COMP0078 Supervised Learning cheat sheet

Timuel the Great

February 2019

Contents

Introduction

Cheat sheet for the COMP0078 Supervised Learning exam. This cheat sheet is meant to be used in preparation for the exam, **not during** the exam. Expected background knowledge:

- Probability (Bayes rule, conditional probability, expectation, random variables, basic combinatorics).
- Linear Algebra (singular value decomposition, positive semi-definite, positive definite, rank, linear systems of equations).
- Misc: convexity, boolean functions (and, or, not, conjunctive normal form, disjunctive normal form, conjunction, disjunction).

Terminology

- **m** is the size of the training set.
- **n** is (usually) the number of dimensions in the input.
- $\hat{y}_i$ is the predicted output for input x_i.
- S is the training set.
- $\mathcal{E}$ is the expected error of a learning algorithm. $\mathcal{E}_{\text{emp}}(S, f)$ is the empirical error of an estimator f on training set S. When target (labels) are categorical, the error is measured as **error rate** - the proportion of cases where the prediction is wrong.
- MSE stands for mean square error.
- **i.i.d.** stands for Independent and Identically Distributed
- $\mathcal{I}\left[x\right]$ is the **indicator function**, $\mathcal{I}\left[x \in R_n\right] = 1$ if $x \in R_n$, and 0 otherwise.
- $\mathcal{H}$ is the hypothesis space

1 Introduction to supervised learning

1.1 Supervised learning model

Given training data, the (pattern, label) pairs $S = \{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$, infer a function f_s such that: $f_s(x_i) \approx y_i$ for future data $S' = \{(x_{m+1}, y_{m+1}), (x_{m+2}, y_{m+2}), ...\}$.

Supervised learning problem: compute a function which "best describes" I/O relationship

1.2 Learning algorithms

Given training set $S = \{(x_i, y_i)_{i=1}^m \subseteq \mathcal{X} \times \mathcal{Y}\}$, a learning algorithm is a mapping $S \to f_s$. The new input x is predicted as $f_s(x)$. Example algorithms are linear regression, neural networks, decision trees, support vector machines.

Generalization error tells us how well f_s performs on new data. Computational complexity and sample complexity tells us how complex the learning task is.

1.3 Least squares

Least squares is often used in linear regression. Least squares solution gives us the parameters that minimize the square error in our predictions, i.e.:

$$\operatorname{argmin}_w \left[\mathcal{E}_{\text{emp}}(S, \mathbf{w})\right] = \operatorname{argmin}_w \left[\sum_{i=1}^{m} (y_i - \hat{y}_i)^2\right] = \operatorname{argmin}_w \left[\sum_{i=1}^{m} (y_i - f(x_i, w))^2\right]$$

.

Note that this minimization problem is the same as minimizing MSE. For a linear predictor $\hat{y} = w \cdot x$, the least squares solution can be derived from the following relationship: $X^T X w = X^T y$. This relationship is established by taking the derivative of $\mathcal{E}\text{emp}(S, \mathbf{w})$ and setting it to zero, then solving for **w**. If matrix $X^T X$ is invertible, the solution is unique: $w = (X^T X)^{-1} X^T y$. Otherwise, the solution is not unique and we need to use some other method to solve the system

1.4 Bias term and normal equations

Bias term (mostly for linear regression) lets us achieve better fit for the data that doesn't go through origin. Without the bias term, our linear estimator f will be forced to go through the origin, which might not represent the data realistically.

Normal equations represent a way to compute coefficients for (i.e. *solve*) linear regression. Normal equations are used to derive the coefficients analytically. They produce the solution in one step (as opposed to iterative algorithms like gradient descent).

1.5 k-NN

k-NN determines the label of input x_i by taking a local majority vote from its neighbours (local rule). Closeness is measured by some metric, usually (but not always) the Euclidean norm. Decision boundary is non-linear! Smaller k results in a more irregular decision boundary.

Optimality of 1-NN (aka Cover's bound)

The error rate for 1-nn as number of samples goes to infinity is no more than twice the Bayes error rate.

Optimality of k-NN

One can show that optimality for k-NN, i.e. $\mathcal{E}(k(m)\text{-NN}) \to \mathcal{E}(f^*)$, is achieved as $m \to \infty$ given:

1. $k(m) \to \infty$
2. $\frac{k(m)}{m} \to 0$

1.6 Optimal supervised learning (Bayes estimator and classifier)

We assume that data is sampled i.i.d. from some **fixed but unknown** probability distribution $P(x, y)$. The expected error is then:

$$\mathcal{E}(f) := \mathbb{E}\left(y - f(x))^2\right] = \int (y - f(x))^2 dP(y, x)$$

The goal is to minimize $\mathcal{E}$. The optimal solution, $f^* = \mathrm{argmin}_f \mathcal{E}(f)$, is called the **Bayes estimator**. In most real-world problems, we can't compute the Bayes estimator f^* since we don't know the true distribution $P(x, y)$.

A Bayes estimator for a particular problem can be derived by minimizing the expected error for a fixed value of $x = x'$, which fixes $z = f(x')$. Then you do the usual minimization - take derivative, set it to 0, solve for z.

For C-class classification, the Bayes estimator is $f^*(x) = \mathrm{argmax}_{c \in \{1,\ldots,C\}} P(Y = c \mid x)$. The Bayes error rate (which is optimal) then becomes:

$$\int (1 - \Pr(Y = f(x) \mid x)) \, d \Pr(x)$$

.

1.7 Bias and variance

Let $f^*(x)$ be the optimal estimator, let $A(x)$ be our estimator, and (x', y') be the data point. Then **bias** is $f^*(x') - E[A(x')]$, i.e. the discrepancy between our algorithm and the truth. **Variance** is $E\big[(A(x') - E[A(x')])^2\big]$, i.e. how the performance of our algorithm differs between data sets (variance wrt. different data sets).

Bias and variance tend to trade-off against one another. In general, more parameters means better fit to the data, hence less bias, but more variance from over-fitting. Examples: Linear regression is high bias low variance, while k-NN is low bias high variance.

Appears on: 17/18 Q1(a);

1.8 NFL theorems

Several No Free Lunch (NFL) theorems exist showing that no machine learning algorithm can perform as well as the Bayes estimator in the general case. One example is from [Shales-Shwartz and Ben-David 2014, Section 5.1], which states that for binary classification with labels $\mathcal{Y} = \{0, 1\}$ and training set size $m < |\mathcal{X}|/2$, there exist some distributions $P(x, y)$ such that there exist a function $f : \mathcal{X} \to \mathcal{Y}$ with Bayes error 0, but with probability at least $1/7$ w.r.t. to a random draw of examples S from P, of size m, the prediction of $(m + 1)$st example will be wrong with probability at least $1/8$. In other words, if the $(m + 1)$st draw is (x', y'), we have $\Pr(A_s(x') \neq y') \geq 1/8$.

1.9 Hypotheses space

Hypotheses space is a space of "solutions" we consider, e.g. the set of potential functions we can use to solve a particular problem. Defining a hypotheses space can help avoiding the curse of dimensionality, as it limits our search space. We can assign each hypothesis a complexity, and then prefer “simpler” hypothesis.

In real-world problems, we can’t compute expected error $\mathcal{E}$ (since true distribution is not known) hence we approximate it with empirical error $\mathcal{E}_{emp}$. Note that we could, in theory, achieve $\mathcal{E}_{emp}$ by iterating over all possible estimator functions, but this would be too expensive. Instead, we limit our search to some hypotheses space $\mathcal{H}$.

This way, our problem becomes $\mathrm{argmin}_{f \in \mathcal{H}} \mathcal{E}_{emp}(S, f)$, which is often called empirical error (risk) minimization. Setting $\mathcal{H}$ to be too large can result in overfitting, while setting $\mathcal{H}$ too small results in underfitting.

1.10 Model selection and cross-validation

Model selection refers to an approach you use to select a model from the hypotheses space. Some examples are K-fold cross-validation, Bayesion model selection via evidence, structure risk minimization.

K-fold cross-validation is good in practice. Cross-validation error is computed by averaging the error over K folds. Small K are easier to compute but give less reliable results. m-fold validation is called leave-one-out (LOO) validation.

2 Kernels and regularization

2.1 Inner product/vector/normed space

TODO. Revise these from the slides.

2.2 Ill-posed problems

A problem is well-posed if: (1) a solution exists, (2) the solution is unique, (3) the solution depends continuously on data.

If solution is not well-posed, it is ill-posed. Learning problem are generally ill-posed, due to (2). Regularization theory provides a general framework for ill-posed problems.

2.3 Ridge regression

Ridge regression is an example of regularization. It is similar to linear regression but it adds a complexity term to the error function. This complexity term, e.g. ... $+ \lambda ||w||_2^2$, penalizes the solution for “overly complex” predicted weights, thus avoiding overfitting. Ridge regression normal equations look like $-2X^T(y - Xw) + 2\lambda w = 0$, giving us the regularized solution

$$w = (X^T X + \lambda I_n)^{-1} X^T y$$

.

2.4 Dual representation

We consider dual representation of regularized linear regression (Ridge regression). When $n \gg m$, dual representation is faster to compute during the training phase. Dual representation is even faster if the input is sparse.

Primal form (functional representation)

Primal form is the standard form we're familiar with - it learns the coefficients in one step, $w = (X^T X + \lambda I_n)^{-1} X^T y$, and then uses them to make predictions, $f(x) = w \cdot x$.

Training (solving for **w**) requires $O(mn^2 + n^3)$ operations. Testing requires $O(n)$ operations.

Dual form

Dual form does not compute w explicitly, but rather breaks it down by forming a Lagrangian $L(x, y, \alpha) = f(x, y) - \alpha g(x, y)$. It learns a vector of parameters $\alpha = (\alpha_1, ..., \alpha_m)^T = (XX^T + \lambda I_m)^{-1} y$. Then, to make predictions, it uses the formula $f(x) = \sum_{i=1}^m \alpha_i x_i^T x$, which has to iterate over all inputs x_i from the training set.

Training (solving for **w**) requires $O(nm^2 + m^3)$ operations. Testing requires $O(mn)$ operations.

The dual problem will find the highest lower bound (infimum) for the primal problem in general. The difference between the lower bound and the solution in primal form is called "duality gap". When the problem is convex (convexify using regularization) then the duality gap is 0.

Use Primal when n << m, use Dual when m << n.

2.5 Feature maps

The ideas from dual representation can be generalised to nonlinear function regression in the form of feature maps. A **feature map** is some function $\phi : \mathbb{R}^n \to \mathbb{R}^q$ that maps the input vector x to some feature vector $\phi(x)$, where q can be chosen arbitrarily. In other words,

$$\phi(x) = (\phi_1(x), \phi_2(x), ..., \phi_q(x))^T, \quad x \in \mathbb{R}^n$$

The $\phi_i(x)$, $i = 1, ..., q$, above are called **basis functions**. Basis functions define a feature map explicitly. The space containing all possible feature vectors, $\{\phi(x) : x \in \mathbb{R}^n\}$, is called the feature space.

Non-linear regression then has primal representation

$$f(x) = \sum_{j=1}^{q} w_j \phi_j(x) = \langle w, \phi(x) \rangle$$

However, same as before, if $q \gg m$, using the dual representation is more efficient (see next section).

2.6 Kernels

Note that in dual representation for non-linear regression, we don't need to know the feature map ϕ explicitly. Kernels can be considered to be implicit feature maps. Given a feature map ϕ, the associated kernel function $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is given by:

$$K = \langle \phi(x), \phi(t) \rangle, \quad x, t \in \mathbb{R}^n$$

The advantage is that $K(x,t)$ in some cases is independent of q, and hence is faster to compute than computing $\phi(...)$ explicitly. Note that different feature maps can generate the same kernels, and thus feature maps are not unique.

2.6.1 Representer theorem

Representer theorem basically shows that dual form of ridge regression also holds for other loss functions. The theorem itself is shown below.

Let $V : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be some loss function, not necessarily linear. If V is differentiable w.r.t. its second argument and **w** is a minimizer of $\mathcal{E}_{emp_\lambda}$, then it has the form

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i \phi(x_i) \ \Rightarrow \ f(x) = \langle w, \phi(x) \rangle = \sum_{i=1}^{m} \alpha_i K(x_i, x)$$

TODO: Add concise proof of Representer Theorem.

2.6.2 Kernel definition

A function $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is positive semidefinite (PSD) if it is symmetric and matrix defined by $m_{ij} = K(x_i, x_j)$ is positive semidefinite for every $k \in \mathbb{N}$ and every $x_1, ..., x_k \in \mathbb{R}^n$. A symmetric matrix is positive semidefinite if and only if its eigenvalues are non-negative.

More importantly, K is positive semidefinite iff

$$K(x,t) = \langle \phi(x), \phi(t) \rangle, \quad x, t \in \mathbb{R}^n$$

for some feature map $\phi : \mathbb{R}^n \to \mathcal{W}$ and some Hilbert space $\mathcal{W}$. Therefore a function $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is a kernel if it is positive semidefinite.

A kernel is **translation invariant** if it has the form $K(x,t) = H(x-t)$, $x, t \in \mathbb{R}^d$ where $H : \mathbb{R}^d \to \mathbb{R}$ is a differentiable function. A kernel is **radial** if it has the form $K(x,t) = h(\|x-t\|)$, where $h : [0, \infty) \to [0, \infty)$ is a differentiable function.

2.6.3 Kernel construction

If K_1, K_2 are kernels, $a \geq 0$, A is a symmetric positive semidefinite matrix, K is a kernel on $\mathbb{R}^q$ and $\phi : \mathbb{R}^n \to \mathbb{R}^q$ then the following functions are positive semidefinite kernels on $\mathbb{R}^n$:

1. $x^T A t$

2. $K_1(x,t) + K_2(x,t)$

3. $aK_1(x,t)$
4. $K_1(x,t) * K_2(x,t)$
5. $K(\phi(x), \phi(t))$

2.6.4 Polynomial kernel and Anova kernel

Polynomial kernel is expressed as:

$$K_p(x,t) = (1 + x^T t)^d$$

While Anova kernel is:

$$K_a(x,t) = \prod_{i=1}^{n} (1 + x_i t_i)$$

Note that Anova kernel has a much simpler feature map, which is basically multiplying all possible combination of variables together. One can show this by expanding the product in the Anova kernel.

2.6.5 Polynomial of kernels

By properties, 2, 3 and 4 from the list above we can show that the following function will make a valid kernel:

$$p(K(x,t)) = \sum_{i=1}^{d} a_i \left(K(x,t)\right)^i$$

This implies that if $p : \mathbb{R} \to \mathbb{R}$ is a polynomial with non-negative coefficients, then $p(x^T t)$ is a a valid kernel on $\mathbb{R}^n$.

Important: The infinite polynomial kernel shown below uniformly converges to the expression shown on the right below:

$$\sum_{i=1}^{r} \frac{a^i}{i!} (x^T t)^i \quad \longrightarrow^{r=\infty} \quad e^{a*\langle x,t \rangle}$$

This shows that the latter is also a valid kernel. We use that to motivate Gaussian kernels and other kernels of that form (see next section). **This result is useful when it comes to proving if a particular function is a valid kernel.**

2.6.6 Gaussian kernel

Gaussian kernel has the following form:

$$K(x,t) = exp(-\beta \left\| x - t \right\|^2), \quad \beta > 0, \ x, t \in \mathbb{R}^d$$

It is a valid kernel because it is a product of two other $exp(...)$ kernels. This can be shown by rewriting the square of the norm as an inner product and expanding it.

3 Tree-based learning algorithms and Boosting

Tree-based learning methods work by partitioning the input space into a set of hyperrectangles and fitting a simple model (a constant) to each rectangle. The estimator can then be expressed as follows:

$$f(x) = \sum_{n=1}^{N} c_n [x \in R_n]$$

where R_n represents a particular rectangle and $\{c_n\}_{n=1}^{N}$ are some real parameters. Note also that $\cup_{n=1}^{N} R_n = \mathbb{R}^d$ and $R_a \cap R_b = \emptyset$ if $a \neq b$.

One natural choice for c_n is $c_n = average(y_i \mid x_i \in R_n)$, i.e. the average output from all inputs in that rectangle.

3.1 Classification and regression trees (CART)

3.1.1 Recursive binary partition

This method of partitioning first splits the entire input space into two parts, then breaks each of the 2 parts into halves, and so on. It's simple to express and can be easily interpreted. It suffers from **risk of overfitting**, as enough splits can fit any arbitrary data.

3.1.2 Regression trees

TODO. Include the "ideal" (and intractable) formulation of the problem

Regression trees work by building recursive binary partitions one step at a time. At each level, the algorithm picks a dimension $j \in \{1, ..., d\}$ (corresponding to jth element of input x) and a value at which the split will occur, s. **The optimisation problem** that determines j and s looks as follows:

$$\text{argmin}_{j,s} \left(\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right)$$

The internal minimization is solved by simply setting c_i to the average of values in the rectangle created by the split at s.

For each variable j the best split can be found in $O(m)$ computations (since the split can occur between any two points), meaning that one iteration of the problem is solved in $O(dm)$ steps.

How do we know when to stop growing a tree? Using a fixed tree depth is a bad idea since it can underfit or overfit depending on the data, and stopping after particular accuracy improvement threshold might give bad results since accuracy improvement at every step is unpredictable.

A good approach is to first generate a big tree $\hat{T}$ (stopping when each node has some minimum number of data points assigned to it, e.g. 5), then consider smaller subtrees $T_\lambda \subseteq \hat{T}$ and pick the "best" one.

Define $Q_n(T)$ as the **node impurity measure** (i.e. "training error" for node n). For regression tree, $Q_n(T)$ is the square error of prediction made by the hyperrectangle associated with that node. The formula is:

$$Q_n(T) = \frac{1}{m_n} \sum_{x_i \in R_n} (y_i - c_n)^2$$

where m_n is the number of data points assigned to node n. The best subtree is then picked using **cost-complexity pruning**, i.e. we choose T_λ that minimises

$$C_\lambda(T) = \sum_{n=1}^{|T|} m_n Q_n(T) + \lambda |T|$$

λ determines how much we penalize "complex" trees, and bigger λ will result in smaller trees. It can be shown that there is a unique T_λ that minimizes C_λ.

One possible way of finding T_λ is **weakest link pruning**, which involves exhaustively collapsing internal nodes that produce smallest per-node increase in $\sum_{n=1}^{|T|} m_n Q_n(T)$ and computing C_λ after each collapse. The intermediate subtree with smallest C_λ is then guaranteed to be the optimal subtree.

As final steps, one would compute best value for λ using cross-validation, and then retrain on all data and prune the resulting tree to find T_λ.

3.1.3 Classification trees

Classification trees are similar to regression trees, except instead of just constants for each rectangle we compute the probability that $x_i \in R_n$ belongs to a particular class $k \in \{1, ..., K\}$. This probability can be expressed as $p_{nk} = \frac{1}{m_n} \sum_{x_i \in R_n} \mathcal{I}[y_i = k]$. Our prediction is then the class with highest probability:

$$f(x) = \mathrm{argmax}_{k \in \{1,...,K\}} \sum_{n=1}^{N} p_{nk} \mathcal{I}[x \in R_n]$$

We also use different **node impurity measures** $Q_n(T)$. To grow the tree, we usually use Gini index, $\sum_k p_{nk}(1 - p_{nk})$, or cross-entropy, $\sum_k p_{nk} \log \frac{1}{p_{nk}}$, since they are more sensitive to changes in probabilities. To prune the tree, we often use misclassification error, which is the probability that the input does not belong to the class we labelled it with.

3.2 Ensemble methods

Ensemble methods rely on the principle of "wisdom of crowds". If each member of the crowd is correct with probability $p > 1/2$, the probability that their combined vote is wrong **exponentially decays** to zero as crowd size grows.

3.3 Bagging

Bagging decreases the variance of a classifier by training multiple instances of the classifier on different subsets of the data, and then producing a prediction by combining the "votes" of all classifiers.

A classifier that uses bagging can be expressed as

$$H(x) := sign\left(\sum_{t=1}^{T} h_{S(t)}(x)\right)$$

where $S(t)$ is a set of M examples sampled **with replacement** from the training data and $h_{S(t)}(x)$ is the same classifier trained on that set.

Usually people set $M = m$ - since we're sampling with replacement, the set $S(t)$ will contain some duplicate data points and hence the condition $S(t) \cap S = S$ is **not** guaranteed. The probability that a particular example is not included in $S(t)$ is $(1 - \frac{1}{m})^m = \frac{1}{e} \approx 0.368$, which means that there are about 63% of distinct examples in each $S(t)$.

3.3.1 Random forests

Since individual predictors in "wisdom of crowds" need to be independent, decision trees are ideal for bagging even though they have high variance. To de-correlate the trees, during generation of each tree we only consider a subset of all input features. The size of that subset is usually set to $\sqrt{d}$ or $\log d$, which works well in practice.

3.4 Boosting

A **weak learner** is a learner that can predict the output slightly better than random guessing, i.e. with accuracy 55%. Boosting is a method of combining week learners to produce a highly accurate classifier.

In general, boosting works by iteratively training a collection of weak learners, then combining them using a majority vote to determine the final prediction. At each time step t, we update the distribution $D_t(i)$, in which the "hardest" examples have high probability. An example (data point) is "hard" if it has been misclassified a lot by classifier in the previous steps.

3.5 Adaboost

In Adaboost, we walk through $t = 1, ..., T$ iterations, and since we create a new classifier at every iteration, we can also use t as the index of the classifier. $D_t(i)$ denotes the distribution of "hard" examples at step t. It's initialised to uniform distribution since we know nothing about the data. ϵ_t then denotes the *weighted* training error of classifier t:

$$\epsilon_t = \sum_{i=1}^{m} D_t(i)\, \mathcal{I}\left[h_t(x_i) = y_i\right]$$

Let S be our training set, with labels $y_i \in \{-1, +1\}$. The Adaboost algorithm is described below.

1. We fit a classifier $h_t : \mathbb{R}^d \in \{-1, +1\}$ (from some previously established hypotheses space $\mathcal{H}$).

2. We set $\alpha_t = \frac{1}{2} \log \frac{1-\epsilon_t}{\epsilon_t}$ (it could be another value, but this one works well for our purposes).

3. We update the distribution of hard examples by running the following update function for every data point i:

$$D_{t+1}(i) = \frac{D_t(i)e^{-\alpha_t y_i h_t}}{Z_t}$$

. Z_t is just the normalization constant that makes sure our distribution adds up to 1.

4. We repeat the previous 3 steps T times, and then output the final classifier as:

$$f(x) = \sum_{t=1}^{T} \alpha_t h_t(x) \quad \Rightarrow \quad H(x) = sign\left(f(x)\right)$$

Note that usually $\epsilon_t < 0.5$ (i.e. weak learners perform better than random guess, as expected). Note also that if a weak learner h_t has error rate above 50%, it's counterpart, $-h_t$, will have accuracy above 50%.

Since $\epsilon_t < 0.5$, we have $\alpha_t > 0$ and thus our final classifier $H(x)$ is just a linear combination of classifier h_t, $t = 1, ..., T$, with weights controlled by the error.

3.5.1 Exponential convergence of training error

Denote empirical error on the training set S as

$$\mathcal{E}_S(H) := \frac{1}{m}\sum_{i=1}^{m} \mathcal{I}\left[H(x_i) \neq y_i\right]$$

Assume each weak learner h_t achieves weighted training error $\epsilon_t < 1/2 - \gamma_t$, where $\gamma_t \in (0, 0.5]$. Now pick γ such that $0 < \gamma \leq \gamma_t$ for all t. We can show that the error of final classifier drops exponentially fast:

$$\mathcal{E}_S(H) \leq exp(-2\gamma^2 T)$$

We can prove this in several steps:

1. Let D_{T+1} be the final "hardness" distribution after the algorithm terminates. If we unroll the recursive update formula of $D_t(i)$, we can write it explicitly as:

$$D_{T+1}(i) = \frac{1}{m}\frac{\prod_{t=1}^{T} e^{-\alpha_t y_i h_t(x_i)}}{\prod_{t=1}^{T} Z_t} \quad \Rightarrow \quad D_{T+1}(i)\left(\prod_{t=1}^{T} Z_t\right) = \frac{1}{m}\prod_{t=1}^{T} e^{-\alpha_t y_i h_t(x_i)}$$

We can now simplify the result on the right further using the fact:

$$\begin{aligned} D_{T+1}(i)\left(\prod_{t=1}^{T} Z_t\right) &= \frac{1}{m}\prod_{t=1}^{T} e^{-\alpha_t y_i h_t(x_i)} \\ &= \frac{1}{m} e^{-y_i\left[\sum_{t=1}^{T}\alpha_t h_t(x_i)\right]} \qquad \text{because } \prod e^a = e^{\sum a} \\ &= \frac{1}{m} e^{-y_i f(x_i)} \end{aligned}$$

Now we sum each side over all data points and simplify further:

$$\sum_{i=1}^{m}\left[D_{T+1}(i)\left(\prod_{t=1}^{T}Z_t\right)\right]=\sum_{i=1}^{m}\left[\frac{1}{m}e^{-y_i f(x_i)}\right]$$
$$\Downarrow$$
$$\left(\prod_{t=1}^{T}Z_t\right)\sum_{i=1}^{m}D_{T+1}(i)=\frac{1}{m}\sum_{i=1}^{m}e^{-y_i f(x_i)}$$
$$\Downarrow$$
$$\prod_{t=1}^{T}Z_t=\frac{1}{m}\sum_{i=1}^{m}e^{-y_i f(x_i)}$$

2. Note that $H(x_i) \neq y_i \;\Rightarrow\; sign(y_i) \neq sign(f(x_i)) \;\Rightarrow\; y_i * f(x_i) = -1 \;\Rightarrow\; e^{-y_i f(x_i)} \geq 1$. Thanks to this result, and the result from step 1, we can bound $\mathcal{E}_S(H)$ from above:

$$\mathcal{E}_S(H) \leq \frac{1}{m}\sum_{i=1}^{m}e^{-y_i f(x_i)} \quad \Rightarrow \quad \mathcal{E}_S(H) \leq \prod_{t=1}^{T}Z_t$$

.

3. Based on result in step 2, we can say that if aim to minimize Z_t at every time step, we can very quickly minimize the training error. We know that if prediction is correct, we'll have $y_i h(x_i) = 1$. If prediction is wrong, we have $y_i h(x_i) = -1$. We can use that to simplify the formula for Z_t:

$$\begin{aligned} Z_t &= \sum_i D_t(i)e^{-\alpha_t y_i h_t(x_i)} \\ &= \sum_{i \in wrong} D_t(i)e^{\alpha_t} + \sum_{i \in correct} D_t(i)e^{-\alpha_t} \\ &= e^{\alpha_t}\sum_{i \in wrong} D_t(i) + e^{-\alpha_t}\sum_{i \in correct} D_t(i) \\ &= e^{\alpha_t}\,\epsilon_t + e^{-\alpha_t}\,(1-\epsilon_t) \end{aligned}$$

Solving the final result above for $\frac{dZ_t}{d\alpha_t} = 0$ gives us the desired value of α:

$$\alpha = \frac{1}{2}\log\frac{1-\epsilon_t}{\epsilon_t}$$

4. Recall that each learner h_t achieves training error $\epsilon_t = 1/2 - \gamma_t$. We can plug this expression

and the final result from previous step into the expression for Z_t:

$$\begin{aligned}
Z_t &= e^{\alpha_t}\ \epsilon_t + e^{-\alpha_t}\ (1-\epsilon_t) \\
&= e^{\frac{1}{2}\log\frac{1-\epsilon_t}{\epsilon_t}}\ \epsilon_t + e^{-\frac{1}{2}\log\frac{1-\epsilon_t}{\epsilon_t}}\ (1-\epsilon_t) \\
&= e^{\log\sqrt{\frac{1-\epsilon_t}{\epsilon_t}}}\ \epsilon_t + e^{\log\sqrt{\frac{\epsilon_t}{1-\epsilon_t}}}\ (1-\epsilon_t) \\
&= \sqrt{\frac{1-\epsilon_t}{\epsilon_t}}\ \epsilon_t + \sqrt{\frac{\epsilon_t}{1-\epsilon_t}}\ (1-\epsilon_t) \\
&= 2\sqrt{\epsilon_t(1-\epsilon_t)} \\
&= 2\sqrt{(1/2-\gamma_t)(-1/2+\gamma_t)} \\
&= \sqrt{(1-2\gamma_t)(-1+2\gamma_t)} \\
&= \sqrt{1-4\gamma_t^2}
\end{aligned}$$

5. Putting the result from above into the bound from step 2:

$$\begin{aligned}
\mathcal{E}_S(H) &\leq \prod_{t=1}^{T} Z_t \\
&= \prod_{t=1}^{T} \sqrt{1-4\gamma_t^2} \\
&\leq e^{-2\sum_t \gamma_t^2} \\
&\leq e^{-2T\gamma^2} \qquad \textbf{QED}
\end{aligned}$$

3.5.2 Additive models

Note that Adaboost can be seen as "growing" our final classifier $H(x)$ step-by-step. The growing is achieved by adding extra $\alpha_t h_t(x)$ at every step. In literature, *forward stagewise additive models* have the form as shown below.

$$\text{At step } t: \quad \min_{\alpha_t, h_t} \sum_{i=1}^{m} V\left(y_i, f^{t-1}(x_i) + \alpha_t h_t(x)\right)$$

Note that if we set V to be exponential loss, the problem turns into Adaboost.

3.5.3 Why use exponential loss?

In classification problems, the misclassification loss is not smooth and hard to optimize. To work around this issue, we usually pick a loss that maps to $\mathbb{R}$ rather than $\{-1,+1\}$, which makes the loss function *rich* and *smooth*. We use exponential loss because it is diffentiable everywhere (unlike hinge-loss) and promotes predictions with high positive margin (which is good). In contrast, square loss unnecessarily punishes predictions with positive margin.

3.6 Boosting vs. bagging

Bagging usually uses full trees, while boosting uses single-split trees. In the best case, **bagging reduces variance** while **boosting reduces bias**. In general, boosting has more problems with noisy data.

4 Support vector machines (SVM)

4.1 Separating hyperplanes

A hyperplane is an n-dimensional plane parametrised by $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ (also called an affine linear space). The hyperplane is represented by the set

$$H_{w,b} = \{x \in \mathbb{R}^n : w^T x + b = 0\}$$

Let our data set be $S = \{(x_i, y_i)\}_{i=1}^m$, where $x_i \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$. Then, if data is linearly separable, there exist a separating hyperplane $H_{w,b}$ such that

$$y_i(w^T x + b) > 0 \qquad (\equiv) \qquad sign(y_i) = sign(w^T x + b), \qquad i = 1, ..., m$$

Note that we want inequality to be strict, i.e. we don't allow any points to lie on the separating hyperplane.

4.2 Margin

The distance from a point x to the hyperplane is defined as $\rho_x(w, b) := \frac{|w^T x + b|}{\|w\|}$. If $H_{w,b}$ separates the training S, we define its **margin** as

$$\rho_S(w, b) := \min_{i=1}^{m} \rho_{x_i}(w, b)$$

We also define a *margin for a single point*, that can be negative or positive - $\frac{w^T x + b}{\|w\|}$

4.3 Optimal separating hyperplane (OSH)

Intuitively, the optimal separating hyperplane splits the data in such a way that the distance from the plane to the closest point is maximised. Mathetmatically, this can be expressed as

$$\rho(S) := \max_{w,b} \min_{i} \left[\frac{y_i(w^T x_i + b)}{\|w\|} : y_i(w^T x + b) \geq 0, i = 1, ..., m \right]$$

We require that $\rho(S) > 0$ which means that the data has to be linearly separable. This, in turn, implies that we can achieve positive margin for every point x, i.e. $\frac{w^T x + b}{\|w\|} > 0$.

Note that in this problem, the w, b parameterization is not unique. For example, rescaling with a positive constant gives the same hyperplane. There are two ways to fix the parametrization to limit the solutions.

4.4 Normalized hyperplane

We require that $\|w\| = 1$. Substituting this into equations from above, we get:

$$\rho_x(w,b) = |w^T x + b| \quad \textbf{(point)} \qquad\qquad \rho_S(w,b) = \min_{i=1}^{m}\ y_i(w^T x_i + b) \quad \textbf{(margin)}$$

If we consider normalized hyperplanes, the optimization problem for OSH becomes:

$$\rho(S) = \max_{w,b} \min_{i} \left[y_i(w^T x + b) : y_i(w^T x_i + b) \geq 0, \|w\| = 1 \right]$$

4.5 Canonical hyperplane

We require that margin is $\min_{i=1}^{m} y_i(w^T x_i + b) = 1$. This, in turn, means that we require $\|w\|$ such that $\rho_S(w,b) = \frac{1}{\|w\|}$. This parameterization is data-dependent.

If we consider canonical hyperplanes, the optimization problem for OSH becomes:

$$\rho(S) = \max_{w,b} \left[\frac{1}{\|w\|} : \min_{i} \big(y_i(w^T x_i + b) \big) = 1, y_i(w^T x_i + b) \geq 0 \right]$$

4.6 Primal and dual solution forms for OSH

We can now define our actual optimization problem. Problem **P1**: Need to minimize $\frac{1}{2} w^T w$ subject to $y_i(w^T x_i + b) \geq 1$. The solution to this problem is equivalent to finding the saddle point of Lagrangian function:

$$\begin{aligned} L(w,b;\alpha) &= \frac{1}{2} w^T w - \sum_{i=1}^{m} \alpha_i \left[y_i(w^T x_i + b) - 1 \right] \\ &= \frac{1}{2} w^T w - \sum_{i=1}^{m} \alpha_i y_i w^T x_i - b \sum_{i=1}^{m} \alpha_i y_i + \sum_{i=1}^{m} \alpha_i \end{aligned}$$

where α_i are langrangian multipliers. This form of the optimization problem is called the **primal form**.

If a solution exists, this problem can be solved by first minimizing w.r.t. to w and b (assuming other parameters are constant) and then maximizing it w.r.t. $\alpha_i \geq 0$. This way of solving the problem is called **dual form**. Kuhn-Tucker theorem shows that the solution to dual problem is the same as the solution for the primal form.

When we take the derivative with respect to w and b, we get the following results:

$$\begin{aligned} \frac{\partial L}{\partial b} &= -\sum_{i=1}^{m} \alpha_i y_i = 0 \quad \Rightarrow \quad \sum_{i=1}^{m} \alpha_i y_i = 0 \\ \frac{\partial L}{\partial w} &= w - \sum_{i=1}^{m} \alpha_i y_i x_i = 0 \quad \Rightarrow \quad w = \sum_{i=1}^{m} \alpha_i y_i x_i \end{aligned}$$

If we plug these values back into the expression for $L(w,b;\alpha)$ we get:

$$\begin{aligned}L(w,b;\alpha) &= \frac{1}{2}\left(\sum_{i=1}^{m}\alpha_i y_i x_i\right)^T\left(\sum_{i=1}^{m}\alpha_i y_i x_i\right) - \sum_{i=1}^{m}\alpha_i y_i\left(\sum_{j=1}^{m}\alpha_j y_j x_j\right)^T x_i + 0 + \sum_{i=1}^{m}\alpha_i \\ &= \frac{1}{2}\left(\sum_{i=1}^{m}\alpha_i y_i x_i\right)^T\left(\sum_{i=1}^{m}\alpha_i y_i x_i\right) - \left(\sum_{i=1}^{m}\alpha_i y_i x_i\right)^T\left(\sum_{i=1}^{m}\alpha_i y_i x_i\right) + \sum_{i=1}^{m}\alpha_i \\ &= -\frac{1}{2}\left(\sum_{i=1}^{m}\alpha_i y_i x_i\right)^T\left(\sum_{i=1}^{m}\alpha_i y_i x_i\right) + \sum_{i=1}^{m}\alpha_i\end{aligned}$$

Let's define a matrix $A \in \mathbb{M}_{m\times m}$, where each element is $a_{ij} = y_i y_j \langle x_i, x_j\rangle$, we can rewrite the above more concisely as:

$$L(w,b;\alpha) = -\alpha^T A\alpha + \sum_{i=1}^{m}\alpha_i$$

This leads to the **dual** problem:

$$\begin{aligned}\textbf{Maximize:}\quad & Q(\alpha) = -\alpha^T A\alpha + \sum_{i=1}^{m}\alpha_i \\ \textbf{Subject to:}\quad & \sum_{i=1}^{m}\alpha_i y_i = 0 \\ & \alpha_i \geq 0,\ i = 1,...,m\end{aligned}$$

Let vector $\bar{\alpha}$ be the solution to dual problem. Then solution $(\bar{w},\bar{b})$ to the primal problem is given by:

$$\bar{w} = \sum_{i=1}^{m}\bar{\alpha}_i y_i x_i$$

Note that for some examples (x_i, y_i), the coefficient is $\alpha_i = 0$. This means that these examples do not contribute anything to the solution of the primal form. The vectors that **do** actually contribute to the solution are called Support Vectors. Support Vectors are usually a small subset of the data.

$\bar{b}$ can be derived by solving expressions for Kuhn-Tucker (note that x_i has to be a support vector, otherwise the equality is trivial):

$$\begin{gathered}\bar{\alpha}_i(y_i(\bar{w}^T x + \bar{b}) - 1) = 0 \\ \Downarrow \\ \bar{b} = y_i - \bar{w}^T x\end{gathered}$$

4.7 Final classifier and generalization error

Once the solution to dual problem is found, the final classifier is:

$$f(x) = sign\left(\sum_{i=1}^{m} y_i\bar{\alpha}_i x_i^T x + \bar{b}\right)$$

Let n_{SV} be the expected number of support vectors of SVM trained on m i.i.d. samples. Then we can bound generalization error of SVM trained on $m-1$ samples:

$$\textbf{generalization error} \leq \frac{n_{SV}}{m-1}$$

4.8 Non-separable case for OSH

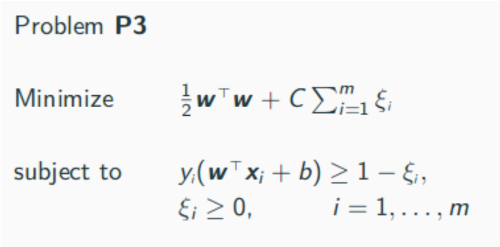
The results above work for the linearly separable case, but if data is not linearly separable? In the separable case, our misclassification error (after finding the solution) is zero. In the non-separable we'd like to find a solution that minimizes the misclassification loss (note that we're still considering canonical hyperplanes):

$$L(w, b; C) = \frac{1}{2}w^T w + C\sum_{i=1}^{m} V_{misclassification}(y_i, w^T x_i + b)$$

Recall that misclassification loss is not a smooth function (since it's discrete), and hence it's hard to optimize. We relax the problem by using hinge loss instead, which is smooth, convex, and differentiable:

$$L(w, b; C) = \frac{1}{2}w^T w + C\sum_{i=1}^{m} V_{hinge}(y_i, w^T x_i + b) \qquad \text{where} \qquad V_{hinge}(y, \hat{y}) = max(0, 1 - y\hat{y})$$

The optimization problem now becomes:



Problem **P3**

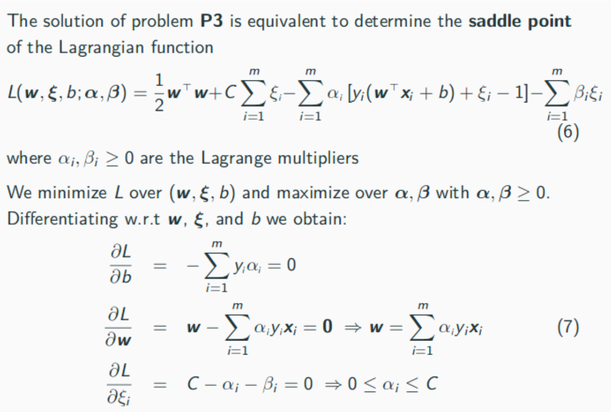
Minimize $\frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\sum_{i=1}^{m}\xi_i$

subject to $y_i(\mathbf{w}^\top\mathbf{x}_i + b) \geq 1 - \xi_i,$
$\xi_i \geq 0, \qquad i = 1, \ldots, m$

The ξ_i's are just slack variables. Normally, we would require the separation between the hyperplane and the closest point (i.e. *margin* of the solution) is at least 1. Now, for every point x_i, we allow margin to be $1-\xi_i$. This, in turn, means that margin can becomes negative, allowing incorrectly-classified points, just like we wanted.

Role of C: C represents the trade-off between the size of w ($\|w\|^2$) and the training error ($\sum_{i=1}^{m}\xi_i$). C is selected by minimizing the LOO cross-validation error. It can also be shown that α_i are piece-wise quadratic functions of C, which makes it easy to recompute the solution if C changes.

The solution to problem **P3** mimics the linearly separable case closely (it also results in a dual problem), with the addition of parameter C and the slack variables ξ_i. Slides below summarise it.



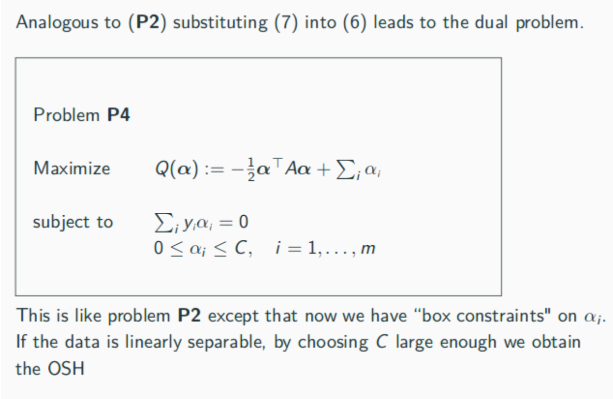
The solution of problem **P3** is equivalent to determine the **saddle point** of the Lagrangian function

$$L(\boldsymbol{w},\boldsymbol{\xi},b;\boldsymbol{\alpha},\boldsymbol{\beta}) = \frac{1}{2}\boldsymbol{w}^\top\boldsymbol{w}+C\sum_{i=1}^{m}\xi_i-\sum_{i=1}^{m}\alpha_i\left[y_i(\boldsymbol{w}^\top\boldsymbol{x}_i+b)+\xi_i-1\right]-\sum_{i=1}^{m}\beta_i\xi_i \qquad (6)$$

where $\alpha_i,\beta_i \geq 0$ are the Lagrange multipliers

We minimize L over $(\boldsymbol{w},\boldsymbol{\xi},b)$ and maximize over $\boldsymbol{\alpha},\boldsymbol{\beta}$ with $\boldsymbol{\alpha},\boldsymbol{\beta}\geq 0$. Differentiating w.r.t $\boldsymbol{w}$, $\boldsymbol{\xi}$, and b we obtain:

$$\begin{aligned}
\frac{\partial L}{\partial b} &= -\sum_{i=1}^{m} y_i\alpha_i = 0 \\
\frac{\partial L}{\partial \boldsymbol{w}} &= \boldsymbol{w}-\sum_{i=1}^{m}\alpha_i y_i\boldsymbol{x}_i=\boldsymbol{0} \;\Rightarrow\; \boldsymbol{w}=\sum_{i=1}^{m}\alpha_i y_i\boldsymbol{x}_i \qquad (7) \\
\frac{\partial L}{\partial \xi_i} &= C-\alpha_i-\beta_i=0 \;\Rightarrow 0\leq\alpha_i\leq C
\end{aligned}$$



Analogous to (**P2**) substituting (7) into (6) leads to the dual problem.

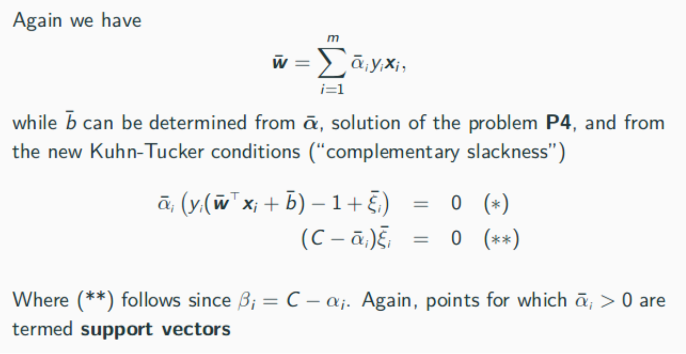
Problem **P4**

Maximize $\quad Q(\boldsymbol{\alpha}) := -\frac{1}{2}\boldsymbol{\alpha}^\top A\boldsymbol{\alpha}+\sum_i\alpha_i$

subject to $\quad \sum_i y_i\alpha_i = 0$

$\quad 0\leq\alpha_i\leq C, \quad i=1,\ldots,m$

This is like problem **P2** except that now we have "box constraints" on α_i. If the data is linearly separable, by choosing C large enough we obtain the OSH



Again we have

$$\bar{\boldsymbol{w}}=\sum_{i=1}^{m}\bar{\alpha}_i y_i\boldsymbol{x}_i,$$

while $\bar{b}$ can be determined from $\bar{\boldsymbol{\alpha}}$, solution of the problem **P4**, and from the new Kuhn-Tucker conditions ("complementary slackness")

$$\begin{aligned}
\bar{\alpha}_i\left(y_i(\bar{\boldsymbol{w}}^\top\boldsymbol{x}_i+\bar{b})-1+\bar{\xi}_i\right) &= 0 \quad (*) \\
(C-\bar{\alpha}_i)\bar{\xi}_i &= 0 \quad (**)
\end{aligned}$$

Where (**) follows since $\beta_i = C-\alpha_i$. Again, points for which $\bar{\alpha}_i > 0$ are termed **support vectors**

4.9 SVMs with feature maps

All of the results above hold for SVMs using kernels and feature maps too. We simply replace x with $\phi(x)$ and x^Tt with $\langle\phi(x),\phi(t)\rangle = K(x,t)$. The matrix A in dual problem now becomes $a_{ij}=y_iy_jK(x_i,x_j)$. The final classifier is then:

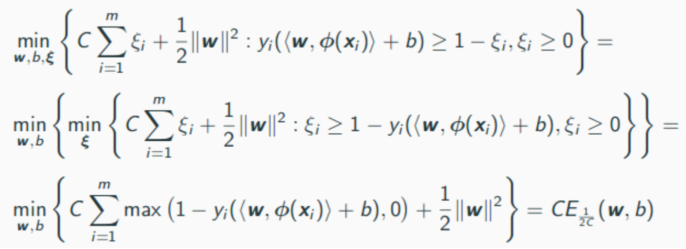
$$f(x) = sign\left(\sum_{i=1}^{m} y_i\bar{\alpha}_iK(x_i,x)+\bar{b}\right)$$

4.10 Connection to regularization

Note that the result above is equivalent to minimizing our hinge loss with a regularization term $\lambda\left\|w\right\|^2$ with $\lambda=\frac{1}{2C}$. That is, minizing the following function:

$$\begin{aligned}
E_\lambda(w,b) &= \sum_{i=1}^{m} V_{hinge}(y_i,\langle w,\phi(x_i)\rangle+b)+\lambda\left\|w\right\|^2 \\
&= \sum_{i=1}^{m}\max(0,1-y_i\langle w,\phi(x_i)\rangle+b)+\frac{1}{2C}\left\|w\right\|^2
\end{aligned}$$

Formally, this can be proven as follows:



$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \left\{ C\sum_{i=1}^{m} \xi_i + \frac{1}{2}\|\boldsymbol{w}\|^2 : y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b) \geq 1 - \xi_i, \xi_i \geq 0 \right\} =$$

$$\min_{\boldsymbol{w},b} \left\{ \min_{\boldsymbol{\xi}} \left\{ C\sum_{i=1}^{m} \xi_i + \frac{1}{2}\|\boldsymbol{w}\|^2 : \xi_i \geq 1 - y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b), \xi_i \geq 0 \right\} \right\} =$$

$$\min_{\boldsymbol{w},b} \left\{ C\sum_{i=1}^{m} \max\left(1 - y_i(\langle \boldsymbol{w}, \phi(\boldsymbol{x}_i)\rangle + b), 0\right) + \frac{1}{2}\|\boldsymbol{w}\|^2 \right\} = CE_{\frac{1}{2C}}(\boldsymbol{w}, b)$$

5 Online learning

5.1 Online learning model

Model for online learning: There exist some online sequence of data (usually no distributional assumptions). **Aim:** To sequentially predict and update classifier to predict well on the sequence (i.e. no training/test set distinction). **Evaluation metric:** Cumulative error.

This contrasts with batch learning model, which has a distinct training and test set sets. Batch model also uses generalization error instead of cumulative error.

We use online learning because it's faster, has small memory footprint, no statistical assumptions required, and is often used for BIG DATA classifiers. There also exist techniques that convert cumulative error guarantees into generalization error guarantees. We will mainly focus on the **worst-case** model for online learning.

5.2 Learning with expert advice

In learning with expert advice, for every new data point (x_t, y_t) we get multiple predictions from different experts (which come from either humans or estimator functions). The goal of the master algorithm is to combine these predictions while minimising the loss:

$$L_A(S) := \sum_{t=1}^{m} |y_t - \hat{y}_t|$$

(Recall that $x_t \in \{0,1\}^n$ and $y_t, \hat{y}_t \in \{0,1\}$). Note that our master algorithm operates in online fashion, so it can update some internal parameters after checking its performance after each time step t.

5.2.1 Regret bound

Let N be the number of experts. Let $L_i(S)$ be the loss of expert E_i, defined as:

$$L_i(S) := \sum_{t=1}^{m} |y_t - x_{t,i}|$$

We want to obtain bounds of the form shown below, which is known as the **regret** bound (a.k.a. bound "relative" best expert that holds even in "the worst case"):

$$\forall S : \quad L_A(S) \;\leq\; a \min_i L_i(S) + b \log(N)$$

where a and b are some "small" constants.

5.2.2 Halving algorithm

Assume at least one of the experts is consistent, i.e. consistently makes correct predictions. The halving algorithm works by having a white-list of experts who are considered in the majority vote. If prediction at step t is correct, white-list remains unchanged. If prediction is wronged, than all of the experts who voted in favour of the incorrect prediction are blacklisted.

Note that for every incorrect prediction, we eliminate at least half of the remaining experts (since they won the majority vote). This means we can make at most $\log_2 N$ mistakes, since by the time we make the $(\log_2 N)$th mistake we'd only have one expert left, who should be consistent by assumption.

5.2.3 Weighted majority algorithm

Clearly, halving algorithm doesn't work when all experts are inconsistent. A better algorithm is to assign weights to each expert, and then decay these weights based on their performance.

Let $w_{t,i}$ be the weight of expert E_i at time t. Let $M_{t,i}$ be the number of mistakes of that expert. We define the weight of an expert and the sum of all experts at time t:

$$w_{t,i} = \beta^{M_{t,i}}, \quad \beta \in [0,1)$$
$$W_t = \sum_{i=1}^{N} w_{t,i}$$

That is, all experts start with weight 1, and everyone they make a mistake their weight is decreased by multiplying it with β. To make a decision, we combine the weights of all experts that chose a particular prediction, and if their sum is greater than $W_t/2$, they are the majority, otherwise they are the minority.

If no mistake occurs, we multiply the minority by β. In this case, the total weight might or might not change: $W_t \leq W_{t+1}$. If we make a mistake, we multiply the majority by β, which means that at least half of total weight will decrease:

$$\begin{aligned} W_t &\leq \frac{1}{2}W_{t-1} + \frac{\beta}{2}W_{t-1} \\ &= \frac{1+\beta}{2}W_{t-1} \end{aligned}$$

Note that initial weight for each expert is 1, so initial total weight is $W_1 = n$. We can use the results from before to bound the total final weight W_{m+1} from above and from below:

$$W_{m+1} = \sum_{j=1}^{N} w_{m+1,j} = \sum_{j=1}^{N} \beta^{M_{m+1,j}} \geq \min_i \beta^{M_{m+1,i}} \qquad \textbf{and} \qquad W_{m+1} \leq \left(\frac{1+\beta}{2}\right)^M n$$
$$\Downarrow$$
$$\left(\frac{1+\beta}{2}\right)^M n \;\geq\; \min_i \beta^{M_{m+1,i}}$$

Solving for M by taking log of both sides, we get the result below (note that inequality changes since we're taking the log of values in interval $[0, 1]$):

$$M \leq \frac{\ln \frac{1}{\beta}}{\ln \frac{2}{1+\beta}} \min_i M_i + \frac{1}{\ln \frac{2}{1+\beta}} \ln n$$

Setting $\beta = 1/e$, we can get $M \leq 2.63 \min_i M_i + 2.63 \ln n$.

5.2.4 Generalizing experts model

We would like to obtain bounds for arbitrary loss functions of form $L : \mathcal{Y} \times \hat{\mathcal{Y}} \rightarrow [0, +\infty]$. We now use a more precise notion of regret as bounds in the form:

$$L_A(S) - \min_i L_i(S) \ \leq \ o(m)$$

Where $o(m)$ is some function sublinear in m. Note that if we divide both sides by m and take the limit as $m \rightarrow \infty$, we get:

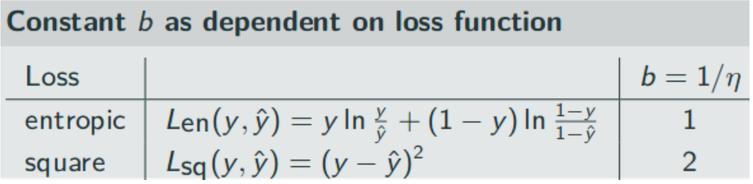
$$\frac{L_A(S)}{m} \leq \frac{\min_i L_i(S)}{m}$$

Which means that the mean asymptotic loss of our algorithm is bounded from above by the mean asymptotic loss of the best expert.

We also relax our predictions, allowing them to be any real value in $[0, 1]$. Now, for each expert, instead of number of mistakes at time step t, we maintain the cumulative loss for that expert, $L_{t,i}$. The weight for each expert is then defined as $w_{t,i} = \beta^{L_{t,i}} = e^{-\eta \, L_{t,i}}$.

Before making a prediction, our algorithm normalizes the weight vector using $v = w/(\sum_i w_i)$. Thus initial normalized weights for each expert are all $1/N$, and final output is a dot product between the normalized weights and predictions made by each individual expert: $\sum_{i=1}^{N} w_{t,i} \, pred_{t,i}$.

There is a theorem showing some important bounds for entropic and square losses:



Constant b as dependent on loss function

Loss		$b = 1/\eta$
entropic	$L_{\text{en}}(y, \hat{y}) = y \ln \frac{y}{\hat{y}} + (1-y) \ln \frac{1-y}{1-\hat{y}}$	1
square	$L_{\text{sq}}(y, \hat{y}) = (y - \hat{y})^2$	2

5.2.5 Hedge

In the hedge algorithm, at every step the learner chooses an allocation v_t (which basically looks like a probability distribution). The learner than receives a loss vector l_t. The total loss of the learner at that step is then $L_A(t) = v_t \cdot l_t$. The update rule is:

$$w_{t+1,i} = w_{t,i} \cdot exp(-\eta \, l_i^t), \qquad \forall i \in [n]$$

where $\eta > 0$ is the learning rate.

One can show that for any sequence of losses $S = l_1, ..., l_m \in [0, 1]^N$, the regret bound of weighted average algorithm with $\eta = \sqrt{\frac{\ln N}{2m}}$ is :

$$E[L_{WA}(S)] - \min_i L_i(S) \leq \sqrt{2m \ln N}$$

5.3 Learning with thresholded linear combinations

In the case with experts, the bounds are logarithmic in the number of experts N, which implies that we can just use a large number of experts to solve the problem. This is not always computationally feasible, so we'll now consider linear combinations of experts.

Let $\mathcal{U}$ be our comparison class (a.k.a. hypothesis space). For linear classifiers, $\mathcal{U}$ is a set of linear threshold functions. Let $Loss_u(S)$ be the loss associated with some function $u \in \mathcal{U}$ on training set S. The relative loss is then:

$$\textbf{Relative loss:} \qquad L_A(S) - \inf_{u \in \mathcal{U}} Loss_u(S)$$

(Note that $\inf$ is similar to $\min$ with several minor differences). We will focus on the realizable case, that is, we will assume there exists $u \in \mathcal{U}$ s.t. $Loss_u(S) = 0$.

5.3.1 Linear classifiers with disjunctions

Classical AI focused on working with symbolic knowledge. One example of such knowledge is boolean logic, and we will show that linear classifiers can simulate logical conjunctions and disjunctions.

Terminology recap: A literal is a single variable, e.g. x_1. A literal can be negated, e.g. $\bar{x}_1$. A **term** is a conjunction (AND) of literals, e.g. $x_1 \bar{x_2} x_3$. A **clause** is a disjunction (OR) of literals, e.g. $x_1 V x_2$. A **monotone** disjunction or conjunction must contain no negated literals.

Recall that a linear threshold function $f_{u,b} : \mathbb{R}^n \to \{-1, 1\}$ determines the label based on a separating hyperplane. It can be written as:

$$f_{u,b} := sign(u \cdot x + b)$$

For this problem, our input will be pairs (x_i, y_i), where $x_i \in \{True, False\}^n$ and $y_i \in \{True, False\}$. Numerically, we will always represent $True$ as 1 and $False$ as either 0 or -1, as convenient.

Note that we only really need to solve the problem for monotone disjunctions. To solve a non-monotone disjunction, we can simply introduce a feature map $\phi = (x_1, 1 - x_1, ..., x_m, 1 - x_m)$. To solve a conjunction, we can simply convert it into a disjunction using De Morgan's theorem.

5.3.2 Representing a monotone disjunction using a linear threshold function

Imagine we have N experts, denoted E_i. Each expert votes $True$ or $False$, represented as 1 or 0 respectively. As an example, let's say we only want use $z < N$ experts in our prediction. To simulate a disjunction, we set $b = -1/2$, and set u to be all zeros, except for indices of our chosen experts, which will be 1's. Then, evaluating $sign(u \cdot x + b)$ will return 1 if at least one expert has voted $True$, and -1 otherwise, perfectly simulating a disjunction.

5.3.3 Sub-optimal solution

One solution is to consider all possible disjunction functions (simulated using $u \in \mathcal{U}$) as our experts. This will require $\binom{n}{k}$ weights, resulting in time and space complexity exponential in k. This is computationally unfeasible for large inputs, so we will now consider solutions with one weight per literal.

5.4 Algorithms for learning boolean functions

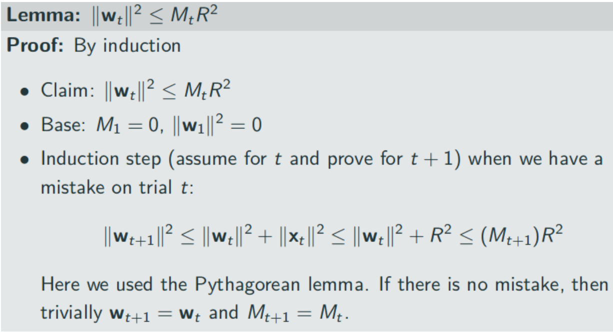
5.4.1 Perceptron

Perceptron algorithm assumes that data is linearly separable with some margin γ. Algorithms work by initializing the weight vector w and mistake count M to 0. At every step t, we make a prediction $\hat{y}_t = sign(w_t \cdot x_t)$. If a mistake is made, we increment the mistake count M by 1, and update the weights using:

$$w_{t+1} = w_t + y_t x_t$$

Let R be a number s.t. $\|x_t\| \leq R$. We can show that the number of mistakes perceptron can make is at most $\left(\frac{R}{\gamma}\right)^2$, by combining the upper and lower bounds on $\|w\|$. We can prove this in 3 steps:

- Steps 1 and 2:



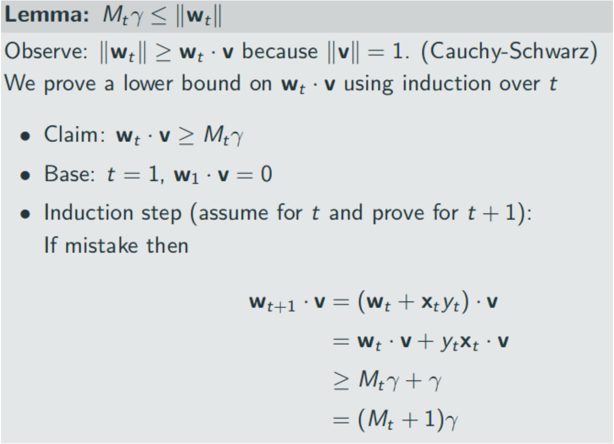
Lemma: $\|\mathbf{w}_t\|^2 \leq M_t R^2$

Proof: By induction

- Claim: $\|\mathbf{w}_t\|^2 \leq M_t R^2$
- Base: $M_1 = 0$, $\|\mathbf{w}_1\|^2 = 0$
- Induction step (assume for t and prove for $t+1$) when we have a mistake on trial t:

$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_t\|^2 + \|\mathbf{x}_t\|^2 \leq \|\mathbf{w}_t\|^2 + R^2 \leq (M_{t+1})R^2$$

Here we used the Pythagorean lemma. If there is no mistake, then trivially $\mathbf{w}_{t+1} = \mathbf{w}_t$ and $M_{t+1} = M_t$.



Lemma: $M_t\gamma \leq \|\mathbf{w}_t\|$

Observe: $\|\mathbf{w}_t\| \geq \mathbf{w}_t \cdot \mathbf{v}$ because $\|\mathbf{v}\| = 1$. (Cauchy-Schwarz)
We prove a lower bound on $\mathbf{w}_t \cdot \mathbf{v}$ using induction over t

- Claim: $\mathbf{w}_t \cdot \mathbf{v} \geq M_t\gamma$
- Base: $t = 1$, $\mathbf{w}_1 \cdot \mathbf{v} = 0$
- Induction step (assume for t and prove for $t+1$):
If mistake then

$$\begin{aligned}\mathbf{w}_{t+1} \cdot \mathbf{v} &= (\mathbf{w}_t + \mathbf{x}_t y_t) \cdot \mathbf{v} \\ &= \mathbf{w}_t \cdot \mathbf{v} + y_t \mathbf{x}_t \cdot \mathbf{v} \\ &\geq M_t\gamma + \gamma \\ &= (M_t + 1)\gamma\end{aligned}$$

- Step 3: Finally, we have $M_y\gamma \leq \|w_t\| \leq M_t R^2$. Rearranging, we get the desired result, $M \leq \left(\frac{R}{\gamma}\right)^2$

Novikoff convergence theorem for perceptron is then: $M \leq \left(\frac{R}{\gamma}\right)^2$ and $R = \max_t ||x_t||$. Only holds if there exists vector v s.t. $||v|| = 1$ and γ s.t. $(v \cdot x_t)y_t \geq \gamma$.

More precisely, using the feature map $\phi(x) = (x, 1)$ we can use perceptron to learn monotone disjunctions so that:

$$M \leq (4k+1)(n+1)$$

5.4.2 Winnow

For winnow, we initialize the weights vector to 1. On input x_t we predict using:

$$\begin{aligned} w_t \cdot x_t \geq n \quad &\Rightarrow \quad \textbf{predict } 1 \\ w_t \cdot x_t < n \quad &\Rightarrow \quad \textbf{predict } 0 \end{aligned}$$

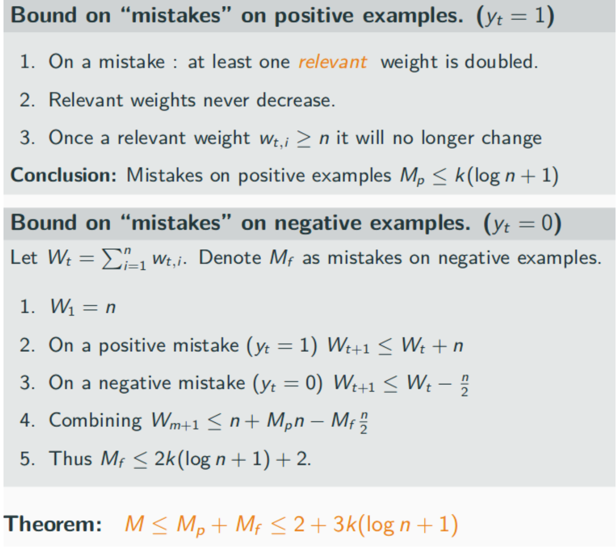
where n is the dimensionality of the input. When a mistake occurs, we update each element of the weight vector using:

$$w_{t+1,i} = w_{t,i}\dot{2}^{y_t-\hat{y}_t}x_{t,i}$$

We can show that number of mistakes of winnow is bounded by:

$$M \leq 3k(\log n + 1) + 2$$

If there exists a consistent k-literal monotone disjunction. See proof below.



Bound on "mistakes" on positive examples. ($y_t = 1$)

1. On a mistake : at least one *relevant* weight is doubled.
2. Relevant weights never decrease.
3. Once a relevant weight $w_{t,i} \geq n$ it will no longer change

Conclusion: Mistakes on positive examples $M_p \leq k(\log n + 1)$

Bound on "mistakes" on negative examples. ($y_t = 0$)

Let $W_t = \sum_{i=1}^{n} w_{t,i}$. Denote M_f as mistakes on negative examples.

1. $W_1 = n$
2. On a positive mistake ($y_t = 1$) $W_{t+1} \leq W_t + n$
3. On a negative mistake ($y_t = 0$) $W_{t+1} \leq W_t - \frac{n}{2}$
4. Combining $W_{m+1} \leq n + M_p n - M_f \frac{n}{2}$
5. Thus $M_f \leq 2k(\log n + 1) + 2$.

Theorem: $M \leq M_p + M_f \leq 2 + 3k(\log n + 1)$

5.4.3 Winnow vs Perceptron

Both algorithms are efficient, in as sense that they maintain only one weight per literal and the update step is $O(1)$. Winnow offers better performance guarantees (at least for disjunctions). Perceptron algorithm, on the other hand, is compatible with the kernel trick.

5.4.4 Learning disjunctive normal form (DNF)

DNFs can be learnt be efficiently learnt by perceptron (with poor prediction performance) and inefficiently by Winnow (with good performance).

Note that the feature map of the Anova kernel takes in a vector x_t of size n, and outputs a vector $\phi(x_t)$ of all possible disjunctions of elements of x_t, of size 2^n. That said, computing $K(x,t)$ only takes n steps because Anova kernel uses an implicit feature map. Because of this, using Anova kernel singificantly speeds up learning DNF with Perceptron.

On the other hand, Winnow is not compatible with the kernel trick, so we need to consider all elements of $\phi(x_t)$ explicitly, which results in slower predictions. The differences in performance between Perceptron and Winnow on DNF can be summarised as follows:

Perceptron:	**prediction speed** $= O(n \cdot M)$	**mistakes** $= O(k\, 2^n)$
Winnow:	**prediction speed** $= O(2^n \cdot M)$	**mistakes** $= O(k \ln 2^n) = O(k \cdot n)$

6 Sparsity and matrix estimation

Not examined.

7 Learning theory

7.1 Learning model

Data is sampled i.i.d. from a distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, where $\mathcal{Y} = \{0, 1\}$. The expected error (a.k.a. generalization error) of a estimator $h : \mathcal{X} \to \mathcal{Y}$ is:

$$L_{\mathcal{D}}(h) \ = \ \mathcal{D}(\{(x, y) : h(x) \neq y\}) \ = \ \Pr_{(x,y)\sim\mathcal{D}} [h(x) \neq y]$$

The empirical error of h is:

$$L_S(h) = \sum_{i=1}^{m} \frac{1}{m} \mathcal{I} \left[h(x_i) \neq y\right]$$

7.2 Validation set bound

Validation set bound tells us that we can bound the generalization error of a predictor h by its empirical error. Formally, the theorem is: Select an $h : \mathcal{X} \to \mathcal{Y}$, then for any $\delta \in (0, 1)$, with probability at least $1 - \delta$ over the random sample V of size m from $\mathcal{D}$ we have that

$$L_{\mathcal{D}}(h) \leq L_V(h) + \sqrt{\frac{\ln \frac{1}{\delta}}{2m}}$$

TODO: *Include proof?*

Note that the validation set V cannot be used in training for the bound to work, which makes this data expensive.

7.3 Empirical risk minimization (ERM)

Assume we have a learning agent $\mathcal{A}$ that choose some hypthesis function $\mathcal{A}_{''(S)}$ from hypothesis space $\mathcal{H}$ in response to training set S. We will focus on empirical risk minimzation:

$$ERM_{\mathcal{H}}(S) := \text{argmin}_{h\in\mathcal{H}} L_S(h)$$

Note that there may be many possible empirical risk minimizers.

7.4 Expected vs. confident bounds

TODO: *TODO.*

7.5 PAC model

7.5.1 Realisibility assumption

We assume that there exists some true function f^* with zero error, i.e. a perfect classifier such that for all $x \in \mathcal{X}$ we have $f^*(x) = y$. This trick lets us assume that $\mathcal{D}$ is a distribution over $\mathcal{X}$ on its own, without $\mathcal{Y}$. We will indicate this dependence on f^* in notation:

$$L_{\mathcal{D},f^*}(h) = \Pr_{x \sim \mathcal{D}}[h(x) \neq f^*(x)]$$

7.5.2 Role of ϵ and δ

Can we find a learning algorithm $\mathcal{A}$ such that $L_{\mathcal{D},f^*}(h)$ is small? Sadly, it's impossible to find an algorithm such that $L_{\mathcal{D},f^*}(h) = 0$.

We can prove that by considering $\mathcal{X} = \{x_1, x_2\}$. Set probabilities in distribution $\mathcal{D}$ of x_1 to $1 - \epsilon$ and x_2 to ϵ, for any $\epsilon \in (0, 1)$. The probability to not see x_2 among m examples picked i.i.d. is then $(1 - \epsilon)^m \approx e^{-\epsilon m}$. Finally, if $\ll 1/m$, we might not see x_2 at all, so we can't know its label.

We can relax this constraint to allow $L_{\mathcal{D},f^8}(h) < \epsilon$ for user specified ϵ. But, recall that input for the algorithm is randomly generated, which means there's a small chance the same input will be seen again and again. As a result we need another relaxation - we allow the algorithm to fail with probability $\delta \in (0, 1)$.

In PAC learning, learner is given ϵ and δ but doesn't know $\mathcal{D}$ and f^*. Learner requests training data S that can only depend on ϵ and δ. The learner should output a hypothesis h s.t. with probability at least $1 - \delta$ it holds that $L_{\mathcal{D},f^*}(h) \leq \epsilon$.

7.5.3 NFL lower bound

Theorem states: Fix $\delta \in (0, 1), \epsilon < 1/2$. For every learner $\mathcal{A}$ and training set size m, there exist $\mathcal{D}, f^*$ such that with probability at least δ over the generation of training data S of m examples, it holds that $L_{\mathcal{D},f^*}(h) \geq \epsilon$

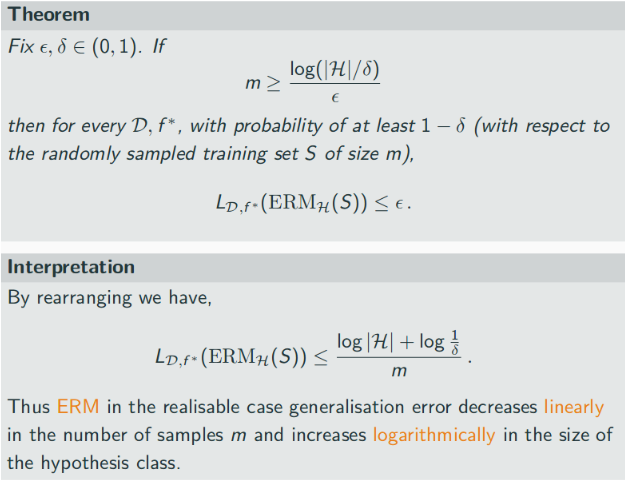
This theorem basically tells us that for some cases there is no PAC learner that satisfies our requirements. If we provide the learner with some prior knowledge of $\mathcal{H}$, solution might exist.

7.5.4 Learning with finite hypothesis classes

Let $\mathcal{H}$ be a finite hypothesis class. We will use consistent learning (a.k.a. empirical risk minimization, ERM), which takes in a training set and returns some $h \in \mathcal{H}$ that minimizes the empirical risk:

$$L_S(h) = \frac{1}{m}|\{i : h(x_i) \neq y_i\}|$$

Recall that function $ERM_{\mathcal{H}}(S)$ an estimator that minimises the empirical risk (not necessarily unique). There is an important bound for the loss:



Theorem

Fix $\epsilon, \delta \in (0,1)$. If

$$m \geq \frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$$

then for every $\mathcal{D}, f^*$, with probability of at least $1-\delta$ (with respect to the randomly sampled training set S of size m),

$$L_{\mathcal{D},f^*}(\mathrm{ERM}_{\mathcal{H}}(S)) \leq \epsilon\,.$$

Interpretation

By rearranging we have,

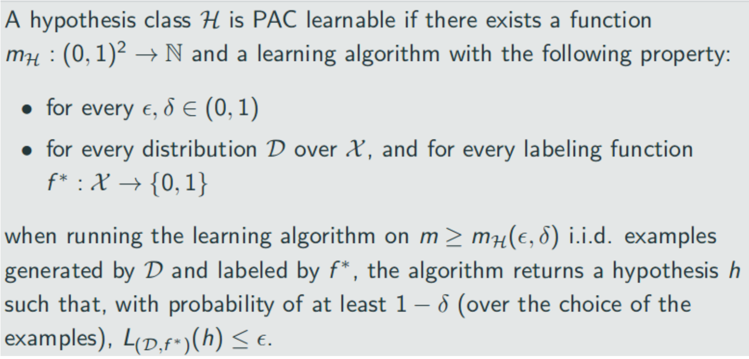
$$L_{\mathcal{D},f^*}(\mathrm{ERM}_{\mathcal{H}}(S)) \leq \frac{\log|\mathcal{H}| + \log\frac{1}{\delta}}{m}\,.$$

Thus ERM in the realisable case generalisation error decreases linearly in the number of samples m and increases logarithmically in the size of the hypothesis class.

7.5.5 Sample complexity

Sample complexity, denoted $m_{\mathcal{H}}$, is the number of examples (size of the training set) required by the learning algorithm to achieve $L_{\mathcal{D},f^*}(h) \leq \epsilon$ with probability at least $1-\delta$, for some $\epsilon, \delta \in (0,1)$. Sample complexity is obtained by using the $ERM_{\mathcal{H}}$ learning rule, i.e. by finding a predictor that is always correct after being trained on some number of examples.

PAC learnability is then defined using $m_{\mathcal{H}}$:



A hypothesis class $\mathcal{H}$ is PAC learnable if there exists a function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a learning algorithm with the following property:

- for every $\epsilon, \delta \in (0,1)$
- for every distribution $\mathcal{D}$ over $\mathcal{X}$, and for every labeling function $f^* : \mathcal{X} \to \{0,1\}$

when running the learning algorithm on $m \geq m_{\mathcal{H}}(\epsilon,\delta)$ i.i.d. examples generated by $\mathcal{D}$ and labeled by f^*, the algorithm returns a hypothesis h such that, with probability of at least $1-\delta$ (over the choice of the examples), $L_{(\mathcal{D},f^*)}(h) \leq \epsilon$.

7.6 VC-dimension

Let $\mathcal{H}$ be our hypothesis space. Consider some arbitrary subset of $\mathcal{X}$ of size $|C|$. Denote this subset C.

Recall the definition of our output space: $\mathcal{Y} \in \{-1,+1\}$. Applying some predictor h to every element of C will create a vector of outputs u, where $u \in \{-1,+1\}^{|C|}$. Note that $|\{-1,+1\}^{|C|}| = 2^{|C|}$, which means there are $2^{|C|}$ potential output vectors that we could generate.

We now limit our hypothesis space $\mathcal{H}$ to functions that only take the elements of C as input. Denote this limited space as $\mathcal{H}_C$. Then we have:

$$\begin{aligned} \textbf{Old:} \quad & \mathcal{H} = \{h_i : \mathcal{X} \to \mathcal{Y}\}_i \\ \textbf{New:} \quad & \mathcal{H}_C = \{h_i : C \to \mathcal{Y} \ \text{ s.t. } \ h_i \in \mathcal{H}\}_i \end{aligned}$$

Note that since there are only $2^{|C|}$ output vectors, the maximum size of set $\mathcal{H}_C$ is $2^{|C|}$. At the same time, the hypothesis space $\mathcal{H}$ might not be complex enough to contain functions that can generate every single output vector based on inputs from C. This gives us the following relationship:

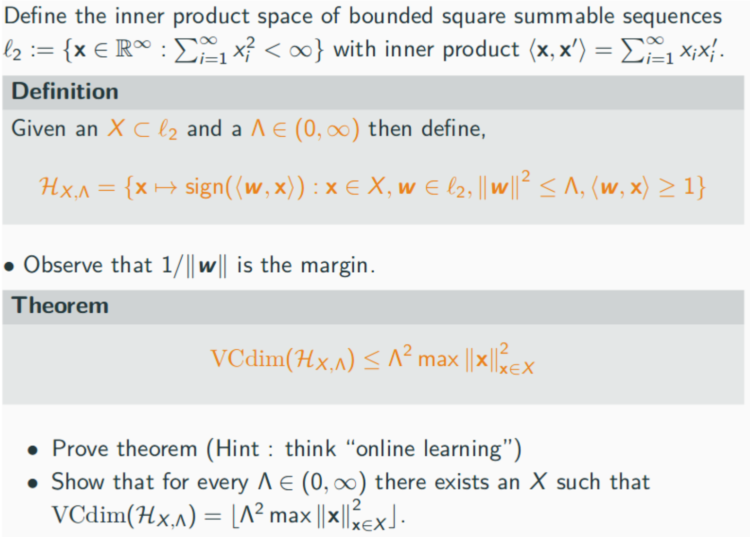
$$|\mathcal{H}_C| \leq 2^{|C|}$$

We say that $\mathcal{H}$ *shatters* C if $|\mathcal{H}_C| = 2^{|C|}$, that is, $\mathcal{H}_C$ has enough functions in it to generate all possible output vectors. This, in turn, means that hypothesis space $\mathcal{H}$ is "complex" enough to achieve perfect accuracy on any training set of size C and any labelling on that training set. Finally, we define VC dimension as:

$$VCdim(\mathcal{H}) = \sup\{|C| : \mathcal{H} \text{ shatters } C\}$$

To prove that $VCdim(\mathcal{H}) = d$, we need to prove that there exists set C of size d that is shattered by $\mathcal{H}$, and that every set C of size $d + 1$ cannot be shattered by $\mathcal{H}$.

7.6.1 Large margin halfspaces



Define the inner product space of bounded square summable sequences $\ell_2 := \{\mathbf{x} \in \mathbb{R}^\infty : \sum_{i=1}^\infty x_i^2 < \infty\}$ with inner product $\langle \mathbf{x}, \mathbf{x}' \rangle = \sum_{i=1}^\infty x_i x_i'$.

Definition

Given an $X \subset \ell_2$ and a $\Lambda \in (0, \infty)$ then define,

$$\mathcal{H}_{X,\Lambda} = \{\mathbf{x} \mapsto \text{sign}(\langle \boldsymbol{w}, \mathbf{x} \rangle) : \mathbf{x} \in X, \boldsymbol{w} \in \ell_2, \|\boldsymbol{w}\|^2 \leq \Lambda, \langle \boldsymbol{w}, \mathbf{x} \rangle \geq 1\}$$

- Observe that $1/\|\boldsymbol{w}\|$ is the margin.

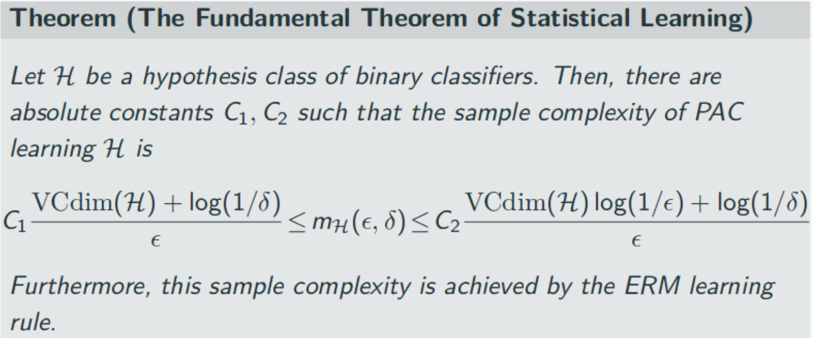
Theorem

$$\text{VCdim}(\mathcal{H}_{X,\Lambda}) \leq \Lambda^2 \max \|\mathbf{x}\|^2_{\mathbf{x} \in X}$$

- Prove theorem (Hint : think "online learning")
- Show that for every $\Lambda \in (0, \infty)$ there exists an X such that $\text{VCdim}(\mathcal{H}_{X,\Lambda}) = \lfloor \Lambda^2 \max \|\mathbf{x}\|^2_{\mathbf{x} \in X} \rfloor$.

7.6.2 VC-dimension upper bound for PAC

TODO: *Show relation to finite hypothesis classes.*



Theorem (The Fundamental Theorem of Statistical Learning)

Let $\mathcal{H}$ be a hypothesis class of binary classifiers. Then, there are absolute constants C_1, C_2 such that the sample complexity of PAC learning $\mathcal{H}$ is

$$C_1 \frac{\text{VCdim}(\mathcal{H}) + \log(1/\delta)}{\epsilon} \leq m_{\mathcal{H}}(\epsilon, \delta) \leq C_2 \frac{\text{VCdim}(\mathcal{H}) \log(1/\epsilon) + \log(1/\delta)}{\epsilon}$$

Furthermore, this sample complexity is achieved by the ERM learning rule.

7.7 Agnostic PAC model

Before we assumed that the labels for the data are generated by some deterministic function f^*. We now relax this constraint and assume that the label for each point x is generated using some distribution over $\mathcal{Y}$. In fact, we will now assume that data points are sampled from some joint distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$.

We now redefine the loss for a single estimator as:

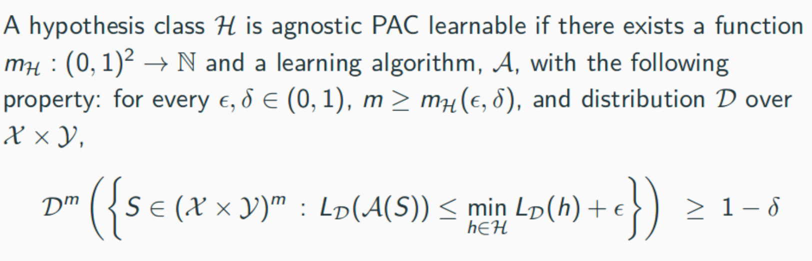
$$L_{\mathcal{D}}(h) \;:=\; \Pr_{(x,y)\sim\mathcal{D}}[h(x) \neq y] \;=\; \mathcal{D}(\{(x,y) : h(x) \neq y\})$$

We redefine the notion of "approximately correct" to:

$$L_{\mathcal{D}}(A(S)) \leq \min_{h\in\mathcal{H}} L_{\mathcal{D}}(h) + \epsilon$$

We now want to **probably approximately** solve the following problem: Learner knows $\mathcal{H}$ and parameters δ, ϵ. Learner can request m data points, with m being dependent on δ and ϵ. Learner outputs hypothesis $\mathcal{A}(S)$, and it must be the case that with probability at least $1-\delta$, our notion of approximate correctness, i.e. $L_{\mathcal{D}}(A(S)) \leq \min_{h\in\mathcal{H}} L_{\mathcal{D}}(h) + \epsilon$, is satisfied.

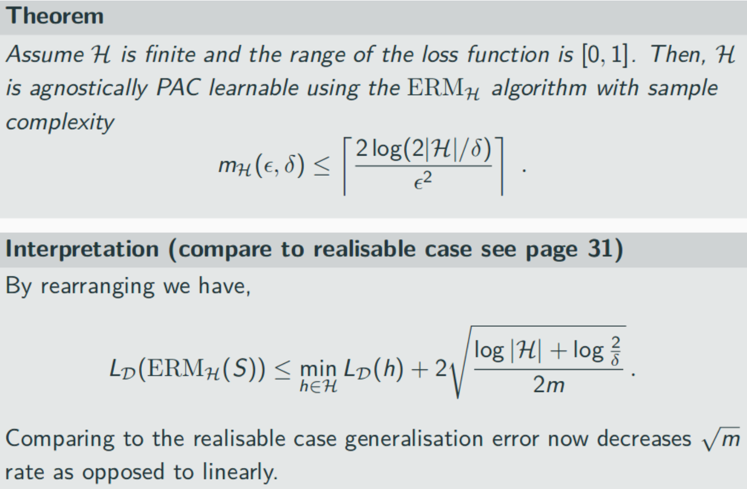
The **formal definition** of agnostic PAC is shown below.



A hypothesis class $\mathcal{H}$ is agnostic PAC learnable if there exists a function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a learning algorithm, $\mathcal{A}$, with the following property: for every $\epsilon, \delta \in (0,1)$, $m \geq m_{\mathcal{H}}(\epsilon, \delta)$, and distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$,

$$\mathcal{D}^m\left(\left\{S \in (\mathcal{X}\times\mathcal{Y})^m \;:\; L_{\mathcal{D}}(\mathcal{A}(S)) \leq \min_{h\in\mathcal{H}} L_{\mathcal{D}}(h) + \epsilon\right\}\right) \;\geq\; 1-\delta$$

A training set S is called ϵ-representative if $\forall \dot{h} \in \mathcal{H},\ |L_S(H) - L_{\mathcal{D}}(h)| \leq \epsilon$. **TODO:** *Include uniform convergence.*



Theorem

Assume $\mathcal{H}$ is finite and the range of the loss function is $[0,1]$. Then, $\mathcal{H}$ is agnostically PAC learnable using the $\mathrm{ERM}_{\mathcal{H}}$ algorithm with sample complexity

$$m_{\mathcal{H}}(\epsilon, \delta) \leq \left\lceil \frac{2\log(2|\mathcal{H}|/\delta)}{\epsilon^2} \right\rceil .$$

Interpretation (compare to realisable case see page 31)

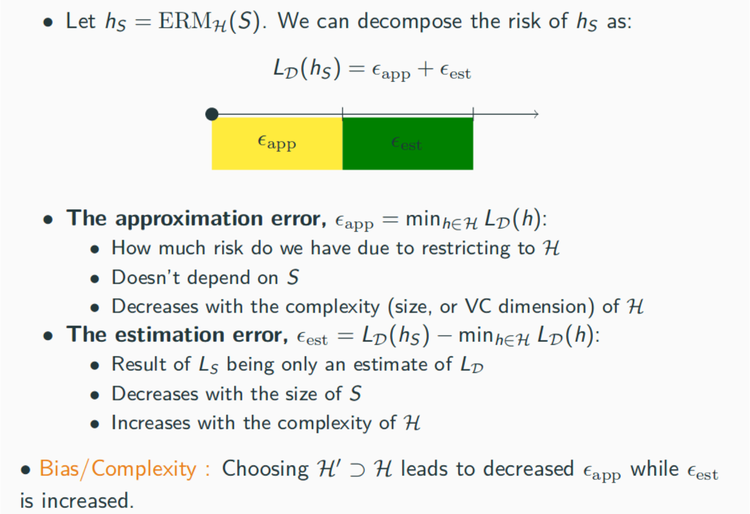
By rearranging we have,

$$L_{\mathcal{D}}(\mathrm{ERM}_{\mathcal{H}}(S)) \leq \min_{h\in\mathcal{H}} L_{\mathcal{D}}(h) + 2\sqrt{\frac{\log|\mathcal{H}| + \log\frac{2}{\delta}}{2m}} .$$

Comparing to the realisable case generalisation error now decreases $\sqrt{m}$ rate as opposed to linearly.

TODO: *Include proof for sample complexity?*

7.8 Error decomposition



- Let $h_S = \mathrm{ERM}_{\mathcal{H}}(S)$. We can decompose the risk of h_S as:

$$L_{\mathcal{D}}(h_S) = \epsilon_{\mathrm{app}} + \epsilon_{\mathrm{est}}$$

- **The approximation error,** $\epsilon_{\mathrm{app}} = \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$:
  - How much risk do we have due to restricting to $\mathcal{H}$
  - Doesn't depend on S
  - Decreases with the complexity (size, or VC dimension) of $\mathcal{H}$
- **The estimation error,** $\epsilon_{\mathrm{est}} = L_{\mathcal{D}}(h_S) - \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$:
  - Result of L_S being only an estimate of $L_{\mathcal{D}}$
  - Decreases with the size of S
  - Increases with the complexity of $\mathcal{H}$

- Bias/Complexity : Choosing $\mathcal{H}' \supset \mathcal{H}$ leads to decreased ϵ_{app} while ϵ_{est} is increased.

8 Advanced online learning

8.1 Partial feedback setting

Consider the following problem. You can pull one of the n levers. The lever you pick at step t is your "action" $\hat{y}_t$, where $\hat{y}_t \in [n]$. At the end of each step, nature produces the loss vector $l_t \in [0,1]^n$, which indicates the loss for each lever. This loss vector may be revealed to you fully or partially (see below). We aim to achieve "small regret":

$$\sum_{t=1}^{m} l_{t,\hat{y}_t} - \sum_{t=1}^{m} \min_{i \in [n]} l_{t,i} \leq o(m)$$

In the **full feedback setting**, after picking an action $\hat{y}$, you get to know the loss of each of the levers. So, looking back, you could see if you chose the "best" lever or not.

In **partial feedback setting** (the *bandit* setting), we only get to know the loss of the action we just performed. That is, we know how "good" the action we picked was, but we don't know if other actions were better or worse.

8.2 Exploration vs. exploitation

Note that each level we pull will have a different "reward" associated with it. Initially, we don't know the valuation of each lever so we will need to "explore" by trying levers at random. Once we find the best lever, we can "exploit" it by picking the same lever continuously.

Oftentimes the trade-off between exploration and exploitation is not trivial - the search space can be too big, rewards can be noisy, etc. This means that we need to find a good trade-off between exploitation and exploration to make sure we minimize the loss.

8.3 Importance weighting

An **estimator** $\hat{\theta}$ estimates the parameter θ of some distribution based on samples from that distribution. The estimator is **unbiased** if $E[\hat{\theta}] = \theta$.

Assume that the loss for each lever is not some fixed number l_i, but rather is sampled i.i.d. from the distribution $\mathcal{D}_i$ associated with that lever. That is, if we pick action $\hat{y} = i$, the loss we observe will be $l_{t,i} \sim \mathcal{D}_i$. Note that if we save $l_{t,i}$ every time we pick action i, and then take the average of all values we've seen, we're going to get an unbiased estimator of $l_{t,i}$.

However, if $\mathcal{D}_i$ changes over time, using the simple approach from above **will not result in an unbiased estimator**. To work around this issue, we introduce the **hallucinated loss vector** $l^h_{t,i}$.

Let v_t be our internal discrete distribution over actions $[n]$. The higher the probability $v_{t,i}$, the better the chance of an action i being picked. We maintain and update this distribution however we want.

Let $\hat{y}_t$ be the action we picked at step t, sampled from v_t. We set the **hallucinated loss vector** $l^h_{t,i}$ to be:

$$l^h_{t,i} := \frac{l_{t,i}}{v_{t,i}} \mathcal{I}[i = \hat{y}_t] \qquad \text{for all } i \in [n]$$

Where l_t is the loss vector given to us. Note that in partial feedback setting, l_t will be zeros everywhere except for the $\hat{y}_t$th element. This means the resultant hallucinated loss vector looks like:

$$l^h_{t,i} = (0, 0, ..., \frac{l_{t,\hat{y}_t}}{v_{t,\hat{y}_t}}, 0, ..., 0)$$

We can show that this means that $l^h_{t,i}$ is an unbiased estimator of the true loss vector:

$$E_{\hat{y}_t \sim v_t}[l^h_{t,i}] \;=\; \sum_{j=1}^{n} v_{t,j} \frac{l_{t,i}}{v_{t,i}} \mathcal{I}[i = j] \;=\; l_{t,i}$$

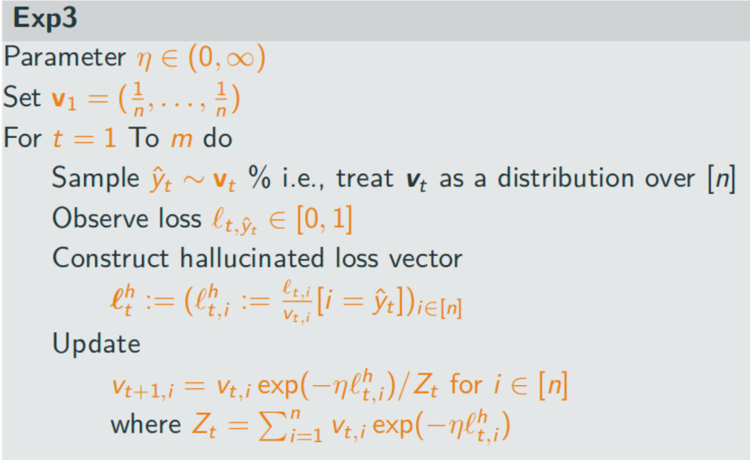
8.4 Exponential-weight algorithm for Exploration and Exploitation (EXP3)

The EXP3 algorithm is outlined below. It is based on applying the Hedge algorithm to the hallucinated loss vector, with the main difference that $l^h_{t,i}$ is unbounded while Hedge expects bounded loss vectors. The other problem stems from the source of randomness in l_t - we will clarify how l_t is generated by the adversarial model.

Note that **EXP3 is basically Hedge** applied the hallucinated loss vector.

8.4.1 Deterministic oblivious adversary

In this model, the adversary determines the loss vectors $l_1, ..., l_m$ before the algorithm is even started. This way, l_t are not allowed to change on-the-fly - the adversary cannot adapt after seeing what predictions we make. However, the adversary can base l_t's on our learning algorithm - that is, they can try to predict what conclusions we will make and change l_t's as to maximise our loss. Note in this model, adversary can simply simulate the stochastic model, sampling l_t's from static distributions.



```
Exp3
Parameter $\eta \in (0,\infty)$
Set $\mathbf{v}_1 = (\frac{1}{n},\dots,\frac{1}{n})$
For $t = 1$ To $m$ do
    Sample $\hat{y}_t \sim \mathbf{v}_t$ % i.e., treat $\boldsymbol{v}_t$ as a distribution over $[n]$
    Observe loss $\ell_{t,\hat{y}_t} \in [0,1]$
    Construct hallucinated loss vector
        $\ell_t^h := (\ell_{t,i}^h := \frac{\ell_{t,i}}{v_{t,i}}[i = \hat{y}_t])_{i\in[n]}$
    Update
        $v_{t+1,i} = v_{t,i}\exp(-\eta\ell_{t,i}^h)/Z_t$ for $i \in [n]$
        where $Z_t = \sum_{i=1}^{n} v_{t,i}\exp(-\eta\ell_{t,i}^h)$
```

8.4.2 EXP3 regret bound

One can show that for any sequence of losses $S = l_1, ..., l_m \in [0,1]^N$, the regret bound of EXP3 with $\eta = \sqrt{\frac{\ln N}{2mN}}$ is :

$$E[L_{WA}(S)] - \min_i L_i(S) \le \sqrt{2mN \ln N} \qquad \text{(for Hedge it was } \sqrt{2m \ln N}\text{)}$$

8.5 Matrix completion

Let $U \in -1, 1^{m\times n}$ be a matrix that encodes some boolean function, where rows and columns represent different inputs. Matrix completion problem requires our learner to "learn" the matrix structure in online fashion. That is, at every step nature selects some entry (i_t, j_t) from the matrix, and the learner must predict the output. If a mistake occurs, the learner updates some internal state and proceeds. Our goal is to bound the number of mistakes made by the learner using some function matrix complexity.

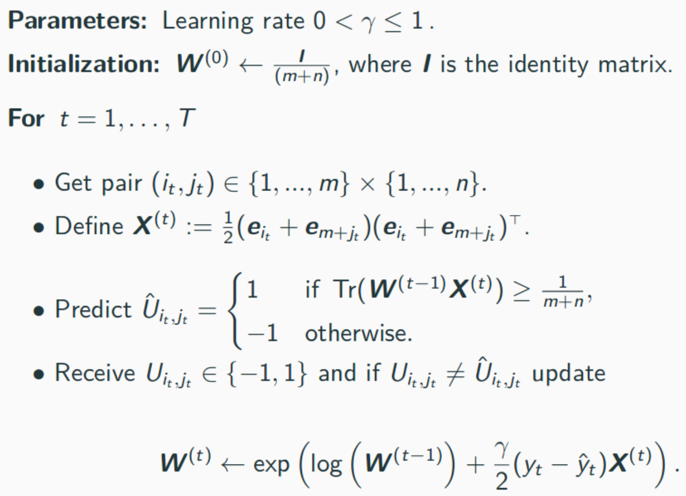
8.5.1 Factor models

We can think of this problem in terms of **factor models**. Consider the Netflix problem, where user i rates movie j. We can say that each movie has k factors associated with it (e.g. "how much" of movie is comedy, horror, thriller, etc.). Let $q_j \in \mathbb{R}^k$ represent these movie factors. Each user will then have some score with respect to each individual factor. Denote the full vector for each user as $p_i \in \mathbb{R}^k$. We can then predict the score user i will give to movie j by computing the inner product $p_i \cdot q_j$.

We can now define a matrix $P \in \mathbb{M}_{m\times k}$, where each row is p_i, and a matrix $Q \in \mathbb{M}n \times k$, where each row is q_j. We can rewrite matrix U as $U = PQ^T$, which is known as rank-k decomposition of U.

8.5.2 Matrix winnow

Consider the algorithm for **Matrix Winnow**:



```
Parameters: Learning rate $0 < \gamma \leq 1$.
Initialization: $\boldsymbol{W}^{(0)} \leftarrow \frac{\boldsymbol{I}}{(m+n)}$, where $\boldsymbol{I}$ is the identity matrix.
For $t = 1, \ldots, T$
    • Get pair $(i_t, j_t) \in \{1, ..., m\} \times \{1, ..., n\}$.
    • Define $\boldsymbol{X}^{(t)} := \frac{1}{2}(\boldsymbol{e}_{i_t} + \boldsymbol{e}_{m+j_t})(\boldsymbol{e}_{i_t} + \boldsymbol{e}_{m+j_t})^\top$.
    • Predict $\hat{U}_{i_t,j_t} = \begin{cases} 1 & \text{if } \mathrm{Tr}(\boldsymbol{W}^{(t-1)}\boldsymbol{X}^{(t)}) \geq \frac{1}{m+n}, \\ -1 & \text{otherwise.} \end{cases}$
    • Receive $U_{i_t,j_t} \in \{-1, 1\}$ and if $U_{i_t,j_t} \neq \hat{U}_{i_t,j_t}$ update
        $\boldsymbol{W}^{(t)} \leftarrow \exp\left(\log\left(\boldsymbol{W}^{(t-1)}\right) + \frac{\gamma}{2}(y_t - \hat{y}_t)\boldsymbol{X}^{(t)}\right)$.
```

Note that we can take $\log$ and $\exp$ of matrices by performing eigen-decomposition, then applying respective operations to the diagonal eigen value matrix.

8.5.3 Matrix complexity measures

Rank complexity: Low rank decomposition can act as a complexity measure for matrices. A matrix $U \in \mathbb{M}_{m\times n}$ has a rank-k decomposition if there are exist matrices $P \in \mathbb{M}_{m\times k}$ and $Q \in \mathbb{M}_{n\times k}$ such that $U = PQ^T$. Observe that a rank-k decomposition is specified by $k(m+n)$ parameters.

We define a quantity called "decomposition complexity" of matrix U as:

$$dc(U) := \min\left\{\dot{k} \; : \; P \in \mathbb{M}_{m\times k},\; Q \in \mathbb{M}_{n\times k},\; sign\left((PQ^T)_{ij}\right) = sign(U_{ij}) \;\; \forall\, i, j\right\}$$

$dc(U)$ basically means "the size of the smallest rank-k decomposition that generates some matrix M such that $sign(m_{ij}) = sign(u_{ij})$". Note that we only care about the sign of each element, we don't care about its actual value.

Margin complexity: Let P, Q be the matrices that achieve the $dc(U) = \min k$. Not that these matrices are not necessarily unique. With this in mind, we define margin complexity as:

$$mc(U) := \min_{P,Q} \max_{i\in[m], j\in[n]} \|p_i\|_2 \, \|q_j\|_2$$

That is, we take the largest rows p_i and q_j from the smallest such matrices P, Q and multiply their norms together.

The following relationships then hold:

$$\begin{aligned} vcd(U) &\leq mc^2(U) \leq rank(U) \\ vcd(U) &\leq \;\; dc(U) \;\; \leq O(mc^2(U)\log(mn)) \end{aligned}$$

8.5.4 Matrix winnow mistake bound

One could derive a tight upper bound (up to a logarithimic factor) for the number of mistakes:

$$\textbf{Matrix winnow mistakes} \leq 3.55\; \textbf{mc}^2(U) \cdot (m+n) \cdot \log(m+n)$$

We can also bound the number of mistakes from below. Given any learning algorithm $\mathcal{A}$ and any constant $l \in [n]$, one can show that there exists a matrix U with margin complexity $\textbf{mc}^2(U) \leq l$ and a sequence matrix of entries s.t.

$$lm \leq \textbf{Matrix winnow mistakes}$$

.

8.6 Multi-task learning

Multi-task learning is concerned with predicting entries of some vector $u \in \{-1, +1\}^n$ (one at a time). Given some kernel $K(x_1, x_2)$, we can use the perceptron to bound the number of mistakes.

Novikoff theorem: One can show that the number of mistakes to predict elements of u is bounded by:

$$\textbf{mistakes} \leq \min_{w \in \mathbb{R}^n} w^T K^{-1} w \max_{j \in [n]} K_{jj}$$

Assuming K is a strictly PD kernel and $\hat{u}_j w_j \geq 1$ for all $j \in [n]$. We can adjust our definition of margin complexity $mc(U)$ to rely on kernels and relevant weights instead - now, in place of P and Q, we consider the minimum kernel matrix and all possible weights that solve the problem with perfect accuracy. We then maximize over the quadratic of the max weight vector with the kernel matrix multiplied by the maximum diagonal element of the kernel matrix. Mathematically, we have:

$$mc^2(U) = \min_{p.d.K,\ w^{(1)},\dots,w^{(m)} \in \mathbb{R}^n} \max_{i \in [m], j \in [n]} {w^{(i)}}^T K^{-1} w^{(i)} K_{jj}$$

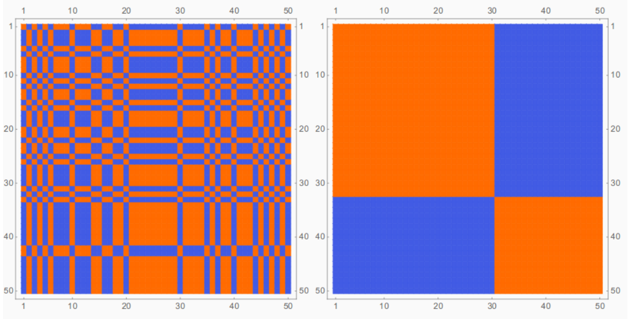
where $U_{ij} w_j^{(i)} \geq 1$ for all i, j. Note that this alternative definition formulation of margin complexity is identical to our original one, except this one uses kernel. Recall that our matrix winnow algorithm (without kernels) has successfully achieved the following bound:

$$\textbf{Matrix winnow mistakes} \leq 3.55\ \textbf{mc}^2(U) \cdot (m + n) \cdot \log(m + n)$$

This means that by applying matrix winnow (without kernels) to the multi-task problem, we can predict as well as the best kernel on the worst task (given $|tasks| > |items|$).

8.7 (k, l)-biclustered matrices

A matrix is (k, l)-biclustered if, after swapping its rows and columns, we can end up with a matrix with $k \times l$ grid of rectangles, each labelled 1 or -1. See example below.



A (2,2)-biclustered matrix

After permutation

If $U \in -1, +1^{m \times n}$, one can show that:

$$mc^2(U) \leq \min(k, l)$$

Let $\mathbb{B}_{k,l}^{m,n}$ denote the set of all (k, l)-biclustered matrices of size $m \times n$. It is also the case that $VCdim(\mathbb{B}_{k,l}^{m,n}) \geq k \cdot l$.

9 Graph-based semi-supervised learning (SSL)

9.1 Why SSL?

Semi-supervised learning lets us learn from labelled and unlabelled data at the same time. The motivation for this is that often data points which are close to each other will have the same label, so we only need to know the label of one point in a cluster. Unlabelled data is also much cheaper to obtain than labelled data.

9.2 SL vs USL vs SSL

Supervised learning aims to learn how to accurately predict an input/output problem.
Unsupervised learning aims to model the data, e.g. by clustering or running PCA.
Semi-supervised learning learns to approximate the input/output problem using a mixture of labelled and unlabelled data.

Inductive SSL aims to learn a function $f(x) \approx y$ for all available data (including labelled and unlabelled).

Transductive SSL aims to learn $f(x) \approx y$ only for unlabelled data.

9.3 Building graphs

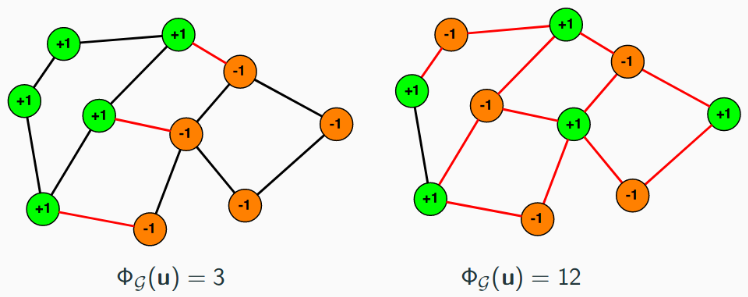
Intrinsic graphs are provided to us from existing problems - protein interaction, highway systems.
Extrinsic graphs are built by some algorithm, using k-NN, ϵ-ball, tree-based methods (MST or SPT), weighted graphs (e.g. edges are weighted using $e^{-c*D(p,q)}$. Different methods can also be combined to create a single graph.

The quality of predictions usually depends on the quality of the graph.

9.4 Algorithmic frameworks

9.4.1 Labelling as a graph complexity measure

The "complexity" of a graph can be measured by considering the final labelling achieved on that graph. Consider the image below.



$\Phi_{\mathcal{G}}(\mathbf{u}) = 3$ $\Phi_{\mathcal{G}}(\mathbf{u}) = 12$

Each of the graphs above was labelled by some classifier. For example, if the node of the graph represent cat and non-cat pictures connected using the ϵ-ball technique, nodes labelled 1 could represent cats and nods labelled -1 could represent non-cats.

The measure of complexity $\Phi_G(u)$ is defined as the number of edges that have different labels on each end. Intuitively, small $\Phi_G(u)$ means you'll need to make less cuts to split the graph into 2 clusters, which means that the graph is "smoother". Note that $\Phi_G(u)$ is discrete and hence can be hard to optimize. The graph Laplacian works around this issue by extending $\Phi_G(u)$ to real values.

9.4.2 Graph Laplacian

Let n be the number of nodes in the graph. Let $W \in \mathbb{M}_{n\times n}$ be the weight matrix, where W_{ij} represents the weight of the edge from i to j if such edge is present, 0 otherwise. Note that W is symmetric.

Let $D \in \mathbb{M}_{n\times n}$ be a diagonal matrix where D_ii is the sum of weights of all edges that involve node i. Graph Laplacian is them defined as follows:

$$L(G) := D - W$$

Note that graph Laplacian is positive semi-definite. The associated semi-inner product and semi-norm are:

$$\langle u, v\rangle_G := u^T Lv \qquad\qquad ||u||_G^2 = \langle u, u\rangle_G = \sum_{(i,j)\in E(G)} w_{ij}(u_i - u_j)^2$$

This semi-norm and semi-inner product work for any vector $u, v \in \mathbb{R}^n$ such that $\sum_{i=1}^n u_i = 0$ and $\sum_{i=1}^n v_i = 0$ (i.e. all elements must sum up to zero).

Let L^+ denote the standard pseudoinverse of the graph Laplacian. The associated kernel is then $K_G(i,j) = e_i^T L^+ e_j = L_{ij}^+$. It is not hard to show the associated feature map is $\phi(i) = e_i^T L^+$.

Note that the smallest eigen value is always 0 and the corresponding eigen vector is the 1 vector. For connected graphs, this is the only eigen value/vector.

Things you may not have known:

If $||v|| = 0$ for non-zero v then $||v||$ is a semi-norm.

If $||v|| = 0$ for $v = 0$ then $||v||$ is a norm.

9.4.3 Minimum cut and spectral clustering

Minimum cut tries to split the graph into two "natural" clusters by minimising some cut function. The standard mincut minimizes $cut(A,B) = \sum_{i \in A, j \in B} w_{ij}$, i.e. the total weight of all edges that will be broken by the cut. Note that the naive mincut will tend to cut off a single node to minimize the weight of broken edges. Mincut can be efficiently computed in $O(n^3)$ time.

Ratio cut also takes into account the sizes of the resultant clusters: $RatioCut(A,B) = cut(a,b)(\frac{1}{|A|} + \frac{1}{|B|})$. Ratio cut is better in practice because it tries to balance the size of each cluster, but it's NP-hard to compute.

We will consider a balanced version of the mincut that creates clusters of equal size, i.e. we optimize the following objective:

$$\min_{A,B:|A|=|B|} cut(A,B)$$

We will represent the solution as a vector $u \in \{-1,+1\}^n$, where $u_i = +1$ implies node i is in cluster A and $u_i = -1$ implies that it is in cluster B. Note that since we require $|A| = |B|$, the solution must satisfy $\sum_{i=1}^n u_i = 0$. We can now rewrite $cut(A,B)$ in terms of u:

$$cut(A,B) = \frac{1}{4} \sum_{i<j} w_{ij}(u_i - u_j)^2 = \frac{1}{4} u^T L u$$

With this definition of a cut, our problem now becomes to minimize $u^T L u$ w.r.t u such that $u \in \{-1,+1\}^n, u \perp 1$ and $\|u\|^2 = n$. Note that this problem is basically integer optimization, and cannot be solved efficiently.

If we relax the problem to allow $u \in \mathbb{R}^n$, we can obtain the solution by exploiting variational characterization of eigenvalues:

Generalized Rayleigh-Ritz: If $\lambda_1 \leq ... \leq \lambda_n$ are the eigenvalues of a positive symmetric matrix M, and $v_1, ..., v_n$ are the corresponding eigenvectors, it holds that:

$$\lambda_k = \min_u \frac{u^T M u}{u^T u} : u \perp v_1, ..., v_{k-1}$$

As mentioned in previous sections, the smallest eigenvalue of a Laplacian is always 0, with 1 being the corresponding eigenvector. This means that the second eigenvector is the solution to out relaxed balanced cut problem. This method is called **spectral graph clustering**.

Note that we can also obtain a solution for $RationCut(A,B)$ using a similar technique. Now, we require $u \in \{-\sqrt{\frac{|A|}{|B|}}, +\sqrt{\frac{|B|}{|A|}}\}$, and our optimization problem becomes:

$$u^T L u = \left(\frac{|B|}{|A|} + \frac{|A|}{|B|} + 2 \right) cut(A,B) = (|A| + |B|) RatioCut(A,B)$$

9.5 Graph Laplacian

9.5.1 Regularization and interpolation

The semi-norm of a graph Laplacian now gives us a natural measure for complexity of the graph. Using semi-norm as a complexity measure, we can add it as a regularization parameter to our problem.

For **interpolation**, we're going to assume there exists at least one $f^* \in \mathcal{F}$ that achieves perfect accuracy on training data. We then select the zero-error f^* with minimum complexity, using the limit case of regularization:

$$f_\lambda = \text{argmin}_{f \in \mathcal{F}}\, error(\text{training data}, f) + \lambda\, complexity(f)$$
$$\Downarrow$$
$$f^* := \lim_{\lambda \to \infty} f_\lambda$$

9.5.2 Minimum cut transduction

Assume we're given a set of labelled vertices, i.e. $(v_i, y_i) \in V(G) \times \{-1, 1\}$. We could also assign non-trivial weights to each edge. The hypothesis is given by:

$$\hat{u}_t = \text{argmin}_{u \in \{-1,+1\}^n} \{ \sum_{(i,j) \in E(G)} : w_{ij} |u_i - u_j| : u_{i_1} = y_1, ..., u_{i_l} = y_l \}$$

Again, this problem can be solved min-cut/max-flow algorithm in $O(n^3)$. Note that the objective minima is unchanged if we relax the problem, and that this problem doesn't have a unique solution (and hence is ill-posed). Mincut is not very popular as it tends to produce unbalanced labellings.

9.5.3 Laplacian-based transduction

Laplacian based transduction is very similar to minimum cut transduction, except now our solution is the (already relaxed) interpolant vector:

$$\bar{u}_t = \text{argmin}_{u \in R^n} \{ \sum_{(i,j) \in E(G)} : w_{ij} |u_i - u_j|^2 : u_{i_1} = y_1, ..., u_{i_l} = y_l \}$$

To generate a prediction for vertex v, we simply evaluate $\hat{u}_v = sign(\bar{u}_v)$. The solution can be found in cubic time by solving a system of linear equations.

Alternatively, one could optimize Laplacian interpolation using a consensus algorithm: At every time step t, we pick one of the free nodes (the ones with no prior label) uniformly at random and average the values of their neighbours. This method is useful as it is fault tolerant, and can be performed asynchronously and in parallel. Additionally, the calculation for each individual node is simple and local. For a weighted graph, we replace mean with a weighted mean.

Theorem (Harmonic solution): The problem defined using the interpolant vector from above implies that the solution is:

$$\begin{aligned} \bar{u}_i &= \frac{\sum_{i=1}^n w_{ij} u_j}{d_i} \quad \text{for } i = l+1, ..., n \\ \bar{u}_i &= y_i \quad \text{for } i = 1, ..., l \end{aligned}$$

9.6 Interpreting Laplacian-based transduction

9.6.1 Interpreting using resistive networks

One way to interpret the solution is to treat the graph as a resistive network, were labels specify the voltage constraints and weights correspond to inverse resistances.

We define the power of a labelling u as:

$$P(u) := \|u\|_G^2 = \sum_{(i,j)\in E(G)} w_{ij}|u_i - u_j|^2 = u^T Lu$$

We then obtain the optimal solution by minimizing the power:

$$\bar{u} = \mathrm{argmin}_{u\in\mathbb{R}^n}\{P(u) : u_{i_1} = y_1, ..., u_{i_l} = y_l\}$$

Recall the basic rules of electric circuits:

- $Voltage = Current * Resistance.$
- In a series circuit, we just add up resistance of all components to get total resistance. The current through all components is the same, but the voltage is split up based on the resistance.
- In a parallel circuit, we compute total resistance through all branches using $\frac{1}{R_{eff}} = \frac{1}{R_1} + \frac{1}{R_2} + ...$. The voltage is the same through every branch, but the current is split up based on the resistance.

Now recall that the resistance of an edge is the reciprocal of its weight. We can define **effective resistance** between node i and j as the voltage difference needed to induce a unit current flow between them. This measure is equivalent to the total resistance of the network when a voltage source is connected across i and j.

Finally, one could show that effective resistance can be expressed as follows:

$$r_G(p,q) = \frac{1}{\min_{u\in\mathbb{R}^n}\{\|u\|_G^2 : u_p = 1, u_q = 0\}}$$

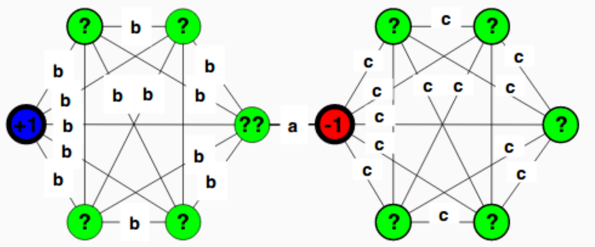
Surprisingly, effective resistance can be shown be the squared distance induced by the kernel L^+:

$$r_G(p,q) = L^+_{pp} + L^+_{qq} - L^+_{pq}$$

Moreover, kernel L^+ encodes the following connectivity information:

$$\begin{aligned} L^+_{vv} &= \frac{\sum_{i=1}^n r_G(v,i)}{n} - \frac{\sum_{i>j} r_G(i,j)}{n^2} \\ &= \frac{R(v)}{n} - \frac{R_{tot}}{n^2} \end{aligned}$$

Note that effective resistance between any two nodes within a clique whose edge-resistances are all $\leq d$ is $r_{m-clique} \leq \frac{2d}{m}$.



Two $m-cliques$ connected by an edge with distances $a<b$

Consider the example below, where numbers next to edges indicate their effective resistance. The node $??$ would receive the label $+1$ as long as $\frac{2b}{m} < a$, which implies that labelings respect cluster structure.

9.6.2 Interpreting using random walks

We can denote a transition probability P_{ij} of going from vertex i to vertex j on a particular step as:

$$P_{ij} := \frac{w_{ij}}{\sum_j w_{ij}}$$

Assume the labels are now $\{0, 1\}$. Denote the set of all fixed nodes labelled 0 as V_0, and the set of all nodes labelled 1 as V_1. Now, let p_i be the probability then a random walk started at node i reaches V_1 before it reaches V_0. One could show that p_i is equal to weighted average of probabilities of neighbours of i, which is equivalent to consensus approach shown above (which is sometimes called label propagation).

Theorem: Given an unweighted graph G and adjacency matrix W, and reusing definitions of p_i, V_0, V_1 from above, one could show that the solution is:

$$p = \operatorname{argmin}_{p\in\mathbb{R}^n}\{\sum_{i<j} : w_{ij}|p_i - p_j|^2 : (p_k = 0)_{v_k\in V_1}, (p_k = 1)_{v_k\in V_1}\}$$

Let $C_G(i, j)$, the commute time, denote the expected time it will take to travel from i to j and back again (based on probabilities from above). We can link commute time and effective resistance together as commute time between i and j is twice the total degree of the graph times effective resistance between i and j:

$$C_G(i, j) = r_G(i, j)\sum_{i,j}^{n} w_{ij}$$