KNN, Kernels, Margins

Grzegorz Chrupała and Nicolas Stroppa

Google
Saarland University

META

Outline

1. K-Nearest neighbors classifier
2. Kernels
3. SVM

Outline

1. K-Nearest neighbors classifier
2. Kernels
3. SVM

KNN classifier

K-Nearest neighbors idea

When classifying a new example, find k nearest training example, and assign the majority label

- Also known as
  - Memory-based learning
  - Instance or exemplar based learning
  - Similarity-based methods
  - Case-based reasoning

Distance metrics in feature space

- Euclidean distance or L_2 norm in d dimensional space:

$$D(\mathbf{x}, \mathbf{x}') = \sqrt{\sum_{i=1}^{d} (x_i - x_i')^2}$$

- L_1 norm (Manhattan or taxicab distance)

$$L_1(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{d} |x_i - x_i'|$$

- L_∞ or maximum norm

$$L_\infty(\mathbf{x}, \mathbf{x}') = \max_{i=1}^{d} |x_i - x_i'|$$

- In general, L_k norm:

$$L_k(\mathbf{x}, \mathbf{x}') = \left(\sum_{i=1}^{d} |x_i - x_i'|^k \right)^{1/k}$$

Hamming distance

- Hamming distance used to compare strings of symbolic attributes
- Equivalent to L_1 norm for binary strings
- Defines the distance between two instances to be the sum of per-feature distances
- For symbolic features the per-feature distance is 0 for an exact match and 1 for a mismatch.

$$\text{Hamming}(x, x') = \sum_{i=1}^{d} \delta(x_i, x'_i) \tag{1}$$

$$\delta(x_i, x'_i) = \begin{cases} 0 & \text{if } x_i = x'_i \\ 1 & \text{if } x_i \neq x'_i \end{cases} \tag{2}$$

IB1 algorithm

For a vector with a mixture of symbolic and numeric values, the above definition of per feature distance is used for symbolic features, while for numeric ones we can use the scaled absolute difference

$$\delta(x_i, x'_i) = \frac{x_i - x'_i}{max_i - min_i}. \quad (3)$$

IB1 with feature weighting

- The per-feature distance is multiplied by the weight of the feature for which it is computed:

$$D_{\mathbf{w}}(x, x') = \sum_{i=1}^{d} w_i \delta(x_i, x_i') \tag{4}$$

  where w_i is the weight of the ith feature.

- We'll describe two entropy-based methods and a χ^2-based method to find a good weight vector $\mathbf{w}$.

Information gain

A measure of how much knowing the value of a certain feature for an example decreases our uncertainty about its class, i.e. difference in class entropy with and without information about the feature value.

$$w_i = H(Y) - \sum_{v \in V_i} P(v) H(Y|v) \tag{5}$$

where

- w_i is the weight of the ith feature
- Y is the set of class labels
- V_i is the set of possible values for the ith feature
- $P(v)$ is the probability of value v
- class entropy $H(Y) = -\sum_{y \in Y} P(y) \log_2 P(y)$
- $H(Y|v)$ is the conditional class entropy given that feature value $= v$

Numeric values need to be temporarily discretized for this to work

Gain ratio

IG assigns excessive weight to features with a large number of values.
To remedy this bias information gain can be normalized by the entropy of the feature values, which gives the gain ratio:

$$w_i = \frac{H(Y) - \sum_{v \in V_i} P(v) H(Y|v)}{H(V_i)} \tag{6}$$

For a feature with a unique value for each instance in the training set, the entropy of the feature values in the denominator will be maximally high, and will thus give it a low weight.

χ^2

The χ^2 statistic for a problem with k classes and m values for feature F:

$$\chi^2 = \sum_{i=1}^{k} \sum_{j=1}^{m} \frac{(E_{ij} - O_{ij})^2}{E_{ij}} \tag{7}$$

where

- O_{ij} is the observed number of instances with the ith class label and the jth value of feature F
- E_{ij} is the expected number of such instances in case the null hypothesis is true: $E_{ij} = \frac{n_{\cdot j} n_{i \cdot}}{n_{\cdot \cdot}}$
- n_{ij} is the frequency count of instances with the ith class label and the jth value of feature F
  - $n_{\cdot j} = \sum_{i=1}^{k} n_{ij}$
  - $n_{i \cdot} = \sum_{j=1}^{m} n_{ij}$
  - $n_{\cdot \cdot} = \sum_{i=1}^{k} \sum_{j=0}^{m} n_{ij}$

χ^2 example

- Consider a spam detection task: your features are words present/absent in email messages
- Compute χ^2 for each word to use a weightings for a KNN classifier
- The statistic can be computed from a contingency table. Eg. those are (fake) counts of **rock-hard** in 2000 messages

	rock-hard	¬ rock-hard
ham	4	996
spam	100	900

We need to sum $(E_{ij} - O_{ij})^2/E_{ij}$ for the four cells in the table:

$$\frac{(52-4)^2}{52} + \frac{(948-996)^2}{948} + \frac{(52-100)^2}{52} + \frac{(948-900)^2}{948} = 93.4761$$

Distance-weighted class voting

- So far all the instances in the neighborhood are weighted equally for computing the majority class
- We may want to treat the votes from very close neighbors as more important than votes from more distant ones
- A variety of distance weighting schemes have been proposed to implement this idea

KNN – summary

- Non-parametric: makes no assumptions about the probability distribution the examples come from
- Does not assume data is linearly separable
- Derives decision rule directly from training data
- "Lazy learning":
  - During learning little "work" is done by the algorithm: the training instances are simply stored in memory in some efficient manner.
  - During prediction the test instance is compared to the training instances, the neighborhood is calculated, and the majority label assigned
- No information discarded: "exceptional" and low frequency training instances are available for prediction

Outline

1. K-Nearest neighbors classifier
2. Kernels
3. SVM

Perceptron – dual formulation

- The weight vector ends up being a linear combination of training examples:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y \mathbf{x}^{(i)}$$

where $\alpha_i = 1$ if the ith was misclassified, and $= 0$ otherwise.

- The discriminant function then becomes:

$$g(x) = \left(\sum_{i=1}^{n} \alpha_i y^{(i)} \mathbf{x}^{(i)} \right) \cdot \mathbf{x} + b \quad (8)$$

$$= \sum_{i=1}^{n} \alpha_i y^{(i)} \mathbf{x}^{(i)} \cdot \mathbf{x} + b \quad (9)$$

Dual Perceptron training

$\textsc{DualPerceptron}(x^{1:N}, y^{1:N}, I)$:

1: $\boldsymbol{\alpha} \leftarrow \mathbf{0}$
2: $b \leftarrow 0$
3: **for** $j = 1...I$ **do**
4: **for** $k = 1...N$ **do**
5: **if** $y^{(k)} \left[\sum_{i=1}^{N} \alpha_i y^{(i)} \mathbf{x}^{(i)} \cdot \mathbf{x}^{(k)} + b \right] \leq 0$ **then**
6: $\alpha_i \leftarrow \alpha_i + 1$
7: $b \leftarrow b + y^{(k)}$
8: **return** $(\boldsymbol{\alpha}, b)$

Kernels

- Note that in the dual formulation there is no explicit weight vector: the training algorithm and the classification are expressed in terms of dot products between training examples and the test example
- We can generalize such dual algorithms to use Kernel functions
  - A kernel function can be thought of as dot product in some transformed feature space

$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{z})$$

    where the map ϕ projects the vectors in the original feature space onto the transformed feature space
  - It can also be thought of as a similarity function in the input object space

Kernel – example

- Consider the following kernel function $K : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$

$$
\begin{aligned}
K(\mathbf{x}, \mathbf{z}) &= (\mathbf{x} \cdot \mathbf{z})^2 && (10)\\
&= \left(\sum_{i=1}^{d} x_i z_i\right) \left(\sum_{i=1}^{d} x_i z_i\right) && (11)\\
&= \sum_{i=1}^{d} \sum_{j=1}^{d} x_i x_j z_i z_j && (12)\\
&= \sum_{i,j=1}^{d} (x_i x_j)(z_i z_j) && (13)
\end{aligned}
$$

Kernel vs feature map

Feature map ϕ corresponding to K for $d = 2$ dimensions

$$\phi \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1 x_1 \\ x_1 x_2 \\ x_2 x_1 \\ x_2 x_2 \end{pmatrix} \tag{14}$$

- Computing feature map ϕ explicitly needs $\mathcal{O}(d^2)$ time
- Computing K is linear $\mathcal{O}(d)$ in the number of dimensions

Why does it matter

- If you think of features as binary indicators, then the quadratic kernel above creates feature conjunctions
- E.g. in NER if x_1 indicates that word is capitalized and x_2 indicates that the previous token is a sentence boundary, with the quadratic kernel we efficiently compute the feature that both conditions are the case.
- Geometric intuition: mapping points to higher dimensional space makes them easier to separate with a linear boundary

Separability in 2D and 3D

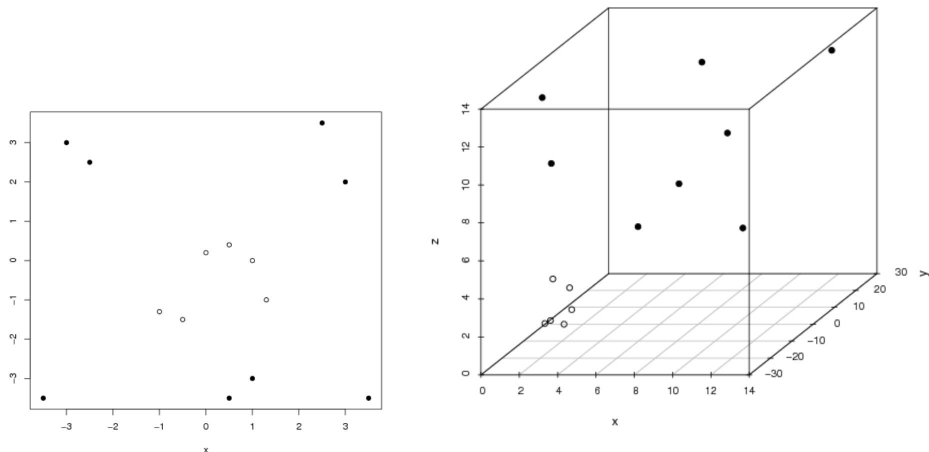


Figure: Two dimensional classification example, non-separable in two dimensions, becomes separable when mapped to 3 dimensions by $(x_1, x_2) \mapsto (x_1^2, 2x_1x_2, x_2^2)$

Outline

1. K-Nearest neighbors classifier
2. Kernels
3. SVM

Support Vector Machines

- Margin: a decision boundary which is as far away from the training instances as possible improves the chance that if the position of the data points is slightly perturbed, the decision boundary will still be correct.
- Results from Statistical Learning Theory confirm these intuitions: maintaining large margins leads to small generalization error (Vapnik 1995)
- A perceptron algorithm finds any hyperplane which separates the classes: SVM finds the one that additionally has the maximum margin

Quadratic optimization formulation

- Functional margin can be made larger just by rescaling the weights by a constant
- Hence we can fix the functional margin to be 1 and minimize the norm of the weight vector
- This is equivalent to maximizing the geometric margin

For linearly separable training instances $((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n))$ find the hyperplane $(\mathbf{w}, b)$ that solves the optimization problem:

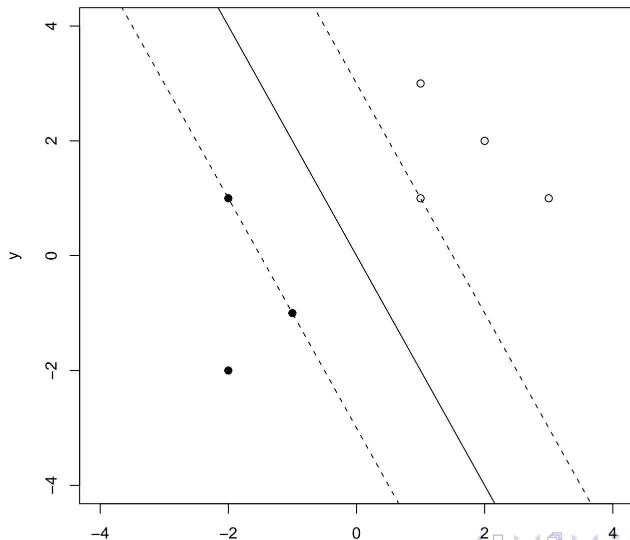
$$\begin{aligned} \text{minimize}_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 \\ \text{subject to} \quad & y_i(w \cdot \mathbf{x}_i + b) \geq 1 \quad \forall_{i \in 1..n} \end{aligned} \tag{15}$$

This hyperplane separates the examples with geometric margin $2/||\mathbf{w}||$

Support vectors

- SVM finds a separating hyperplane with the largest margin to the nearest instance
- This has the effect that the decision boundary is fully determined by a small subset of the training examples (the nearest ones on both sides)
- Those instances are the support vectors

Separating hyperplane and support vectors



Soft margin

- SVM with soft margin works by relaxing the requirement that all data points lie outside the margin
- For each offending instance there is a "slack variable" ξ_i which measures how much it would have move to obey the margin constraint.

$$
\begin{aligned}
\text{minimize}_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i \\
\text{subject to} \quad & y_i(w \cdot \mathbf{x}_i + b) \geq 1 - \xi_i \\
& \forall_{i \in 1..n} \xi_i > 0
\end{aligned}
\tag{16}
$$

where

$$\xi_i = \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b))$$

- The hyper-parameter C trades off minimizing the norm of the weight vector versus classifying correctly as many examples as possible.
- As the value of C tends towards infinity the soft-margin SVM approximates the hard-margin version.

Dual form

The dual formulation is in terms of support vectors, where SV is the set of their indices:

$$f(x, \alpha^*, b^*) = \mathrm{sign}\left(\sum_{i \in SV} y_i \alpha_i^* (\mathbf{x}_i \cdot \mathbf{x}) + b^*\right) \tag{17}$$

The weights in this decision function are the α^*.

$$\begin{aligned} &\text{minimize} \quad W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j (\mathbf{x}_i \cdot \mathbf{x}_j) \\ &\text{subject to} \quad \sum_{i=1}^{n} y_i \alpha_i = 0 \qquad \forall_{i \in 1..n} \alpha_i \geq 0 \end{aligned} \tag{18}$$

The weights together with the support vectors determine $(\mathbf{w}, b)$:

$$\mathbf{w} = \sum_{i \in SV} \alpha_i y_i \mathbf{x}_i \tag{19}$$

$$b = y_k - \mathbf{w} \cdot \mathbf{x}_k \text{ for any } k \text{ such that } \alpha_k \neq 0 \tag{20}$$

Summary

- With the kernel trick we can
  - use linear models with non-linearly separable data
  - use polynomial kernels to create implicit feature conjunctions
- Large margin methods help us choose a linear separator with good generalization properties

Efficient averaged perceptron algorithm

$\textsc{Perceptron}(x^{1:N}, y^{1:N}, I)$:

1: $\mathbf{w} \leftarrow \mathbf{0}$; $\mathbf{w_a} \leftarrow \mathbf{0}$
2: $b \leftarrow 0$; $b_a \leftarrow 0$
3: $c \leftarrow 1$
4: **for** $i = 1...I$ **do**
5:     **for** $n = 1...N$ **do**
6:         **if** $y^{(n)}(\mathbf{w} \cdot \mathbf{x}^{(n)} + b) \leq 0$ **then**
7:             $\mathbf{w} \leftarrow \mathbf{w} + y^{(n)}\mathbf{x}^{(n)}$; $b \leftarrow b + y^{(n)}$
8:             $\mathbf{w_a} \leftarrow \mathbf{w_a} + cy^{(n)}\mathbf{x}^{(n)}$; $b_a \leftarrow b_a + cy^{(n)}$
9:         $c \leftarrow c + 1$
10: **return** $(\mathbf{w} - \mathbf{w_a}/c, b - b_a/c)$

Problem: Average perceptron

Weight averaging

Show that the above algorithm performs weight averaging.
Hints:

- In the standard perceptron algorithm, the final weight vector (and bias) is the sum of the updates at each step.
- In average perceptron, the final weight vector should be the mean of the sum of partial sums of updates at each step

Solution

Let's formalize:

- Basic perceptron: final weights are the sum of updates at each step:

$$\mathbf{w} = \sum_{i=1}^{n} f(\mathbf{x}^{(i)}) \tag{21}$$

Solution

Let's formalize:

- Basic perceptron: final weights are the sum of updates at each step:

$$\mathbf{w} = \sum_{i=1}^{n} f(\mathbf{x}^{(i)}) \tag{21}$$

- Naive weight averaging: final weights are the mean of the sum of partial sums:

$$\mathbf{w} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{i} f(\mathbf{x}^{(j)}) \tag{22}$$

Solution

Let's formalize:

- Basic perceptron: final weights are the sum of updates at each step:

$$\mathbf{w} = \sum_{i=1}^{n} f(\mathbf{x}^{(i)}) \tag{21}$$

- Naive weight averaging: final weights are the mean of the sum of partial sums:

$$\mathbf{w} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{i} f(\mathbf{x}^{(j)}) \tag{22}$$

- Efficient weight averaging:

$$\mathbf{w} = \sum_{i=1}^{n} f(\mathbf{x}^{(i)}) - \sum_{i=1}^{n} i f(\mathbf{x}^{(i)})/n \tag{23}$$

- Show that equations 22 and 23 are equivalent. Note that we can rewrite the sum of partial sums by multiplying the update at each step by the factor indicating in how many of the partial sums it appears

$$\mathbf{w} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{i} f(\mathbf{x}^{(j)}) \tag{24}$$

$$= \frac{1}{n} \sum_{i=1}^{n} (n - i) f(\mathbf{x}^{(i)}) \tag{25}$$

$$= \frac{1}{n} \left[\sum_{i=1}^{n} n f(\mathbf{x}^{(i)}) - i f(\mathbf{x}^{(i)}) \right] \tag{26}$$

$$= \frac{1}{n} \left[n \sum_{i=1}^{n} f(\mathbf{x}^{(i)}) - \sum_{i=1}^{n} i f(\mathbf{x}^{(i)}) \right] \tag{27}$$

$$= \sum_{i=1}^{n} f(\mathbf{x}_i) - \sum_{i=1}^{n} i f(\mathbf{x}^{(i)}) / n \tag{28}$$

Viterbi algorithm for second-order HMM

In the lecture we saw the formulation of the Viterbi algorithm for a first-order Hidden Markov Model. In a second-order HMM transition probabilities depend on the two previous states, instead on just the single previous state, that is we use the following independence assumption:

$$P(y_i|x_1, \ldots, x_{i-1}, y_1, \ldots, y_{i-1}) = P(y_i|y_{i-2}, y_{i-1}).$$

For this exercise you should formulate the Viterbi algorithm for a decoding a second-order HMM.