Regression: (Root) mean square error (RMSE/MSE) is one of the most common metrics, easy to optimize directly, best prediction is the average. Mean absolute error (MAE) is less sensitive to outliers than MSE, best prediction is the median. R^2 is simply MSE divided by the baseline, which gives more meaningful units for MSE.

Classification: Accuracy is simplest, but it only works with hard predictions and is hard to optimize. Los-loss is $-\frac{1}{N}\sum_{i=1}^{N}(y_i \log p_i + (1-y_i)\log(1-p_i))$, directly optimized in many models.

For area under curve (AUC), plot $TPR = \frac{TP}{TP+FN}$ against $FPR = \frac{FP}{FP+TN}$, then intergate to get AUC. Gives probability that the model ranks a random positive example more highly than a random negative example. AUC is **scale invariant** and **classification-threshold-invariant**, note that these two properties are not always desirable.

Standard SNE uses RBF $\mathbf{d}(x,\tilde{x}) = exp(-\left\|x-\tilde{x}\right\|^2/2\sigma_i^2)$ to compute edge weights and make a totally connected graph. σ_i^2 is picked for each data point, want it to be small for dense areas and big for sparse areas. Transition matrix $p_{j|i}$ is then computed by getting random walk probabilities. For low dimensional map, we use $\mathbf{d}(y,\tilde{y})$ with no denominator and form $p_{j|i}$ matrix. **Objective** is minimize sum of KL distances for all data points, giving derivatives:

$$\frac{\partial C}{\partial y_i} = 2\sum_j (p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})(y_i - y_j)$$

t-SNE uses a heavy-tail distribution, changing q_{ij} distance measure to $\mathbf{d}(y,\hat{y}) = (1+\left\|x-\hat{x}\right\|^2)^{-1}$. Heavy-tail t-distribution eliminates crowding, it's invariant to scale and is cheaper to compute. t-SNE also uses a symmetric joint distribution p_{ij} and q_{ij}. q_{ij} is achieved by normalizing each weight by the sum of all edges. p_{ij} is achieved by setting $p_{ij} = \frac{p_{j|i}+p_{i|j}}{2n}$. t-SNE gradient is:

$$\frac{\partial C}{\partial y_i} = 2\sum_j (p_{ij} - q_{ij})(y_i - y_j)(1+\left\|y_i - y_j\right\|^2)^{-1}$$

t-SNE can be modified to display a random subset of points called **landmark points**, while using information from the entire data set. To compute p_{ij} for landmarks L_i, we build a k-NN graph for each landmark for some fixed k, then set p_{ij} to be probability that a random walk through neighbourhood graph of L_i, starting at L_i, will reach L_j.

The cost of each iteration is $O(N^2)$, which is prohibitively expensive on large data sets. We use the following **optimization tricks** to improve compute speed:

- We can replace p_{ij} by a sparse approximation to decrease computation time - for each input point, we only compute probabilities for the $3u$ closest neighbours. We find neighbours efficiently using **vantage-point trees**.
- We can rewrite t-SNE gradient as $4(p_{ij}-q_{ij})q_{ij}Z(y_i-y_j)$, which can then be multiplied out to give attractive and repulsive forces. $4(F_{attr}+F_{rep})$. F_{attr} is cheap and can be computed in $O(uN)$ steps. F_{rep} is expensive $O(N^2)$, but can be decreased to $O(N\log N)$ using **Barnes-Hut**. Compute quad or octree, at every step decide whether we can replace the branch with the corresponding non-leaf node (is it far enough?).

Triangle inequality is $\mathbf{d}(y,q) \le \mathbf{d}(x,y) + \mathbf{d}(x,q)$, useful when $\mathbf{d}(y,q)$ is unknown. If $\mathbf{d}(y,q) \le \frac{1}{2}\mathbf{d}(x,y)$, point y is guaranteed to be further away from q than x and we don't need to compute the distance.

Orchard's algorithm works by precomputing all distances, then forming sorted lists for each training data point. On new input, pick the first guess at random, then "work" through the list be either by terminating, or switching to a better list. An example where **Orchard is worse** than naive approach is a straight line graph, where the input is on one end, and our first guess is on another end. Orchard is better on the same graph when input is in the middle of the graph.

AESA works by forming maximum lower bounds for all unknown $\mathbf{d}(y,q)$. We eliminate all y's whose lower bound is above our current best guess. We then pick y with the lowest bound as next candidate.

Both Orchard and AESA take $O(N^2)$ storage, $O(DN^2)$ time during precomputation. Orchard needs to examine $M < N$ list members, giving total time $O(DM)$ on new input. Orchard needs to compute $M < N$ bounds, taking $O(M(N-M))$ for each bound.

Buoys can decrease storage to $O(BN)$, and can speed up AESA and Orchard by preelimnating points based on upper/lower bounds. Linear AESA uses can use the Buoy lower bound (slightly increasing the number of distance calculations), and Orchard can use upper bounds instead of actual distance (though it doesn't make much of an improvement).

KD-trees always have storage $O(N)$ and require $O(N)$ steps to evaluate on new input. Formed by recursively considering the dimension with largest spread and picking the median. To evaluate on new input, find a leaf node by going left or right in the tree, then check distances to all parents (and update best guess accordingly), then check whether you need to check other branches. **Vantage point** trees simply use radius instead of hard "splits".

SVD: $A = ULV^T$. It describes A as a sum of components ranked by importance. Can be computed in $O(\min(mn^2, m^2n))$. All singular vectors are orthogonal and elements can be negative so its not good for topic modelling.

PCA: Similar to SVD but only selects the first k eigenvectors corresponding to the highest k eigenvalues (in order).

NNMF: $\min \left\|X - WH\right\|_F^2, \quad W \ge 0,\ H \ge 0$. Find W and H by iteratively fixing one and optimise for the other.

NNMF using ALS: Can simply solve unconstrained optimization problem, then project using: $W = max(0, (XH^T)/(HH^T))$. Usually does not converge but is good for initializing values before switching to another algorithm.

NNMF using NNLS: $W = \text{argmin}_{W\ge 0}\left\|X - WH\right\|_F$. Requires **separability assumption** implies that there is an identity matrix embedded inside A. Since $M = AB$, this implies that k rows of M embedded in B! We can use a greedy algorithm to find these k rows - find a row that is in the convex hull of all other rows, then remove it.