

NONPARAMETRIC BAYESIAN CLASSIFICATION WITH MASSIVE DATASETS: LARGE-SCALE QUASAR DISCOVERY

ALEXANDER GRAY

Georgia Institute of Technology, College of Computing
agrays@cc.gatech.edu

GORDON RICHARDS

Princeton University, Department of Astrophysical Sciences
gtr@astro.princeton.edu

ROBERT NICHOL

University of Portsmouth, Institute of Cosmology and Gravitation
bob.nichol@port.ac.uk

ROBERT BRUNNER

University of Illinois, Department of Astronomy and NCSA
rb@ncsa.uiuc.edu

ANDREW MOORE

Carnegie Mellon University, School of Computer Science
aum@cs.cmu.edu

The kernel discriminant (a nonparametric Bayesian classifier) is appropriate for many scientific tasks because it is highly accurate (it approaches Bayes optimality as you get more data), distribution-free (works for arbitrary data distributions), and it is easy to inject prior domain knowledge into it and interpret what it's doing. Unfortunately, like other highly accurate classifiers, it is computationally infeasible for massive datasets. We present a fast algorithm for performing classification with the kernel discriminant exactly (i.e. without introducing any approximation error). We demonstrate its use for quasar discovery, a problem central to cosmology and astrophysics, tractably using 500K training data and 800K testing data from the Sloan Digital Sky Survey. The resulting catalog of 100K quasars significantly exceeds existing quasar catalogs in both size and quality, opening a number of new scientific possibilities, including the recent empirical confirmation of cosmic magnification which has received wide attention.

1. Introduction and Approach

Quasars are star-like objects which are not very well understood yet play a critical role in cosmology. As the most luminous (and thus the most distant) objects in the universe, they can be used as markers of the mass in the distant (early) universe. With the very recent advent of massive sky surveys such as the Sloan Digital Sky Survey (SDSS), it is now conceivable in principle to obtain a catalog of the locations of quasars which is more comprehensive than ever before, both in sky coverage and depth (distance). Such a catalog would open the door to numerous powerful analyses of the early/distant universe which were never before possible. A central challenge of this activity is the question of how to use the limited information we have in hand (a tiny set of known, nearby

quasars) to extract a massive amount of more subtle information from the SDSS dataset (a large set of faint quasar candidates). In this paper we describe a method which has yielded the most comprehensive and accurate quasar catalog to date. The catalog and data methodology have been previously described¹; here we describe the algorithm for the first time.

Nonparametric Bayesian classification. We wish to classify a set of unlabeled objects (the *test set*, or *query points*) as either stars or quasars. We first create samples of “stars” and “quasars” that will serve as *training sets* or *reference points*. The probability that an object producing data x (represented by four color measurements) is a star is proportional to the probability that x would be produced by a star, $p(x|C_1)$; this is the *likelihood* under the the probability density function (pdf), which must be

estimated, describing the star class C_1 .

To incorporate prior or subjective information, in our case the fraction of an unseen set of objects which the user roughly expects to be stars, we use a simple application of Bayes' Rule:

$$P(C_1|x) = \frac{p(x|C_1)P(C_1)}{p(x|C_1)P(C_1) + p(x|C_2)P(C_2)}.$$

Objects with $P(C_1|x) > 0.5$ are classified as stars, which we denote by assigning class label $c(x) = C_1$, otherwise as quasars.

Kernel density estimator. For the likelihood of each class we will use the nonparametric *kernel density estimate* (KDE) ² of the pdf, a mature statistical method which can be thought of as a generalization of the histogram, having the form

$$p(x|C) = \frac{1}{N} \sum_i^N K_h(x, x_i)$$

where the kernel is for example, the Gaussian $K_h(x, x_i) = K(h, \|x - x_i\|) = \frac{1}{C(h)} \exp\|x - x_i\|^2/h^2$ where $C(h)$ is a normalizing constant which depends on h . h , called the *bandwidth*, is the critical parameter which controls the smoothness of the estimate. In our method we use a slight generalization in which each point may be given different weights: $p(x|C) = \frac{1}{W} \sum_i^N w_i K_h(x, x_i)$ where $W = \sum_i^N w_i$, in order to possibly allow for measurement uncertainty or other prior knowledge. In the algorithm we'll work with unnormalized sums such as $\Phi(x|C) = \sum_i^N w_i K_h(x, x_i)$ so that $p(x|C) = \frac{1}{W} \Phi(x|C)$. We'll compress the notation by referring to $p(C_1|x)$ and $\Phi(x|C_1)$ as $p_1(x)$ and $\Phi_1(x)$. We refer to the Bayes classifier using KDE estimates as the *kernel discriminant* and such classification as *kernel discriminant analysis* (KDA).

Computational challenge. Training the kernel discriminant consists of finding the parameters $\{h_1, h_2\}$ which will maximize its performance at predicting the class labels of data drawn from the same distribution as the training data. We use as an estimator of this performance the leave-one-out cross-validated accuracy score. This form of classifier is highly accurate in practice. The main reason it is not commonly used is that it comes with a severe quadratic computational cost. The main problem treated in this paper is that of computing KDA classifications tractably.

2. Algorithm

We developed an algorithm which computes for each query point x_q its class label $c(x_q)$ as if the sums $\Phi_1(x)$ and $\Phi_2(x)$ had been computed *exactly*, though in many cases they need not be.

First, a space-partitioning tree data structure such as a *kd-tree* ³ is constructed on the query (testing) dataset, and another is created on the reference (training) dataset.

The idea is to maintain bounds on $p_1(x)$ and $p_2(x)$ and successively tighten them in a multi-resolution manner, as nodes at increasingly finer levels of the trees are considered, until we can show that the bounds determine that one of these class probabilities must dominate the other. This is true if one is definitely greater than 0.5, definitely less than 0.5, or definitely greater than the other. Initially the class label for each query point $c(x_q)$ is recorded as “?” (unknown), and is updated to C_1 or C_2 when the bounds determine it. Efficiency over the naive algorithm is obtained to the extent that we are able to determine the label for large chunks of the query points simultaneously.

Bounds. We'll maintain various bounds during the run of the algorithm, including bounds on $\Phi_1(x)$ and $\Phi_2(x)$, e.g. $\Phi_1^L(x) \leq \Phi_1(x)$ and $\Phi_1^U(x) \geq \Phi_1(x)$, and bounds on $p_1(x)$ and $p_2(x)$, e.g. $p_1^L(x) \leq p_1(x)$ and $p_1^U(x) \geq p_1(x)$.

We'll also maintain bounds which hold for various subsets X of the points, which correspond to tree nodes, e.g. $\forall x \in X: \Phi_1^L(X) \leq \Phi_1(x)$, $\Phi_1^U(X) \geq \Phi_1(x)$, $p_1^L(X) \leq p_1(x)$, and $p_1^U(X) \geq p_1(x)$. We can utilize bounds on the class-conditional likelihoods to obtain simple bounds on the final class probability:

$$p_1^L(x) := (\Phi_1^L(x)\pi_1) / (\Phi_1^L(x)\pi_1 + \Phi_2^U(x)\pi_2)$$

$$p_1^U(x) := (\Phi_1^U(x)\pi_1) / (\Phi_1^U(x)\pi_1 + \Phi_2^L(x)\pi_2).$$

Within each node X , in an efficient bottom-up (dynamic programming) fashion, we compute and store certain properties of the class 1 points (if any) and the class 2 points (if any) which reside in the node: for each class, the bounding box of the points in that class and the sum of the weights of the points in that class, $W_1(X)$ and $W_2(X)$. Note that expressions like $W_1(X)$ are generally implemented as $X.W_1$, to use a C-like notation.

We can use these bounding boxes to compute simple lower and upper bounds on the distance be-

tween any point in a node Q and any point (of a certain class) in a node R , *e.g.*: $\forall x_q \in Q, \forall x_r \in R$ such that $c(x_r) = C_1$: $\delta_1^L(Q, R) \leq \delta_{qr}$ and $\delta_1^U(Q, R) \geq \delta_{qr}$, where $\delta_{qr} = \|x_q - x_r\|$.

Bound tightening. Let K^L and K^U be constants such that $\forall x, y$: $K^L \leq K(x, y)$ and $K^U \geq K(x, y)$ – for most kernels of interest such as the Gaussian, which are probability density functions, the lower bound is 0 and upper bound is 1. At the beginning of a run of the algorithm, the bounds are initialized using these values, *e.g.*: $\forall x_q$: $\Phi_1^L(x_q) = W_1 K^L$ and $\Phi_1^U(x_q) = W_1 K^U$. For each query x_q , the bounds $\Phi_1^L(x)$ and $\Phi_1^U(x)$ have accounted for each reference point’s potential contribution to the sum in a worst-case manner.

Nodes are examined in pairs $\{Q, R\}$ – one node Q from the query tree and one node R from the reference tree. The idea is that when we see a new reference node R , we can tighten our bounds on the contribution of the reference points in R to the sum for each query point. When doing so, we must also undo the previous contribution of the points in R to each of our bounds. For example the new contribution of R to $\Phi_1^L(Q)$ is $W_1(R)K(h_1, \delta_1^U(Q, R))$ whereas the old contribution was implicitly $W_1(R)K^L$. So we update $\Phi_1^L(Q)$ by adding to it

$$\Delta\Phi_1^L(Q, R) := W_1(R)K_{h_1}(\delta_1^U(Q, R)) - W_1(R)K^L.$$

Similarly, we change $\Phi_1^U(Q)$ by adding to it

$$\Delta\Phi_1^U(Q, R) := W_1(R)K_{h_1}(\delta_1^L(Q, R)) - W_1(R)K^U.$$

Because we always move downward in the tree, these updates are always improvements to the bounds or at worst leave them unchanged.

Control flow. The order in which nodes are examined is determined by a min-priority queue which stores node-pair objects $\{Q, R\}$. Note that values such as $\Delta\Phi_1^L(\{Q, R\})$ ($\Delta\Phi_1^L$ for short) are often implemented as $\{Q, R\}.\Delta\Phi_1^L$. A node-pair object stores the change values that are computed for it, the *previous* such values (denoted by apostrophes), and its priority.

Node-pair $\{Q, R\}$ is assigned priority

$$f(Q, R) := |(\Delta\Phi_1^U - \Delta\Phi_1^{U'}) + (\Delta\Phi_2^U - \Delta\Phi_2^{U'}) - (\Delta\Phi_1^L - \Delta\Phi_1^{L'}) - (\Delta\Phi_2^L - \Delta\Phi_2^{L'})|,$$

the difference in improvement (*i.e.* current values minus previous values) of the upper bounds and the lower bounds. A procedure **makePair**(Q, R, \dots)

creates the node-pair structure $\{Q, R\}$ and stores the other arguments in its slots for $\Delta\Phi_1^{L'}$, $\Delta\Phi_1^{U'}$, $\Delta\Phi_2^{L'}$, $\Delta\Phi_2^{U'}$, respectively. **computeBounds**(Q, R) computes the Δ values and the priority for node-pair $\{Q, R\}$. Node-pairs are expanded further by placing every pairwise combination of their respective children on the queue. Each node-pair also stores an “undo” flag $\text{undo}(Q, R)$ which determines whether it should be expanded. Whenever improvements are made to the bounds of a query node, they are updated in all the children of the query node with a simple recursive routine **passDown**(Q, \dots). For each node Q in the query tree we store $M(Q)$, the number of points in the tree which have known class labels (definitely C_1 or C_2). If we encounter a node for which all $N(Q)$ of the query points have known labels, we can stop recursing on it.

When both Q and R are leaf nodes, this corresponds to the base case of the recursion. In this case we compute the contribution of each point in R to each point in Q exhaustively. Because this direct type of contribution is exact and thus unchangeable, while other contributions tighten bounds which can change, it is useful to record it separately – we denote it by $\phi(x_q)$ for each query point.

In the pseudocode, for brevity, we use the convention that children of leaves point to themselves, and redundant node pairs are not placed on the priority queue. In the pseudocode, following a C-like notation for compactness, the $a += b$ denotes $a = a + b$ and $a != b$ denotes $\overline{(a = b)}$. The pseudocode shows a version of the algorithm which only computes and uses the bounds for one of the classes.

3. Results

Using the algorithm described, we were able tractably estimate (find optimal parameters for) a classifier based on a large training set consisting of 500K star-labeled objects and 16K quasar-labeled objects, and predict the label for 800K faint (up to $g = 21$) query objects from 2099 deg² of the SDSS DR1 imaging dataset. Of these, 100K were predicted to be quasars, forming our catalog of quasar candidates. This significantly exceeds the size, faintness, and sky coverage of the largest quasar catalogs to date. Based on spectroscopic hand-validation of a subset of the candidates, we estimate that 95.0% are truly quasars, and that we have identified 94.7% of

the actual quasars. These efficiency and completeness numbers far exceed those any previous catalog, making our catalog both the most comprehensive and most accurate to date. The recent empirical confirmation of cosmic magnification⁴ using our catalog is an example of the scientific possibilities opened up by this work. In ongoing efforts we are exploring ways to make the method both more computationally and statistically efficient, with the goal of obtaining all 1.6M quasars we estimate are detectable in principle from the entire SDSS dataset.

References

1. Richards, G., Nichol, R., Gray, A., *et. al*, Efficient Photometric Selection of Quasars from the Sloan Digital Sky Survey: 100,000 $z > 3$ Quasars from Data Release One, *Astrophysical Journal Supplement Series* 155, 2, 257–269, 2004.
2. Silverman, B. W., Density Estimation for Statistics and Data Analysis, Chapman and Hall/CRC, 1986.
3. Preparata, F. P. and Shamos, M., *Computational Geometry*, Springer-Verlag, 1985.
4. Scranton, R., *et. al*, Detection of Cosmic Magnification with the Sloan Digital Sky Survey, *Astrophysical Journal* 633, 2, 589–602, 2005. Described in *Nature*, April 27, 2005.

kdaBase(Q, R)

```
forall  $x_q \in Q$ ,
  if  $c(x_q) = \text{"?"}$ ,
    forall  $x_r \in R$ ,
      if  $c(x_q) = C_1$ ,  $\phi_1(x_q) += w_r K_{h_1}(\delta_q r)$ .
      if  $c(x_q) = C_2$ ,  $\phi_2(x_q) += w_r K_{h_1}(\delta_q r)$ .
```

```
 $\Phi_1^L(x_q) := C(h_1)(\Phi_1^L(Q) + \phi_1(x_q) - W_1(R)K^L)$ .
 $\Phi_2^L(x_q) := C(h_2)(\Phi_2^L(Q) + \phi_2(x_q) - W_2(R)K^L)$ .
 $\Phi_1^U(x_q) := C(h_1)(\Phi_1^U(Q) + \phi_1(x_q) - W_1(R)K^U)$ .
 $\Phi_2^U(x_q) := C(h_2)(\Phi_2^U(Q) + \phi_2(x_q) - W_2(R)K^U)$ .
```

```
 $p_1^L(x_q) := \Phi_1^L(x_q)\pi_1 / (\Phi_1^L(x_q)\pi_1 + \Phi_2^L(x_q)\pi_2)$ .
 $p_1^U(x_q) := \Phi_1^U(x_q)\pi_1 / (\Phi_1^U(x_q)\pi_1 + \Phi_2^U(x_q)\pi_2)$ .
```

```
if  $p_1^L(x_q) \geq 0.5$ ,  $c(x_q) := C_1$ .
if  $p_1^U(x_q) < 0.5$ ,  $c(x_q) := C_2$ .
if  $c(x_q) \neq \text{"?"}$ ,  $M(Q) += 1$ .
```

```
 $\phi_1^L(Q) := \min_{x_q \in Q} \phi_1(x_q)$ .
 $\phi_2^L(Q) := \min_{x_q \in Q} \phi_2(x_q)$ .
 $\phi_1^U(Q) := \max_{x_q \in Q} \phi_1(x_q)$ .
 $\phi_2^U(Q) := \max_{x_q \in Q} \phi_2(x_q)$ .
```

```
 $\Phi_1^L(Q) := W_1(R)K^L$ ,  $\Phi_2^L(Q) := W_2(R)K^L$ .
 $\Phi_1^U(Q) := W_1(R)K^U$ ,  $\Phi_2^U(Q) := W_2(R)K^U$ .
```

kda(Q_{root}, R_{root})

```
{ $Q_{root}, R_{root}$ } := makePair( $Q_{root}, R_{root}, 0, 0, 0, 0$ ).
computeBounds({ $Q_{root}, R_{root}$ }).
insertHeap( $H, \{Q_{root}, R_{root}\}$ ).
```

while H is not empty,

```
{ $Q, R$ } := extractMin( $H$ ).
```

```
if  $M(Q) = N(Q)$ , skip.
```

```
if !leaf( $Q$ ),
```

```
 $\Phi_1^L(Q) := \min(\Phi_1^L(\text{ch}_1(Q)), \Phi_1^L(\text{ch}_2(Q)))$ .
```

```
 $\Phi_2^L(Q) := \min(\Phi_2^L(\text{ch}_1(Q)), \Phi_2^L(\text{ch}_2(Q)))$ .
```

```
 $\phi_1^L(Q) := \min(\phi_1^L(\text{ch}_1(Q)), \phi_1^L(\text{ch}_2(Q)))$ .
```

```
 $\phi_2^L(Q) := \min(\phi_2^L(\text{ch}_1(Q)), \phi_2^L(\text{ch}_2(Q)))$ .
```

```
(similar for upper bounds)
```

```
 $M(Q) := M(\text{ch}_1(Q)) + M(\text{ch}_2(Q))$ .
```

```
 $\Delta\Phi_1^L := \Delta\Phi_1^L(\{Q, R\}), \dots$ 
```

```
if undo( $Q, R$ ), passDown( $Q, \Delta\Phi_1^L - \Delta\Phi_1^{L'}$ ,
   $\Delta\Phi_1^U - \Delta\Phi_1^{U'}$ ,  $\Delta\Phi_2^L - \Delta\Phi_2^{L'}$ ,  $\Delta\Phi_2^U - \Delta\Phi_2^{U'}$ ).
else, passDown( $Q, \Delta\Phi_1^L, \Delta\Phi_1^U, \Delta\Phi_2^L, \Delta\Phi_2^U$ ).
```

```
 $\Phi_1^L := C(h_1)(\Phi_1^L(Q) + \phi_1^L(Q))$ .
```

```
 $\Phi_2^L := C(h_2)(\Phi_2^L(Q) + \phi_2^L(Q))$ .
```

```
 $\Phi_1^U := C(h_1)(\Phi_1^U(Q) + \phi_1^U(Q))$ .
```

```
 $\Phi_2^U := C(h_2)(\Phi_2^U(Q) + \phi_2^U(Q))$ .
```

```
 $p_1^L(Q) = \Phi_1^L\pi_1 / (\Phi_1^L\pi_1 + \Phi_2^L\pi_2)$ .
```

```
 $p_1^U(Q) = \Phi_1^U\pi_1 / (\Phi_1^U\pi_1 + \Phi_2^U\pi_2)$ .
```

```
if  $p_1^L(Q) \geq 0.5$ ,  $c(Q) := C_1$ .
```

```
if  $p_1^U(Q) < 0.5$ ,  $c(Q) := C_2$ .
```

```
if  $c(Q) \neq \text{"?"}$ ,
```

```
 $M(Q) := N(Q)$ . skip.
```

```
if leaf( $Q$ ) and leaf( $R$ ),
```

```
passDown( $Q, \Delta\Phi_1^L, \Delta\Phi_1^U, \Delta\Phi_2^L, \Delta\Phi_2^U$ ).
```

```
kdaBase( $Q, R$ ).
```

```
else,
```

```
{ $\text{ch}_1(Q), \text{ch}_1(R)$ } := makePair( $\text{ch}_1(Q)$ ,
   $\text{ch}_1(R), \Delta\Phi_1^L, \Delta\Phi_1^U, \Delta\Phi_2^L, \Delta\Phi_2^U$ ).
```

```
{ $\text{ch}_1(Q), \text{ch}_2(R)$ } := makePair( $\text{ch}_1(Q)$ ,
   $\text{ch}_2(R), \Delta\Phi_1^L, \Delta\Phi_1^U, \Delta\Phi_2^L, \Delta\Phi_2^U$ ).
```

```
computeBounds({ $\text{ch}_1(Q), \text{ch}_1(R)$ }).
```

```
computeBounds({ $\text{ch}_1(Q), \text{ch}_2(R)$ }).
```

```
if  $\rho(\{\text{ch}_1(Q), \text{ch}_1(R)\}) < \rho(\{\text{ch}_1(Q), \text{ch}_2(R)\})$ 
  undo({ $\text{ch}_1(Q), \text{ch}_1(R)$ }) := true.
```

```
else, undo({ $\text{ch}_1(Q), \text{ch}_2(R)$ }) := true.
```

```
insertHeap( $H, \{\text{ch}_1(Q), \text{ch}_1(R)\}$ ).
```

```
insertHeap( $H, \{\text{ch}_1(Q), \text{ch}_2(R)\}$ ).
```

```
{ $\text{ch}_2(Q), \text{ch}_1(R)$ } := makePair( $\text{ch}_2(Q)$ ,
   $\text{ch}_1(R), \Delta\Phi_1^L, \Delta\Phi_1^U, \Delta\Phi_2^L, \Delta\Phi_2^U$ ).
```

```
{ $\text{ch}_2(Q), \text{ch}_2(R)$ } := makePair( $\text{ch}_2(Q)$ ,
   $\text{ch}_2(R), \Delta\Phi_1^L, \Delta\Phi_1^U, \Delta\Phi_2^L, \Delta\Phi_2^U$ ).
```

```
computeBounds({ $\text{ch}_2(Q), \text{ch}_1(R)$ }).
```

```
computeBounds({ $\text{ch}_2(Q), \text{ch}_2(R)$ }).
```

```
if  $\rho(\{\text{ch}_2(Q), \text{ch}_1(R)\}) < \rho(\{\text{ch}_2(Q), \text{ch}_2(R)\})$ 
  undo({ $\text{ch}_2(Q), \text{ch}_1(R)$ }) := true.
```

```
else, undo({ $\text{ch}_2(Q), \text{ch}_2(R)$ }) := true.
```

```
insertHeap( $H, \{\text{ch}_2(Q), \text{ch}_1(R)\}$ ).
```

```
insertHeap( $H, \{\text{ch}_2(Q), \text{ch}_2(R)\}$ ).
```