Machine Learning (Part 1): Introduction

Objective of a *brief* introduction to ML

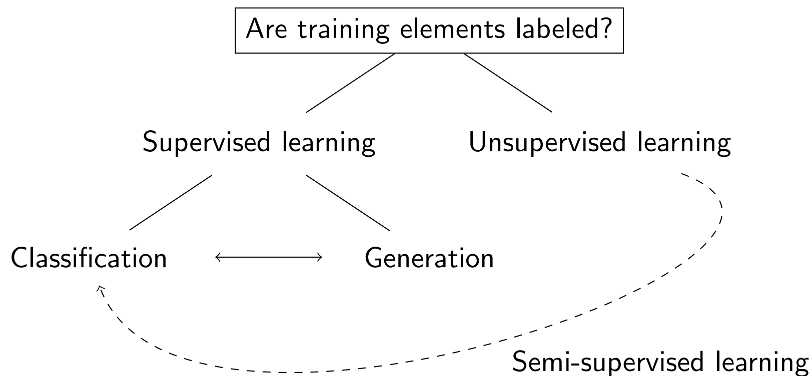
We want to give you:

- The tools and terminology used in ML, focusing on classification
- The critical thinking skills needed to consider experimental setups, tools, and evaluation results
- The context to understand how ML has changed over the years and how this has affected the development of security & privacy

Divided into three parts:

1. Introduction to ML, general strategies and terminology
2. Non-NN Classifiers
3. NN Classifiers

Overview



The basics of classifiers

A classifier aims to determine which class $C(x)$ input instances x belong to. It does so by training on labeled instances (**supervised** learning), and its performance is evaluated through testing on other labeled instances.

- **Parameters**: Internal values of the classifier learned during training; determines how the classifier classifies
  - **Weights**: Multiplied onto extracted features or raw data for classification. Often the main parameters of the classifier
  - **Hyperparameters**: Set by the experimentor, generally controls unlearnable behavior. Examples: Learning speed, number of epochs
- **Architecture**: The design of the classifier; always specified for neural networks
- **Ground truth**: Correct labels for training/testing elements. Often the main challenge of a research work

The basics of classifiers

- **Feature engineering**: We can extract feature sets from data elements to simplify the input and to avoid the **curse of dimensionality**. Often done manually with domain expertise

Example

For network traffic: packets per second, ratio of outgoing packets, sizes of packet bursts, median interpacket time, etc.

- **Encoding**: Some types of inputs need to be encoded into a numerical structure for machine processing

Example

Byte-pair encoding: Assign a token number to each character (a=1, b=2...). Then, iteratively, assign a number the most frequent pair of tokens. Repeat until a predefined vocabulary size is reached.

Simple example: Naive Bayes

- We can treat classification as a Bayesian question: What is the probability that element x with features $\{f_1, f_2, ..., f_F\}$ belongs to class $C(x)$?
- Features are seen as the evidence for a claim
- Naive assumption gives us:

$$p(C(x)|(f_1, f_2, ..., f_F)) \propto p(f_1|C(x))p(f_2|C(x))...p(f_F|C(x))$$

- Estimate each $p(f_i|C_k)$ using a simple count from the training dataset

Simple example: Naive Bayes

- Suppose we have the following training set, bracketized:

Packets per second	Outgoing %	Data exfiltration attack
1∼10	25∼50	No
10∼100	50∼75	No
10∼100	75∼100	Yes
10∼100	75∼100	Yes
100∼1000	25∼50	Yes
100∼1000	50∼75	Yes

  Test input: 50 packets per second, 30% outgoing packets

- $p(10 \sim 100p/s, Yes) = 0.66, p(25\% \sim 50\%, No) = 0.5- > p(Yes) \propto 0.33$
- $p(10 \sim 100p/s, No) = 0.33, p(25\% \sim 50\%, No) = 0.5- > p(No) \propto 0.17$

  After weighing, the classifier believes there is a 67% chance this is an attack

Data sets

- We usually segregate our data into three data sets:
  1. Training
  2. Testing
  3. Validation: Used to evaluate classifier performance and adjust parameters **during training**. Avoids overfitting to the training set
- k-fold cross validation: Divide the data set into k folds. Repeat for i from 0 to $k-1$: In experiment i, use fold i for testing, fold $i+1/i-1$ for validation, and the rest for training
  - This allows us to set the testing set to be small

Binary classification

Binary classification is classification with two classes, a positive and a negative.

- Positives and negatives are defined by the context. In security, a negative is usually a benign event and a positive is a malicious event

Example

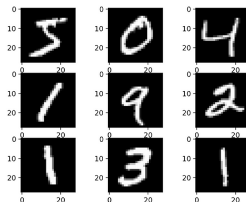
A packet for C&C communication (botnets) is positive.

- A correct classification is true, an incorrect classification is false
  - This gives us True Positives (TP), False Positives (FP), TN, and FN
- Be wary of class imbalance (more later): there are usually far more negatives than positives in security

Multi-class classification

Multi-class classification is classification with more than two classes.

- Often uses the **closed-world** assumption: there are no **out-of-distribution** elements
- Some classifiers are inherently binary and extended to multi-class classification using certain strategies:
  - One-vs-all: A classifier is trained for each class vs "the class of all other elements". When testing, the class of the classifier that scores the highest confidence is chosen
  - One-vs-one: A classifier is trained for each class vs each other class. When testing, a voting strategy is used to determine the class
- Famous visual computing problems: MNIST, CIFAR
  - MNIST: Recognize a handwritten numeral
  - CIFAR: Recognize a drawing of one of ten classes (automobile, cat, dog...)



Evaluating classifiers

- Multi-class: Accuracy, the chance that a classifier is correct
- Binary:
  - True Positive Rate (recall/sensitivity) and False Positive Rate

$$TPR = TP/P \qquad\qquad FPR = FP/N$$

  - P is the number of positives in the testing set and N is the number of negatives
  - TNR is also known as specificity
  - Precision: Usually given as $TP/(TP+FP)$
  - F1-score: Harmonic mean of recall and precision

Probability and thresholding

- Classifier outputs the probability of each class

Example

Neural networks frequently use softmax on the last layer, which take inputs $y_1, y_2, ..., y_K$ and output $\frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}}$

- This gives us an easy confidence thresholding mechanism to trade off recall for precision: Classifier rejects all classifications with probability less than threshold T (classify as negative event)
  - What does decreasing T give us?
- Recall/precision tradeoff can be expressed as AUC of the ROC

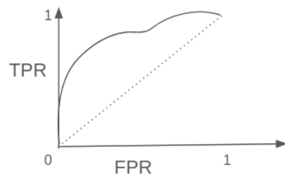


Figure: Receiver Operating Characteristic (ROC) curve. y-axis is TPR, x-axis is FPR. Dotted line is baseline.

Class imbalance and the Base Rate Fallacy

- Suppose a classifier is trained to distinguish images of cats and non-cats.
    - Used for automatic tagging on an image-uploading website
- Testing environment: $P = n$ cat images and $N = n$ non-cat images. Achieved TPR $= 0.9$, TNR $= 0.99$

$$\begin{aligned} \text{Precision} &= TP/(TP + FP) \\ &= 0.9n/(0.9n + (1 - 0.99)n) \\ &\approx 0.99 \end{aligned}$$

- In the above we used $TP = TPR \cdot P$, $FP = FPR \cdot N$

Class imbalance and the Base Rate Fallacy

- However, suppose only 1% of the images uploaded to this website are actually cats.
- Assuming TPR and TNR were correctly evaluated, we need to re-calculate real life precision.
- We can replace the above n non-cat images with $99n$ non-cat images:

$$\begin{aligned}\text{Precision} &= TP/(TP + FP) \\ &= 0.9n/(0.9n + (1 - 0.99)99n) \\ &\approx 0.47\end{aligned}$$

Class imbalance and the Base Rate Fallacy

- The base rate fallacy: ignoring base rate context when evaluating probability
- Seen in:
  - DNA testing: if a criminal DNA match has 99.999% TNR, does that prove whomever it matches is a criminal? How much? What about facial recognition?
  - Bird explains the replication crisis as a base rate fallacy [Bir21]
- Frequently occurs in security because of class imbalance: it is hard to know how unbalanced the classes are
  - How many attack packets are there in your average network? (How many can pass through a matching filter?)
  - Does a facial recognition system need very low FPR?

[Bir21]: Bird Alexander. Understanding the replication crisis as a base rate fallacy. British Journal for the Philosophy of Science, 2021

Objective function

- During training, classifier aims to minimize **loss**

Cross entropy (log-loss)

For element e_i with true class C_i with output probability p_i, log loss is

$$\mathcal{L}(e_i) = -(p_i \log(p_i))$$

- We sum up log loss for training elements
- An objective function for a search in parameter space $\mathcal{W}$ would be:

$$\arg\min_{w \in \mathcal{W}} \sum_i \mathcal{L}(e_i, w)$$

- In the above, w are the weights/learnable parameters
- Cross entropy is used almost universally in deep learning

Gradient descent

- To solve the objective function, we can use differentials:

$$\nabla_w \mathcal{L} = (\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, ..., \frac{\partial \mathcal{L}}{\partial w_W})$$

- ML classifiers are generally designed to ensure $\mathcal{L}$ is differentiable
- Then we take a step in the direction of minimizing loss:

$$w \leftarrow w - \nabla_w \mathcal{L} \cdot \eta$$

- η, the step size, is also called the learning rate
- **Stochastic gradient descent**: Compute $\nabla_w \mathcal{L}$ and take gradient step on a random training batch instead of the full set

Learning rate

- In learning, we need to avoid converging to **local minima** of the loss function
  - At local minima, $\nabla_w \mathcal{L} = 0$, so SGD does not step any further
- We can follow a defined learning rate schedule that starts with large learning rates and decreases it over time, e.g. exponential with hyperparameter d:

$$\eta_t = \eta_0 e^{-dt}$$

- These strategies are sensitive to hyperparemeter choice
- The same learning rate for different features may not be appropriate

Learning rate (ADAM)

ADAM: Adaptive Moment Estimation [Kin15]:

- Step size is based on sliding moment estimation:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_w \mathcal{L} \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_w \mathcal{L})^2$$

- β_1 and β_2 are hyperparameters (suggested $\beta_1 = 0.9$, $\beta_2 = 0.999$)
- In a (very common) degenerate case where gradient is frequently 0, the learning rate decays unnecessarily, so we correct for bias:

$$\hat{m}_t = m_t/\beta_1^t \qquad \hat{v}_t = v_t/\beta_2^t$$

- Now our learning step would be:

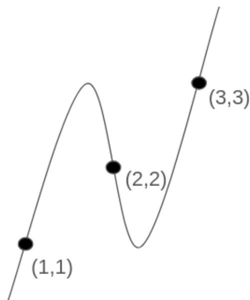
$$w \leftarrow w - \eta \hat{m}_t/(\sqrt{\hat{v}_t} + e)$$

- η and e are also hyperparameters

[Kin15]: Diederik P. Kingma, Jimmy Ba. *Adam: A Method for Stochastic Optimization. ICLR (Poster), 2015.*

Overfitting

- Classifiers of excessive dimension (complexity) gravitate towards complicated explanations of the training set
  - Poor generalization to unseen input
- Validation can help but does not eliminate overfitting
- Also known as the Hughes Phenomenon: increasing feature dimensions weakens the classifier after a point ("curse of dimensionality")



Overfitting

Approaches to mitigate overfitting are known as **regularization**, and include:

- Data Augmentation: Add slightly modified elements to the data set
- Reduce the complexity of the classifier; remove/shrink layers
- Dropout: During training, randomly set neuron weights to zero to force other neurons to learn
- Penalize large weights as part of the loss function:

$$\mathcal{L}(e_i, w) = -(p_i \log(p_i)) + R(w) \text{ where}$$
$$R(w) = w_1^2 + w_2^2 + ... + w_W^2$$

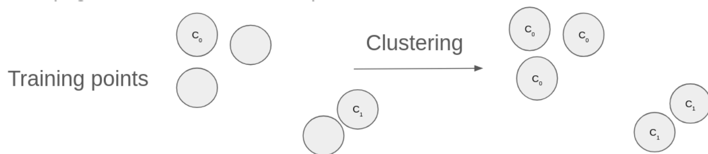
- Regularization terms can depend on the exact ML approach: e.g. in decision trees, penalize the number of leaves

Bias and variance

- **Bias**: How much the classifier fails to learn on the training data, i.e. underfitting. Measured with how well the classifier performs on the training set (training error).
- **Variance**: How much the classifier fails to extrapolate to the testing data, i.e. overfitting. Measured with how well the classifier performs on the testing set (generalization error).
- Training strategies often have to trade off the two

Domain adaptation

- Knowledge from training in one domain (with abundant training data) can be transferred to another domain (without enough data)
- **Covariate shift** describes a difference between testing set Y and training set X
- Some techniques are:
  - Reweighing classes to adapt to the new class balance
  - Transferring common features found in both sets
  - Semi-supervised learning: Collect a new training set from the distribution of Y, label them based on labels in X



Concept drift

- Over time, changes in the domain will weaken a classifier's performance
  - Significant issue in domains that change rapidly
- Mitigation methods:
  - Simplest/most costly: Regular re-training with new data
  - Detection of drift with statistical analysis, re-train when drift is too large
- Domain adaptation could also work
- Can you defeat these methods as an adversary?

Learning theory

Learning theory is concerned with proving statements about how machines learn. Some results include:

- VC-theory: Dimensionality of the classifier can be used to upper bound its test error relative to its training error
  - More complex classifiers have a higher bound
- Stability: Certain classifiers can be proven to be stable, meaning we can bound how much its learning deviates with changes in the data set
- Convergence: Certain classifiers can be proven to converge to an optimal solution (for the training set)
- PAC-learning: There are deep connections between learning (i.e. choosing the best function that minimizes generalization error) and computational complexity theory

Machine Learning (Part 2): Non-NN Classifiers

Multinomial Naive Bayes

- Original Naive Bayes: Calculates class probability independently from occurrences of discretized feature sets
  - e.g. Instance format: {color = "orange", edible = "yes"}
  - Doesn't work well with numerically valued features; can have null features
- Multinomial: use **frequencies** of occurence as well, suppose feature x_i occurs f_i times:

$$p(C_k|((x_1, f_1), (x_2, f_2), ..., (x_k, f_k))) \propto p(x_1|C_k)^{f_1} p(x_2|C_k)^{f_2} ... p(x_k|C_k)^{f_k}$$

- Used in text classification (e.g. information retrieval, sentiment analysis) [Kib04]
  - Each document is an unordered "bag of words", e.g. word1: freq1, word2: freq2, ...

[Kib04]: Kibriya, A. M., Frank, E., Pfahringer, B., & Holmes, G. "Multinomial naive bayes for text categorization revisited." Australasian joint conference on artificial intelligence, 2004.

Multinomial Naive Bayes

Useful techniques to enhance classification include:

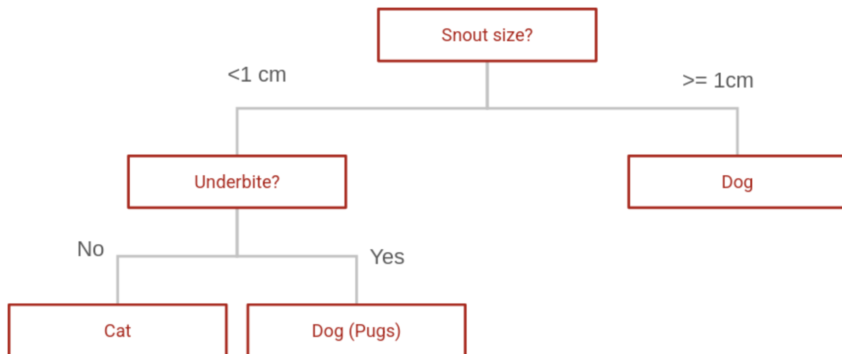
- Laplace smoothing: Start each feature count at 1
- TF-IDF: instead of raw frequencies f_i, instead use:

$$\hat{f}_i = \frac{f_i}{\sum_i f_i} \cdot \log(\frac{N}{N_{x_i}})$$

- N is the total number of training elements, and N_{x_i} is the number of training elements in which x_i appears at least once

Decision trees

- Features are numerical values, $\{f_1, f_2, ..., f_F\}$
- Each node is a rule branching on if a certain feature f_i is less than or greater than a threshold T
- Classification of an element starts at the root and applies branching rules until it reaches a class



Decision trees

Training:

- Suppose the probabilities of each class at a node is p_i for class C_i, calculated from the training set
  - Gini impurity of the node: $1 - \sum p_i^2$
  - Shannon entropy of the node: $1 - \sum p_i \log(p_i)$
- Compare the sum Gini impurity/entropy of the child nodes to its parent node; the more impurity lost/information gained, the better
- We can evaluate all possible feature-threshold pairs exhaustively, or randomly

It is easy to achieve a decision tree with low bias and high variance.

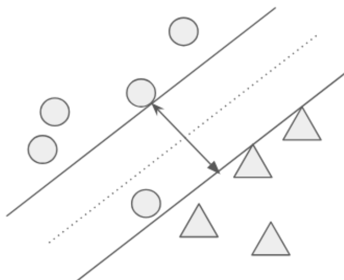
Random forests

An **ensemble** is multiple classifiers working together. A forest is an ensemble of decision trees.

- Each decision tree is trained with a **bag** (random selection) of the training set
- When evaluating splits, at each node only a random selection of the feature space is available to the tree (random subspace)
- Threshold choice is also randomized
- Has desirable theoretical properties: bounds on generalization error, resistance to noise
- Loses interpretability of decision trees

Support Vector Machines

- Formulated as finding the maximal separator between two classes of elements in a feature space



- The separator is written as $w \cdot x + b = 0$
- One of the dominant classifiers (together with random forests) before NN prevalence

Support Vector Machines

- The gap between the two classes can be found to be $2/||w||$, so we have the objective:

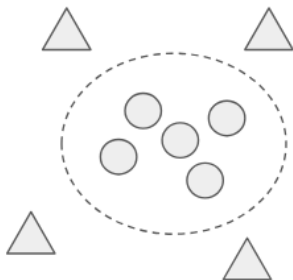
$$\arg\min_{w,b} ||w||^2 \text{ s.t. } C(x_i)(w \cdot x_i + b) \geq 1$$

- In the above, $C(x_i)$ is the class, either 1 or -1
- This is a convex objective function with a single minimal solution
- In practice, the above suffers from overfitting; we instead use a **soft margin** rule by adding a cost hyperparameter C and calculate how much each training sample lies on the wrong side of the separator as ζ_i:

$$\arg\min_{w,b} (||w||^2 + C \sum \zeta_i) \text{ s.t. } C(x_i)(w \cdot x_i + b) \geq 1 - \zeta_i$$

Support Vector Machines

- SVMs described as above are **linear** classifiers, but can be extended to nonlinear classification using a transformation
- Consider the problem of finding an elliptical separator:



1. Transform each point (x, y) into $(z_1 = x_1^2, z_2 = x_2^2)$
2. Find the linear separator on 2-dimensional space: $w \cdot z + b = 0$, which can be easily transformed into the ellipsis $\frac{x_1^2}{a_1^2} + \frac{x_2^2}{a_2^2} = 1$

Support Vector Machines

- The **kernel trick** is to without having to explicitly define the transformation by observing that x_i are only used in a dot product:

$$w \cdot x_i$$

- We replace the above with a kernel $k(w, x_i)$, which can be:
  - Polynomial kernel: $k(x, y) = (x \cdot y + r)^d$
  - Gaussian/radial kernel: $k(x, y) = e^{-\gamma ||x-y||^2}$
  - Sigmoid kernel: $k(x, y) = \tanh(ax \cdot y + c)$
- Gaussian and tanh kernel have infinite dimension, and can capture arbitrary curves in a lower dimension
- SVMs suffer from computational complexity with big data

k Nearest Neighbors

- Simple algorithm: Classify based on the k nearest neighbors in the training set
- k neighbors vote; vote can be weighed based on distance
- No training involved, but testing is slow: naive approach requires distance computation with all N training points
  - Optimization can be done with tree-based structure to compare with only $O(\log N)$ points
- Resistance to noise (with large k)

Ensemble learning

Ensemble learning combines many weak learners into one strong learner.

- We saw Bagging (bootstrap aggregating) in random forests: Each weak learner (decision tree) is trained (in parallel) with a subset of the training set **sampled with replacement**
- Boosting: Weak learners are trained sequentially. Mis-classified elements in each learner are given higher weights for the next successive learner
  - Examples: AdaBoost, XGBoost
  - AdaBoost (1995) creators Freund and Schapire won the Gödel prize in 2009
  - Easily overfits if too many learners are used
- Ensemble learning usually chooses decision trees as the weak learner

Ensemble learning

To combine decisions for an ensemble:

- Simple voting: Majority count of weak learners. In binary classification, can use thresholding for bias-variance tradeoff
- **Weighted voting**: Each weak learner's vote is weighed by a parameter α. Weights for each weak learner are learned during training.
  - Training on ML outputs is known as stacked learning/meta-learning
- Gradient tree boosting: learning weighted voting on decision trees
  - AdaBoost is a common algorithm
  - XGBoost is a popular library that implements a number of algorithms

AdaBoost (Adaptive Boosting)

- Classifier decision is a sum of weak classifiers k_i with weights α_i:

$$M_n(x_i) = \alpha_1 k_1(x_i) + ...\alpha_n k_n(x_i)$$

- We determine the weights iteratively as follows. Suppose we want to extend to M_{n+1} by adding α_{n+1} and k_{n+1}, we:
  1. Calculate the sum error of the machine on all instances as:

$$Err(M_{n+1}) = \sum_{i=1}^{N} w_i^{n+1} e^{-C(x_i)\alpha_{n+1}k_{n+1}(x_i)}$$

  2. In the above, w_i^{n+1} is how poorly the current classifier classifies x_i:

$$w_i^{n+1} = e^{-C(x_i)M_n(x_i)}$$

  3. It can be shown that k_{n+1} that minimizes $\sum_{C(x_i)\neq k_{n+1}(x_i)} w_i^{n+1}$ will minimize the total error, so we train k_{n+1} with that objective
  4. After training k_{n+1} as above, we then determine α_{n+1} as:

$$\alpha_{n+1} = \frac{1}{2} \ln \frac{\sum_{C(x_i)=k_{n+1}(x_i)} w_i^{n+1}}{\sum_{C(x_i)\neq k_{n+1}(x_i)} w_i^{n+1}}$$

XGBoost [Chen2016]

Gradient tree boosting with focus on scalability to billions of examples.

- In general, we can use a second order approximation to minimize the loss function:

$$\mathcal{L}(C(x_i), M_{n+1}(x_i)) = \mathcal{L}(C(x_i), M_n(x_i)) + g_i k_{n+1}(x_i) + \frac{1}{2} h_i (k_{n+1}(x_i))^2$$

- where g_i and h_i are the first and second order moments of the loss function gradient, $g_i = \partial\mathcal{L}(C, M_n)(x_i)/\partial x_i$
- XGBoost stores information in blocks to enable parallelization with randomized split finding
- XGBoost also uses regularization term to punish total weight as well as number of leaves

---

[Chen2016]: Chen, T., & Guestrin, C. "XGBoost: A scalable tree boosting system." Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, 2016.