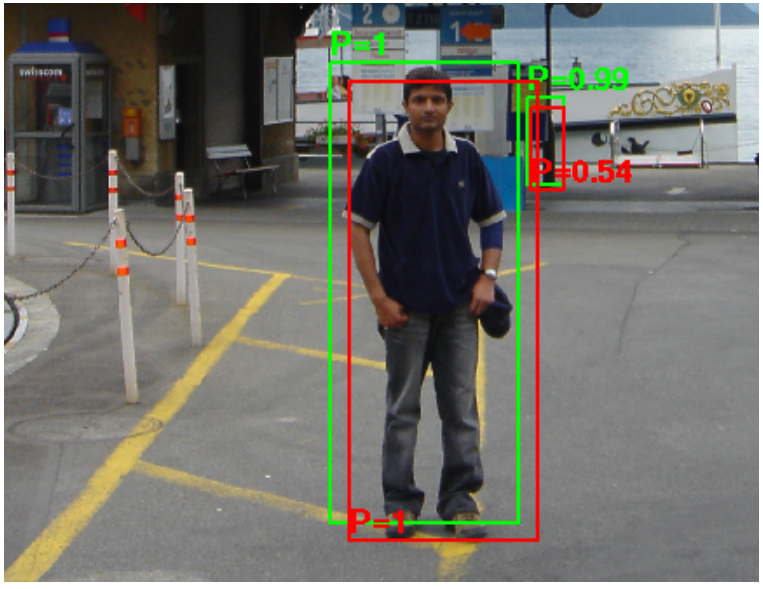
GPU-Accelerated Gaussian Processes for Object Detection

Calum Blair*, John Thompson*, Neil Robertson†
*Institute for Digital Communications,
University of Edinburgh
†Institute for Sensors, Signals and Systems,
Heriot-Watt University

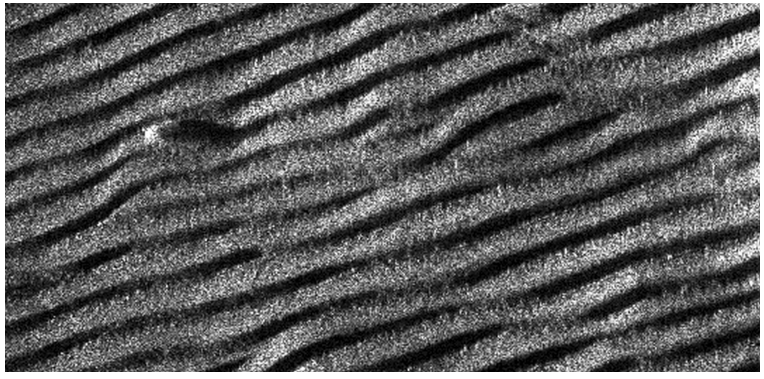
SSPD 2015
9th September 2015

Contents

Motivation



Pedestrian or object detection in images with realistic confidence measures†



Object detection in Sonar (SAS) imagery*

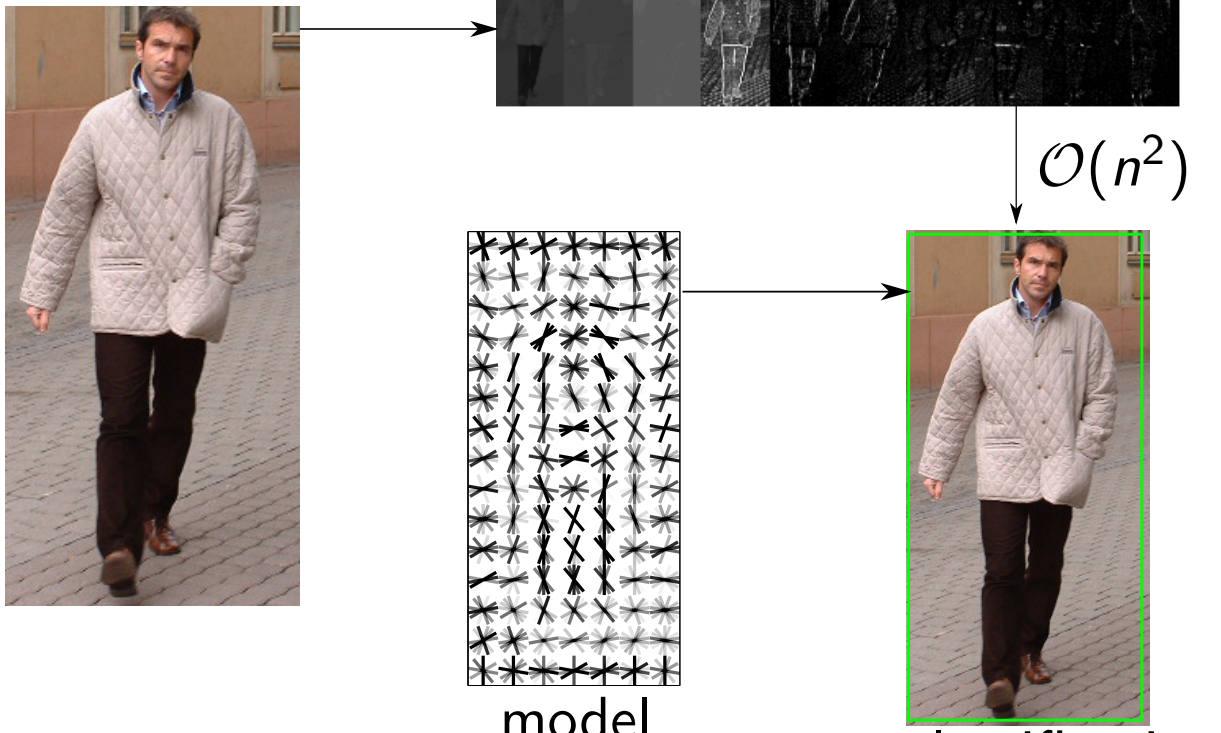
†Blair, Thompson, Robertson, *Introspective Classification for Pedestrian Detection*, SSPD 2014

*Blair, Thompson, Robertson, *Identifying Anomalous Objects in SAS Imagery Using Uncertainty*, Fusion 2015

Goals

Reliable and fast object detection: use Gaussian Processes as complete or final-stage classifier. Use support vector machines (**SVM**s) as baseline. GPU acceleration needed.

Classification Algorithm Structure



Gaussian process Classifiers (GPCs)

Given training data $\mathbf{X}$ and matching labels $\mathbf{y} \in \{0, 1\}$, do parameter learning. Perform probabilistic prediction $p(y = +1|\mathbf{x}_*)$ of new data sample $\mathbf{x}_*$.

Stage 1: define latent functions $f(x)$ as Gaussian distribution: $\mathcal{N}(\mu(x), k(\mathbf{X}, \mathbf{x}_*))$.

Covariance function k can be linear:

$$k(x_i, x_j) = \sigma \mathbf{x_i} \cdot \mathbf{x_j} \,. \tag{1}$$

or squared-error:

$$k(x_i, x_j) = \sigma \exp\left(-\frac{(\mathbf{x_i} - \mathbf{x_j})^2}{2\ell^2}\right) \,. \tag{2}$$

where σ, ℓ learned during training.

Classifier algorithm

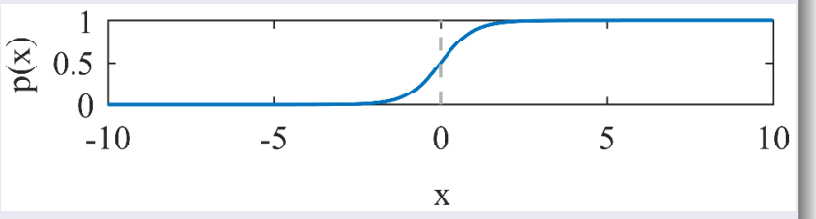
Estimate distribution of f_* which best fits $\mathbf{x}_*$:

$$p(f_*|\mathbf{X}, \mathbf{y}, \mathbf{x}_*) = \int p(f_*|\mathbf{X}, \mathbf{x}_*, \mathbf{f}) p(\mathbf{f}|\mathbf{X}, \mathbf{y}) d\mathbf{f}. \quad (3)$$

given $\mathbf{f}$ is the distribution of the latent function over $\mathbf{X}$.

Stage 2: 'squash' f_* using sigmoid with output range $[0, 1]$:

$$\sigma(x) = \frac{1}{(1 + \exp(-f(x))}. \quad (4)$$



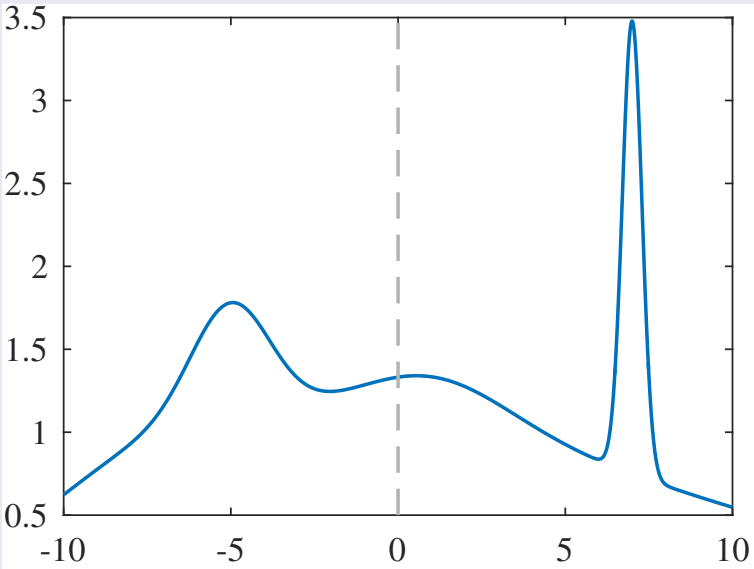
Final class membership probability π:

$$\pi \triangleq p(y = +1|\mathbf{X}, \mathbf{y}, \mathbf{x}_*) = \int \sigma(f_*) p(f_*|\mathbf{X}, \mathbf{y}, \mathbf{x}_*) df_* . \quad (5)$$

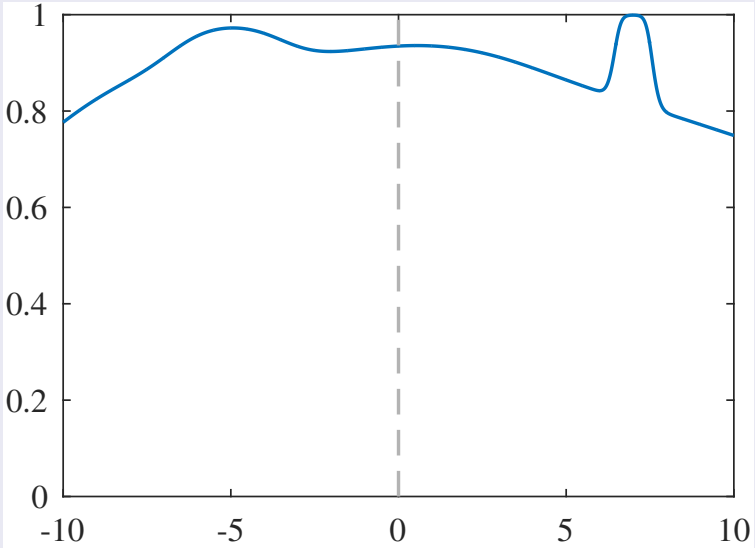
training process is $\mathcal{O}(n^3)$, while testing is $\mathcal{O}(n^2)$.

Graphical Interpretation

Stage 1: $\{\mathbf{X}, \mathbf{x}_*\} \rightarrow \mathbf{f}$



Stage 2: $\mathbf{f} \rightarrow \pi$



Baseline Algorithm

Support Vector Machine Comparison

Test equation:

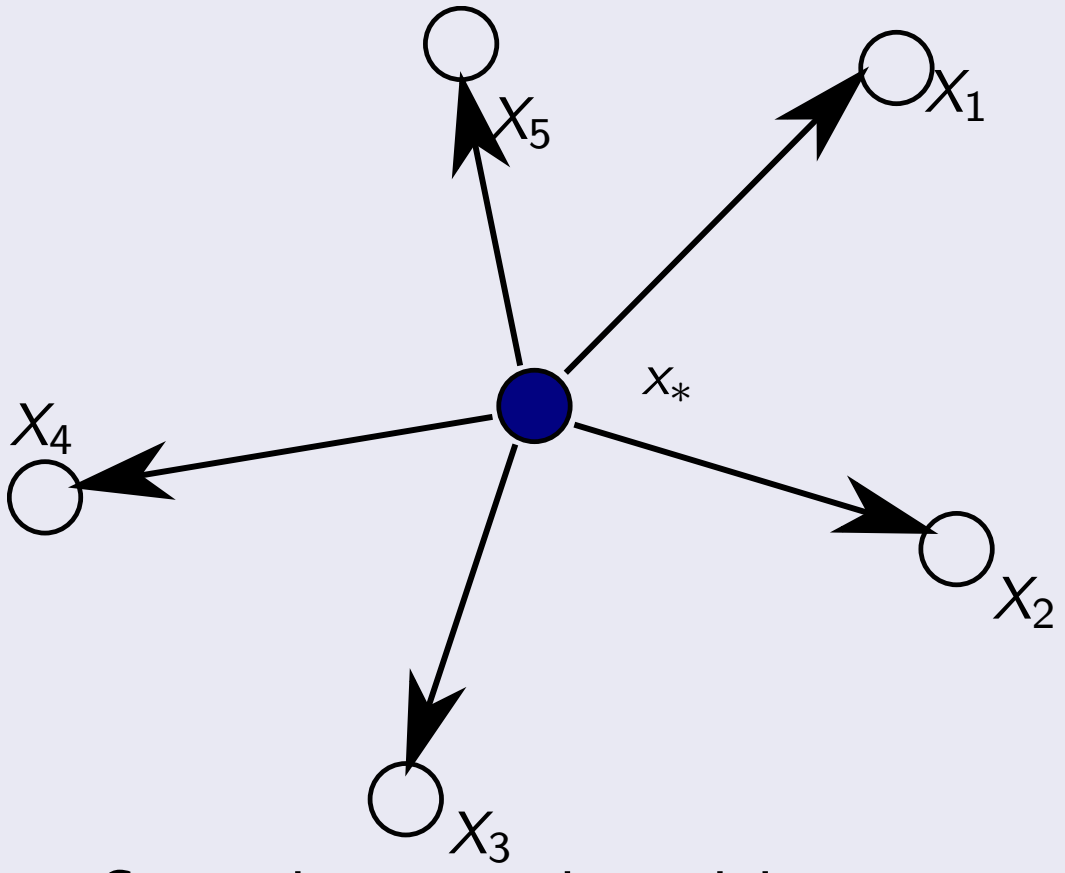
$$f(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i K(\mathbf{x}, \mathbf{w}_i) + b \tag{6}$$

$\alpha, \mathbf{w}$ and b learned during training. Use radial basis function (RBF) kernel: same as (2) above.

Difference with GPCs: $\mathbf{w}$ is condensed model of training data, but $\mathbf{X}$ is all samples seen.

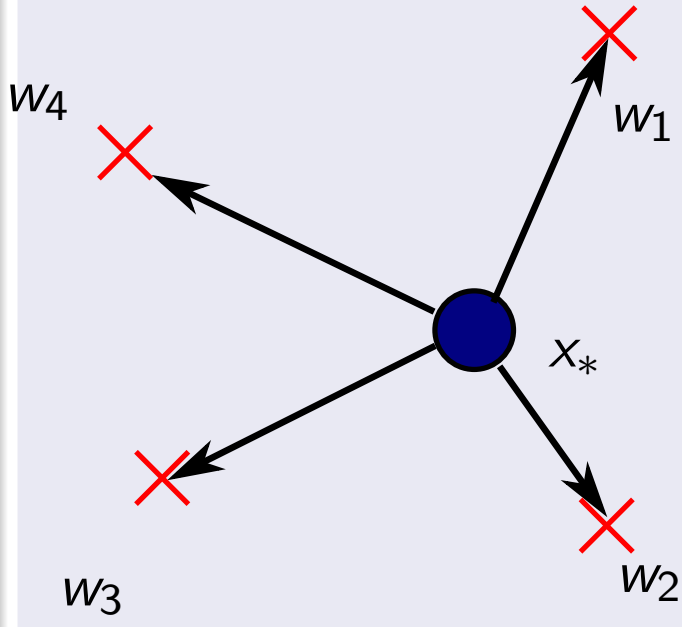
Graphical Interpretation

GPC-RBF: $\mathbf{X} \longleftrightarrow \mathbf{x}_*$



Comparison to entire training set

SVM-RBF: $\mathbf{x}_* \longleftrightarrow \mathbf{w}$



Smaller set of supporting points

Accelerating Matrix Computations

LAPACK (Linear Algebra Package) standard library.
Uses **BLAS** (Basic Linear Algebra Subprograms): vector, matrix and vector-matrix algorithms for multiplication and linear equations.
Highly optimised versions (tweak order of operations and cache contents), available for **Intel x86** (MKL, gotoBLAS, ...) and NVIDIA CUDA **GPU** (cuBLAS, MAGMA, nvBLAS).

MATLAB/ Numpy etc. make BLAS calls: $C = AB \rightarrow$

$$C = \mathrm{sgemm}(A, B)$$

single-precision, **ge**neral **m**atrix-**m**atrix multiply

BLAS Limitations

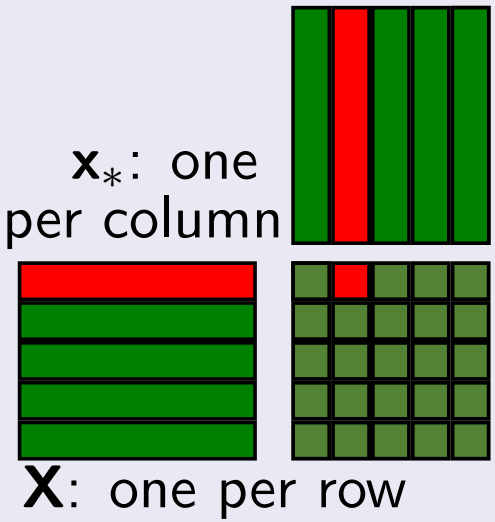
Must reformulate high-level operations to match available subroutines:

$$\exp\left(-\frac{(\mathbf{x_i}-\mathbf{x_j})^2}{2\ell^2}\right) \tag{7}$$

expands to:

$$(\mathbf{x}_i-\mathbf{x}_j)^2=\mathbf{x}_i^2+\mathbf{x}_j^2-2\mathbf{x}_i\mathbf{x}_j\,. \tag{8}$$

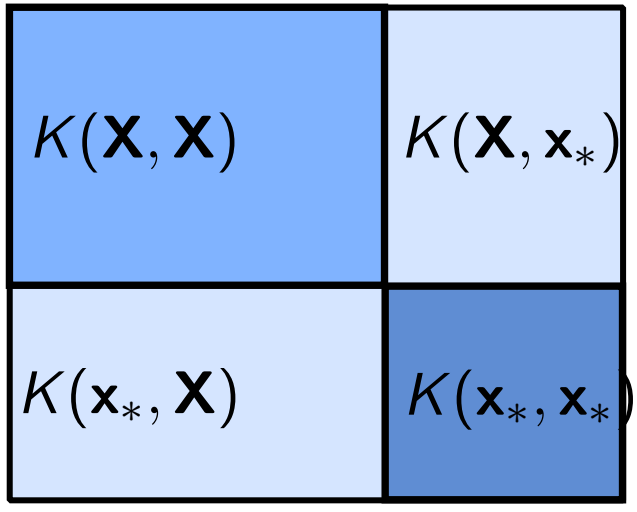
3 separate calls, 3 separate data accesses: problem when A, B are $\rightarrow 1GB$.
GPU model: limited memory, latency dominates. Here, $\mathbf{X}$ (training samples) is huge.
Now describe modification of GPC algorithm.



Inference

Goal: find π in (5) via predictive mean $\mathbb{E}[\mathbf{f}_*|\mathbf{X}, \mathbf{y}, \mathbf{x}_*]$ and predictive variance $\mathbb{V}[\mathbf{f}_*|\mathbf{X}, \mathbf{y}, \mathbf{x}_*]$†.
Training and test covariances form part of larger matrix:

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(\mathbf{X}, \mathbf{X}) & K(\mathbf{X}, \mathbf{x}_*) \\ K(\mathbf{x}_*, \mathbf{X}) & K(\mathbf{x}_*, \mathbf{x}_*) \end{bmatrix}\right) \tag{9}$$



Define K_X as $K(\mathbf{X}, \mathbf{X})$, K_{X*} as $K(\mathbf{X}, \mathbf{x}_*)$ and K_* as $K(\mathbf{x}_*, \mathbf{x}_*)$. Write a conditional Gaussian on (9) as:

$$\mathbf{f}_*|\mathbf{X}, \mathbf{x}_*, \mathbf{f} \sim \mathcal{N}(K(\mathbf{x}_*, \mathbf{X})K_X^{-1}\mathbf{f}, K_* - K(\mathbf{x}_*, \mathbf{X})K_X^{-1}, K_{X*}) \tag{10}$$

Now have one term in $\mathbf{f}_*$-expression.

†Ch.3, Rasmussen & Williams, *Gaussian Processes for Machine Learning* (2006).

Abridged Maths

Approximate posterior term with a Gaussian and:

$$p(\mathbf{f}|\mathbf{X},\mathbf{y}) \approx q(\mathbf{f}|\mathbf{X},\mathbf{y}) = \mathcal{N}(\hat{\mathbf{f}}, A^{-1}) \tag{11}$$

Obtain **predictive mean** as:

$$\mathbb{E}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] = K_{X*}^{\mathsf{T}} \nabla \log p(\mathbf{y}|\hat{\mathbf{f}}). \tag{12}$$

Define **predictive variance** as:

$$\mathbb{V}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] = K_* - K_{X*}^{\mathsf{T}}(K_X + W^{-1})^{-1}K_{X*}\,, \tag{13}$$

Using $W \triangleq -\nabla\nabla \log(p(\mathbf{y}|\mathbf{f}))$, $\mathbf{L} = \mathrm{cholesky}(I + W^{\frac{1}{2}} K_X W^{\frac{1}{2}})$, and $\mathbf{v} = \mathbf{L}\backslash(W^{\frac{1}{2}} K_{X*})$, simplify to:

$$\mathbb{V}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] = K_* - \mathbf{v}^{\mathsf{T}}\mathbf{v} \tag{14}$$

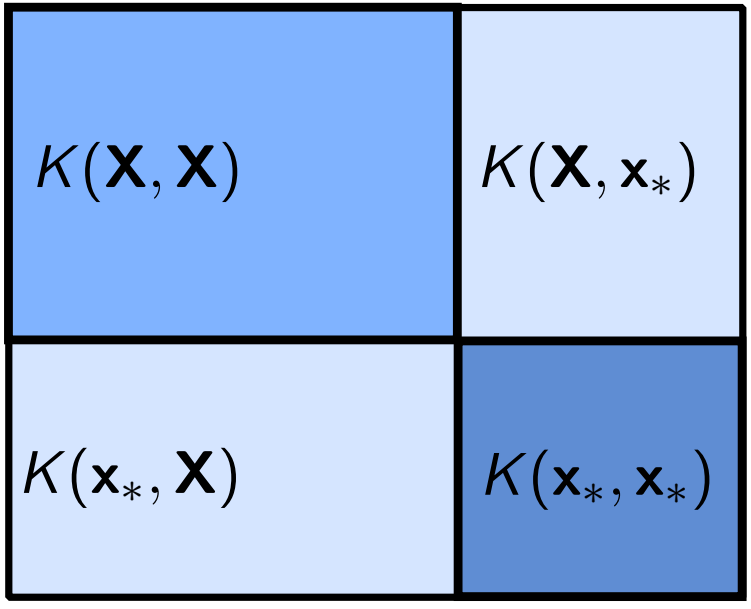
See paper for complete derivations

Probabilistic Prediction: Full algorithm

Require: $\mathbf{X}, \mathbf{x}_*, \mathbf{y}, \hat{\mathbf{f}}, W, L$, kernel function $k(x_i, x_j)$

1: $K_{X*} = K(\mathbf{X}, \mathbf{x}_*)$ ▶
2: $K_* = K(\mathbf{x}_*, \mathbf{x}_*)$ ▶
3: $\mathbb{E}_q[f_*|X, \mathbf{y}, \mathbf{x}_*] = K_{X*}^\intercal \nabla \log(p(\mathbf{y}|\hat{\mathbf{f}}))$ // latent mean
4: $\mathbf{v} = L \backslash (W^{\frac{1}{2}} K_{X*})$ ▶
5: $\mathbb{V}_q[f_*|X, \mathbf{y}, \mathbf{x}_*] = K_* - \mathbf{v}^\intercal \mathbf{v}$ // latent variance
6: $\bar{\pi}_* = \int \sigma(z) \mathcal{N}(z|\mathbb{E}_q[f_*], \mathbb{V}_q[f_*]) dz$ // prediction
7: **return** $\bar{\pi}$

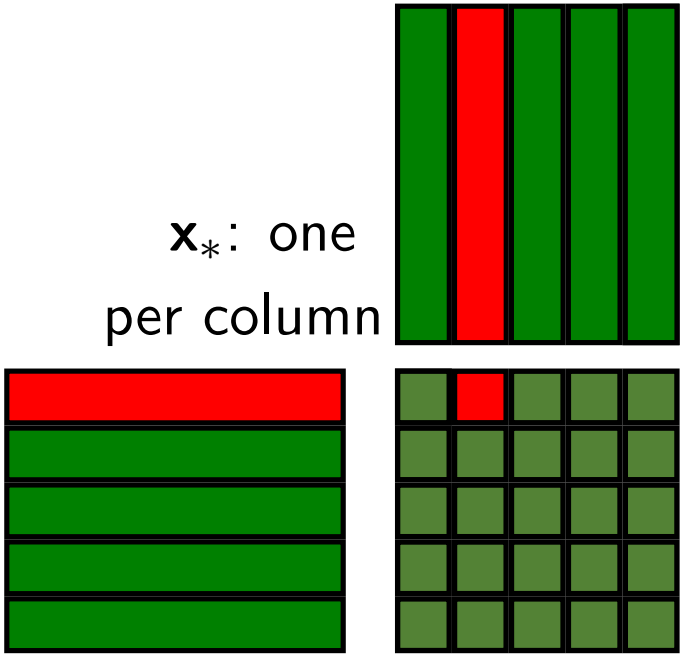
Figure: Calculate π at test time. Compute-heavy lines marked with ▶.



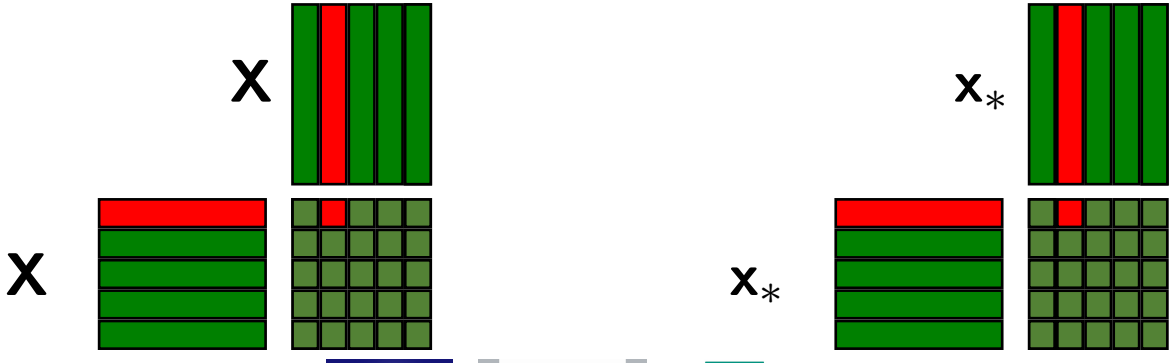
Optimisation

Each sample has $d \sim 5000$.
Same block-level data reused for ~ 100 sliding windows in image.

Optimised Matrix Multiplication

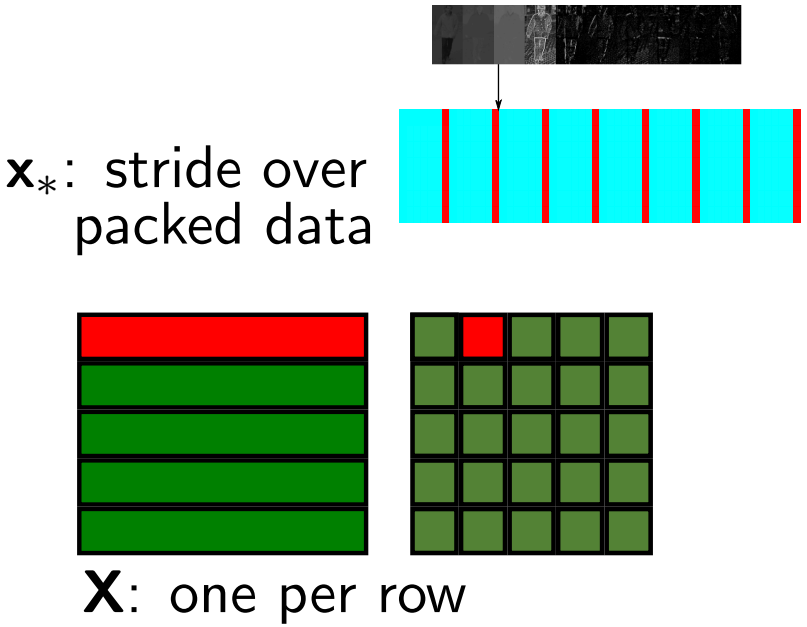


$C = AB$: load tiles of A and B into fast memory.
(Existing work optimised tile sizes via automated parameter exploration.)
For K_{X*} and K_X, $A = \mathbf{X}$; usual matrix structure, one sample per row, no overlap.



Improvements

When B is $\mathbf{x}_*$: densely packed; instead of one row per window, re-use nearby data already in fast shared memory.
Time to access A and C (the resulting K_{X*} matrix) dominates. Big reductions in time & memory consumption. Further improvements from instruction level parallelism.



Results

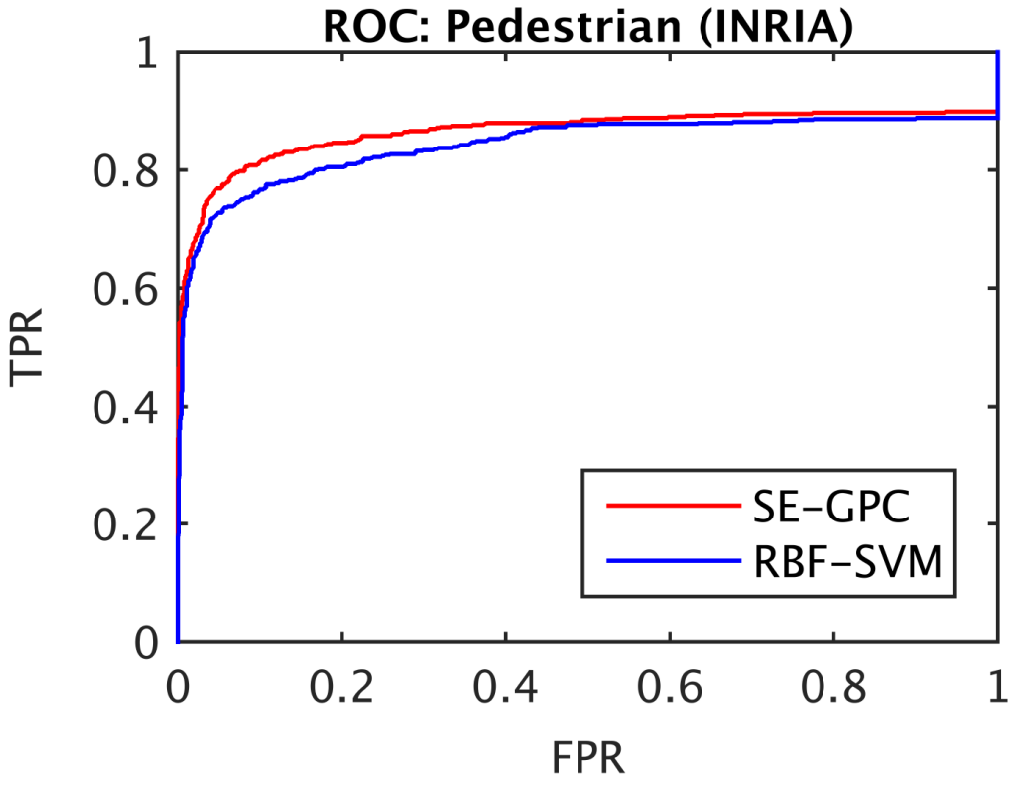
- Timing: test processing speed on single image
- Accuracy: test on large dataset

Timing

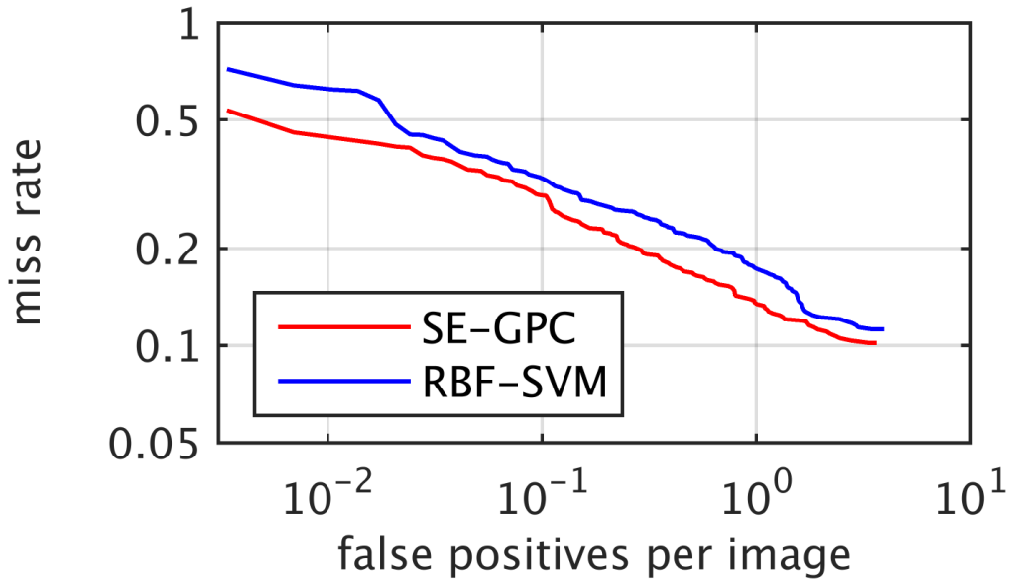
Algorithm	Processor	Implementation	Time(s)	Speedup
GPC	CPU	MATLAB BLAS	10.28	
GPC	**GPU**	GPGPGPU	2.77	3.7×
SVM	CPU	LIBSVM	66.92	
SVM	**GPU**	cuSVM	1.74	38.5×

Matrix multiplication stage for 640×480 image on CPU (12-core Intel Xeon X5650, 2.67GHz) and GPU (NVIDIA GeForce GTX 680, 1536 cores, 2GB RAM).
cuSVM implementation is faster as SVM needs fewer support vectors ($\sim$3000 vs $\sim$14000 GPC training vectors).

Receiver Operating Characteristic



Detection Error Tradeoff



Reliability Diagram

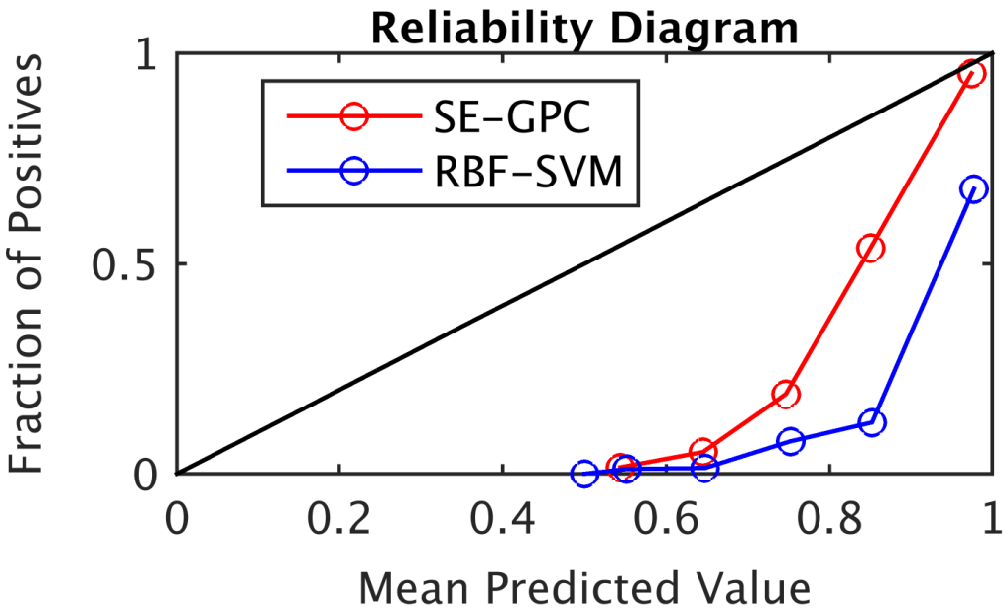


Figure: GPC and SVM Reliability; classifiers closer to black line are more reliable.

Conclusion

GPC compared to baseline SVM: similar speed but gain in reliability.

Best case is 3.7× speedup compared to an optimised implementation on CPU.

Improvements usually possible even over heavily optimised initial code, when matched to application.

Code available for download†.

Questions?

†http://homepages.ed.ac.uk/cblair2/

Appendix

Training posterior

Require: $\mathbf{X}, \mathbf{y}, \mathbf{f}$, kernel function $k(x_i, x_j)$

1: $\hat{\mathbf{f}} \triangleq \mathbb{E}_q[\mathbf{f}, \mathbf{X}, \mathbf{y}] = \operatorname{argmax}_{\mathbf{f}} p(\mathbf{f}|\mathbf{X}, \mathbf{y})$ // Using Newton's method

2: $K_X = K(\mathbf{X}, \mathbf{X})$

3: $W = -\nabla\nabla \log(p(\mathbf{y}|\hat{\mathbf{f}}))$

4: $L = \text{cholesky}(I + W^{\frac{1}{2}} K W^{\frac{1}{2}})$

5: **return** $W, L, \hat{\mathbf{f}}, K_X$

Figure: Prepare training posterior. This only needs to be done once and can be re-used during testing.

Derivation of Mean

Laplacian approximation: treat posterior over the training data and labels in our $\mathbf{f}_*$ term (3) as a Gaussian:

$$p(\mathbf{f}|\mathbf{X},\mathbf{y}) \approx q(\mathbf{f}|\mathbf{X},\mathbf{y}) = \mathcal{N}(\hat{\mathbf{f}}, A^{-1}), \quad (15)$$

where

$$\hat{\mathbf{f}} = \arg\max_{\mathbf{f}} p(\mathbf{f}|\mathbf{X},\mathbf{y}), \quad (16)$$

and (where ∇ represents differentiation):

$$A = -\nabla\nabla \log(p(\mathbf{f}|\mathbf{X},\mathbf{y})|_{\mathbf{f}=\hat{\mathbf{f}}}. \quad (17)$$

$\hat{\mathbf{f}}$ can thus be found by applying Bayes' rule to the posterior distribution over the training variables,
$p(\mathbf{f}|\mathbf{X},\mathbf{y}) = p(\mathbf{y}|\mathbf{f})p(\mathbf{f}|\mathbf{X})/p(\mathbf{y}|\mathbf{X})$. Discard $p(\mathbf{y}|\mathbf{X})$ as maximising $\mathbf{f}$. Take log and differentiate $p(\mathbf{f}|\mathbf{X},\mathbf{y})$ to get predictive mean:

$$\mathbb{E}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] = K_{X*}^{\mathsf{T}} \nabla \log p(\mathbf{y}|\hat{\mathbf{f}}). \quad (18)$$

Derivation of Variance

Define predictive variance as:

$$\mathbb{V}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] = K_* - K_{X*}^{\mathsf{T}}(K_X + W^{-1})^{-1}K_{X*}\,, \tag{19}$$

using $W \triangleq -\nabla\nabla\log(p(\mathbf{y}|\mathbf{f}))$. Defining the symmetric positive definite matrix $\mathbf{B}$ as $\mathbf{B} = I + W^{\frac{1}{2}}K_X W^{\frac{1}{2}}$, $\mathbf{L}\mathbf{L}^{\mathsf{T}} = \mathbf{B}$ so $\mathbf{L} = \mathrm{cholesky}(\mathbf{B})$, and $\mathbf{v} = L\backslash(W^{\frac{1}{2}}K_{X*})$ simplify to:

$$\begin{aligned}\mathbb{V}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] &= K_* - K_{X*}^{\mathsf{T}}W^{\frac{1}{2}}(LL^{\mathsf{T}})^{-1}W^{\frac{1}{2}}K_{X*} &(20)\\ \mathbb{V}_q[\mathbf{f}_*|\mathbf{X},\mathbf{y},\mathbf{x}_*] &= K_* - \mathbf{v}^{\mathsf{T}}\mathbf{v} &(21)\end{aligned}$$

Posterior term in π (5) now approximated as a Gaussian $q(f_*|\mathbf{X},\mathbf{y},\mathbf{x}_*)$ with mean $\mathbb{E}$ and variance $\mathbb{V}$.

The solution of the division involving the lower triangular matrix L on line 4 requires too much memory to obtain any benefit from performing the calculation on a GPU. In our experiments it proved to be faster to execute this on the CPU; the memory limitations on the GPU meant that the test covariance matrix K_{X*} had to be partitioned into very small batches, because of the large size of K_X.