Efficient Model Selection for Large-Scale Nearest-Neighbor Data Mining

Greg Hamerly and Greg Speegle

Baylor University, Waco, TX 76798, USA
greg_hamerly@baylor.edu, greg_speegle@baylor.edu

Abstract. One of the most widely used models for large-scale data mining is the k-nearest neighbor (k-nn) algorithm. It can be used for classification, regression, density estimation, and information retrieval. To use k-nn, a practitioner must first choose k, usually selecting the k with the minimal loss estimated by cross-validation. In this work, we begin with an existing but little-studied method that greatly accelerates the cross-validation process for selecting k from a range of user-provided possibilities. The result is that a much larger range of k values may be examined more quickly. Next, we extend this algorithm with an additional optimization to provide improved performance for locally linear regression problems. We also show how this method can be applied to automatically select the range of k values when the user has no *a priori* knowledge of appropriate bounds. Furthermore, we apply statistical methods to reduce the number of examples examined while still finding a likely best k, greatly improving performance for large data sets. Finally, we present both analytical and experimental results that demonstrate these benefits.

Key words: data mining, k nearest neighbor, optimal parameter selection

1 Introduction

The nearest neighbor algorithm is a powerful tool for finding data similar to a query example. It is popular because it is simple to understand and easy to implement, has good theoretical properties and flexible modeling assumptions, and can operate efficiently in low dimensions with indexes [7, 13, 3] and in high dimensions via approximations [1, 25]. The nearest neighbor algorithm is used in applications such as bioinformatics [17], multimedia databases [3], collaborative filtering [33] and many others. In fact, the k-nearest neighbor (k-nn) algorithm was chosen as one of the top 10 data mining algorithms in ICDM 2006 [32].

Before using k-nn, a practitioner must select the size of the neighborhood such that all neighbors are considered similar. The best size will depend on the application and the data. For k-nn, the neighborhood size is k, the number of nearby examples that are part of the neighborhood. In other contexts, the neighborhood size is the radius r of a sphere centered at the query example, and examples contained in the sphere are neighbors of the query example.

The choice of k has a large impact on the performance and behavior k-nn models. A small k allows simple implementation, permits efficient queries, and enjoys provable theoretical properties [4]. However, larger values of k tend to produce smoother models and are less sensitive to label noise. Ferrer-Troyano et al. [6] show that for some data sets the prediction error varied greatly depending on the value selected for k. Thus, the choice of k must be carefully made for the task at hand.

While the model selection problem is important, there has been little published research on selecting a good k for k-nn-based prediction models. Typically a practitioner tries several different k values, using cross-validation to measure prediction loss for each k, and chooses a k which has sufficiently small loss. This can be a very slow task on large datasets and when there are many possible k values. Thus, the focus of this work is on automating, accelerating, and extending this method of selecting k based on cross-validation in varying contexts.

In this paper, we provide algorithms for efficiently determining the best k (the one having the least prediction loss) under diverse situations. First, we consider the type of data analysis to be performed, classification or constant regression versus linear regression. While the first two types are easy to optimize, we show how to efficiently determine the best k when using linear regression. Second, we consider the case where the maximum possible best k (denoted K^*) is initially unknown, and show how to efficiently determine the appropriate K^*. Finally, we consider the case where the data set is extremely large. Statistical methods allow us to analyze a small fraction of the data with a high confidence of selecting a k with an estimated loss very close to that of the best k.

Section 2 contains related work on efficient k-nn search, optimal k selection, and efficient cross-validation. In Section 3, we present two algorithms for finding the best k within a user-specified range. The first is the naïve algorithm, which is presented for comparison purposes. The second incorporates a known optimization to provide clear benefits for some data mining applications. We provide explanation and analysis of these benefits. In Section 4, we show how this optimization does not provide the same benefits for locally linear regression problems, but additional techniques can be applied to achieve the same speedup. In Section 5, we consider the case of environments without *a priori* knowledge of an upper bound on k. Under these circumstances, it is possible for the naïve algorithm to *outperform* the optimized algorithm. We present a technique with asymptotic running time better than the naïve algorithm. Section 6 contains a methodology to apply statistical methods (BRACE [19]) to greatly reduce the number of examples examined in cross-validation on large data sets. Section 7 gives experimental evidence of the effectiveness of our proposed algorithms. In Section 8 we conclude with a discussion of our findings in this work, and future work.

2 Related work

Cross-validation is a simple and popular method to estimate the loss of a model in a way that maximizes the use of training data. Rather than just train on one subset of the data and estimate loss on the remaining part, cross-validation tries multiple train/test splits to get an average loss. Cross-validation was popularized by Stone [26] and Geisser [8]. Two common types of cross validation are leave-one-out (LOOCV) and m-fold (or k-fold as it's more commonly called). In m-fold cross-validation we partition the training data into m folds, train a model on $m - 1$ of the folds and then estimate loss by making predictions on the held-out fold. This is repeated m times, holding out a different fold each time. LOOCV is a particular type of m-fold cross-validation where m is the number of examples in the training set.

LOOCV has been shown to be optimal in the sense of minimizing the squared prediction error [16]. Ouyang et al. [23] recently showed asymptotic results for k-nn model selection using cross-validation, in the context of least-squares regression. Cross-validation can be computationally costly, since it requires training m models. In Section 3, we exploit the properties of LOOCV to efficiently perform k-nn model selection. In the remainder of this section, we review previous work on speeding up k-nn queries, model selection for k-nn, and speeding up cross-validation.

2.1 Speeding up k-nn

The vast majority of work on k-nn involves improving the query efficiency of finding nearest neighbors. The simplest solution requires $O(nd)$ time to perform a linear scan on the dataset and $O(n \log k)$ to maintain the k nearest neighbors with a distance-ordered priority queue. Indexing techniques reduce to this time when d is large, due to the curse of dimensionality [15]. Both memory- and disk-based indexes have been developed to help with specific types of queries, including k-d trees [7], R-trees [13] and metric trees [28].

In order to avoid the curse of dimensionality, dimension reduction techniques such as principal components analysis have been applied [18]. Other efficient approaches have been developed, including approximate solutions [25, 1] and techniques such as locality sensitive hashing [11]. All of these techniques can be used in conjunction with the algorithm improvements, as these issues are orthogonal to this work.

There are several papers on using locally adaptive methods for neighborhood size [30, 14, 29]. This paper is concerned with finding a globally-best k. Although there is an obvious application of the technique in this paper to locally adaptive k-nn, applying the optimization in this environment is ongoing research.

2.2 Selecting k for k-nn

Cover and Hart [4] proved that there exists a distribution for which no choice of k is better than $k = 1$. They also showed that the error rate of a 1-nn classifier is at most twice the Bayes' rate (the theoretical lower bound). Obviously, $k = 1$ is not always the best solution, as Devroye et al. [5] discuss the asymptotic prediction error bounds of nearest neighbor classifiers as $n \to \infty$ and $k/n \to 0$.

From an applied perspective, Ferrer-Troyano et al. [6] present a comparison of k-nn over various UCI datasets, showing that finding a 'best' k can be difficult. They observe that larger values of k produce smaller errors on some datasets, but low values of k are similar on other datasets. Additionally, some examples could not be classified correctly for any value of k. They do not mention an efficient way to determine the best k.

Olsson [22] studied the impact of training size on the best k for text retrieval problems. On large training sets a broad set of k values were similar, while small training sets required careful selection of k. The paper uses an algorithm similar to Algorithm 1, presented in Section 3, for finding the optimal k in each case. As an aside, our techniques in Section 5, would improve the performance of their work.

Loop reordering is mentioned briefly in [10] as an efficient means to perform many k-nn searches over the same data. The authors of Weka [31] implement the loop reordering presented in Section 4. However, this paper significantly extends these works by analyzing the efficiency gains, extending the technique to additional decision models, discovering novel algorithms resulting from the loop reordering, determining reasonable bounds for k when the maximum is not known, and using statistical techniques to efficiently approximate k for very large data sets.

2.3 Speeding up cross-validation

Several techniques have been developed to improve the performance of cross-validation. Mullin and Sukthankar [21] developed an efficient method of calculating the 'complete cross-validation' error. Complete cross-validation considers all possible divisions of the data into training and test data, for a fixed training data size. Their approach uses combinatorics to determine the number of training sets that would correctly classify a test example. Similar to our approach, their optimization comes from a view centered on the entire data set, rather than on the training sets. Their approach is only directly applicable to binary classification tasks, not for multiclass or regression problems.

Moore and Lee [19] used LOOCV to compare the accuracies of many different learning models and to select a best model in terms of minimal loss. Their approach looks at each example in turn, computing its loss for every model in contention. Whenever a model is statistically unlikely to be the best, it is eliminated from consideration (called *racing*). Likewise, if two models are statistically similar, one is dropped from the competition (called *blocking*). Combining these yields their BRACE algorithm, which can usually select a best model without having to compute the cross-validation loss for every example. We show how the BRACE method can be used to select the likely best k in Section 6.

Blockeel and Struyf [2] use a technique similar to the loop reordering of this paper for efficient cross-validation during the decision tree induction process. They apply each test only once per test example and store the results for each possible training set that contains that example. This generates a $O(m)$ speedup (where m is the number of folds). Unfortunately, in some cases, other factors dominate the cost of the execution so that overall performance is not always improved. Blockeel and Struyf have also proposed methods of parallelizing cross-validation for more general inductive logic programming tasks [27].

Racine [24] showed how to reduce the computation necessary for cross-validation of globally linear models, using incremental update techniques. His work was specifically for 'h-block' cross-validation, which is where observations close in time are assumed to be dependent, and observations that are far apart in time are assumed independent. In h-block cross-validation, making a prediction on an observation at time t involves leaving out the examples at times $t - h$ through $t + h$. We show in Section 4 how to use the optimized algorithm for efficiently updating linear models in Racine's work.

3 Efficient cross-validation for selecting k

To serve as the basis for improved best k selection, we first present Algorithm 1, a naïve algorithm for best k selection using LOOCV. The algorithm selects a value of k from a range of values $\{1 \dots K^*\}$.

- $\mathcal{X}, \mathcal{Y}$ – Domain, range of prediction problem.

Algorithm 1 Naive K-nn LOOCV(K^*, λ, X, Y)

```
 1: for $k = 1$ to $K^*$ do
 2:     loss($k$) $\leftarrow 0$
 3:     for $i = 1$ to $n$ do
 4:         $N \leftarrow$ KnnQuery($k, x_i, X - \{x_i\}$)
 5:         $s \leftarrow 0$
 6:         for all $j \in N$ do {Sum neighbor labels}
 7:             $s \leftarrow s + y_j$
 8:         $\hat{y} \leftarrow s/k$ {Prediction is average of labels}
 9:         loss($k$) $\leftarrow$ loss($k$) $+ \lambda(\hat{y}, y_i)$
10: return argmin$_k$ loss($k$)
```

Algorithm 2 Optimized K-nn LOOCV(K^*, λ, X, Y)

```
 1: loss($k$) $\leftarrow 0$ for $1 \leq k \leq K^*$
 2: for $i = 1$ to $n$ do
 3:     $N \leftarrow$ KnnQuery($K^*, x_i, X - \{x_i\}$)
 4:     Sort($N$) {by increasing distance from $x_i$}
 5:     $s \leftarrow 0$;    $k \leftarrow 1$
 6:     for all $j \in N$ do {From nearest to furthest}
 7:         $s \leftarrow s + y_j$
 8:         loss($k$) $\leftarrow$ loss($k$) $+ \lambda(y_i, s/k)$
 9:         $k \leftarrow k + 1$
10: return argmin$_k$ loss($k$)
```

– n – Number of training examples.
– $X = \{x_1, \ldots, x_n \mid x_i \in \mathcal{X}\}$ – Training examples.
– $Y = \{y_1, \ldots, y_n \mid y_i \in \mathcal{Y}\}$ – Training labels.
– K^* – Maximum reasonable value of k.
– KnnQuery : $k \times \mathcal{X} \times \mathcal{X}^{n-1} \rightarrow \mathcal{Z}^k$ – Search algorithm that finds the k nearest neighbors of a query example from a dataset with $n - 1$ examples (the query example being held out).

Algorithm 1 runs in $O\left(n \sum_{k=1}^{K^*} Q(k, n) + n{K^*}^2\right)$ time, where $Q(k, n)$ is the running time for finding the k nearest neighbors in a database with n rows. The summation is not in closed form because $Q(k, n)$ depends on the search algorithm used, which has not yet been specified (see Table 1 for specific examples). We expect that in most applications the calls to KnnQuery will dominate this algorithm's running time. Making a k-nn prediction on held-out x_i requires the following steps:

1. Finding the k nearest neighbors of x_i from $X - \{x_i\}$ (line 4). In practice, to avoid actually removing x_i from the search space, we request the $k + 1$ nearest neighbors of x_i and discard x_i from the result set.
2. Building a model based on those neighbors, and making a prediction for x_i (in this case, constant regression, lines 5-9 average the labels of the k nearest neighbors).

These two steps are **computationally wasteful**, since the cross-validation repeats these computations over multiple values of k. For a single held-out example x_i, the innermost loop computes the $k = 1$ nearest neighbors, then the $k = 2$ nearest neighbors, etc., as well as summing the labels for these nearest neighbor sets. Obviously, the nearest neighbors for $k = m$ includes the $k = m - 1$ nearest neighbors. Thus, it should be sufficient to search for the K^* nearest neighbors of each held-out example **only once**, and derive necessary information for all smaller k. This simple observation has been previously exploited (e.g. in Weka) and provides a speedup for constant regression and classification that will be the basis of our further optimizations.

3.1 Optimized LOOCV to select k

Three changes to Algorithm 1 enable a large potential speedup. These key changes are (1) reordering the nesting of the two outermost loops, (2) searching for all K^* nearest neighbors at once, and (3) computing label predictions and loss for each value of k in an incremental fashion. The result is shown in Algorithm 2.

Algorithm 2 will generally greatly outperform Algorithm 1 because it calls KnnQuery only once (instead of K^* times) per held-out example. The algorithm uses the K^* neighbors to construct the predictions (and losses) for all $k \leq K^*$ nearest neighbors. To do this incrementally, it considers the K^* neighbors of the held-out example in order of increasing distance from the example. Thus, Algorithm 2 requires a sort which Algorithm 1 does not. All these transformations incur a negligible amount of extra memory over Algorithm 1.

Algorithm 2 runs in $O\left(nQ(K^*, n) + nK^* \log(K^*)\right)$ time. The term $nQ(K^*, n)$ comes from executing KnnQuery n times, once per held-out example. The term $K^* \log(K^*)$ comes from the call to Sort, which

Table 1. Running time comparison for Algorithms 1 and 2 for two different k-nn implementations.

k-nn query	$Q(k,n)$	Algorithm 1	Algorithm 2	**Speedup**
Linear scan	$n\log(k)$	$O(n^2K^*\log(K^*))$	$O(n^2\log(K^*))$	$O(K^*)$
Tree-based search	$k\log(n)$	$O(n\log(n)K^{*2})$	$O(n\log(n)K^*)$	$O(K^*)$

is also called once for each held-out example. As before, $Q(k,n)$ represents the running time for the k-nn query algorithm. In the following subsection, we investigate the difference in running time between Algorithms 1 and 2 for particular k-nn query algorithms.

3.2 Algorithm analyses for specific Q

There are many ways to implement the KnnQuery algorithm. Significant research has gone into speeding up k-nn queries under different assumptions or conditions, such as low dimension indexes [7, 13, 28], approximate answers [25, 1, 15, 11], high-dimensional data [18, 15], etc. While there are many possible implementations, we consider the running times of two typical representatives for comparison under our proposed cross-validation algorithm. Table 1 shows an analysis of the running times of Algorithms 1 and 2 when using either a linear scan algorithm or an efficient tree-based index search for KnnQuery.

The linear scan examines all $n-1$ examples from $X - \{x_i\}$ and keeps track of the k nearest neighbors using a priority queue ordered by distance from the query example. This leads to a running time of $Q(k,n) = O(n\log(k))$. It may seem inefficient to scan all examples, but most indexing methods degrade to this behavior in high dimensions, due to the curse of dimensionality.

Searching a tree-based index, such as a k-d tree, can often be much more efficient than linear scan, especially in low dimensions and clustered data. It saves work by pruning branches of the tree based on its current estimate of the furthest nearest neighbor. Like the linear scan, it also maintains a priority queue of nearest neighbors. Our empirical evidence strongly suggests that for uniform data distribution, a simple k-d tree search will yield an *expected* running time of $O(k\log(n))$ for exact search of the k nearest neighbors.

For each analysis, we start from the running times of Algorithms 1 and 2 and use the appropriate form of $Q(k,n)$. The analyses simplify to the given forms under the natural assumption that $n \gg k$. In both cases, Algorithm 2 gives a speedup of $O(K^*)$ over Algorithm 1. This speedup is conserved over more costly k-nn implementations, as long as $Q(k,n)$ is polynomial in k and n.

4 Locally linear regression problems

The loop reordering optimization proposed in Section 3 is simple and mentioned elsewhere [10, 31]. It provides benefits for selecting k under different scenarios: constant regression (as already demonstrated), binary classification, and multi-class classification. However, the beauty of this algorithm (and the contribution of this paper), becomes more evident when it is applied in less obvious cases where it affords advantages not possible under the naïve cross-validation algorithm. One such application is locally-linear regression.

We can use k-nearest neighbor search in combination with a linear regression model to form a locally-linear regression model. This model adapts its regression coefficients based on the location of the desired prediction. Making a prediction on a query example requires finding its nearest neighbors, fitting a linear model to the neighbors, and using the model to make a prediction at the query example.

Consider adapting Algorithms 1 and 2 for finding an appropriate k for the locally-linear regression task. Let $X_k \in \mathbb{R}^{k\times d}$ represent the matrix of d attributes from each of the k nearest neighbors, and let $Y_k \in \mathbb{R}^k$ represent the column vector of output values for those neighbors. Estimating model coefficients for the locally linear model via least-squares requires finding $\hat{\beta}_k = (X_k^TX_k)^{-1}X_k^TY_k$, which can be computed in $O(kd^2)$.

Algorithms 1 and 2 make the same number of model predictions, namely nK^*. Thus, computing $\hat{\beta}$ in $O(kd^2)$ for each model prediction adds a term of $O(nK^{*2}d^2)$ to the run time of both algorithms. While Algorithm 2 is still faster, it does not maintain an $O(K^*)$ speedup, since this additional term is *added* to both algorithms' run times. Nevertheless, the optimization unlocks further possibilities for speedup.

We can improve the performance of Algorithm 2 by incrementally calculating $\hat{\beta}_{k+1}$ from the partial calculations for $\hat{\beta}_k$, and the (k+1)st neighbor, rather than calculating it from scratch from all $k+1$ nearest neighbors. Racine [24] used this approach for global linear regression. To initialize this process, prior to the loop, we calculate the value of $\hat{\beta}_d$ (for the first d neighbors). We retain the two parts used to compute $\hat{\beta}_d$, i.e. $(X_d^T X_d)^{-1}$ and $X_d^T Y_d$, and update them during each iteration.

In particular, $X_{k+1}^T X_{k+1} = X_k^T X_k + x^T x$, where x is the (k+1)st nearest neighbor, and $x^T x$ is a $d \times d$ matrix of rank 1. The key observation is that the Sherman-Morrison formula [12, p. 50] allows us to calculate the inverse of $X_{k+1}^T X_{k+1}$ from $x^T x$ and the (already-computed) inverse of $X_k^T X_k$ in only $O(d^2)$ operations, which is asymptotically faster than taking the inverse of $X_{k+1}^T X_{k+1}$ directly.

The other part used in calculating $\hat{\beta}_k$, namely $X_k^T Y_k$, can be updated with the (k+1)st nearest neighbor to form $X_{k+1}^T Y_{k+1}$ in $O(d)$ time. Putting together these two parts, we can compute $\hat{\beta}_{k+1}$ incrementally more efficiently than computing it from the original matrices. This reduces the cost of estimating the linear model coefficients to $O(nK^* d^2)$. Thus, with incremental computation of $\hat{\beta}$, Algorithm 2 maintains a speedup of $O(K^*)$ over Algorithm 1. This additional speedup due to the Sherman-Morrison formula is not possible in Algorithm 1. In fact, this algorithm demonstrates one of the fundamental properties for improving performance with Algorithm 2– incremental evaluation of the prediction error.

The model we have discussed is linear in the coefficients of the model. Thus, we can apply this locally-linear regression speedup to any basis expansion model that is linear with respect to the estimated coefficients. For example, this speedup applies directly to polynomial regression.

5 Execution under Unknown K*

Algorithms 1 and 2 both require K^* as input. However, in some applications a reasonable upper bound on k is unknown. For instance, in [22], the best k is determined when no improvement in the error has been seen for 15 iterations. Fortunately, it is straightforward to modify both algorithms to dynamically determine an appropriate value for K^*. We first introduce a predicate called *nextk()*, which examines the results of cross-validation thus far and returns true as long as a larger value of k is worth examining. In later analysis we assume the running time of *nextk()* is negligible.

5.1 Modifications to determine K*

We modify Algorithm 1 to dynamically determine K^* by substituting '**while** *nextk()* **do**' in place of '**for** $k = 1$ to K^* **do**'. This supports both the case where K^* is known in advance and the case where it must be determined dynamically. For example, in [9], $K^* = \sqrt{n}$, so *nextk()* would return true while $k \leq \sqrt{n}$. Alternatively, in [22], where K^* is unknown, *nextk()* would return true as long as $k \leq \text{argmin}_i \text{ loss}(i) + 15$.

In Algorithm 2, it is assumed that K^* is known. Therefore, our approach is to wrap the algorithm in an outer loop which chooses K^*, runs Algorithm 2, and then consults *nextk()* to determine if K^* is large enough. If not, the outer loop doubles K^* and repeats. This wrapper can start with any value of K^* that prior knowledge suggests; otherwise, it starts with $K^* = 1$. We investigate the efficiency of this wrapper under two conditions: the initial guess for K^* is too high, or too low. If the initial K^* is exactly right, then Algorithm 2 keeps the same $O(K^*)$ speedup over Algorithm 1.

For the remainder of this section, when referring to Algorithms 1 and 2, we mean the modified versions of these algorithms that dynamically determine K^* by using *nextk()*, as described above. We define K_N^* (K_O^*) to be the maximum value of k considered in Algorithm 1 (2). Note that $K_N^* \leq K_O^*$, since K_N^* increments by one until *nextk()* returns false, but K_O^* doubles each time.

5.2 When K* is larger than necessary

If K_O^* is chosen to be larger than necessary, then Algorithm 2 wastes computation. In particular, KnnQuery searches for more neighbors than necessary. Until *nextk()* returns false, however, Algorithm 1 performs more k-nn lookups than Algorithm 2. Thus, there is a break-even point between the two algorithms.

Using Table 1, with tree-based search, Algorithms 1 and 2 have running times of $O(n\log(n){K_N^*}^2)$ and $O(n\log(n)K_O^*)$, respectively. This suggests that as long as $K_O^* = O({K_N^*}^2)$, Algorithm 2 is the best choice. Even if K_O^* is too large, it is unlikely to be a factor of K_N^* too large in practice.

If KnnQuery is a linear scan, the running times for Algorithms 1 and 2 are $O(n^2K_N^*\log(K_N^*))$ and $O(n^2\log(K_O^*))$. In this case, Algorithm 2 will perform better as long as the initial value of $K_O^* = O({K_N^*}^{K_N^*})$, which is likely to be true in practice unless an initial guess is grossly too large.

5.3 When K* is too small

When the initial guess of K_O^* is too small, all of the work performed by Algorithm 2 is wasted; K_O^* must be doubled and the process restarted. We double K_O^* because the penalty for guessing too high is small. Eventually, we reach the case where $K_O^* \geq K_N^*$. Assuming that K_O^* starts at 1, the number of times that Algorithm 2 must double K_O^* is $t = \lceil \log_2(K_N^*) \rceil$. With the tree-based search, Algorithm 1 requires $O(n\log(n){K_N^*}^2)$ time, and Algorithm 2 requires $O(n\log(n)\sum_{i=0}^{t} 2^i) = O(n\log(n)K_N^*)$ time. Note that this analysis gives the same result as the analysis of Algorithm 2 in Table 1. This analysis suggests that Algorithm 2 is still asymptotically faster than Algorithm 1, even when searching for K_O^*.

When using a linear scan, Algorithm 1 has a running time of $O(n^2K_N^*\log(K_N^*))$, while Algorithm 2 now requires $O(\sum_{i=0}^{t} n^2\log(2^i)) = O(n^2\log^2(K_N^*))$ time. Clearly, the wrapped version of Algorithm 2 should outperform Algorithm 1, even when K^* is unknown.

If K_N^* is large, larger initial values of K_O^* will improve the performance of Algorithm 2 (over choosing $K_O^* = 1$). A larger guess avoids lost work on too-small K_O^* values. For example, in [22], the desired R-precision influences best k selection. For high R-precision, the initial guess for K^* should be 64, while for lower R-precision, an initial guess of 16 would find the best k with less work. Other optimization techniques, including parallelization across different values for K_O^*, are part of our ongoing research.

6 Statistical Methods for Large Data Sets

Although the optimized algorithm improves the performance for finding the best k by a factor of K^*, the dominant term in the running time is the number of times KnnQuery must be called. Clearly, reducing the number of queries could greatly improve the running time of the optimized algorithm over large data sets.

Selecting the best k can be thought of as selecting the best model from a set of K^* models. As such, the BRACE algorithm [19] can be combined with Algorithm 2 to determine when there is a low probability that any other model will have a lower loss than the selected value of k. We denote this new algorithm as Alg 2+BRACE. BRACE eliminates models which are unlikely to be the best by regular statistical comparisons. Models may be eliminated before seeing all the examples. BRACE assumes that model losses are distributed independently throughout the dataset. As a result, the best k returned by Alg 2+BRACE may not be the same as under the Algorithm 2, but it is likely to have a very similar loss value.

Alg 2+BRACE requires two parameters to determine the likelihood the chosen model is the best. Larger values for these parameters yield quicker results, but with greater probability that the selected model does not have the lowest loss. The hypothesis test error rate δ indicates the probability of making an error on eliminating a model. The loss indifference parameter γ defines the range of loss values within which two models are considered similar. Alg 2+BRACE works as follows:

- Start with an initial K^*.
- For each k maintain a status (active/inactive) and two sufficient statistics: the sum of prediction losses, and the sum of the squared prediction losses (on examples seen thus far).
- After each interval of t examples, compare each pair of active k values using hypothesis tests. If some k is unlikely to have significantly lower loss than another, mark k as inactive.
- If two values of k are determined to have similar loss, mark the greater k as inactive
- Reduce K^* to the largest active k.
- Stop if only one k is active or all examples have been examined. Choose the active k with minimum loss.

Table 2. A summary of the experiments according to dataset, size, task, and implementation. The two implementations are ANN tree-based index search [20], and linear scan which examines every example. Note that we test sine10d on two different tasks.

	dataset	n	d	# predictions	task	KnnQuery
K^* known (section 7.1)	magic04	19,020	10	19,020	binary classification	tree index (ANN)
	sine10d	10,000	10	10,000	locally constant regression	tree index (ANN)
	sine10d	10,000	10	10,000	locally linear regression	tree index (ANN)
	Netflix	480,189	17,770	100,480,507	locally constant regression	linear scan
K^* unknown (section 7.2)	sine4d	10,000	4	10,000	locally constant regression	tree index (ANN)
	sine32d	10,000	32	10,000	locally constant regression	linear scan
BRACE (7.3)	sine2d	$10e3$ - $10e7$	2	$10e3$ - $10e7$	locally constant regression	tree index (ANN)

This algorithm has the potential for computational savings by reducing K^* whenever possible, and also by stopping before examining the entire training set. However, it does introduce overhead in the hypothesis testing and, to a much lesser extent, in the maintenance of additional statistics. The hypothesis testing is between all pairs of models, resulting in $O(K^{*2})$ model comparisons. The hypothesis tests can be applied after the examination of each example, but the number of models eliminated per example is typically very small. Thus, we introduce an interval, t, as the number of examples processed before the hypothesis testing is applied. Optimal selection of t is ongoing work.

Experimental results (Section 7.3) indicate this overhead can be significant for small data sets, causing Alg 2+BRACE to be slower than Algorithm 2. However, with an application specific well-chosen t and large data sets, the savings far outweigh the overhead.

7 Experimental results

We now consider the running time of Algorithms 1 and 2 under various conditions. We report three categories of experiments: when K^* is known; when K^* is unknown; and when applying the BRACE algorithm. When K^* is known, we compare the running times of these algorithms on several classification and regression datasets, including using the Sherman-Morrison improvement. When K^* is unknown, we compare both the linear scan and tree-indexed approaches. For BRACE, we use one very large regression dataset, allowing us to compare it directly with algorithm without BRACE. The datasets we used are (see also Table 2):

Magic04 is a binary classification dataset from the UCI repository with 19,020 examples, each having 10 features. This data comes from a Monte Carlo simulation of high-energy gamma particles. The task is to predict whether a record indicates a gamma particle or background noise.

SineXd represents a set of similar synthetic regression datasets having $2 \leq X \leq 32$ features. Each feature is drawn from the $[0, 1]$ uniform distribution. Denoting feature j of example i as x_{ij}, we generate the label for x_i as $y_i = \sin(\sum_{j=1}^{X} x_{ij}^2) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, 0.05)$.

Netflix is a large, sparse regression dataset. It contains 480,189 users who have provided 100,480,507 ratings of 17,770 movies. The goal is to predict a user's movie rating, based on ratings other similar users gave that movie. User similarity is based on Pearson correlation between the ratings on movies both users have rated (excluding the query movie). Each rating requires a k-nn query for prediction.

Our primary focus is on running time rather than prediction accuracy. Thus, these large datasets which cover a wide range of prediction tasks are sufficient for showing the speedups of the proposed methods.

Our experiments run on a four-core, 3 GHz Intel Xeon with 8 GiB of RAM running Linux 2.6. We use the Approximate Nearest Neighbor (ANN) library [20] as a k-d tree search algorithm for k-nn queries on low-dimensional datasets. For high-dimensional datasets, we use a linear scan algorithm, and on the Netflix dataset we optimize the fact that there are multiple similar k-nn queries per user. Further, we parallelized the Netflix experiment across four processors. For experiments comparing known and unknown K^*, we report only running times, since both algorithms produce the same cross-validation results. Figure 3 reports the differences in best k reported using Alg 2+BRACE.

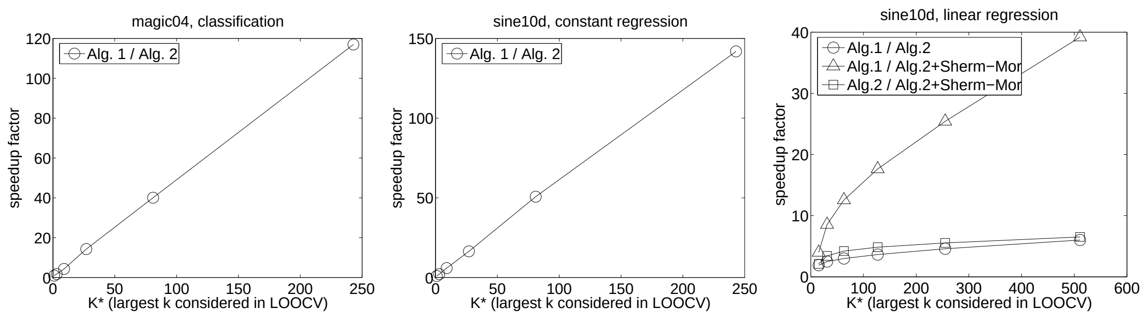

Fig. 1. These graphs show the relative speedup of Algorithm 2 over Algorithm 1 for three datasets. Each line shows speedup: the time of the slower algorithm divided by that of the faster algorithm. Speedup is linear with K^* for classification, constant regression, and linear regression with the Sherman-Morrison incremental evaluation.

7.1 Experiments with known K*

Figure 1 shows the relative speedup of Algorithm 2, defined as the time for Algorithm 1 divided by the time for Algorithm 2. Each experiment performed LOOCV for varying values of K^*. Clearly, the optimization gives vast improvements in running time in each task, allowing searches over much larger values of k much more quickly. For instance, we could search with $K^* = 250$ with Algorithm 2 in less time than it takes Algorithm 1 to search with $K^* = 30$.

For the locally linear regression experiment, we implemented two versions of Algorithm 2 – one that recomputed the linear model each time, and one that used the Sherman-Morrison incremental calculations discussed in Section 4. The Sherman-Morrison optimization yields significant additional speedup, and is not possible with the original algorithm.

We ran the parallel optimized algorithm on the Netflix dataset with $K^* = 500$ in 78 wall-clock hours (310 CPU hours). This was with a linear scan-based k-nn algorithm; a highly optimized algorithm should be able to easily handle the entire dataset even more quickly. For a fixed k, there are over 100 million rating predictions to make. Trying $K^* = 500$ predictions per user rating meant that the algorithm actually made 50 billion predictions in that time – approximately 180,000 predictions per wall-clock second. We did not run the naïve algorithm on this dataset, which we predict would take about 2.2 wall-clock years (based on relative speeds observed in other experiments).

7.2 Experiments with unknown K*

For the algorithms in Section 5, we verify the analysis with experiments on both low dimensional data with effective indexes and high dimensional data using a linear scan. In all cases, 10,000 examples are used. Table 2 reports additional details of the experiments.

Figure 2 reports the results of our experiments. Each line represents a different initial guess for K^*_O (1, 8, 32 and 128). The x-axis represents the range of values for K^*_N. A point on the line represents the speedup of Algorithm 2 over 1. Note the jagged lines for most initial guesses – the drops correspond to those points where Algorithm 2 restarts with a larger K^*, so the work up to that point is wasted. As predicted, the better the initial guess, the better the performance gain.

As K^*_N increases, even vaguely reasonable guesses for K^*_O give significant speedup. The middle graph provides greater detail of the rectangle in the left graph, showing those conditions which allow Algorithm 1 to outperform Algorithm 2. Points below the line speedup=1 (middle graph only) indicate these conditions, e.g., an initial guess of a very large K^*_O results in poor performance, if K^*_N actually is small. However, an initial guess of $K^*_O = 1$ performs at least as well as the Algorithm 1 under all conditions *except* when $K^*_N = 3$. In the right graph, the performance gains are even higher because the linear scan k-nn search is even more costly (relative to other cross-validation calculations) than an indexed search.

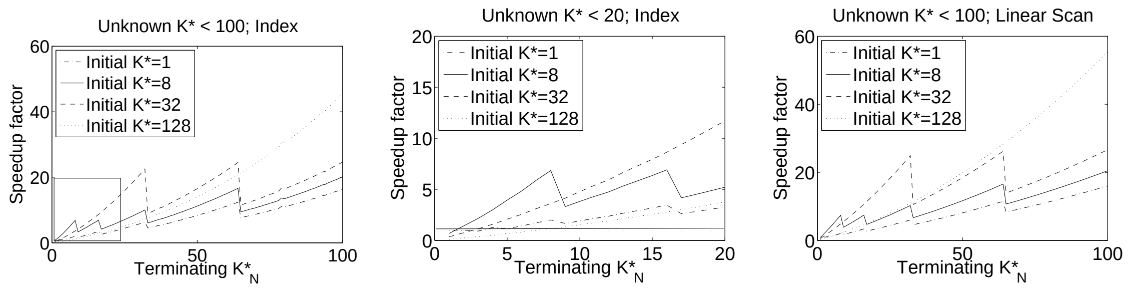

Fig. 2. The running time of Algorithm 1 divided by the running time of Algorithm 2 for various ranges of K_N^*. The task is constant regression. The data have 4 dimensions (indexed search) or 32 dimensions (linear scan). Each line is a different initial guess for K_O^*. The rectangle in the left graph is blown up to give the middle graph. Note that the initial guess of 128 outperforms a guess of 32 when $K_N^* \geq 33$. This is exactly the point where a new K_O^* is required. This implies that the best performance comes from an intelligent, sufficiently large initial guess.

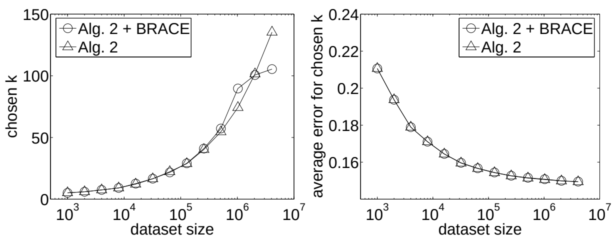

Fig. 3. The left plot shows the k chosen by Algorithm 2with and without BRACE on the sine2d dataset for varying n. The right graph shows average LOOCV error for both algorithms (the lines overlap).

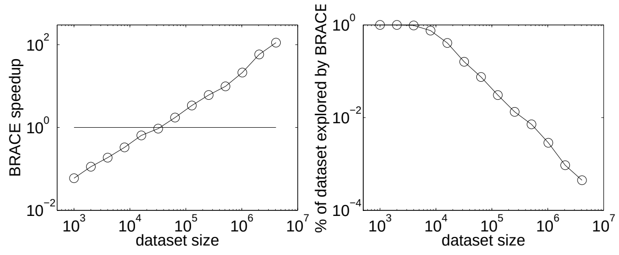

Fig. 4. The left plot shows the speedup of using BRACE with Algorithm 2. The right plot shows the percentage of examples that BRACE examines. Both are on a log-log scale. The speedup plot shows a line at speedup=1. For large datasets using BRACE gives 100-fold speedup.

7.3 Experiments with BRACE

We perform a number of experiments combining the BRACE algorithm with Algorithm 2 to see how much further the model selection process can be accelerated. We compare Algorithm 2 for selecting k in k-nn search with the Alg 2+BRACE version. We use 0.001 for both BRACE parameters γ and δ, and compare models every 10 examples. We show average performance over 20 synthetically generated versions of the sine2d dataset. Alg 2+BRACE produces an extremely fast model selection algorithm for k-nearest neighbor search. While the two versions do not always choose the same k, the chosen k gave nearly the same loss, so that the average loss Alg 2+BRACE is always within 0.003 of the best loss. Figure 3 shows the results over various dataset sizes of the chosen k and the resulting losses, which are indistinguishable due to overlap.

Figure 3 shows that the LOOCV-selected k increases as n increases. The reason for this is as n increases the density of samples across the domain increases, and so does the expected number of neighbors within a fixed radius of any point. Nearby neighbors in regression tend to have the same labels, so using more neighbors in a denser space is beneficial. In this experiment the best k for small n is still a good k for larger n in terms of providing low loss, it's just that an even larger k provides even lower loss for large n.

Despite giving nearly equal losses, Alg 2+BRACE was much faster for large datasets. Figure 4 shows the computational speedup as well as the percentage of the total dataset that Alg 2+BRACE examined. Recall that Alg 2+BRACE stops execution early if it determines a single best k, so on large datasets it usually will stop before it has examined all the examples. Indeed, the second plot shows that as n increases, the percentage of the dataset examined drops dramatically. The details are even more surprising – as n increases, the total number of examples that BRACE examines initially increases, and then actually decreases. While this might seem counter-intuitive, there's a good reason.

On this dataset (and likely most regression datasets) the best k is positively related to n. When comparing two model's estimated losses, BRACE's hypothesis tests use both the difference and the variance of the difference. This variance tends to go down as more examples are examined, at a rate of $1/\sqrt{i}$ for i examples. When k is large, regression noise tends to average out, resulting in lower prediction variance. When the best k is small, BRACE has a difficult time differentiating between the various k values because their predictions tend to have high variance. Thus BRACE must examine many examples until the estimator variance is low enough to cause a significant test, thus eliminating a model. When the best k is large, small values of k are quickly eliminated because their errors are significantly larger. Then, deciding among different large values of k proceeds quickly, because the variance between them is low. BRACE's indifference parameter γ helps speed up this process significantly by eliminating near-identical models.

8 Conclusion

We present detailed analysis and experimental evidence for improving k-nn model selection. Given a maximum reasonable k value, called K^*, the optimized algorithm is $O(K^*)$ faster than the naïve algorithm for leave-one-out cross-validation. This speedup holds for both indexed and linear-scan k-nn queries. The optimization enables additional performance enhancements, such as using the Sherman-Morrison formula for incremental calculations in locally linear regression tasks.

When K^* is not known *a priori*, we present an algorithm to efficiently find it, given a stopping criterion. It starts with an initial guess for K^* and doubles K^* if the criterion has not been met. This adaptive version still has asymptotic running time less than the naive algorithm.

Finding the k-nn for very large datasets is computationally demanding. The BRACE algorithm [19] can be combined with Algorithm 2 to both reduce the number of examples examined and K^*. The resulting algorithm provides dramatic additional speedup for large data sets.

Although the results we have shown in this paper are significant, additional issues include:

- Search for K^* with a number of parallel processes which use different K^* values (using powers of two). All machines terminate when any finds a satisfactory K^*.
- Leave-one-out cross validation is easy to optimize, but m-fold and h-block cross-validation appear more difficult. This is especially true when using indexes such as k-d trees.
- In this work we considered the problem of finding a single best k. We would like to apply this optimization to locally adaptive k-nn problems [30].
- When the underlying dataset is changing, as in a large dynamic database, the question arises of how to efficiently maintain the best k.
- A key to efficiency in the optimized algorithm is the fact that nearest-neighbor models are nested. We would like to explore speedups in other similarly structured models.

Although the basic ideas behind Algorithm 2 are straightforward and known, their implications are significant and underexploited. The optimization allows far more values of k to be considered in the same amount of time. It also allows further optimizations such as Sherman-Morrison, and it surprisingly outperforms Algorithm 1 in the case where K^* is unknown. Finally, BRACE provides significant additional speedup by avoiding examining all examples. Combined, these ideas provide a powerful tool for practitioners to automatically perform k-nn model selection easily and efficiently.

References

1. S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1999.
2. H. Blockeel and J. Struyf. Efficient algorithms for decision tree cross-validation. In *International Conference on Machine Learning*, pages 11–18, 2001.
3. C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

4. T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.
5. L. Devroye, L. Györfi, and G. Lugosi. *A Probabilistic Theory of Pattern Recognition*. Springer, 1996.
6. F. Ferrer-Troyano, J. Aguilar-Ruiz, and J. Riquelme. Empirical evaluation of the difficulty of finding a good value of k for the nearest neighbor. In *International Conference on Computational Science*, pages 766–773, 2003.
7. J. H. Friedman, J. L. Bentley, and R. A. Finkel. Two algorithms for nearest-neighbor search in high dimensions. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.
8. S. Geisser. The predictive sample reuse method with applications. *Journal of the American Statistical Association*, 70(350):320–328, 1975.
9. A. Ghosh, P. Chaudhuri, and C. Murthy. On visualization and aggregation of nearest neighbor classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(10):1592–1602, 2005.
10. A. K. Ghosh. On nearest neighbor classification using adaptive choice of k. *Journal of Computational and Graphical Statistics*, 16(2):482–502, 2007.
11. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *International Conference on Very Large Data Bases*, pages 518–529, 1999.
12. G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, second edition, 1996.
13. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM International Conference on Management of Data*, pages 47–57, 1984.
14. T. Hastie and R. Tibshirani. Discriminant adaptive nearest neighbor classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(6):607–616, 1996.
15. P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *ACM Symposium on Theory of Computation*, pages 604–613, 1998.
16. K.-C. Li. Asymptotic optimality for c_p, c_l, cross-validation, and generalized cross-validation: Discrete index set. *The Annals of Statistics*, 15(3):958–975, 1987.
17. L. Li, C. Weinberg, T. Darden, and L. Pederson. Gene selection for sample classification based on gene expression data: Study of sensitivty to choice of parameters of the ga/knn method. *Bioinformatics*, 17(12):1131–1142, 2001.
18. K.-I. Lin, H. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *The International Journal on Very Large Databases*, 3(4):517–542, 1994.
19. A. Moore and M. S. Lee. Efficient algorithms for minimizing cross validation error. In *International Conference on Machine Learning*, pages 190–198, 1994.
20. D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. `http://www.cs.umd.edu/~mount/ANN/`, 2006.
21. M. Mullin and R. Sukthankar. Complete cross-validation for nearest neighbor classifiers. In *International Conference on Machine Learning*, pages 639–646. Morgan Kaufmann, 2000.
22. J. S. Olsson. An analysis of the coupling between training set and neighborhood sizes of the knn classifier. In *SIGIR*, pages 685–686, 2006.
23. D. Ouyang, D. Li, and Q. Li. Cross-validation and non-parametric k nearest-neighbor estimation. *Econometrics Journal*, 9:448–471, 2006.
24. J. Racine. Feasible cross-validatory model selection for general stationary processes. *Journal of Applied Econometrics*, 12(2):169–179, 1997.
25. G. Shakhnarovich, P. Indyk, and T. Darrell, editors. *Nearest-Neighbor Methods in Learning and Vision*. MIT Press, 2006.
26. M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society, Series B*, 36(2):111–147, 1974.
27. J. Struyf and H. Blockeel. Efficient cross-validation in ILP. In *International Conference on Inductive Logic Programming*, pages 228–239, 2001.
28. J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Applied Mathematics Letters*, 4:175–179, 1991.
29. J. Wang, P. Neskovic, and L. N. Cooper. Neighborhood size selection in the k-nearest-neighbor rule using statistical confidence. *The Journal of the Pattern Recognition Society*, 39(3):417–423, 2006.
30. D. Wettschereck and T. G. Dietterich. Locally adaptive nearest neighbor algorithms. *Advances in Neural Information Processing Systems*, 6:184–191, 1994.
31. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. 1999.
32. X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.
33. G.-R. Xue, C. Lin, Q. Yang, W. Xi, H.-J. Zeng, Y. Yu, and Z. Chen. Scalable collaborative filtering using cluster-based smoothing. In *SIGIR*, pages 114–121, 2005.