for a small enough α, unless the gradient isn't 0:

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathscr{J}}{\partial w_j}$$

We'll worry about how α is chosen later.

We decide to stop with gradient descent when we feel like we've done it enough time. Here are some guidelines for the learning rate α:

- The larger α is, the faster $\vec{w}$ changes
- Be careful of overshooting
- Values are typically very small
- Minimizing total loss requires a smaller learning rate: $\alpha^{'} = \frac{\alpha}{N}$.

We can write this in vector form:

$$\vec{w} \leftarrow \vec{w} - \alpha \frac{\partial \mathscr{J}}{\partial \vec{w}}$$

For linear regression:

$$\vec{w} \leftarrow \vec{w} - \frac{\alpha}{N} \sum_{i=1}^{N} \left(y^{(i)} - t^{(i)} \right) \vec{x}^{(i)}$$

6.5.1 Why do we use it

- Way more applicable
- Easier to implement
    - Many ML libraries can implement that efficiently
- For efficient than direct solution

  - Linear regression solution using matrix inversion, $\left(X^T X\right)^{-1} X^T \vec{t}$ is an $\mathscr{O}\left(D^3\right)$ operation
  - Gradient regression update costs $\mathscr{O}(ND)$
  - Huge difference if D is large.

6.6 Feature Mapping

Nonlinear regression:

$$y = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{i=0}^{M} w_i x^i$$

The feature mapping is $\psi(x) = \left[1,\, x^2,\, \ldots, x^M\right]^T$

M is the no. of degrees of the polynomial that is used to regress.

The higher the degree, the more prone it is to overfitting.

- Underfitting gives us high bias

6.7 Regularization

Prevents weights from being too large. Regularizer is a function that quantifies how much we prefer one hypothesis over another and alludes to Occam's razor.

L^2 regularization goes like the following:

- It penalizes weights that are too large:

$$\mathscr{R}\left(\vec{w}\right) = \frac{1}{2}\left|\left|\vec{w}\right|\right|_2^2 = \frac{1}{2}\sum_j w_j^2$$

The regularized cost function makes a trade-off between the fit to the data and the norm of the weights.

$$\mathscr{J}_{\text{reg}}(\vec{w}) = \mathscr{J}(\vec{w}) + \lambda\mathscr{R}(\vec{w})$$

Large λ penalizes for more error. Best to choose λ that isn't way too large.

$$\vec{w}_{\lambda}^{\text{Ridge}} = \left(X^T X + \lambda N \cdot I\right)^{-1} X^T \vec{t}$$

The word ridge is just another term for l-2 regularization

6.8 Gradient Descent Under L2-Regulaization

$$w \leftarrow (1 - \alpha\lambda)\vec{w} - \alpha\frac{\partial \mathscr{J}}{\partial \vec{w}}$$

Each step along the way, $\vec{w}$ will be decreased a bit towards zero, then updating the gradient. We choose λ via the validation set.

6.9 Conclusion

Linear regression exemplifies themes in ML:

- Choose a model and a loss function
- Formulate an optimization problem
- Vectorize the algorithm
  - Makes things way faster
- Use linear models using feature mappings
- Improve generalization by adding a regularize

Linear models with neutral nets are very nice to deal with, as they are very interpretable.