K-NN Algorithm .•**Given** X to classifieds ,and K. • **find** its **K nearest neighbor** in the training set (k smallest d(x,xi)) •**classify** according the majority for the nearest neighbor.•**Supervised Learning**

Dis: **Euclidean**(x,x') = sqrt(sum_d=1_To_D{ (x_d - x'_d)^2 }) / **Manhattan**(x,x') = sum_d=1_To_D{ |x_d - x'_d| }

Assumption: Training set and test set drawn(took) from the same **statistical distribution** If not the **decision boundary** not be useful on test set •**Training set** must representative all objects the system is likely to encounter.

Notes: **Test set:** Use the test set to gauge its performance.•**small k**: Overfitting •**Big K:** smoother boundary, training error increases •**misclassifying** points near the decision boundary •**more training examples** of one kind than other represented this class unduly(no fair) favored •**Real data often lies in smaller subspaces.**

Knn Limitations: •**Performance issues:** Must load all of the training data and calculate distances to all training samples. It can be done in a naive way or using fancier data structures such as **K-D trees.** However, this is still **slow** for large datasets. •**Distance metric:** The **vanilla** version is used with the simple **Euclidean distance**, which is a **problematic** distance metric in high dimensions as well as with **noisy features** or features of **different type.**

Improve:• **Weighing neighbors** by the inverse of their class size converts neighbor counts into the fraction of each class that falls in your K nearest neighbors. •**Under-sampling** the dominant class. •**Augmenting t**he other class by generating synthetic examples.

Choosing k/ cross-validation :•**Split** training set to training set and validation set •**Choose** k that minimizes the classification error on the validation set (min on the graph(error/complexity) of the validation set) •**Train** for different K's and pick k that **y**ields the highest **accuracy** on the validation set

graph(error/complexity): •**training curve** decreases when the degree/K increases •**test curve** eventually increases again •relevant for all all ML algorithms

Validation graph: accuracy percentage /K •**one single** meta-parameter k so easy to be tuned in cross-validation.

Rosenbrock Function: banana , **r(x,y)**=100(y - x^2)^2 + (1-x)^2 , **f(x,y)**=-1 if r(x,y)<T / 1 otherwise

Data Reduction: Condensed Nearest Neighbors Let **X** be the set of **training samples** and **P** the **set of prototypes**: • **Initialize** • **Repeat 1.** Look for x in X such that its nearest prototype in P has a different label than itself. **2.** Remove x from X and add it to P. •**Note:** •**Fast** computation at inference time

Greedy k-NN Graph Construction: **Parallel-iterative algorithm**: •**Given** a random graph •Each node looks for potential new neighbors: **1.** Among random nodes (optional). **2.** Among "friends of friends". **Note: Gossip Based Computing:** •Highly parallel. •Creates a random graph. •Robust to churn, partition, breakdowns. •Well adapted to peer-to-peer networks.

Number of Neighbors: •**More** neighbors: More coverage .•**Fewer** Neighbors: Better accuracy.

2D Voroni Diagram: For N training set divided the diagram for N cells each present the closest points for sample •**Note** depends on distance calculation kernel •**Decision Boundary:** divide the graph for sector of classification •**Note** can formed by selected edges of the Voronoi Diagram.

Multi Class: •**high-dimensional** space, everything far from everything. **Euclidean** becomes less meaningful.

effectiveness estimator: **distance** between neighboring training samples < d that depends on the problem.

Linear Regression: **supervised** •Given N pairs {(xi, ti)}, find the separations for two classes •**Line fitting**

•**Line:** y-intercept w0 The slope **w1 = Δy /Δx** , **y(x; w) = w0 + w1x** •**positive/Negative emotions:**up/down •**2-D** define a plane, y = w0 + w1•x1 + w2•x2 •**N-D** y=w^T•x , **minimize_W** { 1/N• ∑(W^T•xi - ti)^2 }

Alghrithem: **USE for** Predict quantities: •**Collect** dataset with **continuous** target variable.

Preprocess: Clean data, handle missing values, scale features, and split into training and test sets. •**Initialize** linear model parameters (weights and bias). •**Train:** Use gradient descent to minimize the mean squared error. •**Evaluate**: Test the model and measure performance using metrics like RMSE (Root Mean Squared Error).

•**Reggression:** Now you can predict an estimate of the corresponding yt as yt = w* + w*xt

Train : •**Given** Xn data •**Find** w0 & w1 by **min_W {1/N ∑(yi - ti)^2}** (**Leas squares**)

$$\min_{\mathbf{w}} \frac{1}{N}\sum_{i=1}^{N}(\mathbf{w}^T\mathbf{x}_i - t_i)^2$$

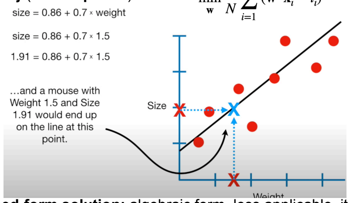
•**Evaluation:** •**compares the predictions** of model with true annotations of test data•**fixed** parameters

•testing data must be **separated**!

Mean Squared Error (MSE): •loss function for Nt test samples

MSE = 1/Nt• ∑ (yi − ti)^2 •where yi = prediction for test sample i and ti = corresponding ground-truth value

•**Root Mean Squared Error (RMSE)** • Square-root of the MSE



Evaluation metrics regression:

•**Mean Absolute Error (MAE):** MAE = (1/Nt) ∑ |yi - ti|

•**Mean Absolute Percentage Error (MAPE):**

MAPE = (1/Nt) ∑ |(yi - ti)/ti| * 100

Minimization problem: •**Gradient descent** (applicable), •**Closed-form solution:** algebraic form, less applicable, it does apply linear regression, $\mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t} = \mathbf{X}^\dagger\mathbf{t}$

Interpreting a linear model:• **Warning**: **coefficient**(מקדם) magnitude(גודל) **depend** on feature/attribute magnitude • **Coefficient** might **small** to compensate that the feature range large • **Normalizing** the data can addressed it

Maximum Margin Classifier, Linear Support Vector Machine (SVM): Finds a **hyperplane** separating classes maximally by maximizing the **margin** between closest points (support vectors).

Algorithm: •**Optimizes** ||w|| subject to **tn * (w · xn) ≥ 1 - ξn**, •**minimizes** (1/2) * ||w||^2 + C * Σ ξn with constraints tn * (w · xn) ≥ 1 - ξn and ξn ≥ 0.

Large C: Prioritizes a **smaller margin** and **fewer training errors**, emphasizing fitting the training data closely, potentially at the cost of overfitting.

Small C: Promotes a **larger margin** but is more lenient on **misclassifications** in the training data, which might lead to better generalization on unseen data by reducing overfitting.

Advantages: •**Effective** in high-dimensional spaces, •**efficient** with memory as it uses only support vectors, •**can** handle non-linearly separable classes using the kernel trick.

Disadvantages: •**Choosing** correct C is critical, sensitive to noisy data (outliers), •**struggles** when the number of features vastly outnumber samples, •**not** inherently suited to multi-class problems.•**requires separable** data

Usage: in applications such as face detection, handwriting recognition, image **classification**, and bioinformatics.

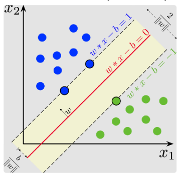
Slack Variables: Allows flexibility in separating data by introducing **ξn** to soften the margin. Points can either lie inside the margin or be misclassified, depending on the value of **ξn**.

Convex Optimization Problem: The problem is **convex**, and efficient algorithms are available for finding the **global optimum** . feature range large • **Normalizing** the data can addressed it

Support Vector Machine (SVM) Find the optimal hyperplane that maximizes the margin between two classes in a dataset. **minimize:** (1/2) * ||w||^2 subject to y_i(w.x_i + b) >= 1 for all i in the training set.

Here, **w** is the weight vector, **b** is the bias term, **x_i** is the feature vector, and **y_i** is the class label (+1 or -1).

Regularization: Introduce slack variables ξ_i to allow some misclassifications.
minimize (1/2) * ||w||^2 + C * Σξ_i subject to y_i(w.x_i + b) >= 1 - ξ_i for all i.
C is a **regularization parameter** controlling the **trade-off** between maximizing the margin and minimizing the classification error. **Higher C** values lead to less margin violation but may overfit.



Lagrangian Formulation & Dual Problem: Convert the constrained optimization problem to **unconstrained** one using **Lagrange multipliers.**
L(w, b, α) = (1/2) * ||w||^2 - Σα_i[y_i(w.x_i + b) - 1]
Dual problem: Maximize the dual objective wrt α subject to α_i >= 0.
Solution: w* = Σ(α_i * y_i * x_i)

Kernel Trick: Transform the input space into a higher-dimensional feature space.
•Use a kernel function
K(x, x') = φ(x).φ(x') to compute dot products in the transformed space without explicitly computing the transformation φ. •**Common kernels:** Linear (K(x, x') = x.x'), Polynomial (K(x, x') = (1 + x.x')^d), Radial Basis Function (RBF) or Gaussian (K(x, x') = exp(-γ||x-x'||^2)).

Support Vectors: •**Data** points that lie on the margin (or violate the margin) are the support vectors. •**Decision** function only depends on support vectors: **f(x) = Σ(α_i * y_i * K(x, x_i)) + b**.

Parameter Selection: Use **cross-validation** to select the best parameters for the SVM (e.g., **C**, kernel parameters).

Training the SVM: Use optimization algorithms like **Sequential Minimal Optimization** (**SMO**) to solve the dual problem and find the Lagrange multipliers (α_i).

Making Predictions: For a new input **x**, classify based on the sign of the decision function **f(x) = Σ(α_i * y_i * K(x, x_i)) + b**.

Multiclass Classification: For multi-class classification, use one-vs-one or one-vs-all strategies.

Scaling and Preprocessing: Scale input features for better performance. •Perform dimensionality reduction if needed.

Model Evaluation: Evaluate the SVM model using metrics like accuracy, precision, recall, or F1-score.

Supervised Learning: Train using an annotated training set •**Unsupervised Learning:**training set is not annotated and the system must also learn the classes.

K-Means: •Group the samples into K clusters •**Minimize**:**∑_K∑_N{Xijk - μk)^2}**
mean μ = 1/N • ∑i=1_To_N {Xi} •Always **converge(**might not for best solution) •**Solution depends** on the initialization •**Best** on **homogeneous** data •**Compactness** of clusters

Algorithm: •**Given** K,X data points •**Initialize** {μ}k randomly •**Until convergence**: **1)Assign** each point xi to the nearest center μ according Euclidean dis **2)Recomputing** the means centers μ

Stop criterion: •**Fixed number** of iterations(arbitrary , small number lead to bad results) •**Difference** of center locations between two iterations •**Converge**

Note: •**Try** different random initialization and keep the best result in term of the sum of square distances.•**Bigger K**: **Average** within-cluster distance decreases ,**too** big clusters make the results meaningless, **elbow curve** is where the drop in within-cluster distances becomes less significant

Inhomogeneous Data: all the entries/ features are of the same type(all being numbers/Or only text) . •**Euclidean** distance is most appropriate • **2D toy** data, both dimensions commensurate(תואם) magnitude. •**Color image**, each dimension represents a color channel and varies in the range .•**In practice**: **Different data** dimensions may have different magnitudes, **They can** encode different types of information. **Heterogeneous data:** •**Large** values boost Euclidean distance and small ones much less. • **It does** not mean that they are more or less meaningful for clustering •**Solutions**: Scale each dimension by subtracting(הפחתה) the smallest value and scaling the result to be between 0 and 1.• Use a different metric such as the Manhattan distance.

Clustering Connectivity: clusters we observe arise from the connectivity of the points.

Graph-based: **Clustering Connectivity** clusters we observe arise from the connectivity of the points. •**Group** the points based on edges in a graph •**Strong connections** indicate points that should be clustered •**Weaker** suggest that the graph can be cut into pieces/ partitions •**Graph** can be **restricted** to K-nearest or similarity (or affinity) matrix neighbor of each point **Algorithm:** •**Input**: graph ,nodes = data points ,edge weights = similarities.

•**Split** and Evaluate: Divide graph for two clusters and calculate cut cost

•**Find** the Best Split: Iterate through different divisions and choose the one that minimizes the cut cost.

Output: The two clusters that result from the best division.

•**Similarity** between points **Wij = exp({-||xi-xj||^2} / σ^2)** (**fully connected**)

•**Graph Cut:** dividing a graph into two parts, A and B **cut(A, B) = ∑ Wij** the cost

•**Normalized Cut:** modification the graph cut that considers the size of each partition to avoid **favoring imbalanced** partitions.

Ncut(A, B) = (cut(A, B) / Vol(A)) + (cut(A, B) / Vol(B))

•**Volume Partition (Vol(A)):** Sum of weights of all edges connected to nodes in partition A.

Vol(A) = ∑ Wij where 'i/j in A/B and Wij is the weight of edge between nodes i and j.

•**Similarity Matrix (W):** An N x N matrix containing similarity scores between all pairs of points. Wij represents the similarity between nodes i and j.

•**Degree Matrix (D):** An N x N diagonal matrix where Dii is the sum of the weights of edges connected to node 'i' **Dii = ∑ Wij** for all 'j'.

•**Graph Laplacian (L):** Defined as (D - W), it's used in solving the **normalized cut problem** L = D - W

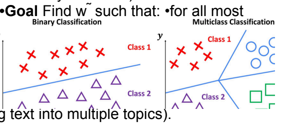
•**Relaxation:** •**Transform** normalized cut problem to an eigenvalue problem:

(D - W)y = λDy, where D = degree matrix (sum of edge weights for each node) and W = similarity matrix (weights between nodes). •**(D - W)** known as Graph **Laplacian** (L). •**Solve** for **eigenvector** associated with the **second smallest eigenvalue.** •**Eigenvector: positive** value suggests the node belongs to one group, **negative** value suggests it belongs to the other group.

•**Note: Separation** isn't perfect. Use a **threshold** (often the median of the eigenvector values) **to assign nodes** to groups more clearly.

Algorithm: **K-way(K>2)** partitioning **options**: •**Recursively** apply 2-way partitioning - not efficient and unstable. •**Use K eigenvectors** to represent each point as a K-dimensional vector, then apply K-means clustering to these vectors (dimensionality reduction followed by K-means)

Linear Classification: •**supervised learning** •**1 Class (binary)** •**Goal** Find w˜ such that: •for all most positive samples w˜•x˜>0 • for all most negative samples w˜•x˜<0.



Algorithm:•**Define** a linear decision boundary.
•**Train** weights and bias by minimizing a loss function.
•**Classify** new data points based on the sign of (w^T * x + b).
Uses: •**Classifying** data into two categories (spam vs.non-spam
•**Can be extended** for multi-class classification (e.g., categorizing text into multiple topics).

Minimize: **E(w˜) = ∑_N{Sign(w˜ • x˜n)•tn}**

Hyperplane: x ∊ RN, w˜•x˜ = 0 with x˜ = [1 | x]. **Signed distance:** w˜ •x˜, with w˜ = [w0|w] and ||w|| = 1

Algorithm Decision boundary : •**Set** w1 to 0 . •**Iteratively**: pick a random index n. **1)** If x˜n is correctly classified, do nothing. **2)** Otherwise, w˜t+1 = w˜t + tn•x˜n. **Test:** **y(x;w˜)** = **1** if W˜•X˜ ≥ 0 / **-1** otherwise

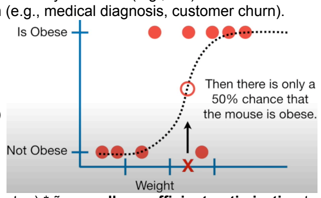
Notes: Randomizing helps,

Centered Perceptron: decision boundary goes through the origin. • **Algorithm**:Center the xns so that w0 = 0….same **Problem**:•**no difference** between close and far from decision boundary. •**We want** positive/ negative examples to be far as possible from it ,•**Two different solutions** among infinitely many perceptron has no way to favor one over other

Convergence Theorem: If **γ > 0** and a parameter vector w*, with **||w*|| = 1**, such that ∀**n, tn•(w* • xn) > γ** the **perceptron** algorithm makes at most **R^2/γ^2 errors, like Linear Classification** just **instead Perceptron** pass into **Sigmoid**:(differentiable •derivatives easy to compute •asymptotically=0/1)

Logistic Regression: •**supervised** •**GLM** •**Algorithm**:•**Compute** a linear combination of features. •**Pass through** the sigmoid function to get probabilities. •**Train weights** and bias by minimizing cross-entropy loss (gradient descent) •**Classify new** points based on probability threshold (e.g., 0.5).

Uses: •Estimating probabilities for **Binary classification** (e.g., medical diagnosis, customer churn).
•**Extendable** to multi-class via softmax
(e.g., digit recognition, classifying images).
•**Minimize** **cross-entropy loss**
E(w) = Σ {t_n * ln(y_n) + (1 - t_n) * ln(1 - y_n)},
where y_n is prediction for input x_n.
t_n is the true label of the n-th data point(= 0 or 1 binary)
•**Minimizing E(w)** pushes y_n close to 1
when t_n = 1, and close to 0 when t_n = 0.
•**Conversely**, if y_n and t_n are mismatched
(e.g. y_n < 0.5 when t_n = 1), the loss is larger.



•**E(w) is convex**; its **gradient** with respect to w is Σ (y_n - t_n) * x̃_n. •**allows efficient optimization** to find **global optimum**. •**y(x; w)** **can be interpreted** as the **probability** of x belonging to one class. It equals 0.5 on the decision boundary, and approach

•**Logistic regression** essentially **finds maximum** like solution assuming **Gaussian** noise.
•**minimize** the error-rate at **training time**.

Multi-Class Logistic Regression: •**linear** problem •**sigmoid** function is **monotonic**, the formulation is almost **unchanged**. •Only the **objective function** being minimized need to be reformulated.

Activation: $a^k(\mathbf{x}) = \tilde{\mathbf{w}}_k^T\tilde{\mathbf{x}}$

Probability that **x** belongs to class k: $y^k(\mathbf{x}) = \frac{\exp(a^k(\mathbf{x}))}{\sum_j \exp(a^j(\mathbf{x}))}$

Multi-class entropy: $E(\tilde{\mathbf{w}}_1, \ldots, \tilde{\mathbf{w}}_K) = -\sum_n \sum_k t_n^k \ln(y^k(\mathbf{x}_n))$

Gradient of the entropy: $\nabla E_{\mathbf{w_j}} = \sum_n (y^k(\mathbf{x}_n) - t_n^k)\mathbf{x}_n$

•**Multi-c Cross-En Loss: Convex/Easy compute**

Problems: •**should** accept to misclassify a few training samples. •**tries** to minimize the error-rate at training time.• **Can** result in poor classification rates at test time •does not guarantee the largest margin

AdaBoost (Adaptive Boosting): Boosts weak classifiers (models slightly better than random guessing) to create a strong classifier through combination.

Algorithm: •**Input**: Training data set {(x_n, t_n)} where t_n ∊ {-1, 1} for n = 1 to N; number of iterations T.
•**Initialize** data weights: for each n, w_1 = 1/N. •**For t = 1 to T: 1)**Find classifier y_t that minimizes weighted error: ε_t = Σ [w_t * I(t_n ≠ y_t(x_n))]. **2)** Evaluate classifier weight: α_t = 0.5 * log((1 - ε_t) / ε_t). **3) Update weights**: w_(t+1) = w_t * exp(α_t * I(t_n ≠ y_t(x_n))).

Final Classifier: Y(x) = sign(Σ(α_t * y_t(x)) from t = 1 to T). •**Weight Adjustment**: Misclassified points gain higher weights, prioritizing them in subsequent iterations.

Training error decreases exponentially if weighted errors (ε_t) of component classifiers are < 0.5: Σ[t_n ≠ h(x_n)] < Σ[ε_t * (1 - ε_t)]^(t) for t = 1 to T. **Testing error** may rise due to overfitting.

Versatility: AdaBoost is generic and can work with various weak classifiers, not just linear.

Cascade Method: Enables efficient object detection by swiftly eliminating negative instances. Vital for real-time applications prioritizing runtime speed.

Validation Set: Vital for tuning the number of iterations and averting overfitting.

Advantages: •Augments accuracy via classifier combination. •Automated classifier weight handling. •Adaptable to various classifier types. •Conducts feature selection, emphasizing critical features.

Disadvantages: •Sensitive to noise and outliers. •Susceptible to overfitting if weak classifiers are too complex or parameters aren't well-tuned.

Applications: Prevalent in face detection, spam filter, medical diagnostics, classification tasks.