Fundamentals of Machine Learning:

Kernel Machines II: The Kernel Trick and Nonlinear SVMs

Prof. Andrew D. Bagdanov (andrew.bagdanov AT unifi.it)

UNIVERSITÀ
DEGLI STUDI
FIRENZE

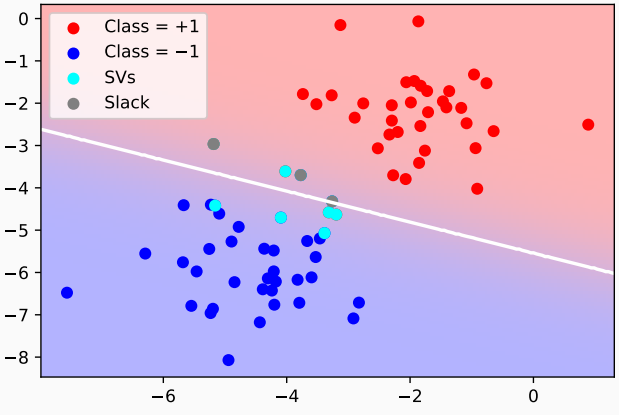
DINFO
DIPARTIMENTO DI
INGEGNERIA
DELL'INFORMAZIONE

Outline

Introduction

Motivations

- We now have a way to fit a linear models for classification problems:



Considerations

- Can we extend this model to capture nonlinear decision boundaries in some way?
- Maybe like we did with the quadratic generative classifier? Or with least squares?
- Is there anyway to do this without complicating the elegant form of the learning objectives?
- It is not immediately evident how we should do this...

Lecture objectives

After this lecture you will:

- Understand how learning the parameters of the linear SVM can be viewed as a loss minimization problem.
- Understand how margin maximization is equivalent to regularization of the loss minimization formulation.
- Recognize how inner products can be replaced with kernel evaluations in the dual SVM formulation.
- Understand how nonlinear SVMs can be fit using an appropriate kernel.
- Be able to apply linear and nonlinear SVMs in practice to solve classification problems.

Another View of the Primal SVM

SVM via Empirical Risk Minimization

- We can derive the Support Vector Machine by devising an empirical risk instead of analyzing the margin.
- We return to our hypothesis class $\mathcal{H}$ of linear discriminant functions:

$$f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b$$

- But what should our loss function be for $\mathcal{H}$ and a dataset $\mathcal{D} = \{ (\mathbf{x_n}, y_n) \mid n = 1, \dots N \}$?
- Recall that the ideal loss would be the zero-one loss:

$$\mathcal{L}(\mathbf{w}, b; \mathcal{D}) = \sum_{n=1}^{N} \mathbf{1}(f(\mathbf{x}_n) \neq y_n)$$

.

- Unfortunately, also recall that this results in a combinatorial optimization problem that is *NP*-hard...

SVM via Empirical Risk Minimization

- So, what should the loss be that results in the same SVM?
- Considering the errors made by an SVM: they grow linearly with how far they are from the correct side of the margin.
- What we need is known as the hinge loss:

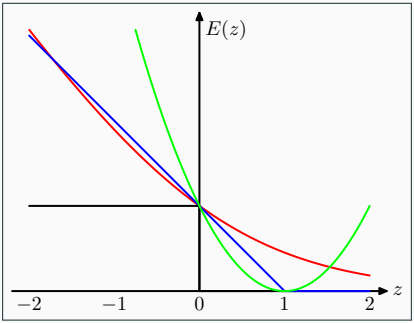
$$\begin{aligned}\mathcal{L}(\mathbf{w}, b; \mathcal{D}) &= \sum_{n=1}^{N} \max\{0, 1 - y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b)\} \\ &= \sum_{n=1}^{N} \ell(\mathbf{x}_n, y_n),\end{aligned}$$

$$\text{where } \ell(\mathbf{x}, y) = \begin{cases} 0 & \text{if } y(\langle \mathbf{w}, \mathbf{x} \rangle + b) \geq 1 \\ 1 - y(\langle \mathbf{w}, \mathbf{x} \rangle + b) & \text{if } y(\langle \mathbf{w}, \mathbf{x} \rangle + b) < 1 \end{cases}$$

SVM via Empirical Risk Minimization

$$\mathcal{L}(\mathbf{w}, b; \mathcal{D}) = \sum_{n=1}^{N} \max\{0, 1 - y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b)\}$$

- This bears consideration:



SVM via Empirical Risk Minimization

- We are just missing one piece of the optimization puzzle: control of model complexity.
- But, we know how to do that from our discussion of linear regression:

$$(\mathbf{w}^*, b^*) = \arg\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{n=1}^{N} \max\{0, 1 - y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b)\}$$

- We have an unconstrained (but still convex) optimization problem expressed in terms of a loss and a regularizer.
- So: *margin maximization can be viewed as regularization.*
- The C hyperparameter is used to weight the loss instead of λ for the regularizer.
- However, we are limited to the primal form of the SVM...

A Closer Look at the Dual Form of the SVM

A Pattern Matching Exercise

- Let's go back to the dual form of the SVM:

$$\max_{\mathbf{a}} \left\{ \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m y_n y_m \langle \mathbf{x}_n, \mathbf{x}_m \rangle \right\}$$

$$\text{subject to} \quad 0 \leq a_n \leq C, \text{ for } n = 1, \ldots, N$$

$$\sum_{n=1}^{N} a_n y_n = 0$$

- Note that the *only* use of the input samples are in the inner product.
- This means that to learn we only need to compute inner products between our input samples $\mathbf{x}_n$.

Explicit embedding

- Hey! We can probably use an explicit embedding $\phi : \mathbb{R}^D \to \mathcal{H}$ of our data into a new space.
- We already did this with a polynomial embedding...
- The new optimization problem:

$$\max_{\mathbf{a}} \left\{ \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m y_n y_m \langle \phi(\mathbf{x}_n), \phi(\mathbf{x}_m) \rangle_{\mathcal{H}} \right\}$$

$$\text{subject to} \quad 0 \leq a_n \leq C, \text{ for } n = 1, \ldots, N$$

$$\sum_{n=1}^{N} a_n y_n = 0$$

- As long as $(\mathcal{H}, \langle \cdot, \cdot \rangle_{\mathcal{H}})$ is an inner product space, we're cool.

Explicit embeddings = Feature maps

- Explicit embeddings like this are usually called feature maps.
- We are mapping a feature representation in one space to another feature representation in a new space.
- Note that there are no limitations on the form of the feature map ϕ.
- The main advantage of explicit feature mapping is that we are free to use nonlinear combinations of the input features.
- The resulting classifier is linear in the “new” feature space, but nonlinear in the original one.

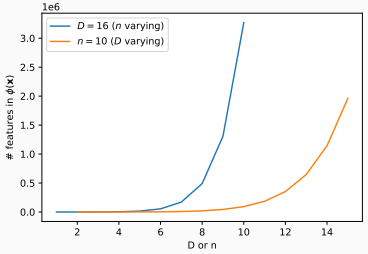
Not all sunshine and lollipops

- Consider the degree-n polynomial feature map from $\mathbb{R}^D \to \mathbb{R}^K$
- Question: what is K? How many monomials of degree $\leq n$ are there in D input variables?

Not all sunshine and lollipops

- Consider the degree-n polynomial feature map from $\mathbb{R}^D \to \mathbb{R}^K$
- Question: what is K? How many monomials of degree $\leq n$ are there in D input variables?
- Answer:

$$\binom{D+n-1}{n} = \frac{1}{(D-1)!}(n+1)^{\overline{D-1}}$$



Are we screwed?

- Well, this seems hopeless...
- If I explicitly embed my data, I (quite literally) explode my space complexity.
- Is there any way we can circumvent this explosion?
- If you think about it, I have already given you a huge spoiler...

The Kernel Trick

We only need the inner product evaluations

- Back to the dual formulation:

$$\max_{\mathbf{a}} \left\{ \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m y_n y_m \langle \phi(\mathbf{x}_n), \phi(\mathbf{x}_m) \rangle_{\mathcal{H}} \right\}$$

$$\text{subject to} \quad 0 \leq a_n \leq C, \text{ for } n = 1, \ldots, N$$

$$\sum_{n=1}^{N} a_n y_n = 0$$

- Observe that we don't need the embeddings $\phi(\mathbf{x}_n)$ and $\phi(\mathbf{x}_m)$.
- All we *actually* need is $\langle \mathbf{x}_n, \mathbf{x}_m \rangle$.
- Let's give it a name (well, two, sort of):

$$\begin{aligned} k(\mathbf{x}, \mathbf{z}) &= \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle_{\mathcal{H}} && \text{(kernel function } k\text{)} \\ K[n, m] &= \langle \phi(\mathbf{x}_n), \phi(\mathbf{x}_m) \rangle_{\mathcal{H}} && \text{(kernel or Gram matrix } K\text{)} \end{aligned}$$

Dual forms of linear models

- Often the dual form of linear models (e.g. the Support Vector Machine, but see also Bishop, chapter 6.1) only rely on linear combinations of the kernel function k.
- The symmetry of the inner product implies that k and K are symmetric.
- The positive definiteness of the inner product implies that K is a positive semidefinite matrix (i.e. $\mathbf{x}^T K \mathbf{x} \geq 0 \, \forall \mathbf{x}$).
- The simplest kernel function is the linear kernel:

$$k(\mathbf{x}, \mathbf{z}) = \mathbf{x}^T \mathbf{z}, \text{ corresponding to } \phi(\mathbf{x}) = \mathbf{x}.$$

- The kernel trick (also called kernel substitution) refers to using kernel functions to substitute inner products in otherwise linear models.

Constructing kernels

- Of course, if the feature map ϕ is tractable, we can just use it to define the corresponding kernel:

$$k(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle \text{ (duh)}$$

- What we really want is to construct kernels directly and avoid embedding.
- Consider this example (assuming $V = \mathbb{R}^2$):

$$\begin{aligned} k(\mathbf{x}, \mathbf{z}) &= (\mathbf{x}^T \mathbf{z})^2 \\ &= (x_1 z_1 + x_2 z_2)^2 \\ &= \dots \\ &= (x_1^2, \sqrt{2} x_1 x_2, x_2^2)(z_1^2, \sqrt{2} z_1 z_2, z_2^2)^T \end{aligned}$$

- So, $k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T \mathbf{z})^2$ corresponds to the polynomial embedding $\phi(\mathbf{x}) = (x_1^2, \sqrt{2} x_1 x_2, x_2^2)$.

Constructing kernels

- We can combine kernels in ways that preserve positive semidefiniteness:

Techniques for Constructing New Kernels.

Given valid kernels $k_1(\mathbf{x}, \mathbf{x}')$ and $k_2(\mathbf{x}, \mathbf{x}')$, the following new kernels will also be valid:

$$
\begin{aligned}
k(\mathbf{x}, \mathbf{x}') &= c k_1(\mathbf{x}, \mathbf{x}') && (6.13)\\
k(\mathbf{x}, \mathbf{x}') &= f(\mathbf{x}) k_1(\mathbf{x}, \mathbf{x}') f(\mathbf{x}') && (6.14)\\
k(\mathbf{x}, \mathbf{x}') &= q\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.15)\\
k(\mathbf{x}, \mathbf{x}') &= \exp\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.16)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}') && (6.17)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') k_2(\mathbf{x}, \mathbf{x}') && (6.18)\\
k(\mathbf{x}, \mathbf{x}') &= k_3\left(\boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}')\right) && (6.19)\\
k(\mathbf{x}, \mathbf{x}') &= \mathbf{x}^{\mathrm{T}} \mathbf{A} \mathbf{x}' && (6.20)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}'_a) + k_b(\mathbf{x}_b, \mathbf{x}'_b) && (6.21)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}'_a) k_b(\mathbf{x}_b, \mathbf{x}'_b) && (6.22)
\end{aligned}
$$

where $c > 0$ is a constant, $f(\cdot)$ is any function, $q(\cdot)$ is a polynomial with nonnegative coefficients, $\boldsymbol{\phi}(\mathbf{x})$ is a function from $\mathbf{x}$ to $\mathbb{R}^M$, $k_3(\cdot, \cdot)$ is a valid kernel in $\mathbb{R}^M$, $\mathbf{A}$ is a symmetric positive semidefinite matrix, $\mathbf{x}_a$ and $\mathbf{x}_b$ are variables (not necessarily disjoint) with $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$, and k_a and k_b are valid kernel functions over their respective spaces.

Constructing kernels

- A very commonly used kernel is the Gaussian:

$$k(\mathbf{x}, \mathbf{z}) = \exp\left\{-||\mathbf{x} - \mathbf{z}||/2\sigma^2\right\}$$

- We can see is a kernel using the table on the previous slide and expanding the square inside the exponential:

$$k(\mathbf{x}, \mathbf{z}) = \exp(-\mathbf{x}^T\mathbf{x}/2\sigma^2)\exp(-\mathbf{x}^T\mathbf{z}/\sigma^2)\exp(-\mathbf{z}^T\mathbf{z}/2\sigma^2)$$

- It is fairly easy to show that this corresponds to an embedding $\phi(\cdot)$ that maps to an *infinite* dimensional feature space.

So what?

- We need only make slight adjustments to our learning problem:

$$\max_{\mathbf{a}} \left\{ \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m y_n y_m k(\mathbf{x}_n, \mathbf{x}_m) \right\}$$

$$\text{subject to} \quad 0 \leq a_n \leq C, \text{ for } n = 1, \ldots, N$$

$$\sum_{n=1}^{N} a_n y_n = 0$$

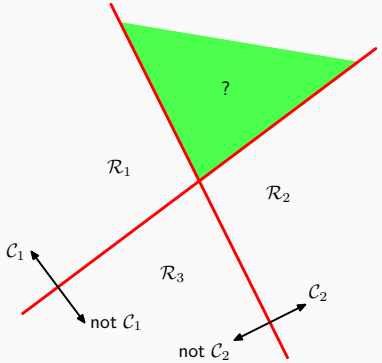
- And to our classifier:

$$\begin{aligned} f(\mathbf{x}) &= \sum_{n=1}^{N} a_n y_n k(\mathbf{x}, \mathbf{x}_n) + b \\ &= \sum_{\mathbf{z} \in \mathrm{SV}} a_n y_n k(\mathbf{x}, \mathbf{z}) + b \end{aligned}$$

Support Vector Machines in Practice

The Multiclass Case

- Hey, wait a minute... We have only developed SVMs for binary classification problems.
- For the general multiclass problem, we usually use a one-versus-rest classifier and just take the max output as out decision rule.



Caveats

- The time and space complexities of SVM solvers can be unpredictable if you don't know what your doing (which we all should now!).
- Time complexity will vary in terms of which solver is used.
- Luckily, there are many rock-solid packages available for training SVMs using various techniques.

LibLinear

- LibLinear is a robust and fast SVM solver with a very long history.
- Its main focus is solving the primal objective formulation.
- It can switch to a dual coordinate descent solver for large-scale problems.
- It can exploit multi-core machines (useful for the multiclass case).

```
class sklearn.svm.LinearSVC(
        penalty='l2', loss='squared_hinge',
        dual=True, tol=0.0001, C=1.0,
        multi_class='ovr', fit_intercept=True,
        intercept_scaling=1,
        class_weight=None, verbose=0,
        random_state=None, max_iter=1000
)
```

LibSVM

- LibSVM (by the same fine folks who brought us LibLinear) is an SVM solver with an even longer history.
- It is a dual solver, so although it provides flexibility via the kernel trick, it can scale poorly in space (or time if K isn't precomputed).
- It supports a HUGE variety of alternate formulations of the SVM objective.

```
class sklearn.svm.SVC(
        C=1.0, kernel='rbf',
        degree=3, gamma='scale', coef0=0.0,
        shrinking=True, probability=False, tol=0.001,
        cache_size=200, class_weight=None, verbose=False,
        max_iter=-1, decision_function_shape='ovr',
        break_ties=False, random_state=None
)
```

Stochastic Gradient Solvers

- For very large datasets, the best option us often to directly solve the primal form using Empirical Risk Minimization.
- Stochastic Gradient Descent (more on this when we get to Deep Learning) implements this in an efficient and sequential way.

```
class sklearn.linear_model.SGDClassifier(
        loss='hinge', *, penalty='l2', alpha=0.0001,
        l1_ratio=0.15, fit_intercept=True, max_iter=1000,
        tol=0.001, shuffle=True, verbose=0, epsilon=0.1,
        n_jobs=None, random_state=None, learning_rate='optimal',
        eta0=0.0, power_t=0.5, early_stopping=False,
        validation_fraction=0.1, n_iter_no_change=5,
        class_weight=None, warm_start=False, average=False
)
```

Custom Kernels

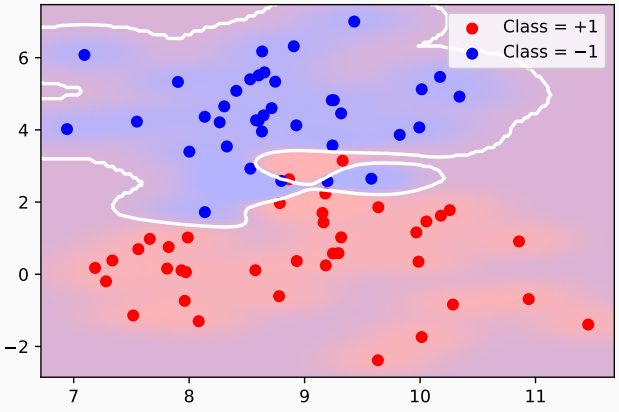
- For dual solvers the selection of the kernel is key (and usually requires extensive crossvalidation!)

```
kernel{‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’}, default=’rbf’

Specifies the kernel type to be used in the algorithm. It must be one
of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable.
If none is given, ‘rbf’ will be used. If a callable is given it is
used to pre-compute the kernel matrix from data matrices; that
matrix should be an array of shape (n_samples, n_samples).
```

Kernel machines and model complexity

- The kernel trick is another way to approach the non-linearly separable case.
- It is the primary way to increase model complexity without changing the underlying model formulation.



Concluding Remarks

The Support Vector Machine

- In the end, Support Vector Machines are always linear learning machines (in some space).
- The kernel trick allows us to define a linear learning problem (in a *potentially infinite dimensional space*) that is nonlinear in the original space.
- As always, care must be taken when applying the SVM in practice.
- Knowing how the primal and dual formulations work will make their performance and robustness much more predictable in practice.

Reading and Homework Assignments

Reading Assignment:

- Bishop: Chapter 7 (7.1), Chapter 6 (6.1, 6.2)

Homework:

- Convince yourself that the embedding $\phi(\cdot)$ corresponding to the Gaussian kernel is infinite dimensional.