CSC311 Notes

Introduction to Machine Learning

https://github.com/ICPRplshelp

Last updated January 9, 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).

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

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

$$\underset{\text{data points}}{N} \times \underset{\text{features}}{d}$$

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

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.

Algorithm. The nearest vector is:

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

Output $y = t^*$. This requires $\vec{x}^*$ (image in training set closest to $\vec{x}$; t^* is $\vec{x}$'s label). The distance function is the norm; you've heard of it in linear algebra.

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

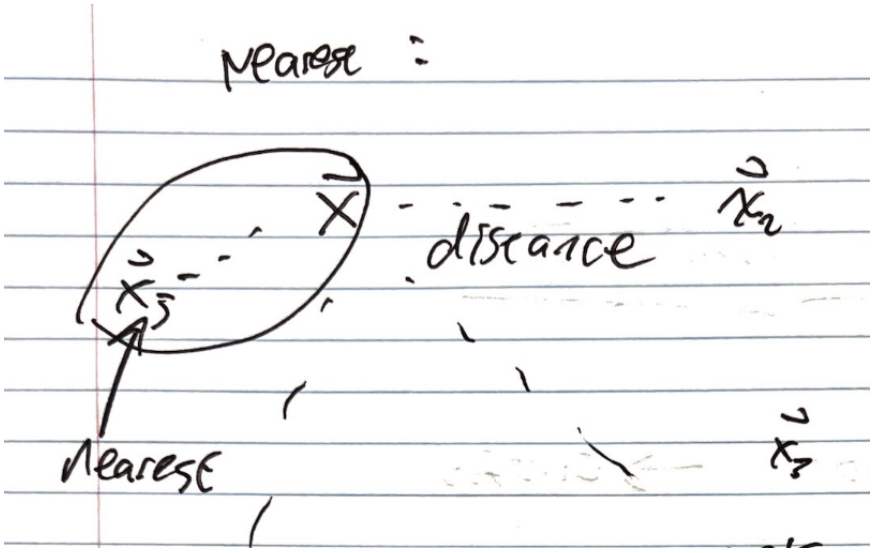

Figure 1: Visualization of the only nearest neighbour to the input vector x

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.

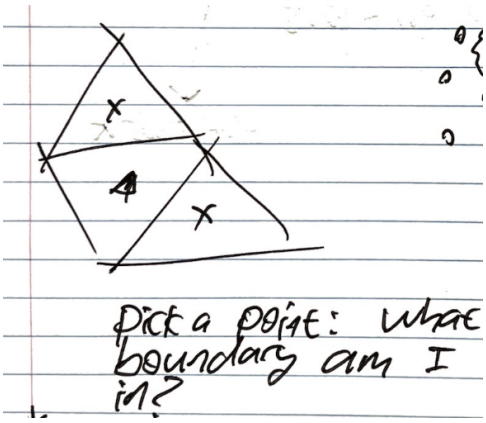

Figure 2: Voronoi Diagram with 3 datapoints and 2 distinct labels

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.

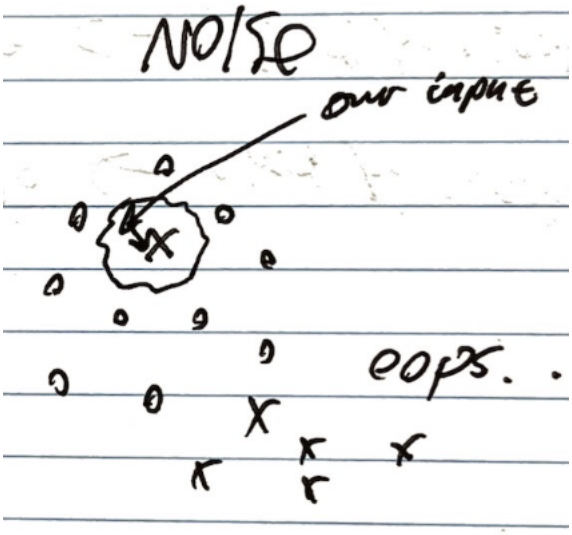

Figure 3: The problem with noise

Solution? Smooth it out by having k-nearest neighbors. Here, **we take up to $\mathbf{k}$ 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.

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

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

This 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.

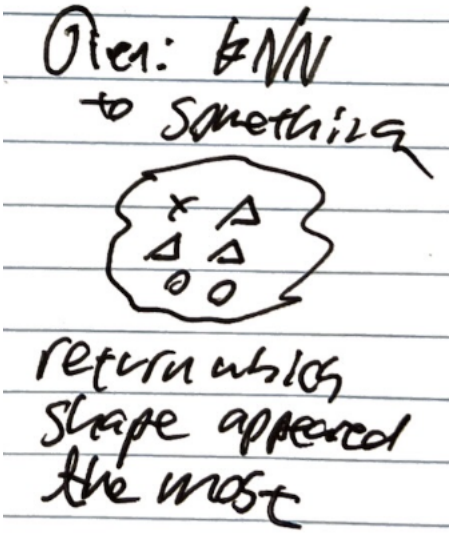

Figure 4: What is that complicated algorithm actually doing?

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.

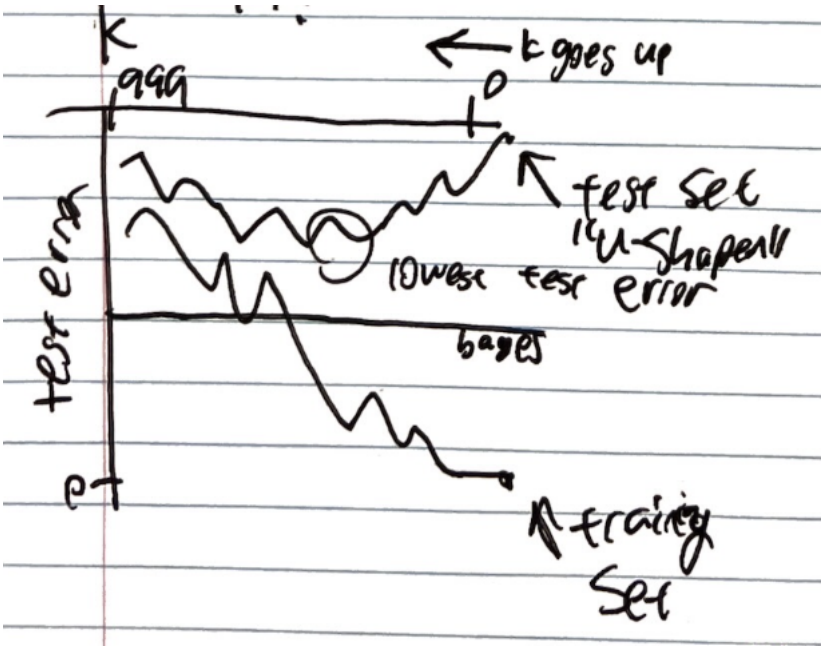

Figure 5: How k impacts the training and test set error.

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.
  - $\widetilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}$ (All variables are in $\mathbb{R}^n$ for any n we can choose, given they're consistent. We use vectors to convinently 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!

3 Parametric Models

TBA