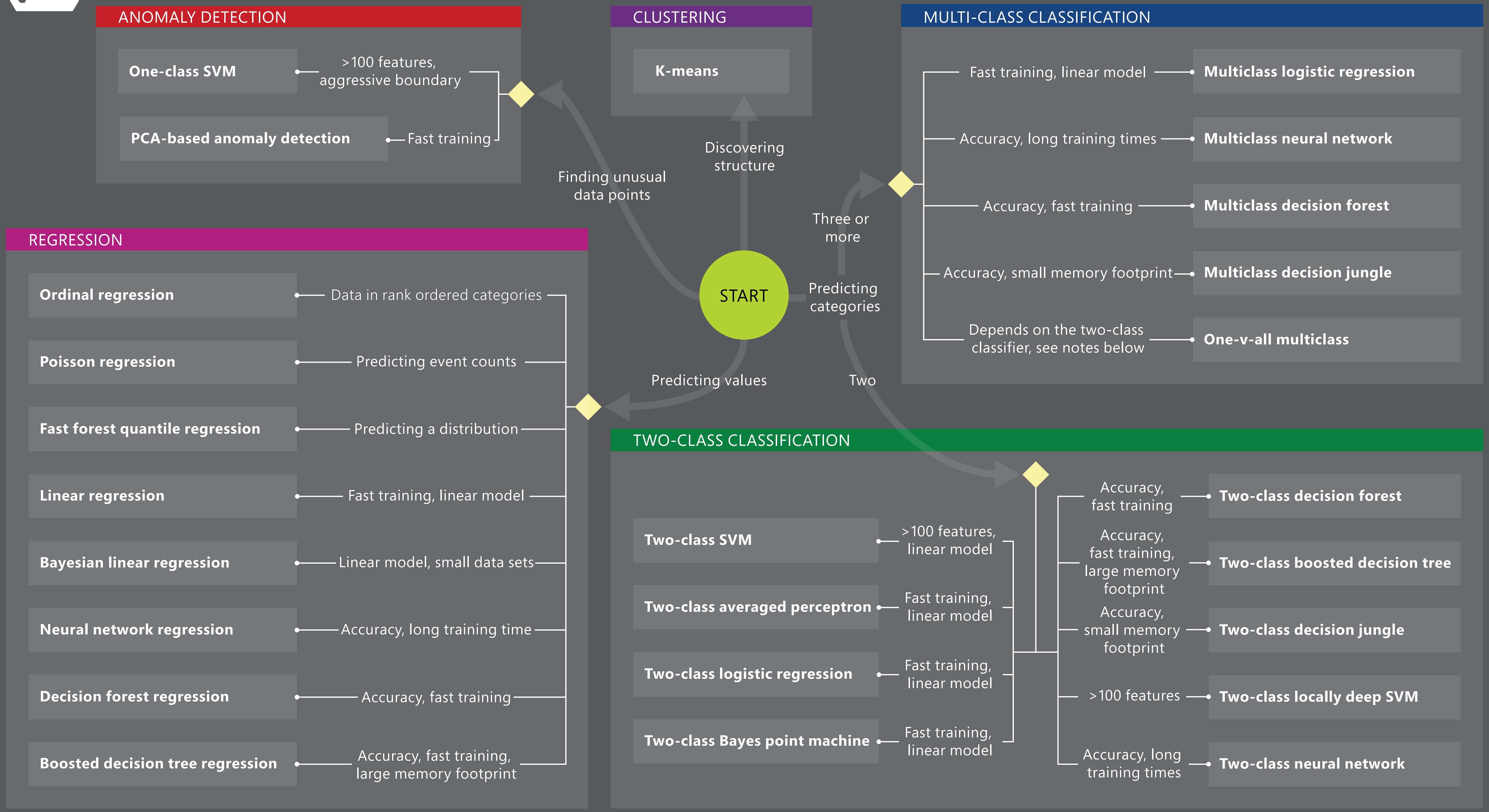
Microsoft Azure Machine Learning: Algorithm Cheat Sheet

This cheat sheet helps you choose the best Azure Machine Learning Studio algorithm for your predictive analytics solution. Your decision is driven by both the nature of your data and the question you're trying to answer.



START

- Finding unusual data points → Anomaly detection
- Discovering structure → Clustering
- Predicting categories: Three or more → Multi-class classification
- Predicting categories: Two → Two-class classification
- Predicting values → Regression

ANOMALY DETECTION

- One-class SVM — >100 features, aggressive boundary
- PCA-based anomaly detection — Fast training

CLUSTERING

- K-means

MULTI-CLASS CLASSIFICATION

- Fast training, linear model — Multiclass logistic regression
- Accuracy, long training times — Multiclass neural network
- Accuracy, fast training — Multiclass decision forest
- Accuracy, small memory footprint — Multiclass decision jungle
- Depends on the two-class classifier, see notes below — One-v-all multiclass

REGRESSION

- Ordinal regression — Data in rank ordered categories
- Poisson regression — Predicting event counts
- Fast forest quantile regression — Predicting a distribution
- Linear regression — Fast training, linear model
- Bayesian linear regression — Linear model, small data sets
- Neural network regression — Accuracy, long training time
- Decision forest regression — Accuracy, fast training
- Boosted decision tree regression — Accuracy, fast training, large memory footprint

TWO-CLASS CLASSIFICATION

- Two-class SVM — >100 features, linear model
- Two-class averaged perceptron — Fast training, linear model
- Two-class logistic regression — Fast training, linear model
- Two-class Bayes point machine — Fast training, linear model
- Accuracy, fast training — Two-class decision forest
- Accuracy, fast training, large memory footprint — Two-class boosted decision tree
- Accuracy, small memory footprint — Two-class decision jungle
- >100 features — Two-class locally deep SVM
- Accuracy, long training times — Two-class neural network

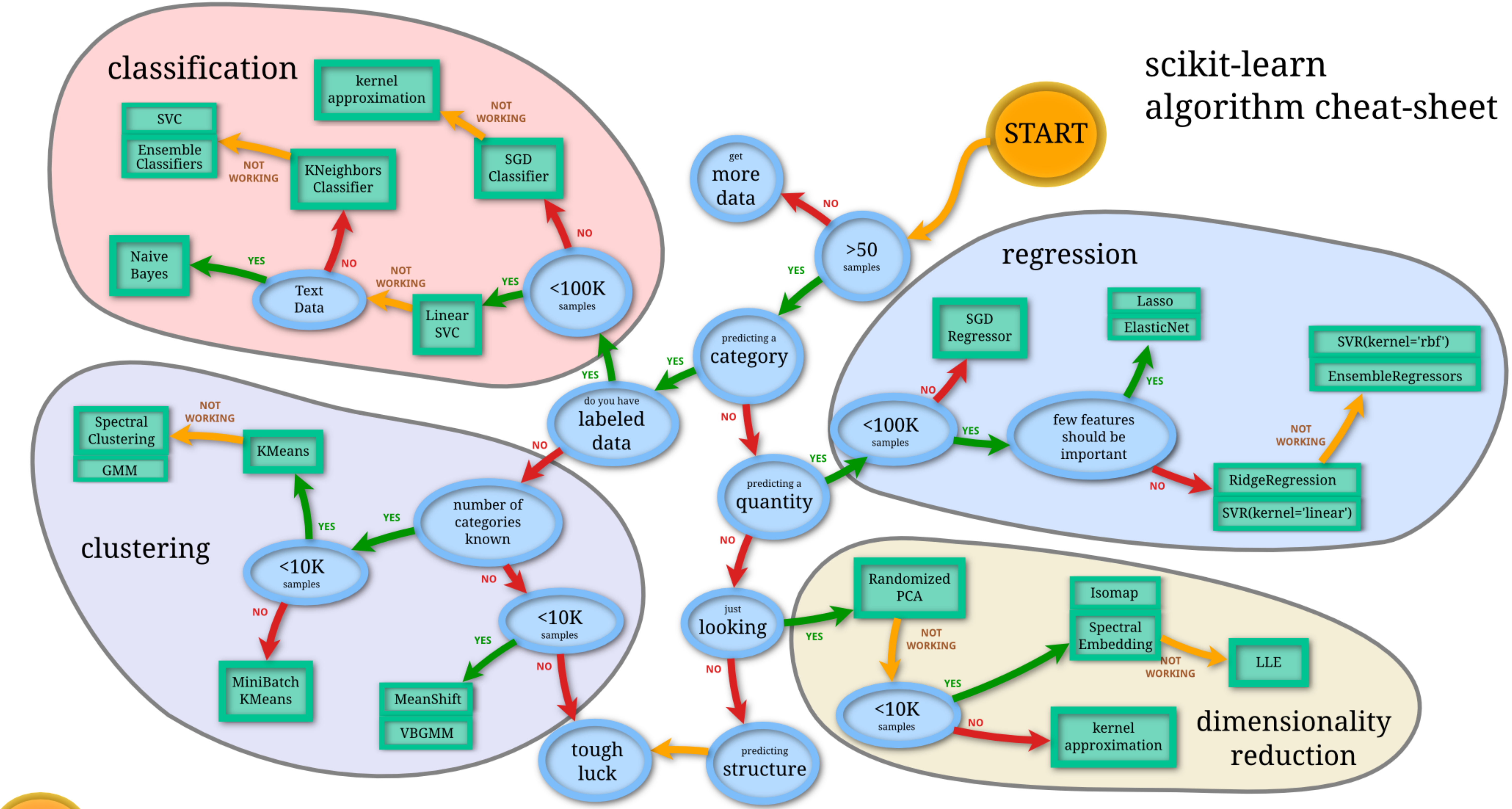
Created by the Azure Machine Learning Team

Email: AzurePoster@microsoft.com

Download this poster: http://aka.ms/MLCheatSheet

Microsoft

scikit-learn algorithm cheat-sheet



Back

scikit learn

Cheat Sheet: Algorithms for Supervised- and Unsupervised Learning [1]

Algorithm	Description	Model	Objective	Training	Regularisation	Complexity	Non-linear	Online learning
k-nearest neighbour	The label of a new point $\hat{x}$ is classified with the most frequent label $\hat{t}$ of the k nearest training instances.	$\hat{t} = \arg\max_{\mathcal{C}} \sum_{i:x_i \in N_k(\boldsymbol{x},\hat{x})} \delta(t_i,\mathcal{C})$ - $N_k(\boldsymbol{x},\hat{x}) \leftarrow k$ points in $\boldsymbol{x}$ closest to $\hat{x}$ - Euclidean distance formula: $\sqrt{\sum_{i=1}^{D}(x_i-\hat{x}_i)^2}$ - $\delta(a,b) \leftarrow 1$ if $a=b$; 0 o/w	No optimisation needed.	Use cross-validation to learn the appropriate k; otherwise no training, classification based on existing points.	k acts as to regularise the classifier: as $k \to N$ the boundary becomes smoother.	$\mathcal{O}(NM)$ space complexity, since all training instances and all their features need to be kept in memory.	Natively finds non-linear boundaries.	To be added.
Naive Bayes	Learn $p(\mathcal{C}_k \mid x)$ by modelling $p(x \mid \mathcal{C}_k)$ and $p(\mathcal{C}_k)$, using Bayes' rule to infer the class conditional probability. Assumes each feature independent of all others, ergo 'Naive.'	$y(\boldsymbol{x}) = \arg\max_k \; p(\mathcal{C}_k \mid x)$ $= \arg\max_k \; p(x \mid \mathcal{C}_k) \times p(\mathcal{C}_k)$ $= \arg\max_k \prod_{i=1}^{D} p(x_i \mid \mathcal{C}_k) \times p(\mathcal{C}_k)$ $= \arg\max_k \sum_{i=1}^{D} \log p(x_i \mid \mathcal{C}_k) + \log p(\mathcal{C}_k)$	No optimisation needed.	**Multivariate likelihood** $p(x \mid \mathcal{C}_k) = \sum_{i=1}^{D} \log p(x_i \mid \mathcal{C}_k)$ $p_{\text{MLE}}(x_i = v \mid \mathcal{C}_k) = \frac{\sum_{j=1}^{N} \delta(t_j = \mathcal{C}_k \wedge x_{ji} = v)}{\sum_{j=1}^{N} \delta(t_j = \mathcal{C}_k)}$ **Multinomial likelihood** $p(x \mid \mathcal{C}_k) = \prod_{i=1}^{D} p(\text{word}_i \mid \mathcal{C}_k)^{x_i}$ $p_{\text{MLE}}(\text{word}_i = v \mid \mathcal{C}_k) = \frac{\sum_{j=1}^{N} \delta(t_j = \mathcal{C}_k) \times x_{ji}}{\sum_{j=1}^{N} \sum_{d=1}^{D} \delta(t_j = \mathcal{C}_k) \times x_{di}}$ . . . where: - x_{ji} is the count of word i in test example j; - x_{di} is the count of feature d in test example j. **Gaussian likelihood** $p(x \mid \mathcal{C}_k) = \prod_{i=1}^{D} \mathcal{N}(v; \mu_{ik}, \sigma_{ik})$	Use a Dirichlet prior on the parameters to obtain a MAP estimate. **Multivariate likelihood** $p_{\text{MAP}}(x_i = v \mid \mathcal{C}_k) = \frac{(\beta_i - 1) + \sum_{j=1}^{N} \delta(t_j = \mathcal{C}_k \wedge x_{ji} = v)}{\lvert x_i \rvert(\beta_i - 1) + \sum_{j=1}^{N} \delta(t_j = \mathcal{C}_k)}$ **Multinomial likelihood** $p_{\text{MAP}}(\text{word}_i = v \mid \mathcal{C}_k) = \frac{(\alpha_i - 1) + \sum_{j=1}^{N} \delta(t_j = \mathcal{C}_k) \times x_{ji}}{\sum_{j=1}^{N} \sum_{d=1}^{D} (\delta(t_j = \mathcal{C}_k) \times x_{di}) - D + \sum_{d=1}^{D} \alpha_d}$	$\mathcal{O}(NM)$, each training instance must be visited and each of its features counted.	Can only learn linear boundaries for multivariate/multinomial attributes. With Gaussian attributes, quadratic boundaries can be learned with uni-modal distributions.	To be added.
Log-linear	Estimate $p(\mathcal{C}_k \mid x)$ directly, by assuming a maximum entropy distribution and optimising an objective function over the conditional entropy distribution.	$y(x) = \arg\max_k \; p(\mathcal{C}_k \mid x)$ $= \arg\max_k \sum_m \lambda_m \phi_m(x, \mathcal{C}_k)$ . . . where: $p(\mathcal{C}_k \mid x) = \frac{1}{Z_\lambda(x)} e^{\sum_m \lambda_m \phi_m(x,\mathcal{C}_k)}$ $Z_\lambda(x) = \sum_k e^{\sum_m \lambda_m \phi_m(x,\mathcal{C}_k)}$	Minimise the negative log-likelihood: $\mathcal{L}_{\text{MLE}}(\lambda, \mathcal{D}) = \prod_{(x,t)\in\mathcal{D}} p(t \mid x) = -\sum_{(x,t)\in\mathcal{D}} \log p(t \mid x)$ $= \sum_{(x,t)\in\mathcal{D}} \left(\log Z_\lambda(x) - \sum_m \lambda_m \phi_m(x,t) \right)$ $= \sum_{(x,t)\in\mathcal{D}} \left(\log \sum_k e^{\sum_m \lambda_m \phi_m(x,\mathcal{C}_k)} - \sum_m \lambda_m \phi_m(x,t) \right)$	Gradient descent (or gradient ascent if maximising objective): $\lambda^{n+1} = \lambda^n - \eta \Delta \mathcal{L}$ . . . where η is the step parameter. $\Delta\mathcal{L}_{\text{MLE}}(\lambda, \mathcal{D}) = \sum_{(x,t)\in\mathcal{D}} \mathbb{E}[\phi(x,\cdot)] - \phi(x,t)$ $\Delta\mathcal{L}_{\text{MAP}}(\lambda, \mathcal{D}, \sigma) = \frac{\lambda}{\sigma^2} + \sum_{(x,t)\in\mathcal{D}} \mathbb{E}[\phi(x,\cdot)] - \sum_{(x,t)\in\mathcal{D}} \phi(x,t)$ . . . where $\sum_{(x,t)\in\mathcal{D}} \phi(x,t)$ are the empirical counts. For each class $\mathcal{C}_k$: $\sum_{(x,t)\in\mathcal{D}} \mathbb{E}[\phi(x,\cdot)] = \sum_{(x,t)\in\mathcal{D}} \phi(x,\cdot) p(\mathcal{C}_k \mid x)$	Penalise large values for the λ parameters, by introducing a prior distribution over them (typically a Gaussian). **Objective function** $\mathcal{L}_{\text{MAP}}(\lambda, \mathcal{D}, \sigma) = \arg\min_\lambda \left(-\log p(\lambda) - \sum_{(x,t)\in\mathcal{D}} \log p(t \mid x) \right)$ $= \arg\min_\lambda \left(-\log e^{\frac{(0-\lambda)^2}{2\sigma^2}} - \sum_{(x,t)\in\mathcal{D}} \log p(t \mid x) \right)$ $= \arg\min_\lambda \left(\frac{\sum_m \lambda_m^2}{2\sigma^2} - \sum_{(x,t)\in\mathcal{D}} \log p(t \mid x) \right)$	$\mathcal{O}(INMK)$, since each training instance must be visited and each combination of class and features must be calculated for the appropriate feature mapping.	Reformulate the class conditional distribution in terms of a kernel $K(x,x')$, and use a non-linear kernel (for example $K(x,x') = (1 + \boldsymbol{w}^T x)^2$). By the Representer Theorem: $p(\mathcal{C}_k \mid x) = \frac{1}{Z_\lambda(x)} e^{\lambda^T \phi(x,\mathcal{C}_k)}$ $= \frac{1}{Z_\lambda(x)} e^{\sum_{n=1}^{N} \sum_{i=1}^{K} \alpha_{nk} \phi(x_n,C_i)^T \phi(x,\mathcal{C}_k)}$ $= \frac{1}{Z_\lambda(x)} e^{\sum_{n=1}^{N} \sum_{i=1}^{K} \alpha_{nk} K((x_n,C_i),(x,C_k))}$ $= \frac{1}{Z_\lambda(x)} e^{\sum_{n=1}^{N} \alpha_{nk} K(x_n,x)}$	Online Gradient Descent: Update the parameters using GD after seeing each training instance.
Perceptron	Directly estimate the linear function $y(x)$ by iteratively updating the weight vector when incorrectly classifying a training instance.	Binary, linear classifier: $y(x) = \text{sign}(\boldsymbol{w}^T x)$ . . . where: $\text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$ Multiclass perceptron: $y(x) = \arg\max_{\mathcal{C}_k} \; \boldsymbol{w}^T \phi(x, \mathcal{C}_k)$	Tries to minimise the Error function; the number of incorrectly classified input vectors: $\arg\min_{\boldsymbol{w}} \; E_P(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}} \; -\sum_{n\in\mathcal{M}} \boldsymbol{w}^T x_n t_n$ . . . where $\mathcal{M}$ is the set of misclassified training vectors.	Iterate over each training example x_n, and update the weight vector if misclassification: $\boldsymbol{w}^{i+1} = \boldsymbol{w}^i + \eta \Delta E_P(\boldsymbol{w})$ $= \boldsymbol{w}^i + \eta x_n t_n$ . . . where typically $\eta = 1$. For the multiclass perceptron: $\boldsymbol{w}^{i+1} = \boldsymbol{w}^i + \phi(x,t) - \phi(x, y(x))$	The Voted Perceptron: run the perceptron i times and store each iteration's weight vector. Then: $y(x) = \text{sign}\left(\sum_i c_i \times \text{sign}(\boldsymbol{w}_i^T x) \right)$ . . . where c_i is the number of correctly classified training instances for $\boldsymbol{w}_i$.	$\mathcal{O}(INML)$, since each combination of instance, class and features must be calculated (see log-linear).	Use a kernel $K(x,x')$, and 1 weight per training instance: $y(x) = \text{sign}\left(\sum_{n=1}^{N} w_n t_n K(x, x_n) \right)$ . . . and the update: $w_n^{i+1} = w_n^i + 1$	The perceptron is an online algorithm per default.
Support vector machines	A maximum margin classifier: finds the separating hyperplane with the maximum margin to its closest data points.	$y(x) = \sum_{n=1}^{N} \lambda_n t_n x^T x_n + w_0$	**Primal** $\arg\min_{\boldsymbol{w},w_0} \; \frac{1}{2} \lVert \boldsymbol{w} \rVert^2$ s.t. $t_n(\boldsymbol{w}^T x_n + w_0) \geq 1 \quad \forall n$ **Dual** $\tilde{\mathcal{L}}(\wedge) = \sum_{n=1}^{N} \lambda_n - \sum_{n=1}^{N} \sum_{m=1}^{N} \lambda_n \lambda_m t_n t_m x_n^T x_m$ s.t. $\lambda_n \geq 0, \quad \sum_{n=1}^{N} \lambda_n t_n = 0, \quad \forall n$	- Quadratic Programming (QP) - SMO, Sequential Minimal Optimisation (chunking).	The soft margin SVM: penalise a hyperplane by the number and distance of misclassified points. **Primal** $\arg\min_{\boldsymbol{w},w_0} \; \frac{1}{2} \lVert \boldsymbol{w} \rVert^2 + C \sum_{n=1}^{N} \xi_n$ s.t. $t_n(\boldsymbol{w}^T x_n + w_0) \geq 1 - \xi_n, \quad \xi_n > 0 \quad \forall n$ **Dual** $\tilde{\mathcal{L}}(\wedge) = \sum_{n=1}^{N} \lambda_n - \sum_{n=1}^{N} \sum_{m=1}^{N} \lambda_n \lambda_m t_n t_m x_n^T x_m$ s.t. $0 \leq \lambda_n \leq C, \quad \sum_{n=1}^{N} \lambda_n t_n = 0, \quad \forall n$	- QP: $\mathcal{O}(n^3)$; - SMO: much more efficient than QP, since computation based only on support vectors.	Use a non-linear kernel $K(x,x')$: $y(x) = \sum_{n=1}^{N} \lambda_n t_n x^T x_n + w_0$ $= \sum_{n=1}^{N} \lambda_n t_n K(x, x_n) + w_0$ $\tilde{\mathcal{L}}(\wedge) = \sum_{n=1}^{N} \lambda_n - \sum_{n=1}^{N} \sum_{m=1}^{N} \lambda_n \lambda_m t_n t_m x_n^T x_m$ $= \sum_{n=1}^{N} \lambda_n - \sum_{n=1}^{N} \sum_{m=1}^{N} \lambda_n \lambda_m t_n t_m K(x_n, x_m)$	Online SVM. See, for example: - *The Huller: A Simple and Efficient Online SVM*, Bordes & Bottou (2005) - *Pegasos: Primal Estimated sub-Gradient Solver for SVM*, Shalev-Shwartz et al. (2007)
k-means	A hard-margin, geometric clustering algorithm, where each data point is assigned to its closest centroid.	Hard assignments $r_{nk} \in \{0,1\}$ s.t. $\forall n \sum_k r_{nk} = 1$, i.e. each data point is assigned to one cluster k. Geometric distance: The Euclidean distance, l^2 norm: $\lVert x_n - \mu_k \rVert_2 = \sqrt{\sum_{i=1}^{D} (x_{ni} - \mu_{ki})^2}$	$\arg\min_{\boldsymbol{r},\mu} \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \lVert x_n - \mu_k \rVert_2^2$ . . . i.e. minimise the distance from each cluster centre to each of its points.	**E**xpectation: $r_{nk} = \begin{cases} 1 & \text{if } \lVert x_n - \mu_k \rVert^2 \text{ minimal for } k \\ 0 & \text{o/w} \end{cases}$ **M**aximisation: $\mu_{\text{MLE}}^{(k)} = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}}$ . . . where $\mu^{(k)}$ is the centroid of cluster k.	Only hard-margin assignment to clusters.	To be added.	For non-linearly separable data, use kernel k-means as suggested in: *Kernel k-means, Spectral Clustering and Normalized Cuts*, Dhillon et al. (2004).	Sequential k-means: update the centroids after processing one point at a time.
Mixture of Gaussians	A probabilistic clustering algorithm, where clusters are modelled as latent Guassians and each data point is assigned the probability of being drawn from a particular Gaussian.	Assignments to clusters by specifying probabilities $p(x^{(i)}, z^{(i)}) = p(x^{(i)} \mid z^{(i)}) p(z^{(i)})$ . . . with $z^{(i)} \sim$ Multinomial(γ), and $\gamma_{nk} \equiv p(k \mid x_n)$ s.t. $\sum_{j=1}^{k} \gamma_{nj} = 1$. I.e. want to maximise the probability of the observed data $\boldsymbol{x}$.	$\mathcal{L}(\boldsymbol{x}, \pi, \mu, \Sigma) = \log p(\boldsymbol{x} \mid \pi, \mu, \Sigma)$ $= \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \pi_k \mathcal{N}_k(x_n \mid \mu_k, \Sigma_k) \right)$	**E**xpectation: For each n, k set: $\gamma_{nk} = p(z^{(i)} = k \mid x^{(i)}; \gamma, \mu, \Sigma) \quad (= p(k \mid x_n))$ $= \frac{p(x^{(i)} \mid z^{(i)} = k; \mu, \Sigma) p(z^{(i)} = k; \pi)}{\sum_{j=1}^{K} p(x^{(i)} \mid z^{(i)} = l; \mu, \Sigma) p(z^{(i)} = l; \pi)}$ $= \frac{\pi_k \mathcal{N}(x_n \mid \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_n \mid \mu_j, \Sigma_j)}$ **M**aximisation: $\pi_k = \frac{1}{N} \sum_{n=1}^{N} \gamma_{nk}$ $\Sigma_k = \frac{\sum_{n=1}^{N} \gamma_{nk} (x_n - \mu_k)(x_n - \mu_k)^T}{\sum_{n=1}^{N} \gamma_{nk}}$ $\mu_k = \frac{\sum_{n=1}^{N} \gamma_{nk} x_n}{\sum_{n=1}^{N} \gamma_{nk}}$	The mixture of Gaussians assigns probabilities for each cluster to each data point, and as such is capable of capturing ambiguities in the data set.	To be added.	Not applicable.	Online Gaussian Mixture Models. A good start is: *A View of the EM Algorithm that Justifies Incremental, Sparse, and Other Variants*, Neal & Hinton (1998).

[1] Created by Emanuel Ferm, HT2011, for semi-procrastinational reasons while studying for a Machine Learning exam. Last updated May 5, 2011.